Modelado e implementación de sistemas de tiempo real en procesadores muchos núcleos basados en malla

Modeling and implementing real-time systems in mesh-based many-core processors

Realizada por: David García Villaescusa

Tesis doctoral
Programa de Doctorado en Ciencia y Tecnología

Dirigida por:
Mario Aldea Rivas
Michael González Harbour

Noviembre 2021

Ítaca te brindó tan hermoso viaje.
Sin ella no habrías emprendido el camino.
Pero no tiene ya nada que darte.
Aunque la halles pobre, Ítaca no te ha engañado.
Así, sabio como te has vuelto, con tanta experiencia,
entenderás ya qué significan las Ítacas.

Fragmento de Ítaca, por Konstantino Kavafis

Para los que no nos pudieron acompañar en el camino.

Declaración

Yo en este texto declaro que salvo las referencias específicas que se han realizado sobre el trabajo de otros, el contenido de este trabajo es original y no ha sido utilizado en parte o totalmente para la obtención de otro título en esta u otra universidad. Este trabajo es mi propia creación y no contiene nada que sea fruto del trabajo de otros, salvo lo especificado en el texto.

Realizada por: David García Villaescusa
Noviembre 2021

Agradecimientos

Y me gustaría agradecer a todo el mundo que me empujado hasta aquí y que quiero que sientan esta tesis como suya.

Lo primero de todo es agradecer tanto a Mario como a Michael todo el apoyo que me han dado no solo en la realización de la tesis. Sin su guía no me hubiera convertido en el profesional que soy ahora con lo que esta tesis no sería posible.

También agradecer a Héctor y Pablo por darme la oportunidad de descubrir una vocación docente que nunca pensé que llegaría a tener. Y por ayudarme a mejorar cada semana.

Estoy seguro que echaré en falta el gran ambiente de trabajo del que he podido disfrutar estos años. Los cafés, las cenas de navidad, paellas y barbacoas han creado grandes enlaces a nivel personal con gente que se ha convertido en más que compañeros de trabajo.

Aunque para ambiente de trabajo, el del despacho 3050. Con Miguel, Baldu, Ricardo, Alex y Piña he compartido grandes momentos. Me han visto sufrir, han celebrado hitos conmigo y siempre han conseguido sacarme una sonrisa (o una canción).

Sin duda alguna he de agradecer los momentos de comedor. Volver al trabajo con la tripa llena y la cabeza despejada ha sido una auténtica maravilla. Muchas gracias a los que habéis compartido mesa conmigo: Fonso, Yael, Andrea, Diego, Rafa, Ujué, Noelia y Jesús. Espero poder disfrutar tanto del descanso para comer en mi nuevo destino.

Para mi es mucho más importante la gente que me llevo de esta etapa que la tesis en sí. Habéis sido un pilar muy importante en mi vida.

Quisiera agradecer también a la gente que ha compartido conmigo mi mayor pasatiempo, tan necesario para poder seguir adelante. El baile. Especialmente a Piña (de nuevo), Marcos, Jose y Gina con los que he compartido grandes momentos de liberación y felicidad.

Otra forma en la que he conseguido despejar mi mente ha sido mediante el gimnasio de la universidad, donde he podido abstraerme además de conocer a gente que forma parte de mi vida. El ambiente ahí es claramente mérito de Javi.

También quiero incluir a mis amigos en este agradecimiento. Me siento muy afortunado de teneros siempre cerca, sin importar la distancia física (en parte gracias al Discord). Muchas gracias Josema, Héctor, Dani, Serrano, Jorge (x2), Bruno, Javi, Chuchi, Varito, Hugo, Gerardo, Iovanna, Juanra, Ruth, Cude, Iñaki y Varo.

Finalmente, quiero dejar para el final a la gente más importante, mi familia. Siempre se dice que la familia de sangre es la que te toca, pero yo no podría estar más contento. Mención especial a mis padres, que han tirado de mi durante gran parte de su vida. No creo que nunca sea capaz de pagarles por todo lo que han hecho.

Resumen

En la actualidad, los procesos de fabricación de procesadores está generando sistemas con cada vez más núcleos de ejecución en el mismo chip. Este tipo de estrategias de fabricación ofrece potentes procesadores enfocados, principalmente, a las necesidades de cálculo computacional. Sin embargo, estos procesadores con tantos elementos en el mismo chip complican mucho el análisis temporal de tiempos de respuesta o pueden resultar siendo sistemas con unos tiempos de respuesta de peor caso mucho mayores que los deseables, lo cual hace que no se utilice su pleno potencial en sistemas de tiempo real estricto. Por contra, los procesadores muchos núcleos con topología de tipo malla parecen un prometedor compromiso entre potencia de cómputo y tiempos de respuesta acotados.

En esta tesis se propone un modelo, basado en el modelo MAST, que se puede utilizar para modelar aplicaciones ejecutadas sobre procesadores muchos núcleos con una topología de tipo malla. Además, el modelado realizado se aplica a un procesador real, el procesador muchos núcleos Epiphany que cuenta con una topología 4x4 de tipo malla. Este procesador se utiliza para aplicar y verificar todo desarrollado en el trascurso de esta tesis.

Otro elemento esencial para poder disponer de un sistema completamente funcional es el sistema operativo de tiempo real. Con este propósito se ha desarrollado M2OS-mc, una adaptación del sistema operativo M2OS al procesador Epiphany. M2OS-mc ha sido diseñado de forma que pueda ser fácilmente adaptable a cualquier otro procesador muchos núcleos con una topología de tipo malla. Para poder disponer en M2OS-mc de mensajes sincronizados, que son necesarios en sistemas concurrentes, se han implementado dos primitivas de comunicación: los puertos de muestreo y los puertos con cola, basados en primitivas similares definidas en el estándar ARINC-653.

Para verificar el correcto funcionamiento del sistema operativo de tiempo real M2OS-mc y la validez del modelado, se ha desarrollado un generador que, partiendo de una definición de alto nivel de una aplicación, genera el fichero MAST que la modela y una implementación de la misma basada en tareas sintéticas. Con este generador se han implementado diferentes sistemas tipo para los que ha sido posible comprobar

tanto el funcionamiento como la temporalidad de los mismos. Se ha podido, por lo tanto, confirmar que se dispone de un sistema funcional para sistemas de tiempo real estricto en procesadores muchos núcleos basados en malla.

Abstract

Currently, the processor manufacturing process is generating systems with increasing execution cores in the same chip. This kind of architecture offers more powerful processors focused, mainly, on the needs of computational calculus. However, these processors with so many elements in the same chip make the temporal analysis of real-time systems more complicated and could even end up in systems with a worst-case response time larger than desired. In hard real-time systems the systems are designed so that the processors do not use all their potential. Nevertheless, 2D mesh-based topology many-core processors seem to offer a promising balance between calculation capacity and the ability to have a bounded response time.

This thesis proposes a model, based on the MAST model, that could be used for modeling any system executed on a 2D mesh-based many-core processor. Furthermore, the developed model is applied to a many-core processor with a 4x4 mesh topology called Epiphany. This processor is used to apply and verify all the techniques developed during this thesis.

Another essential element for being able to have a completely functional system is the real-time operating system. With that purpose we have developed M2OS-mc, targeted for the many-core processor Epiphany but designed for an easy adaptation to any other mesh-based many-core processor. To be able to have synchronized messages in M2OS-mc, which are needed in concurrent systems, we have implemented the sampling port and queuing port communication primitives based in the ARINC-653 standard.

To perform the needed tests that will check the correct behavior of the M2OS-mc real-time operating system as well as the model of a particular system, a code and MAST model generator has been developed that, from a system description in a single file, will generate every needed file for both purposes. With this generator, different kinds of systems have been developed to check both the behavior and the temporal response of those systems. With this generator we have been able to check the timing behavior of hard real-time system in mesh-based many-core processors.

Índice general

Índice de figuras

Índice de tablas

Artículos publicados

Gran parte del trabajo realizado en esta tesis ha sido revisado por pares en diferentes congresos. Asimismo, se ha remitido un artículo a una revista internacional. A continuación se muestra una lista de publicaciones de las contribuciones realizadas en orden cronológico inverso:

- Response-Time Analysis of Mesh-Based Many-Core Systems.
  David García Villaescusa, Mario Aldea Rivas y Michael González Harbour.
  Remitido a IEEE Access.

- Generador de código para un procesador muchos núcleos.
  David García Villaescusa, Mario Aldea Rivas y Michael González Harbour.
  VI Simposio de Sistemas de Tiempo Real, STR 2021

- Queuing ports for mesh based many-core processors.
  David García Villaescusa, Mario Aldea Rivas y Michael González Harbour.
  25th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2021-WiP track)

- M2OS-Mc: An RTOS for Many-Core Processors.
  David García Villaescusa, Mario Aldea Rivas y Michael González Harbour.
  Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021).
  Part of ISBN: 978-3-95977-178-8
  Part of ISSN: 2190-6807
  DOI: 10.4230/OASIcs.NG-RES.2021.5

- Modelado de una arquitectura many-core basada en Network-on-Chip.
  David García Villaescusa, Mario Aldea Rivas y Michael González Harbour.
  XXI Jornadas de Tiempo Real, STR 2019.

- Selección de una arquitectura many-core comercial como plataforma de tiempo real.
  David García Villaescusa, Michael González Harbour y Mario Aldea Rivas
  V Simposio de Sistemas de Tiempo Real, CEDI 2016.
  https://repositorio.unican.es/xmlui/handle/10902/17870

Acrónimos

AADL	Architecture Analysis and Design Language / Lenguaje de análisis y diseño de arquitecturas
API	Application Programming Interface / Interfaz de aplicación
BCET	Best-Case Execution Time / Tiempo de ejecución de mejor caso
BCRT	Best-Case Response Time / Tiempo de respuesta de mejor caso
BCTT	Best-Case Traversal Time Tiempo de travesía de mejor caso
CAST	Certification Authorities Software Team / Equipo de software de las autoridades de certificación
CHA	Caching/Home Agent
DMA	Direct Memory Address / Memoria de acceso directa
e2e	end-to-end / Principio a fin
EASA	European Aviation Safety Agency / Agencia europea de seguridad en la aviación
FAA	Federal Aviation Administration / Administración federal de aviación
GNAT	GNU Ada Translator
GPIO	General Porpoise Input/Output / Entrada/salida de propósito general
HAL	Hardware Abstraction Layer / Capa de abstracción de hardware
IMA	Integrated Modular Avionics / Aviónica integrada modular
LSB	Least-Significant Bit / Bit menos significativo
MRTA	Multicore Response Time Analysis / Análisis del tiempo de respuesta en multinúcleos
NoC	Network-on-Chip / Red de interconexión
NUMA	Non-Uniform Memory Access/ Acceso a memoria no uniforme
QP	Queuing Port / Puerto con cola
RTOS	Real-Time Operating System / Sistema operativo de tiempo real
RTS	Run-Time System / Sistema de tiempo de ejecución

SDK	Software Development Kit / Kit de desarrollo de software
SMP	Symmetric multiprocessing / Multiprocesamiento simétrico
SAF	Store-and-forward / Almacenar y despachar
SDRAM	Synchronous Dynamic Random-Access Memory / Memoria de Acceso Aleatorio Síncrona y Dinámica
SMT	Simultaneous Multithreading / Multihilo simultáneo
SoC	System-on-Chip / Sistema en un chip
SP	Sampling Port / Puerto de muestreo
VPU	Vector Processing Unit / Unidad de procesamiento vectorial
WAR	Write-After-Read / Escribir tras leer
WCET	Worst-Case Execution Time / Tiempo de ejecución de peor caso
WCRT	Worst-Case Response Time / Tiempo de respuesta de peor caso
WCTT	Worst-Case Traversal Time Tiempo de travesía de peor caso

Capítulo 1

Introducción

Los procesadores han evolucionado a lo largo del tiempo siguiendo la ley de Moore [2], lo cual ha implicado un crecimiento exponencial en el número de transistores, como se puede observar en la Figura 1.1 con la leyenda de *Transistors*. Esto es debido a que esta ley de Moore se basa en la observación de que el número de transistores en un procesador se dobla cada, aproximadamente, dos años.

El aumento exponencial del número de transistores ha ido típicamente acompañado con un aumento, también exponencial, en el rendimiento proporcionado por los procesadores a un solo hilo. Esta tendencia se veía fuertemente respaldada por el aumento exponencial en la frecuencia a la que eran capaz de ejecutar los procesadores. Sin embargo, ambas tendencias dejan de tener un progreso exponencial, es más, la frecuencia se ha mantenido prácticamente estable desde principios de los años 2000. Por otro lado, el rendimiento en un solo hilo ha seguido aumentando, aunque ya no de manera exponencial, al margen del estancamiento de la frecuencia. Tanto la progresión en el tiempo de la frecuencia (*Frequency* en la leyenda) como del rendimiento en un hilo (*Single-Thread Performance* en la leyenda) pueden observarse en la Figura 1.1.

La bajada tanto de rendimiento como de frecuencia es debida al alto consumo energético que suponía mantener dichas progresiones, puesto que el consumo aumenta de manera cuadrática respecto a la frecuencia. Esto, además de un problema de consumo, implica una mayor temperatura durante la ejecución de aplicaciones, y por ello es necesario conseguir disipar el calor de los dispositivos electrónicos para alcanzar unas temperaturas adecuadas de funcionamiento. El problema es que en ciertos dispositivos embebidos carentes de ventiladores se era incapaz de mantener el procesador en temperaturas correctas de funcionamiento y en otros sistemas, como las estaciones de cálculo o equipos personales, la refrigeración por aire se encontraba al límite de su capacidad y empezaban a ser necesarios otros mecanismos de refrigeración

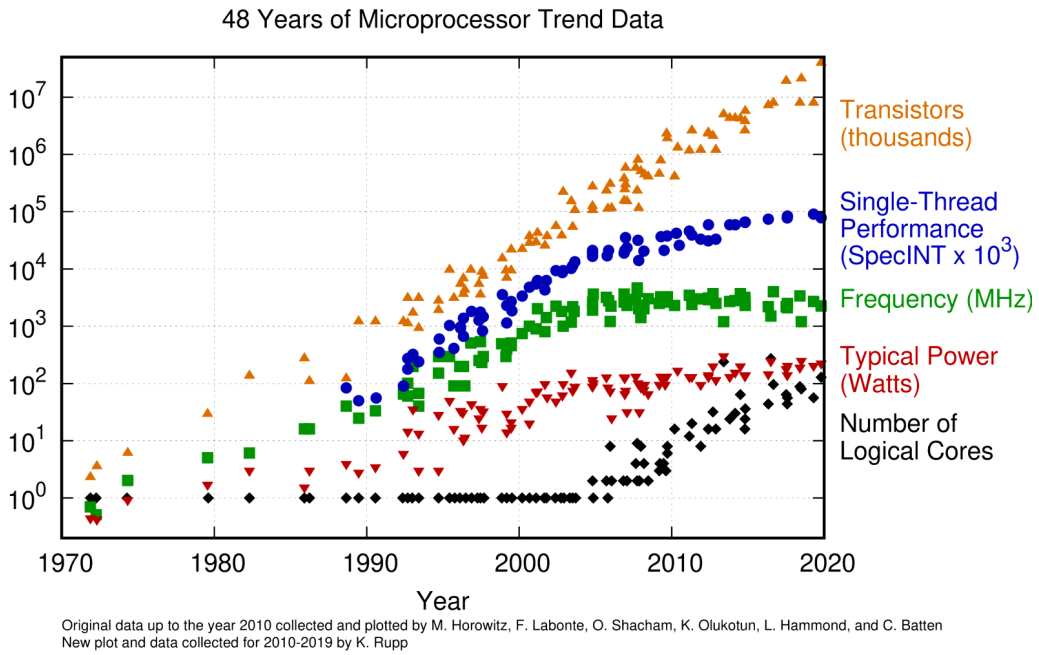

Figura 1.1 Tendencia de los procesadores a lo largo del tiempo (desde 1970 hasta 2020). Los datos hasta el 2010 fueron recogidos por M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond y C. Batten. Los datos a partir de 2010 fueron recogidos por K. Rupp.

que encarecían tanto el producto como el mantenimiento a lo largo del tiempo de los sistemas. Se puede observar como se deja de aumentar el consumo energético en la Figura 1.1 (con la leyenda *Typical Power*).

Sin embargo, el número de transistores sigue cumpliendo la ley de Moore con su incansable crecimiento exponencial. Este factor se aprovecha para seguir aumentando el rendimiento de los procesadores fabricados, solo que en la actualidad se busca un método alternativo para conseguirlo. Se pasa a aumentar el número de núcleos lógicos dentro de un mismo sistema en un chip (*System-on-Chip*, SoC) tal y como se puede observar en la Figura 1.1 (con la leyenda *Number of Logical Cores*). Esto es debido a que el aumento de consumo energético que se produce ante el aumento de núcleos lógicos es lineal en lugar de cuadrático. Es entonces cuando se pasa de tener un procesador con un solo núcleo muy potente a tener varios núcleos en el mismo procesador sin la necesidad de incrementar su capacidad de cálculo en un hilo. Lo que sí se consigue de manera notable es un aumento en el rendimiento general del sistema y, especialmente, cuando el procesador ejecuta aplicaciones paralelizables o múltiples aplicaciones que se pueden ejecutar de forma concurrente. De esta manera se consigue mantener el consumo energético al mismo nivel, pero se sigue aumentando el rendimiento de los

procesadores. Esta tendencia conduce al nacimiento del procesador multinúcleo, el tipo de procesador tan extendido en la actualidad en sistemas de alto rendimiento, computadores personales y electrónica de consumo.

1.1. Procesadores multinúcleo

La arquitectura de los procesadores multinúcleo tiene grandes ventajas con respecto a la de los procesadores mononúcleo:

- Mejora en el rendimiento, especialmente notable en aplicaciones paralelizables o en sistemas en el que las aplicaciones puedan ejecutarse concurrentemente.
- Mayor eficiencia en términos energéticos.
- Mayor rendimiento por coste.
- Permiten la ejecución simultánea de múltiples aplicaciones en un solo procesador, reduciendo el consumo y las altas exigencias de refrigeración con respecto a los procesadores con muy altas frecuencia de reloj. En sistemas embebidos de altas prestaciones esto permite una reducción del número de sistemas de computación, que equivale a una reducción de peso, cableado y espacio.
- Permiten que se efectúen diferentes tareas de una misma aplicación de manera simultánea utilizando diversos hilos de ejecución con concurrencia física. Esto supone un aumento en la capacidad de cómputo.

Como se ha comentado, los procesadores multinúcleo consisten en procesadores que tienen varios núcleos lógicos en el mismo procesador. El tener varios núcleos en un mismo SoC implica, a nivel de memoria, que hay que tener un mecanismo de comunicación entre la memoria caché de los propios núcleos para mantener su coherencia y también de los núcleos hacia la memoria compartida. La típica decisión arquitectónica para realizar dicha comunicación es la de tener un bus de memoria compartido por todos los núcleos, como se muestra en la Figura 1.2, aunque también puede tenerse memoria local exclusiva en cada núcleo. Este bus de memoria también acaba siendo el único método de comunicación de los núcleos con el resto de elementos del mismo procesador o del resto del sistema en el que se encuentra el núcleo. Esta decisión de diseño supone una de las mayores dificultades que se encuentran en el uso de procesadores multinúcleo en sistemas de tiempo real, que radica en que las interacciones temporales introducidas por la gestión del bus de memoria compartido y

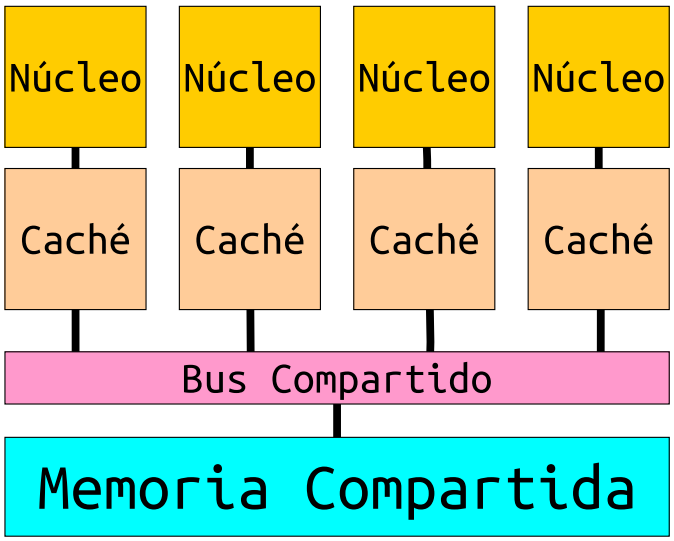

Figura 1.2 Esquema de muestra la arquitectura de memoria de un procesador multinúcleo.

los mecanismos de coherencia de cachés son muy difíciles de analizar temporalmente. Se introduce impredecibilidad en los tiempos de acceso a memoria, que debe mitigarse o evitarse en sistemas de tiempo real estrictos.

Por tanto, aunque la llegada de los procesadores multinúcleo supone continuar con el avance en términos de rendimiento, este no es el principal objetivo a alcanzar en todos los ámbitos en los que se utilizan procesadores. Este es el caso de los sistemas de tiempo real estricto, como por ejemplo el caso de estudio ROSACE [3], en los que lo más importante es asegurar un tiempo de respuesta máximo para las diferentes actividades que se ejecutan. Estos sistemas, presentes en aviones, automóviles, ascensores, etc. podrían ver comprometida la integridad del sistema ante una sola ejecución que excediese el plazo requerido por mucho que el tiempo de ejecución promedio del sistema se vea reducido drásticamente. Es por ello que en estos sistemas supone de mayor interés tener tiempos de respuesta acotados que tener un rendimiento generalmente superior pero que en casos particulares podamos tener una respuesta en un tiempo no acotado. En el caso de los procesadores multinúcleo es el bus compartido la principal causa del aumento del tiempo de ejecución de peor caso, como se puede ver en la evaluación realizada por Kelter [4]. Por otro lado, la arquitectura de los procesadores multinúcleo presenta una baja escalabilidad frente al aumento del número de núcleos, lo que ha conducido a la búsqueda de nuevas estrategias de acceso a memoria.

Otra situación que puede comprometer un sistema de tiempo real es el hecho de compartir la memoria de acceso aleatorio síncrona y dinámica (SDRAM) entre diversos

elementos de un mismo sistema, ya que se requiere entonces la implementación de políticas de arbitrio predecibles. Las implementaciones tradicionales de políticas de arbitrio predecibles no son escalables en términos de frecuencia de reloj (algo que también se puede ver en la evaluación de Kelter [4]), con un número de elementos intentando acceder a la memoria compartida que, como hemos visto en la Figura 1.1, está en continuo incremento. Las implementaciones distribuidas existentes no proporcionan un trato diferente según la prioridad del cliente o lo hacen con bajo rendimiento en términos de área, consumo de potencia y latencia.

Por otro lado, hay ciertas limitaciones hardware que también deben ser tomadas en consideración, aunque la solución se deba dar a nivel de hardware y no será abarcada en esta tesis. Uno de los problemas hardware más inminentes es que si en el futuro se sigue disminuyendo el tamaño del transistor, es posible que el hardware se vuelva inestable, como ha estudiadio Stockman [5], y se tenga que ejecutar en núcleos diferentes la misma operación.

En la actualidad se está trabajando con diferentes decisiones de diseño que afectan a la arquitectura de estos sistemas multinúcleo.

1.1.1. Sistemas de tiempo real

Los sistemas de tiempo real estricto se encuentran ahora ante una situación complicada, ya que la industria del semiconductor sigue evolucionando hacia procesadores multinúcleo con cada vez más núcleos compartiendo el mismo bus de datos, haciendo cada vez más complicado el análisis temporal de un sistema determinado.

El problema radica en que conforme aumenta el número de núcleos que hay en un mismo procesador multinúcleo, aumenta el número de cachés que hay que mantener coherentes. Es por ello que los mecanismos de coherencia de cachés pueden ver su rendimiento impactado negativamente al tener mucho más trabajo que realizar como ya apuntó Gracioli [6]. Estos problemas de rendimiento de los mecanismos de coherencia de cachés pueden suponer también problemas de rendimiento de ejecución de aplicaciones en sus accesos a memoria y, sobre todo, un gran aumento del tiempo de ejecución de peor caso.

Otro de los mecanismos que introduce impredecibilidad en sistemas de varios núcleos en un mismo procesador multinúcleo, es que pueden existir varias aplicaciones ejecutando al mismo tiempo en diferentes núcleos, o incluso en el mismo núcleo si es un procesador con tecnología multihilo simultáneo (SMT). Por lo tanto, varias de esas aplicaciones pueden intentar acceder al mismo recurso compartido del multinúcleo (ya sea memoria compartida, caché de cada núcleo, sistemas de interconexión coherentes

o interfaces externas al procesador) al mismo tiempo, lo cual causaría contención empeorando el tiempo de ejecución de las aplicaciones que pierdan en el arbitrio de los recursos compartidos.

Tampoco se puede extrapolar el análisis temporal de una aplicación en un procesador mononúcleo a cómo sería su comportamiento temporal en un núcleo de un procesador multinúcleo. Esto es debido a que si se require paralelizar la aplicación para aprovechar al máximo todos los núcleos disponibles, puede ser necesario utilizar mecanismos de control o gestión de datos y estudiar las interferencias que puedan producirse entre las diferentes partes concurrentes de la aplicación.

Las interferencias entre las aplicaciones que se ejecutan en un multinúcleo podrían causar un comportamiento no determinista en las aplicaciones de tiempo real estricto en caso de no usar mecanismos de planificación de tiempo real. También podría suceder que no se tenga suficiente tiempo para completar la ejecución cumpliendo los requisitos de la aplicación de tiempo real estricto. Al poder tener múltiples aplicaciones ejecutando a la vez en un mismo procesador multinúcleo, incluso si se trata de un flujo de datos o control explícito entre las aplicaciones, solo por compartir plataforma se pueden crear interferencias que se deberán tener en cuenta al analizar la respuesta temporal del sistema. Para analizar los efectos de estos problemas se han desarrollado técnicas de verificación temporal [7].

El aumento en el número de núcleos también supone un problema de cara a los paradigmas típicos de programación, pensados para procesadores mononúcleo que no aprovechan el potencial ofrecido por los procesadores multinúcleo.

Aparte de toda la complejidad generada por el cambio de la tecnología, siguen existiendo los problemas típicos de un sistema de tiempo real, como la integración de componentes con diferentes niveles de criticidad y el cumplimiento de las diferentes certificaciones requeridas según el tipo de sistema en el que vaya a ser ejecutado.

Por todo lo expuesto anteriormente se genera mayor complejidad a la hora de analizar temporalmente un sistema de tiempo real estricto conforme se aumenta el número de núcleos. Típicamente se ha desaconsejado la implementación de sistemas de seguridad crítica en procesadores multinúcleo [8, 9], o se permitía su implementación restringiendo la ejecución a un solo núcleo a la vez. Actualmente ya se está abriendo la posibilidad de utilizar procesadores multinúcleo incluso en sistemas de aviación. Estos sistemas requieren un proceso de certificación para asegurar que son capaces de cumplir con ciertos criterios de seguridad y fiabilidad.

Tanto la *European Union Aviation Safety Agency* (EASA) [1] como la Federal Aviation Administration (FAA) [2] americana están buscando certificar la utilización de los procesadores multinúcleo, considerando multinúcleo a cualquier procesador con más de un núcleo, independientemente de su arquitectura.

La EASA y la FAA han creado un documento cojunto AMC-20-193 [3] sobre el uso de los procesadores multinúcleo, que pretende asistir el proceso de certificación para los multinúcleo además de recomendar la mejores prácticas para los mismos.

La implantación de la arquitectura multinúcleo en sistemas de tiempo real es compleja debido a lo poco predecibles que resultan estos sistemas con procesadores multinúcleo. En este tipo de certificaciones siempre se es muy cauteloso pues se tratan temas de seguridad con nuevas tecnologías, por lo tanto su penetración en la aviación es notablemente más lenta que en los sistemas dirigidos a propósitos generales. En la actualidad se está trabajando en conseguir que los SoC con más de un núcleo funcional puedan formar parte de un sistema de tiempo real crítico, aprovechándose así todas sus ventajas.

Se ha trabajado muy duro en la implantación de los multinúcleo en los sistemas de tiempo real de seguridad crítica, como se ha podido ver en los análisis de las agencias anteriormente citadas y se puede ver en el trabajo de Nowotsch y Paulitsch sobre los procesadores multinúcleo en aviación [10]. En dicho artículo se indica que para poder utilizar procesadores muchos núcleos en sistemas de tiempo real hay que cumplir ciertos requisitos en sus especificaciones:

- **Núcleos predecibles.** Los núcleos deben proporcionar de manera individual un comportamiento acotado temporalmente. Un aspecto que puede incidir negativamente en la predictibilidad es la ejecución fuera de orden. Esto afecta a los tiempos de ejecución e incide negativamente en los cambios de contexto en los que de manera asíncrona el procesador abandona un punto de la ejecución de un hilo para pasar a ejecutar otro hilo, debiendo por tanto descartar cálculos hechos fuera de orden. Este retardo se puede añadir con facilidad al tiempo de cambio de contexto. Pero por otro lado la ejecución fuera de orden presenta el inconveniente de dificultar el análisis estático de tiempos de ejecución.

- **Memoria local.** La existencia de memoria compartida en un procesador genera una gran incertidumbre en el cálculo de los tiempos de respuesta. En muchos sistemas multinúcleo tanto el último nivel de memoria caché (L2 o L3) como

[1] https://www.easa.europa.eu/
[2] https://www.faa.gov/
[3] https://www.easa.europa.eu/sites/default/files/dfu/npa_2020-09_0.pdf

el bus de memoria son compartidos. Además, la memoria caché privada (L1) necesita mantener la coherencia de sus datos, lo cual provoca que determinados accesos a datos contenidos en esta caché no sean independientes si se hacen accesos a ellos desde diferentes núcleos. La coherencia de las memorias caché genera una gran incertidumbre en el tiempo de respuesta máximo salvo que se tengan mecanismos de coherencia de caché con tiempo de ejecución acotada o se pueda deshabilitar dicha coherencia de caché. Para sistemas de tiempo real, se necesitan cachés privadas o memorias locales a los núcleos de tamaño suficiente para albergar la mayor parte de los accesos de un hilo de ejecución alojado en cada núcleo.

- **Mapeado de memoria.** La existencia de memorias locales o cachés privadas hace necesario disponer de mecanismos para mapear la memoria usada por un hilo de ejecución y bloquearla en la caché privada a voluntad.
- **Red de interconexión.** La red de interconexión entre los núcleos y las memorias también tiene que proporcionar un comportamiento temporal predecible. No basta con que tenga un gran ancho de banda y baja latencia (qué también es algo deseable).
- **Periféricos.** Diferentes aplicaciones pueden intentar acceder a un mismo periférico. El acceso debe ser controlado y sincronizado.
- **Acceso a memoria.** Se requieren tiempos de acceso a memoria acotados.
- **Envío de mensajes.** Es preciso poder acotar el tiempo de envío de mensajes entre los diferentes núcleos del procesador, teniendo en cuenta que en muchas redes el envío de mensajes a procesadores vecinos es más rápido que a otros más lejanos. Por otro lado, la posibilidad de controlar el mapeado de hilos de ejecución a los diferentes núcleos de los procesadores puede facilitar el agrupamiento de aquellos hilos de ejecución que tienen un mayor número de interacciones entre sí.

1.2. Sistemas Operativos

Los sistemas operativos típicos utilizados para procesadores multinúcleo procuran que todos los núcleos del sistema estén lo más ocupados posible para mejorar así su rendimiento. Sin embargo, esto no resulta ser el criterio más importante en un sistema de tiempo real, en el que lo importante es cumplir siempre todos los requisitos temporales impuestos.

Los sistemas operativos presentan requisitos en relación al soporte de la concurrencia tales como: concurrencia a nivel de múltiples hilos o threads y la posibilidad de que estos hilos se sincronicen, tanto por medio de mecanismos de señalización y espera como de exclusión mutua para compartir recursos o datos en memoria. En ocasiones se establecen mecanismos de sincronización por paso de mensajes. También los mecanismos de control de hilos como la suspensión, activación o cancelación son importantes.

En relación a los sistemas de tiempo real, es preciso que el sistema operativo proporcione servicios de temporización y medida del tiempo y que dé soporte a algún tipo de planificación por prioridades, ya sean fijas o dinámicas como las estudiadas por Baruah [11], junto a un protocolo de sincronización entre threads que esté libre de inversión de prioridad no acotada cuyas fuentes expuso Davari [12]. En sistemas multinúcleo que no comparten el reloj se debe establecer algún mecanismo de sincronización de relojes.

Para gestionar los recursos de un procesador multinúcleo existen diferentes tipos de sistemas operativos:

- Una sola instancia del sistema operativo controlando todos los núcleos con memoria compartida como el expuesto por [13]. Existen distintas variantes:
  - **Planificación global**: donde una aplicación puede ser ejecutada en cualquier núcleo. Esto es de aplicación en arquitecturas de procesadores multinúcleo simétricos (SMP).
  - **Planificación particionada**: cada aplicación tiene asignado el núcleo en el que se ejecutará. Esto se puede utilizar tanto en arquitecturas con procesadores simétricos como asimétricos.
  - **Planificación mixta**: cada aplicación puede ser ejecutada solo en un subconjunto de los procesadores. Los subconjuntos se eligen de modo que el acceso a memoria compartida entre sus procesadores sea el más eficiente. Esto es de aplicación en arquitecturas de acceso a memoria no uniforme (NUMA).
- Varias instancias del sistema operativo, cada una con su memoria y que se comunican mediante mensajes. Una aplicación se ejecuta siempre en la misma instancia del sistema operativo. Esto es de aplicación en arquitecturas NUMA y en sistemas multinúcleos donde no siempre hay memoria compartida.

1.3. Arquitecturas muchos núcleos con topología de tipo malla

Para conseguir integrar más de un núcleo en un procesador se creó la arquitectura multinúcleo en la que todos los núcleos comparten un bus de acceso a la memoria y mantienen la coherencia de sus cachés o memorias próximas. Sin embargo, este diseño no resulta escalable ya que cuando el número de núcleos supera la decena los mecanismos de coherencia de cachés son cada vez más complejos y en la práctica el rendimiento del procesador no aumenta tanto como sería esperable debido a los cuellos de botella que expuso Eyerman [14]. Por este motivo nace la arquitectura muchos núcleos [15] en la que se usan redes de interconexión en sustitución del bus compartido de acceso a memoria y en las que puede aparecer memoria local en la que no se mantiene el requisito de mantener la coherencia de cachés.

Las arquitecturas muchos núcleos son incipientes a nivel comercial y su uso en sistemas de tiempo real apenas está extendido. Sin embargo, pueden resultar una buena alternativa de futuro a medio plazo, por lo que investigar el modo en el que un sistema de tiempo real y de seguridad crítica pueda funcionar sobre procesadores muchos núcleos aprovechando todo su potencial es una labor de gran interés. Analizamos este tipo de procesadores a continuación.

La red de interconexión que conecta los diferentes núcleos del procesador muchos núcleos es habitualmente conocida por las siglas de su nombre en inglés *Network on a Chip* (NoC). Puede haber muchas NoCs diferentes dentro de un mismo procesador muchos núcleos y entre las diferentes NoCs puede llegar a haber interferencias o ser ortogonales.

Cada una de las celdas conectadas por la NoC contiene un núcleo de procesamiento, un enrutador y la memoria local, como se muestra en la Figura 1.3. El enrutador tiene la misión de redirigir los paquetes por la red correspondiente o para escribir o leer los datos que circulan por la NoC en o desde la memoria local del núcleo.

Con esta arquitectura dentro de un procesador muchos núcleos, se consigue tener un consumo de energía lineal con el número de núcleos y una latencia predecible en la comunicación entre núcleos. Además, se produce un ahorro de cables con respecto a un sistema mononúcleo distribuido. Por otro lado, uno de los problemas que puede afectar al sistema es que cargar el código en un número elevado de núcleos puede retardar el arranque del sistema de manera significativa.

Un procesador multinúcleo con una arquitectura que utilice una malla 2D como topología para conectar los diferentes núcleos del procesador se llamará en esta tesis a

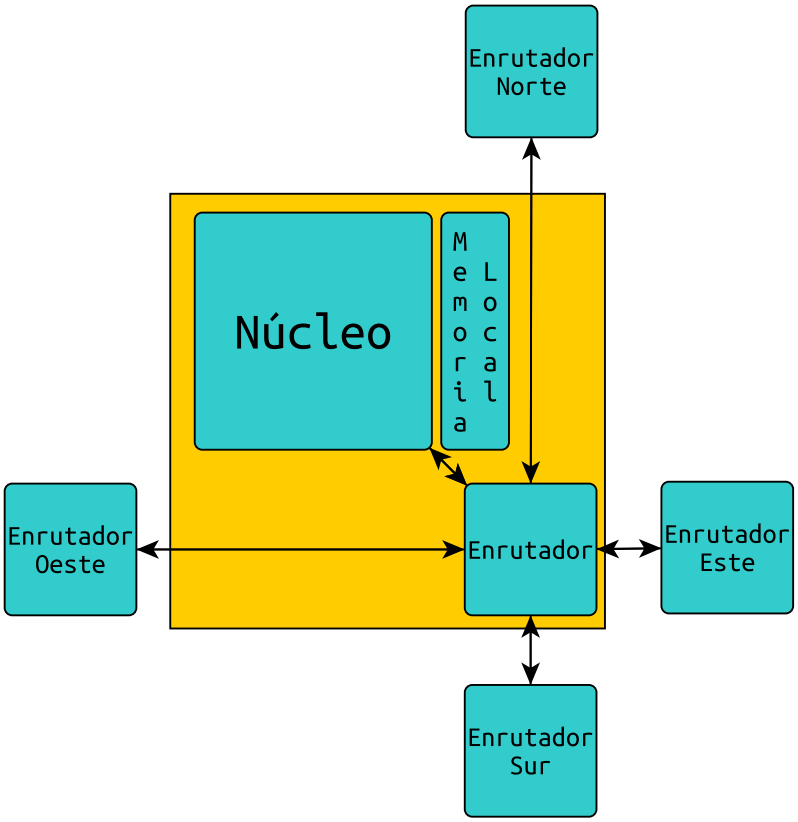

Figura 1.3 Celda en una arquitectura muchos núcleos

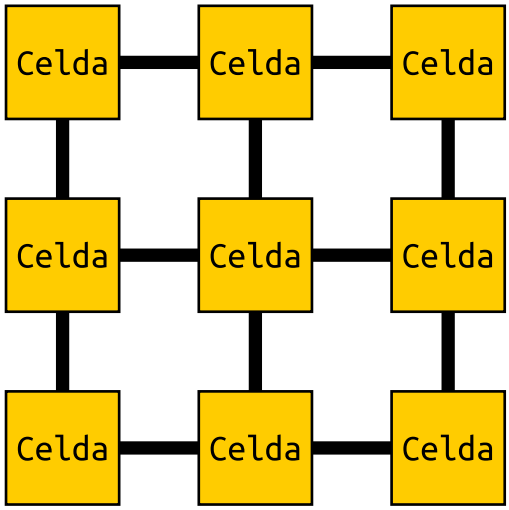

Figura 1.4 Conexión de celdas en un procesador muchos núcleos.

partir de ahora procesador muchos núcleos. Se puede ver una representación gráfica de esta arquitectura en la Figura 1.4, donde se ve cómo cada celda está conectada mediante una red de interconexión a sus celdas vecinas formando un topología malla 2D, en este caso de 3x3. Dentro de cada celda se encontrará el hardware necesario para realizar operaciones de cálculo, memoria e interactuar con la red de interconexión además de dirigir el tráfico que quiera atravesar la celda. El aumento de prestaciones y la contribución a la disminución de tamaño, peso y consumo del sistema completo de un procesador muchos núcleos con respecto a los procesadores multinúcleo muestran un gran potencial para ser usados como plataformas para sistemas de tiempo real.

Para poder enviar mensajes entre los diferentes núcleos del procesador, la red de interconexión utiliza el paquete como unidad de información. Un paquete puede ser una fracción de un mensaje, un mensaje completo o incluso contener varios mensajes. Existen dos políticas de enrutado principales para enviar paquetes a través de la NoC: *wormhole* y *store-and-forward*.

- Utilizando el esquema *wormhole* se divide cada paquete en *flits* de tamaño fijo, siendo el *flit* la unidad indivisible más pequeña de la red. El primer *flit* de un paquete, llamado *flit* de cabecera, posee la información de enrutado y es el único *flit* utilizado para determinar la ruta. El resto de *flits* de un paquete siguen al *flit* de cabecera a través de la NoC enfilados (estilo tubería o *pipeline*) hasta que el último *flit*, llamado *flit* de cola, llega al destino. En caso de que el *flit* de cabecera haya sido bloqueado, el resto de los *flits* del paquete también sufrirán el mismo bloqueo en la NoC.

- El esquema *Store-and-forward* (SAF) opera con el paquete al completo. Los enrutadores esperan la llegada del paquete al completo antes de operar sobre él.

1.4. Análisis temporal

Como hemos podido vislumbrar en la sección anterior, los sistemas de tiempo real se han enfrentado a nuevos retos con la llegada de los procesadores multinúcleo.

Existen ciertos elementos que es interesante modelar en términos temporales, muy dependientes de la arquitectura y de la distancia entre los diferentes elementos implicados:

- **Accesos a memoria.** Cuando se accede a memoria local no se compite con otros núcleos. Al acceder a memoria compartida no solo se compite por el propio acceso

a la memoria si no por el uso de los enlaces de la NoC. Según la arquitectura, la zona de memoria compartida puede tener bloques reservados a ciertos núcleos, pero sigue siendo necesario analizar la contención en la red de interconexión.

- **Paso de mensajes.** Se compite con el tráfico que utilice los mismos enlaces que el mensaje. Hay arquitecturas que dividen el tráfico en diferentes canales para segregar lo máximo posible el tipo de tráfico.

- **Accesos a dispositivos.** En la mayoría de las arquitecturas los dispositivos de entrada/salida están distribuidos a lo largo de la red de interconexión, y por tanto se compite por los enlaces para acceder a ellos.

Para afrontar el reto de poder utilizar las arquitecturas con múltiples núcleos como plataforma para el desarrollo de sistemas de tiempo real, se pueden buscar configuraciones que no generen interferencias a tareas de tiempo real. Esto permitiría tener un análisis temporal muy parecido a como si se estuvieran estudiando un procesador mononúcleo reduciendo significativamente la complejidad del análisis temporal. Por otro lado, esta simplificación puede suponer un gran impacto negativo en cuanto al rendimiento.

Estas configuraciones también reducirían enormemente las capacidades del sistema al no poderse compartir ningún recurso, aislando los núcleos entre sí de manera que se tenga la sensación de tener varios mononúcleos independientes haciendo operaciones de manera aislada. En esta tesis se propone una solución más comprometida con las ventajas que proporciona un procesador con varios núcleos de ejecución y que permite tener una interacción limitada entre las tareas o aplicaciones ejecutadas en diferentes núcleos sin permitir que estas acciones repercutan, de manera excesiva, en el rendimiento del sistema. La propuesta es aplicable a un tipo concreto de multinúcleo, uno que conecta los diferentes núcleos que alberga el procesador en una topología de tipo malla 2D o toro 2D.

1.5. Objetivos

Con esta tesis pretendemos demostrar que es posible implementar aplicaciones con requisitos de tiempo real en procesadores muchos núcleos basados en malla y obtener modelos fiables de su comportamiento temporal encaminados al análisis de planificabilidad que garantice el cumplimiento de todos los requisitos temporales.

Con dicho objetivo final en mente, podemos considerar diferentes hitos para conseguirlo:

- Estudiar las arquitecturas de procesadores muchos núcleos y seleccionar una sobre la que se estime que será posible obtener un comportamiento temporal predecible tanto en la ejecución de tareas en los núcleos como en la comunicación entre ellas a través de la red de interconexión.
- Desarrollar un sistema operativo de tiempo real para procesadores muchos núcleos. Este sistema operativo tendrá que disponer de primitivas de comunicación sincronizadas.
- Realizar un estudio exhaustivo del comportamiento temporal del procesador muchos núcleos seleccionado, con objeto de dar soporte a la ejecución de aplicaciones de tiempo real.
- Modelar de la manera más genérica posible el comportamiento de los procesadores muchos núcleos basados en malla.
- Finalmente, se pretende comprobar que el modelo genérico se puede aplicar al procesador muchos núcleos seleccionado, usado como plataforma para aplicaciones de tiempo real que se ejecutan sobre el sistema operativo y primitivas de sincronización desarrollados.

1.6. Organización de la tesis

Antes de plantear el trabajo realizado se describe el trabajo relacionado en el Capítulo 2. En el Capítulo 3 hablaremos del modelo propuesto para el comportamiento temporal de aplicaciones de tiempo real ejecutando sobre un procesador muchos núcleos, basado en el modelo ya existente utilizado para la herramienta de análisis de planificabilidad MAST [16]. En el Capítulo 4 se introduce el procesador Epiphany, elegido para el desarrollo del sistema operativo de tiempo real y para realizar las pruebas correspondientes en relación a la temporalidad de los sistemas ejecutados en un procesador muchos núcleos. En el Capítulo 5 se describe sistema operativo de tiempo real M2OS-mc [17], desarrollado para la plataforma elegida. En el Capítulo 6 se describirán los mecanismos de sincronización desarrollados en M2OS-mc. En el Capítulo 7 se presenta un generador automático que, a partir de un único fichero de descripción de las características de una aplicación, genera tanto el fichero con el modelo MAST como el código con cargas de ejecución sintéticas. Finalmente, acabaremos en el Capítulo 8 con las conclusiones y el trabajo futuro.

Capítulo 2

Trabajo relacionado

Procederemos a presentar el trabajo relacionado con esta tesis dividiéndolo en diferentes estudios hechos sobre redes de interconexión, procesadores muchos núcleos y sistemas operativos.

Una de las principales diferencias con el trabajo presentado en esta memoria es que en ninguno de los trabajos referenciados se ha realizado una implementación sobre un procesador muchos núcleos existente. Algunos artículos simulan comportamientos de sistemas [18–25] mientras que otros utilizan Ethernet enrutado [26–28]. En esta tesis se implementa el sistema modelado en un procesador muchos núcleos real para poder comparar el modelado propuesto con ejecuciones reales.

2.1. Análisis temporal en sistemas distribuidos

El análisis de sistemas distribuidos de tiempo real tiene gran relación con esta tesis aunque no esté directamente realizado con redes de interconexión o procesadores muchos núcleos.

Podemos establecer los orígenes del análisis de tiempos de respuesta en el trabajo de Tindell [29] con su análisis de planificabilidad holístico aproximado para sistemas de tiempo real de prioridades fijas en un sistema distribuido, sin ser excesivamente pesimista.

Más adelante Palencia [30] mejoró ese análisis holístico y desarrolló el análisis de planificabilidad basado en *offsets* que reduce el pesimismo del análisis holístico además de permitir el análisis de los efectos de la suspensión de tareas.

El conjunto de herramientas MAST [16] puede se utilizado para aplicar y comparar diferentes técnicas de análisis de planificabilidad. MAST define un modelo abierto para describir sistemas de tiempo real dirigidos por eventos. La herramienta no está

adaptada directamente al funcionamiento de un procesador muchos núcleos por lo que se requiere adaptar la herramienta para permitir el análisis temporal en este tipo de plataformas. Esto supone una de las aportaciones de esta tesis.

Los primeros estudios sobre comunicación de tiempo real estricto para sistemas con enrutado multi salto se hicieron sobre hardware Ethernet con latencias de principio a fin acotadas como mostraron Zhang [26] y Yiming [27]. El impacto positivo que se produce al aumentar el número de enrutadores y la influencia de la topología fueron previamente estudiados por Lee [28]. Estos tres estudios han pavimentado un escenario de consideraciones iniciales que han ido evolucionando con el tiempo hasta llegar a los análisis actuales de sistemas con redes de interconexión.

Davis [31] utiliza Multicore Response Time Analysis (MRTA) buscando proporcionar tiempos de respuesta para la utilización de diferentes recursos de un hardware que se define de manera teórica. En ese trabajo también se estudia la política de arbitrio cíclica (*round-robin*), pero sobre el bus de memoria. Con este trabajo se busca una aproximación general a la verificación temporal de sistemas multinúcleo. Se definen dos conjuntos disjuntos de bloqueos de memoria, uno para instrucciones y otro para datos. El modelo de tareas propuesto guarda ciertas similitudes con el modelo MAST presentado en el Capítulo 3, del que parte el modelo propuesto en esta tesis para los sistemas muchos núcleos. Sin embargo, no contempla sistemas con NoCs, solo aplica el *round-robin* al bus.

2.2. Redes de interconexión

Aunque la existencia de las NoC en procesadores de propósito general es de reciente aplicación, el concepto de NoC no es algo novedoso, ya fue presentado por Benini [32] en 2002 y enseguida acaparó la atención de la comunidad de sistemas de tiempo-real estricto [33] ya que las arquitecturas con NoCs en comparación con las arquitecturas basadas en bus tienen una eficiencia energética mucho mejor, mayor escalabilidad, reusabilidad, fiabilidad, proporcionan una distribución de los núcleos homogénea y todo esto con un menor requerimiento de cableado.

Gopalakrishnan [23] estudió cómo definir las cotas temporales en un sistema de tiempo real implementado sobre plataformas que utilizan múltiples buses enlazados. Al igual que en esta tesis, se impone una restricción en el diseño del sistema, en el caso de [23] se trata de un límite del número de saltos que puede dar un mensaje de tiempo-real en lugar de la restricción en la ratio de utilización de los enlaces propuesta en esta tesis.

Cuando existen flujos de tráfico en toda la red de interconexión, determinar exactamente la planificabilidad de todos los conjuntos de flujos es un problema NP-completo, como dijo Shi [18], que además vio que las aplicaciones de tiempo real necesitan una estrategia de planificabilidad y un análisis para predecir si todos los paquetes de tiempo real pueden satisfacer sus límites temporales. En esta tesis se hace un análisis completo que gracias a las limitaciones que se proponen para el sistema evita la necesidad de utilizar grandes recursos computacionales para su cálculo.

Se han realizado varios estudios teóricos sobre las NoCs con malla 2D [19, 24]. En estos estudios se ha analizado la planificabilidad utilizando unos procesadores mucho núcleos teóricos. En caso del artículo de Idrusiak [19] se supone, igual que se hace en los flujos analizados en esta tesis, que las tareas mandan un mensaje al finalizar su tiempo de computación, aunque en esta tesis permitimos el envío de más de un mensaje.

Por otro lado, Nikolić [24] considera para sus análisis el número de enlaces atravesados y la latencia de atravesar un enlace. La latencia de enrutado puede generar un *buffering* o almacenamiento intermedio auto impuesto en cada enrutador atravesado. Esto último no sucede en el planteamiento de esta tesis, donde solo hay un *buffer* en cada enlace del enrutador (lo cual permite tener el mismo análisis para diferentes tamaños de *buffer* en el hardware). En este artículo llegan a la conclusión de que los *buffers* más grandes no proporcionan necesariamente mejores resultados.

Como se propone en el artículo [34], en la NoC de los procesadores muchos núcleos pensados para sistemas de tiempo real los mensajes deben diferenciarse según su prioridad, de manera que se garantice una calidad de servicio en aquellos de alta prioridad, aunque sea a costa de que los de baja prioridad no lleguen nunca, siempre considerando que cualquier mensaje cuyo retardo pueda aumentar la latencia de una tarea de alta prioridad también tendrá la prioridad de la tarea que puede retardar. Con una mayor independencia entre las tareas se evitaría tener que aumentar la prioridad de muchos mensajes, descongestionando la NoC para los mensajes requeridos por tareas de mayor prioridad. El aislamiento de diferentes recursos compartidos puede hacerse de manera temporal a través del planificador y de manera espacial mediante regiones de memoria separadas.

En otras discusiones de modelos sobre el tiempo de respuesta de peor caso (Worst-Case Response Time, o WCRT) hechas sobre NoCs *wormhole* con topología malla de 2D [20] [21] se ha introducido el fenómeno de contrapresión. Este fenómeno puede ser un gran problema a la hora de calcular el impacto de la NoC sobre un determinado paquete que esté viajando por la NoC ya que podría causar un impacto mayor del esperado en la temporalidad del paquete. La contrapresión se define como la situación

que ocurre cuando, debido al tráfico que interfiere entre sí en la red, un paquete de un mensaje se ve bloqueado por el paquete anterior ya que en enlace por el que debe transmitirse está ocupado. Esta contrapresión puede propagarse hacia los enlaces anteriores en la ruta del mensaje, hasta llegar a detener la ejecución del núcleo durante la inyección de tráfico a la red (fenómeno llamado *stall*). Este fenómeno se produce cuando se encuentra ocupado el enlace de salida hacia el primer enrutador en la ruta del paquete generado.

Tanto el esquema *wormhole* como el SAF han sido estudiados por Dridi [22] evitando el análisis cuando el canal de comunicación es compartido por más de un flujo de tráfico, esto resulta más restrictivo que nuestra aproximación en la que se pueden compartir los enlaces siempre que no se supere una ratio determinada. En ese artículo se incluye un simulador para comprobar la exactitud del comportamiento temporal, nosotros hemos preferido probar el comportamiento temporal en una plataforma real utilizando tareas con tiempo de ejecución sintéticos.

Garrido[35] presentó una implementación para el protocolo de comunicación síncrona ARINC-653 en la que se basará el mecanismo de comunicación síncrona utilizado en esta tesis. Dicho protocolo ofrece comunicación entre particiones aisladas con requisitos de tiempo real estricto.

La mayoría de los trabajos realizados sobre el análisis temporal en sistemas que utilizan NoCs se centran en el tiempo de travesía de peor caso (Worst-Case Traversal Time o WCTT), considerando así solo el tiempo necesario para el viaje por la NoC en vez del tiempo de respuesta de peor caso (WCRT) de un conjunto de tareas que están localizadas en diferentes núcleos del procesador necesitando comunicarse entre sí. El análisis del WCRT de flujos de tráfico y las tareas del mismo es una de las aportaciones de esta tesis.

2.3. Procesadores muchos núcleos

Un procesador de muchos núcleos tiene que cumplir cuatro requisitos para servir como plataforma para sistemas de tiempo real, según Metzlaff [25]:

1. Los núcleos tienen que ser pequeños y simples.

2. La jerarquía de memoria tiene que evitar el almacenamiento automático en caché. El acceso a memoria tiene que ser predecible y proporcionar suficiente ancho de banda. En el artículo se propone una memoria de nivel L1 privada y un nivel L2 compartido sin almacenamiento en caché, lo que facilita el análisis de la memoria

y evita una fuente de impredecibilidad ya que se evita la necesidad de mantener la coherencia de la caché.

3. El enrutado de la NoC tiene que ser estático, en busca de predictibilidad temporal y alta utilización garantizando un mínimo de ancho de banda y un máximo de latencia. Se pueden reservar particiones temporales para conexiones individuales punto a punto.

4. El análisis de la tarea y de la red de interconexión se pueden realizar de manera independiente. La respuesta temporal de la tarea se puede determinar con una gran variedad de técnicas ya existentes, lo cual no sucede con el análisis temporal de la NoC, para el que existen menos estudios publicados.

Estos requisitos se pueden comparar con las características del procesador utilizado en esta tesis, el procesador Epiphany:

1. Los núcleos del procesador son pequeños y simples.

2. Cada núcleo tiene su propia memoria local, no hay cachés y la escritura/lectura sobre las memorias locales de los núcleos es directa y se puede hacer entre núcleos.

3. El enrutado de la NoC es XY, el paquete viaja primero en la dirección X y luego en la Y. Se garantiza un ancho de banda mínimo y una latencia máxima entre enrutadores de los paquetes que viajen por la red. Cada enrutador tiene un arbitrio cíclico o *round-robin* que reparte temporalmente la utilización de los enlaces.

4. En la tesis realizamos un análisis de las tareas y la red de interconexión por separado, pero hemos detectado y estudiado el impacto directo que tiene la red con la tarea analizada. Al tener una malla con planificación estática hemos sido capaces de analizar y modelar su funcionamiento.

Además del procesador elegido, también se han analizado ciertos procesadores muchos núcleos como plataforma para este trabajo [36] y que, después de un estudio en gran profundidad, se descartaron. Estos procesadores son:

- **Intel Xeon Phi** es una familia de procesadores que lleva evolucionando desde 2010. Esta arquitectura está orientada a la computación de altas prestaciones. Su arquitectura consiste en una malla 2D que conecta celdas, cada una con 2 núcleos y cada uno con 2 Vector Processing Units (VPUs) que comparten 1MB de caché L2. La coherencia de la caché L2 y la conexión con la NoC se realizan

a través del Caching/Home Agent (CHA). El procesador con la arquitectura Knights Landing [37] tiene 72 núcleos.

La familia Intel Xeon Phi está muy dirigida a un mercado que nada tiene que ver con el tiempo real y eso se nota en que, aparentemente, no se permite la desactivación de la coherencia de caché L2 y que en cada celda se encuentre un sistema de doble núcleo. El pequeño tamaño de la caché privada L1 hace difícil evitar los conflictos debidos a la coherencia de la caché L2.

- **Tilera** es un procesador muchos núcleos cuya fabricación ha sido descontinuada. Los procesadores de la arquitectura Tilera-GX [38] se dividen en celdas, cada una con un procesador de 64-bits, memoria caché (niveles 1 y 2) y un switch que actúa como interfaz con la malla que comunica todas las celdas y que proporciona coherencia de la caché L2 a todos los núcleos. El procesador Tile GX-100, de esta familia, es un modelo que presenta 100 núcleos.

  El procesador con ejecución en orden es predecible. Pese a ser un sistema que no está orientado a tiempo real se permite cierta independencia entre zonas del procesador gracias al sistema de particionado. También se tiene una NoC de comportamiento predecible. Hay una cierta noción de memoria local si mediante el particionado se consigue que cada celda acceda solo a los datos de su propia caché L2 y no a los de otras celdas. Esta caché L2 tiene un tamaño de 256kB, que puede ser suficiente para un gran número de aplicaciones empotradas.

- **Kalray** es un fabricante que con su chip muchos núcleos MPA-256 [39] ha conseguido un procesador expresamente dirigido a sistemas embebidos y de tiempo real. Cuenta con 288 núcleos repartidos en 16 agrupaciones (clústers) de cómputo y 4 subsistemas de entrada/salida cada uno con 4 núcleos capaces de acceder a la memoria externa. Cada agrupación contiene 16 núcleos de ejecución más un núcleo que actúa como gestor de recursos. Pueden llegarse a conectar 4 procesadores Kalray MPPA-256 a través de una placa TurboCard3 en caso de requerirse mayor potencia de cálculo.

  Este procesador muchos núcleos está dirigido a sistemas de tiempo real. La arquitectura anidada nos proporciona un sistema de particionado directo. La NoC tiene tiempos de paso de mensajes acotados y cada núcleo presenta ejecución en orden. Se cuenta con una noción de memoria local, con un tamaño de 128kB.

  Pese a que es un candidato con un gran potencial para sistemas de tiempo real hemos decidido optar por un procesador muchos núcleos más sencillo y con una

arquitectura de red sin agrupaciones, como resulta la red de interconexión de 2D en malla, para que sea más sencilla la adaptación a otros procesadores muchos núcleos.

2.3.1. Análisis de tiempos de ejecución

El análisis de la respuesta temporal de cualquier aplicación concurrente descansa en un conocimiento preciso de los tiempos de ejecución de cada hilo de ejecución considerando su ejecución en aislamiento, sin interferencias con las ejecuciones de otros hilos. Puesto que el tiempo de ejecución del software es dependiente de los datos procesados, hay una variabilidad en el mismo. A los efectos del análisis de la respuesta temporal se modela el tiempo de ejecución por medio de sus cotas inferior y superior. Son los llamados tiempos de ejecución de mejor caso (Best-Case Execution Time o BCET) y peor caso (Worst-Case Execution Time o WCET). En esta tesis no se trata el cálculo de tiempos de ejecución, que se remite al uso de técnicas existentes. A continuación se detallan algunas de ellas.

Becker [40] presentó un framework de ejecución sin interferencias para el procesador muchos núcleos Kalray MPPA-256. Se basa en el encapsulado de los recursos compartidos en componentes, evitando que nadie más pueda acceder a ellos durante la ejecución. El acceso a estos recursos compartidos se bloquea para el resto de componentes desde el principio de la ejecución y se libera al final, no solo durante su uso. Para sincronizar los accesos a la memoria se realiza una planificación temporal concreta para evitar bloqueos en la medida de lo posible. En este tesis, sin embargo, solo se bloquean los recursos en el momento de su uso y se analiza el impacto temporal de encontrarnos esos recursos ocupados.

Perret [41] realizó un análisis de cómo acotar el tiempo de ejecución de peor caso en un procesador muchos núcleos Kalray MPPA-256 a través del aislamiento temporal entre las aplicaciones del sistema implementando dos mecanismos: aislamiento hardware entre las celdas del procesador a modo de particiones y un hipervisor de tiempo real que limita el comportamiento de las aplicaciones. Se realiza un estudio académico utilizando el caso de estudio ROSACE [3].

Skalistis [42] realiza una estimación sobre la cota superior del WCET de las tareas de una aplicación con dependencia de datos. Se concluye que evitando fuentes de interferencias se reduce drásticamente el impacto sobre el WCET, lo cual se acaba traduciendo en una mejora global de latencias.

2.3.2. Análisis de la respuesta temporal

Nikolić [43] utiliza Limited Migrative Model en el que una aplicación podría ejecutar en un núcleo perteneciente a un subconjunto de núcleos, permitiendo el balance de carga en tiempo de ejecución pero limitando entre que núcleos puede ser balanceada una aplicación. En ese artículo se diferencia entre tráfico directo y tráfico indirecto. El análisis de los paquetes se realiza de una forma parecida al propuesto en esta tesis, siendo la suma de la latencia en una situación aislada, el retardo generado por los bloqueos y el de la interferencias.

Rihani [44] realiza un análisis del tiempo de respuesta para una aplicación de flujos de datos. Basado en el MRTA framework, le añade la política de arbitrio del Kalray MPPA-256. Se añaden las interferencias locales entre núcleos y modela el tráfico por la red de interconexión. Se utiliza una limitación hardware del ancho de banda de cada clúster de la red. No hay transacciones de escritura sincronizadas, en esta tesis se desarrollan mecanismos que lo proporcionan, y las lecturas también bloquean el núcleo hasta completarse, lo cual también sucede en los mecanismos de lectura propuesto en esta tesis.

2.4. Sistemas de criticidad mixta

Es habitual que los componentes de un sistema de tiempo real tengan requisitos de fiabilidad y se clasifiquen por ello en diversos niveles de criticidad, según la gravedad de las consecuencias de los fallos que puedan sufrir. En algunos sistemas todos sus componentes tienen el mismo nivel de criticidad, pero también es habitual encontrar sistemas en que se mezclan componentes de diversos niveles: son los sistemas de criticidad mixta.

Pellizzoni [45] es el claro paradigma que consideró introducir los sistemas de criticidad mixta a plataformas embebidas multinúcleo y muchos núcleos. Tiene como objetivo que los componentes de alta criticidad no fallen aunque los componentes no críticos puedan llegar a fallar. Para impedir la propagación de los fallos de los componentes no críticos a los críticos se busca aislarlos, encapsulando cada núcleo en un envoltorio monitorizado con un control sobre la comunicación y el uso de recursos del propio núcleo.

Otro paso adelante en este tipos de sistemas es la definición del lenguaje de análisis y diseño de arquitecturas o Architecture Analysis and Design Language (AADL) [46] para sistemas de criticidad mixta facilitando el análisis temprano del sistema y con el objetivo de reducir la sobrecarga de computación y comunicación.

El uso de un bus compartido en estos sistemas resulta problemático, por lo que se puede considerar el uso de una NoC. Con la NoC se pueden integrar funciones con diferentes niveles de criticidad en el mismo sistema. Zamorano [47] propone la posibilidad de ejecutar sistemas de criticidad mixta en la misma plataforma muchos núcleos en la que haya más núcleos que particiones. Al tener un sistema particionado se evita tener sistemas de planificación más complejos. Se subraya que generalmente no se utiliza memoria virtual en este tipo de sistemas particionados por lo que nos ahorramos la traducción de direcciones de memoria, tal y como sucede en el sistema propuesto en esta tesis.

2.5. Sistemas operativos

Desarrollar software para sistemas de tiempo real en arquitecturas de procesadores muchos núcleos requiere conocimientos de la arquitectura a la hora de diseñar servicios del sistema operativo; tales como sincronización de tareas, localización de tareas, localización del código y datos, actualización de datos, acceso a periféricos, etc.

Gu [48] presenta un planificador maestro-esclavo para sistemas operativos de tiempo real para manejar aplicaciones en sistemas embebidos multinúcleo o muchos núcleos distribuidos sin memoria compartida. El resultado es un planificador de pequeño tamaño, lo cual es necesario para sistemas embebidos. Con este planificador además se proporciona un protocolo para facilitar la programación paralela. En esta tesis se ha decidido tener un planificador independiente ejecutando en cada núcleo siguiendo así el paradigma de *micro kernel*.

Como se dice en [49], donde se propone un sistema operativo para procesadores muchos núcleos desde una perspectiva de sistemas de seguridad crítica, una de las ventajas una plataforma muchos núcleos es que, al tener un gran número de núcleos, podría ser posible que haya núcleos que ejecutan una sola tarea. Incluso sería posible que más de un núcleo se dedique a ejecutar una tarea de manera exclusiva, de modo que paralelizando la tarea se reduciría su tiempo de ejecución de peor caso.

En ese mismo artículo [49] se utiliza la red NoC para el envío de mensajes entre núcleos, lo que implica que los núcleos del procesador tienen que tener un pequeño sistema operativo o *micro kernel* que se encargue de estos mensajes, de ciertos parámetros de configuración y de que cada componente respete su espacio de memoria. Es este *kernel* el encargado de cargar el código de los componentes que debe ejecutar cada núcleo. También puede suceder que en un núcleo se ejecuten varios componentes, por lo que el *kernel* debe tener también un planificador local (siendo innecesario en esta arquitectura

de *micro kernel* tener un planificador global) y medios para proteger el estado de los componentes. Tanto el código como los datos pertenecientes al *micro kernel* deben protegerse de cualquier intento de manipulación por parte de la aplicación. En sistemas de tiempo real todos los *micro kernels* tienen que proporcionar un comportamiento temporal predecible. En el artículo se separan los componentes en diferentes núcleos, asignando también núcleos a ciertos servicios globales (como la entrada/salida), lo cual en ciertas ocasiones empeora el peor caso respecto a ejecutar el servicio en el mismo núcleo que el componente que llama al servicio. Al utilizar la entrada/salida a un servicio proporcionado en un núcleo externo se puede incurrir a un aumento de la variabilidad de los tiempos de respuesta.

En cuanto a la entrada/salida, [49] da una solución que consiste en solo permitir a ciertos núcleos el acceso al hardware de manera directa. Estos núcleos actuarían como un servidor de entrada/salida. Estos núcleos de entrada/salida también podrían recurrir a núcleos dedicados a las interrupciones para no degradar en exceso la respuesta temporal.

Basándose en todo esto [49] propone ciertos retos en los procesadores muchos núcleos para sistemas de criticidad mixta:

- Manejo de la NoC: el sistema operativo debe proporcionar medios para gestionar los requerimientos de la aplicación en lo que respecta al ancho de banda y/o latencia. También debe desarrollarse un mecanismo de tolerancia a fallos.

- Encontrar un balance entre la distribución y centralización de los servicios esenciales: los servicios deberían ejecutarse en los mismos núcleos que la aplicación cliente reduciendo el tiempo de peor caso. Esto implica en algunos casos la duplicación de código.

Se han llevado a cabo proyectos que desarrollado sistemas operativos de tiempo real (Real-Time Operating Systems o RTOS) para plataformas muchos núcleos::

- **P-SOCRATES** [50] desarrolla un nuevo entorno que abarca desde el diseño conceptual de la funcionalidad del sistema hasta su implementación, facilitando así el desarrollo de arquitecturas paralelas estandarizadas en cualquier tipo de sistema. El sistema operativo no tiene memoria compartida y utiliza un protocolo para simplificar la programación paralela siguiendo el modelo OpenMP [51]. Utiliza el paso de mensajes entre objetos. En este proyecto se ha desarrollado una adaptación del RTOS Erika, desarrollado por Erika Enterprise [52], llamado Erika3 tratándose de un RTOS que utiliza una imagen de unos poco kB, puede ser ejecutado en el procesador Kalray MPPA-256.

- **MOSSCA** [49], Manycore Operating System for Safety-Critical, es un sistema operativo basado en una arquitectura *micro kernel*, con una instancia del sistema operativo en cada núcleo, que ofrece servicios en relación al hardware del que dispone cada núcleo. Se basa en cumplir las siguientes propiedades:

  - El sistema debe tener un comportamiento temporal predecible y conducir a la implementación de sistemas analizables.
  - El particionado en tiempo y espacio evita interferencias y facilita el análisis. Hay que tener un cuidado especial con los recursos compartidos.
  - Los procesadores del mismo componente pueden comunicarse entre ellos. Para comunicarse a través de los límites de particionado hay que utilizar el sistema operativo.

- **eSol** ha desarrollado un RTOS para procesadores muchos núcleos llamado eMCOS [53] que implementa una arquitectura *micro kernel* distribuida. Las instancias del *micro kernel* tienen una funcionalidad mínima (paso de mensajes, planificación local y gestión de hilos) mientras las operaciones más avanzadas las realizan unos núcleos servidores. Ha sido desarrollado para una gran variedad de arquitecturas entre las que no se encuentra el procesador Epiphany. La planificación la realiza mediante dos planificadores, uno que asigna las tareas de mayor prioridad a cada núcleo para que puedan ejecutarse siempre que estén listos. El resto de tareas están bajo otro planificador que las distribuye sobre los núcleos balanceando su carga y buscando alto ancho de banda.

- **Altamary** [54] ha portado el popular sistema operativo libre RTEMS [1] para el procesador Epiphany ejecutando en la placa Parallella, que es la utilizada para esta tesis. Como se especifica en la Sección 4.1, la placa Parallella proporciona una memoria global compartida aparte de la memoria local de cada núcleo. RTEMS puede estar localizado en cualquiera de las dos memorias. Cuando RTEMS está en la memoria compartida, su funcionamiento es significativamente más lento. Por otro lado, cuando se utiliza la memoria local de los núcleos, solo quedan 5kB para ejecutar los componentes. RTEMS es un RTOS con cierta complejidad que implementa varios algoritmos de planificación.

[1] https://www.rtems.org/

2.6. Mapeado de tareas

Para conseguir que un sistema de tiempo real sea ejecutado en procesadores muchos núcleos se pueden asignar ciertas tareas de manera permanente a núcleos del procesador, pudiendo llegar a reducir así el tiempo de respuesta de peor caso al minimizar los cambios de contexto. El mapeado de tareas en procesadores muchos núcleos es un problema NP-Completo por lo que las soluciones utilizadas actualmente son heurísticas. En estos algoritmos se tienen en cuenta la distancia entre los núcleos que se van a comunicar y la congestión que pueda haber en los enlaces que se van a utilizar. Debido a que la complejidad de la colocación en un sistema aumenta de manera exponencial con el tamaño del software (número de tareas) y del hardware (cantidad de núcleos), la gran mayoría de algoritmos son heurísticos genéticos [55], simulación [56] o ramificación y poda [57]. También se puede buscar reducir en lo máximo posible el consumo energético utilizando mapeados eficientes de tareas y reduciendo la frecuencia de funcionamiento del procesador mientras se mantiene la planificabilidad [58].

Hay una serie de consideraciones a tener en cuenta a la hora de mapear tareas en un procesador:

- Las tareas que accedan a dispositivos deben estar lo más cerca posible a los mismos.
- Las tareas que utilicen asiduamente la memoria RAM deben colocarse lo más cercanas a su bloque.
- Las tareas que se comuniquen entre ellas deben colocarse cerca.
- Hay que evitar congestiones en la NoC.
- Las tareas más críticas o con menor plazo tienen que estar en una situación más favorable que las demás.
- En aplicaciones complejas que demandan flexibilidad puede ser de gran interés poder modificar el mapeado en tiempo de ejecución, en momentos de baja carga del sistema, según las necesidades que tenga el propio sistema.

Junto al mapeado se suelen aplicar algoritmos de particionado estáticos (igualmente a través de algoritmos heurísticos y genéticos) donde se tienen en cuenta las siguientes características: los ciclos que van a tardar en comunicarse diferentes tareas, la probabilidad de que haya comunicación entre las tareas, la contención en los enlaces debida al tráfico y la latencia y probabilidad de acceso a recursos compartidos.

Para realizar correctamente el particionado se requieren protocolos con los que mantener las diferentes particiones aisladas de manera espacial y temporal. Las particiones no podrán estar completamente aisladas pues es preciso tener en cuenta que hay cierta contención para el intercambio de mensajes por la NoC y por el tráfico que atraviesa otras particiones para acceder a recursos compartidos.

En esta tesis se imponen determinadas restricciones que posibilitan obtener cotas superiores para la interferencia producida por el tráfico de mensajes en la NoC. En base a estas restricciones se desarrollan técnicas de modelado y análisis de estas interferencias.

Capítulo 3

Modelado de procesadores muchos núcleos basados en malla

En este capítulo se describirá el modelo desarrollado para realizar un análisis temporal de aplicaciones ejecutadas en procesadores muchos núcleos basados en malla. Este modelo que describe tanto la plataforma de ejecución y comunicación como la aplicación se ha basado en el modelo MAST [16] utilizando elementos específicos para modelar la plataforma muchos núcleos y las comunicaciones entre los diferentes núcleos en lugar de utilizar mensajes y redes de interconexión que se utilizan habitualmente para modelar elementos de un sistema distribuido de tiempo real. El modelo MAST fue concebido para describir los aspectos fundamentales del comportamiento temporal de un sistema software junto a su plataforma hardware con el objetivo de analizar su planificabilidad, que se define como la capacidad de cumplir los requisitos temporales bajo una situación particular. Utilizando la herramienta MAST también se pueden conocer los tiempos de respuesta, tanto de mejor como de peor caso, de los diferentes elementos ejecutados en el sistema además de la utilización de cada recurso de procesamiento.

Tanto el modelo original de partida como la forma de utilizarlo para disponer de un modelo que sirva para procesadores muchos núcleos basado en malla con un esquema de funcionamiento SAF será explicado en las siguientes secciones de la tesis.

3.1. Elementos básicos del modelo

El modelo MAST para arquitecturas software está basado en flujos de principio a fin, o *end-to-end* (e2e), que son activados mediante eventos externos que determinan la carga de trabajo. Cada uno de estos flujos se describe como un grafo direccional acíclico en el que los eventos se utilizan para interconectar actividades llevadas a cabo

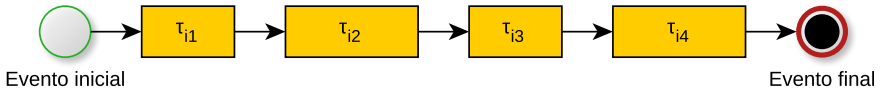

Figura 3.1 Ejemplo de un flujo e2e formado por cuatro actividades.

por el sistema. Dichas actividades se llaman *steps* en el modelo MAST. Cada llegada de un evento externo activa la ejecución de una instancia del flujo e2e. En su forma más sencilla los flujos e2e siguen el llamado modelo de flujo lineal en el que la finalización de una actividad genera un evento interno que puede activar la siguiente actividad o finalizar el flujo e2e en caso de ser la última actividad del flujo. Esto está ilustrado en la Figura 3.1. En un sistema distribuido como es un procesador muchos núcleos que contiene tanto núcleos como elementos de la red de interconexión, se encuentran dos tipos de modelado de actividades: a) la ejecución de un hilo en un núcleo, y b) la transmisión de un mensaje a través de la NoC.

El modelo de un procesador muchos núcleos tiene n recursos de procesamiento $\{PR_1, PR_2, ..., PR_n\}$, uno por cada núcleo del procesador. Cada núcleo tiene un planificador con una política de planificación concreta, como podrían ser las prioridades fijas. Como esta tesis busca el modelado sobre procesadores muchos núcleos que utilicen una red de interconexión de malla 2D hemos adaptado la notación de los recursos de procesamiento definiendo un PR_i con sus coordenadas en la malla convirtiéndolo así en PR_{xy} donde la x sería la columna en la que está situado el núcleo e y la fila.

Γ_i representa un flujo e2e donde cada actividad se denota como τ_{ij}. Cada actividad está asignada de manera estática a un recurso de procesamiento PR_{xy} determinado, con una prioridad fija $prio_{ij}$. Se pueden mapear varias actividades en el mismo núcleo, como se muestra en la Figura 3.2 donde las cuatro actividades de un flujo e2e se han mapeado en dos núcleos. No hay ninguna norma sobre dónde deberían de ser mapeados las actividades de un mismo flujo e2e, esto es, las actividades se pueden mapear en el mismo núcleo o en núcleos diferentes.

El flujo e2e Γ_i es disparado por el evento externo e_i. Dicho evento externo puede seguir unos de los tres siguientes patrones de activación:

- **Evento periódico.** Generado a intervalos de tiempos regulares con un periodo T_i.
- **Evento esporádico.** Generado a intervalos de tiempos irregulares. Existe un tiempo mínimo entre activaciones.

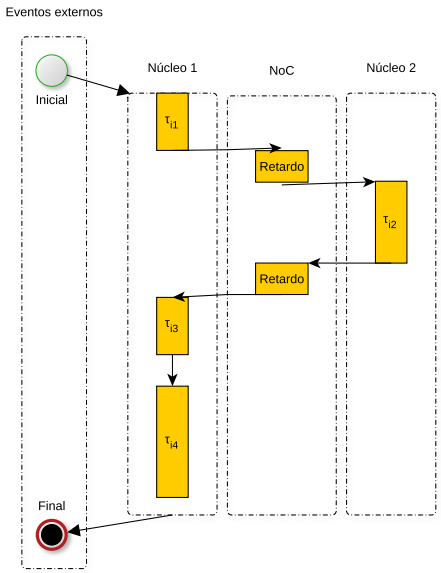

Figura 3.2 Flujo e2e mapeado en dos núcleos.

- **Evento aperiódico.** Generado a intervalos irregulares sin limitación entre intervalos de activaciones.

Cada actividad τ_{ij} perteneciente al flujo e2e Γ_i tiene un tiempo de ejecución de peor caso (Worst-Case Execution Time, WCET) C_{ij}, correspondiente con una cota superior al peor tiempo de ejecución de una instancia del hilo ejecutado en un núcleo sin tener en cuenta la contención de otros hilos o de mensajes que circulan por la red. La actividad también tiene tiempo de ejecución de mejor caso (Best-Case Execution Time, BCET) o una estimación acotada por abajo del mismo, C^b_{ij}.

Se puede ver un flujo e2e de ejemplo en la Figura 3.1. Este ejemplo presenta cuatro actividades ($\tau_{i1}...\tau_{i4}$) con diferentes C_{ij}, que se han representado de manera gráfica variando su anchura, así como un evento inicial y un evento final.

Se considera que los hilos de ejecución, llamados *Scheduling Servers* en MAST, están particionados de manera estática en el conjunto de n núcleos idénticos organizados en m filas y columnas $\{PR_{00}, PR_{01}..., P_{m-1,m-1}\}$. Las actividades en un flujo e2e se asocian a hilos de ejecución. El conjunto de hilos de ejecución asociado a un núcleo PR_{xy} se denomina como PR^T_{xy}.

Cada flujo Γ_i tiene un plazo (*deadline*) relativo D_i, lo cual implica que se tiene que completar la última actividad del flujo antes de que transcurra un intervalo D_i respecto al disparo del evento externo de activación e_i. El plazo D_i puede ser arbitrario, esto es, mayor, menor o igual que el periodo T_i del mismo flujo e2e Γ_i.

Cada actividad τ_{ij} tiene tanto un tiempo de respuesta de peor caso (WCRT) R_{ij} como un tiempo de respuesta de mejor caso (BCRT) R^b_{ij}, relativo a la llegada del evento externo de su flujo e2e e_i. El WCRT R_i de un flujo lineal e2e Γ_i se define como el tiempo de respuesta de peor caso de su última actividad. El sistema se considera planificable en caso de que el WCRT R_i de que todos y cada uno de los flujos e2e del sistema Γ_i sean menores o iguales que su plazo D_i.

El modelo MAST permite analizar el tiempo de respuesta de diferentes sistemas utilizando un amplio abanico de técnicas de análisis, pero no es válido para analizar directamente procesadores muchos núcleos. Es por esto que se requiere un tratamiento especial que convierta un sistema muchos núcleos en un conjunto de recursos de procesamiento donde la interacción con la NoC se modele en base a otros elementos de modelado presentes en MAST.

Modelar un procesador muchos núcleos requiere modelar los mensajes transmitidos a través de la NoC. Estos mensajes son generados por cada salto presente en un flujo e2e, esto es, cuando el recurso de procesamiento de la actividad τ_{ij} es diferente al recurso de procesamiento de la actividad $\tau_{i(j+1)}$. En esta situación la actividad τ_{ij} puede generar uno o más mensajes m_{ijk} al final de su ejecución. Como se justificará en las secciones venideras, el tiempo de generación de todos los mensajes está incluido en el tiempo de ejecución de la actividad τ_{ij} y se introduce un elemento de retardo de MAST para modelar la latencia de la travesía del mensaje por la NoC. Un retardo en MAST es un manejador de eventos que genera su evento de salida después de que haya transcurrido intervalo de tiempo desde la llegada del evento de entrada.

Como resumen, un flujo Γ_i, como el mostrado en la Figura 3.2 tiene un periodo T_i, un plazo D_i y está formado por un conjunto ordenado de actividades $\{\tau_{i1}, \tau_{i2}, ..., \tau_{ix}\}$. Donde cada actividad τ_{ij} está definida como $\{C_{ij}, C^b_{ij}, prio_{ij}, PR_{xy}\}$ que representa respectivamente su WCET, BCET, prioridad y el recurso de procesamiento donde la actividad será ejecutada. Para cada salto del flujo e2e entre un recurso de procesamiento y otro se introduce un retardo para modelar el comportamiento de la NoC. La duración de esté bloque de retardo será discutida en las secciones venideras.

3.2. Parámetros básicos de la red de interconexión

Pese a que en el trabajo realizado en esta tesis se contempla una red de interconexión 2D de tipo malla, el modelo que hemos desarrollado podría utilizarse en otro tipo de NoCs que utilicen la técnica de control de enrutado SAF. Un paquete se considera indivisible, con lo que podemos decir que un paquete equivale al concepto de *flit* que se

utiliza en otras redes de interconexión. Consideramos que un mensaje viaja a través de la red paquete a paquete y cada uno de estos paquetes se enruta de manera individual para realizar su travesía por la NoC.

La NoC se define mediante estos tres parámetros:

- **Frecuencia de la NoC (F_{NoC}).** Este parámetro es el número de paquetes generado por un núcleo que la NoC es capaz de absorber por unidad de tiempo, en ausencia de cualquier otro tráfico en la red. El inverso de esta frecuencia de la NoC es el ciclo NoC, es decir, el tiempo que requiere la NoC para absorber un paquete generado por un núcleo.

- **Latencia de salto (L_H).** Este parámetro es el tiempo requerido por un paquete para pasar de un enrutador a uno de sus enrutadores vecinos sin tener que competir con ningún otro paquete por el enlace que comunica ambos.

- **Latencia de arbitrio del enrutado (L_R).** Este parámetro es el tiempo requerido por el enrutador para tomar una decisión de arbitrio. Esta latencia se añade cada vez que un paquete pierde su arbitrio en un enrutador en favor de otro paquete.

Puede existir más de una NoC en el mismo procesador conectando los mismos núcleos pero utilizadas por diferentes tipos de mensajes (de escritura o de lectura) con valores de los parámetros de la NoC diferentes para cada una de ellas. Los mensajes solo comparten los recursos de la NoC con otros mensajes que viajen por la misma NoC.

3.2.1. Modelado de los mensajes

Los núcleos de un procesador muchos núcleos puede acceder a la memoria local de otras celdas en la NoC. Leer o escribir en las memorias locales de otras celdas implica el envío de un mensaje a través de la red de interconexión que las comunica. Esto implica que la ejecución de una actividad en un núcleo podría requerir múltiples mensajes a ser enviados, al menos uno cada vez que se accede a la memoria local de otra celda. Esto puede complicar el modelado de la aplicación. Para simplificar el modelo, suponemos que la actividad en ejecución en un núcleo determinado utiliza la memoria local de su celda, evitando así el uso de memoria compartida, y limitamos la escritura en la memoria local de otros núcleos al final de la ejecución de la actividad. En definitiva, restringimos la comunicación con los demás núcleos al final de la ejecución de cada actividad, cuando esta ha producido los resultados de su cálculo.

En algunos casos, se utilizarán primitivas de sincronización de alto nivel para el envío de mensajes al final de la ejecución de una actividad, lo cual puede implicar además la utilización de *mutexes* o *spinlocks* distribuidos u otras primitivas similares que requieran la lectura de la memoria local del núcleo destinatario del mensaje. Como consecuencia, también pueden ocurrir operaciones de lectura además de las de escritura al final de una actividad, requiriendo ambas operaciones mensajes que atraviesan la NoC. Más adelante vamos a describir las primitivas de sincronización de alto nivel que se han implementado para este propósito y describiremos cómo modelar los mensajes asociados a las mismas.

La activación de todas las actividades de un flujo e2e, salvo la primera, requiere del envío de un mensaje. Para cada activación, una actividad puede activar a otra actividad mapeada en el mismo o en un recurso de procesamiento diferente. El mensaje de activación entre actividades mapeadas en diferentes PRs tiene que viajar a través de la NoC. Un mensaje se compone de un conjunto de paquetes. La actividad destinatario del mensaje es activada en cuanto se recibe el último paquete del último mensaje.

Un elemento clave en nuestro modelo es la ratio de generación de paquetes de un mensaje. Dado un mensaje, m_{ijk}, definimos la ratio de generación de paquetes de m_{ijk}, denotada como ρ_{ijk}, como el inverso del número mínimo de ciclos de la NoC que pueden transcurrir entre el envío a la NoC de dos paquetes de ese mensaje. En caso de que un mensaje esté formado por un solo paquete, ρ_{ijk} será el inverso del mínimo intervalo entre el envío de esta paquete y el paquete más cercano del mensaje anterior o posterior enviado por el mismo PR por el mismo enlace de salida. En el caso especial de que se envíe un único mensaje desde un PR, ρ_{ijk} será el inverso del intervalo mínimo posible entre dos paquetes enviados para la activación de una actividad: $T_{ij} - J_{ij}$, con un factor de conversión para transformar ese tiempo transcurrido en ciclos de la NoC.

Cuando el tiempo de respuesta de peor caso es superior al periodo y se ha dado esa situación de peor caso, el flujo e2e Γ_i está listo para ejecutar su siguiente instancia de forma inmediata, ya que el evento inicial periódico ya habría llegado en un instante anterior. A partir de ese momento lo mejor que le puede ocurrir al flujo es encontrarse en su mejor caso y tener un tiempo de respuesta R_{ijb}. Por tanto la mínima separación entre esa instancia y la anterior es justamente ese valor.

Sin embargo, si el tiempo de respuesta de peor caso es inferior al periodo, la mínima separación entre instancias será mayor que R_{ijb}. Tomaremos la situación más desfavorable para el cálculo de la ratio máxima:

$$\rho_{ijk} = \frac{1}{R^b_{ij}} \cdot \frac{1}{F_{NoC}} \tag{3.1}$$

Una operación de lectura entre PR_{xy} y $PR_{x'y'}$ necesita dos mensajes: un mensaje desde PR_{xy} hacia $PR_{x'y'}$ que indique el área de memoria a leer y la dirección de memoria local dónde escribir sus contenidos y un segundo mensaje desde $PR_{x'y'}$ hacia PR_{xy}, que llevará el contenido del área de memoria que se quería leer. Esto tiene como consecuencia que todo mensaje de lectura m_{ijk} genera otro mensaje de respuesta *write-back*, $m_{ij(k+1)}$, que escribirá el dato intercambiando la celda de destino y la de origen. Los mensajes de respuesta son escritos en la memoria local de la celda desde la que ejecuta la tarea lectora, directamente por la interfaz de la red, lo que justifica que la petición de lectura contenga las dos direcciones de memoria indicando qué quiere leer y dónde lo quiere depositar.

A modo de resumen, un mensaje m_{ijk}, siendo el k-ésimo mensaje generado al final de una actividad τ_{ij} hacia otra actividad $\tau_{i(j+1)}$ mapeada en un PR diferente de la actividad emisora, se define como una tupla de cuatro elementos $\{\rho_{ijk}, \mu_{ijk}, t_{ijk}, d_{ijk}\}$ donde:

- ρ_{ijk} la ratio de generación de los paquetes del mensaje medida en $ciclos^{-1}$.
- μ_{ijk} el número de paquetes que forman el mensaje.
- t_{ijk} el tipo de mensaje (mensaje de lectura, escritura o respuesta).
- d_{ijk} la celda de destino del mensaje generado por la actividad; también toma el rol de mensaje de origen cuando el mensaje es de tipo de respuesta.

En nuestro modelo, la generación de los paquetes de los mensajes producidos al final de una actividad junto con el tiempo necesario para inyectarlos en la NoC se incluyen en el tiempo de ejecución de la propia actividad, ya que el núcleo que ejecuta la actividad está ocupado durante dichas acciones y no puede ejecutar otras instrucciones. Además, también se considera tiempo de ejecución de la actividad el intervalo de tiempo que transcurre hasta completar una operación de lectura, si la hay, lo cual incluye el tiempo de travesía por la red del mensaje de petición de lectura, el tiempo requerido por la interfaz de red para realizar la operación de lectura en su celda y el tiempo que tarda en volver por la red el mensaje de respuesta generado ya que nuestro modelo supone que que el núcleo está detenido esperando la llegada de este último mensaje.

La ejecución de la actividad finaliza cuando el último paquete del último mensaje a enviar está en el enrutador de la celda en la que se está ejecutando. La siguiente actividad del flujo e2e se activará tan pronto como llegue este último paquete a su dirección de destino de la memoria local del núcleo en el que está mapeado dicha actividad. En nuestro modelo, el tiempo necesario para que este último paquete viaje a

través de la NoC se modela utilizando un elemento de retardo de MAST con un tiempo relativo máximo y mínimo determinado.

El intervalo de tiempo mínimo asignado al elemento de retardo de MAST es el tiempo requerido por el paquete para viajar a través de la NoC en ausencia de otros paquetes competidores. En esta situación, el tiempo de travesía del último paquete de un mensaje m_{ijk} solamente depende del número de saltos o, lo que es lo mismo, del número de enrutadores que tenga que atravesar, H_{ijk}, incluyendo el enrutador de la celda de origen y de la celda de destino, y de la latencia introducida por cada salto L_H. El tiempo de travesía de mejor caso (Best-Case Traversal-Time o BCTT) para el último paquete del mensaje m_{ijk}, TT^b_{ijk}, que se muestra en la Ecuación 3.2.

$$TT^b_{ijk} = L_H \cdot H_{ijk} \tag{3.2}$$

Con el objetivo de obtener el tiempo de intervalo máximo del elemento de retardo de MAST necesitamos estimar el tiempo de travesía de peor caso (Worst-Case Traversal Time o WCTT) del paquete, el cual tiene que tener en cuenta al resto de paquetes que estén viajando por la NoC. Este valor será calculado en la Sección 3.2.3 después de la introducción de la restricción de la ratio de generación de paquetes de la siguiente sección.

3.2.2. Restricción de ratio máxima

Las interferencias entre paquetes hacen que el cálculo del WCTT de los mensajes sea más complicado, tanto que se ha considerado un problema NP-completo [18] ya que el estudio de la gestión de recursos compartidos se vuelve más complicada al aumentar el número de núcleos en el mismo chip. Además, las técnicas de análisis de planificabilidad de NoCs existentes son pesimistas a la hora de acotar los efectos del resto de los mensajes, para poder llegar a técnicas de análisis viables.

En nuestra aportación, hemos reducido la complejidad del análisis al considerar la ratio de generación de paquetes que tienen los mensajes para restringir la ratio de utilización máxima de los enlaces de la NoC de los sistemas que somos capaces de modelar. Describiremos a continuación esta limitación y cómo consigue evitar el fenómeno de la contrapresión.

Vamos a llamar $M_{xy \to x'y'}$ al conjunto de mensajes que van del enrutador perteneciente a PR_{xy} a través de un enlace de salida al enrutador perteneciente a la celda vecina $PR_{x'y'}$. Cuando hay más de un mensaje generado desde el mismo PR que utiliza el mismo enlace que comunica PR_{xy} con $PR_{x'y'}$, solo los mensajes con mayor ratio

de generación son incluidos en $M_{xy\to x'y'}$. Llamamos ratio de transmisión acumulada del enlace, $P_{xy\to x'y'}$, al sumatorio de todas las ratios de generación de mensajes en $M_{xy\to x'y'}$:

$$P_{xy\to x'y'} = \sum_{m_{ijk}\in M_{xy\to x'y'}} \rho_{ijk} \tag{3.3}$$

Proponemos como condición previa al análisis de planificabilidad de un sistema tener todos los enlaces con una ratio de transmisión acumulada no mayor que el inverso de la latencia de arbitrio del enrutador $\frac{1}{L_R}$, o lo que es lo mismo, que $P_{xy\to x'y'} \leq \frac{1}{L_R}$, para todo enlace de la NoC.

Si esta condición se cumple para todos los enlaces podemos asegurar que la red de interconexión es capaz de transmitir los paquetes al menos a la misma ratio con la que son generados.

La restricción previamente mencionada nos permite librarnos de la mayoría de las interferencias entre mensajes. De acuerdo con la bibliografía [20] existen dos fuentes de interferencias entre mensajes a tener en cuenta: la interferencia directa y la interferencia indirecta. La interferencia directa sucede entre mensajes que comparten algún enlace en su ruta cuando el enlace no es capaz de transmitir los paquetes a la ratio a la que le llegan. En esta situación se puede producir contrapresión en los mensajes llegando a detener la generación del mensaje en el núcleo, lo que se denomina *stall*. Por otro lado, las interferencias indirectas pueden suceder cuando un mensaje que ha sufrido contrapresión debido a una interferencia directa entre dos o más mensajes acaba afectando a tiempo de travesía de otros mensajes. Por ejemplo, supongamos que tenemos tres mensajes A, B y C, donde los mensajes B y C comparten el mismo enlace y los mensajes A y B comparten un enlace anterior en el recorrido por la ruta del mensaje B. Si el mensaje B sufre contrapresión debido a la interferencia directa con el mensaje C, el mensaje B será bloqueado en el enrutador y entonces bloqueará su enlace de entrada correspondiente. Si este enlace de entrada es compartido con el mensaje A, entonces este mensaje A sufrirá una interferencia indirecta por el mensaje C, a pesar de que estos dos mensajes no comparten ningún enlace. Como ya introdujo Xiong [21], la contrapresión puede ser un comportamiento problemático que con el que habría que tener consideraciones especiales.

Para aclarar la diferencia entre interferencia directa e indirecta vamos a apoyarnos en la Figura 3.3. Un mensaje generado por la tarea τ_{11} puede tener interferencia directa con los mensajes generados por la tarea τ_{21}. Además de eso, un mensaje generado por la tarea τ_{21} puede tener interferencia directa con los mensajes generados por la tarea τ_{31} y/o por los mensajes generados por la tarea τ_{41}. Es por este motivo que un

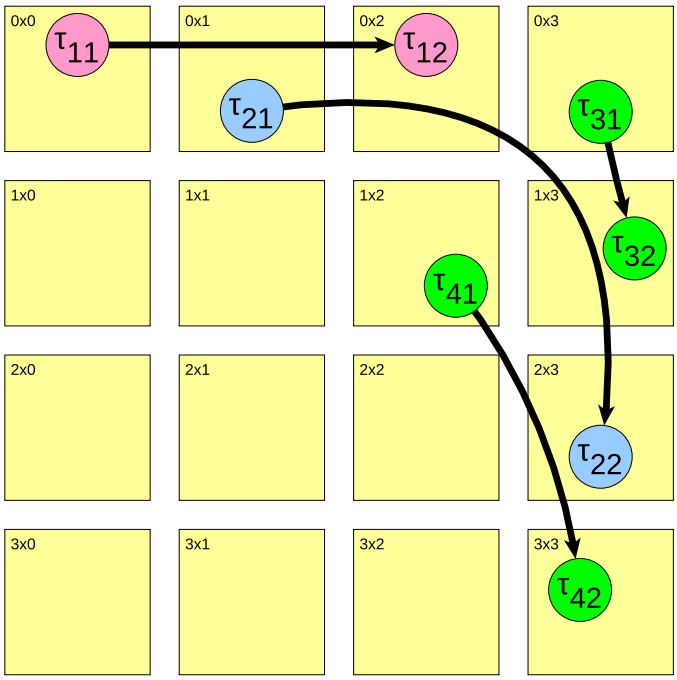

Figura 3.3 Tráfico generado por 4 tareas en una red de tipo malla 4x4 2D con enrutado XY.

mensaje generado por la tarea τ_{11} puede tener una interferencia indirecta causada por los mensajes generados por las tareas τ_{31} y τ_{41}.

La interferencia directa no puede generar contrapresión cuando los enlaces compartidos verifican la restricción de ratio máxima ya que con esta restricción nos aseguramos que los paquetes son capaces de transmitirse por los enlaces de la NoC al mismo ritmo que se generan. La interferencia indirecta tampoco puede existir en una sistema que cumple esta restricción ya que, para que se produzca una interferencia indirecta, se tiene que producir contrapresión en primer lugar.

Sin embargo, hay una pequeña y acotada posibilidad de interferencia que se puede producir en los sistemas que verifican la restricción de ratio máxima. Este tipo de interferencia está relacionada con la política de arbitrio del enrutador y será discutida en la siguiente sección.

La restricción de ratio máxima simplifica enormemente el análisis. Además, la limitación no es muy restrictiva con el diseño del sistema ya que las NoCs suelen tener una gran capacidad. Aparte, si un sistema no es analizable debido a estas restricciones propuestas, las diferentes actividades de un flujo e2e podrían volver a mapearse de otra manera para intentar cumplir las restricciones.

Como veremos en la Sección 4.2.4, cuando este modelo se aplica al procesador muchos núcleos Epiphany, esta restricción es razonable ya que prácticamente no impone restricción alguna al software.

3.2.3. Tiempo de travesía de los paquetes

Si el sistema que se está analizando cumple con la restricción de ratio de transmisión acumulada, $P_{xy \to x'y'} \leq \frac{1}{L_R}$, para todos sus enlaces, las interferencias ente los paquetes del sistema estarán acotadas, ya que solo un número limitado de paquetes pueden afectar a los enlaces compartidos en la ruta que seguirá el paquete por la NoC al estar la utilización de cada enlace limitada por la ratio. La única interferencia entre paquetes será la causada por la política de arbitrio de los enrutadores.

Cuando más de un paquete se localiza, de manera simultánea, en los *buffers* de entrada de un enrutador y se dirigen al mismo enlace de salida, el enrutador debe aplicar la política de arbitrio de acceso al enlace de salida compartido.

La política de enrutado más extendida es la cíclica, o *round-robin*. Utilizando esta política, un paquete que esté intentado acceder a un enlace de salida de un enrutador esperará, en el peor de los casos posibles, hasta que el enrutador haya enviado a través de ese enlace un paquete que estuviera esperando en cada uno de los restantes *buffers* de entrada. El retardo causado por cada paquete competidor es el parámetro L_R de la NoC.

Como suponemos que las rutas de cada paquete del sistema son conocidas, podemos determinar el número máximo de paquetes competidores que un determinado paquete puede encontrar en cada uno de los enrutadores que atravesará. Para cada enrutador, r_{xy}, en la ruta del mensaje m_{ijk}, llamamos nb_{ijkxy} al número de *buffers* de entrada utilizados por mensajes que comparten el mismo enlace de salida que el mensaje que está siendo analizado (siempre sin tener en cuenta el *buffer* que utiliza el propio mensaje analizado). El arbitrio de enrutado genera una interferencia llamada I_{ijk}, que se va acumulando por los enrutadores que atraviesa el mensaje, calculada como:

$$I_{ijk} = \sum_{r_{xy} \in route_{ijk}} nb_{ijkxy} \cdot L_R \tag{3.4}$$

Como consecuencia, el tiempo de travesía de peor caso de cualquier paquete del mensaje m_{ijk}, es el tiempo de mejor caso obtenido en la Ecuación 3.2 más la interferencia que suceda en la ruta:

$$TT_{ijk} = TT^b_{ijk} + I_{ijk} = L_H \cdot H_{ijk} + I_{ijk} \tag{3.5}$$

Un ejemplo del cálculo del WCTT de un paquete transmitido por un sistema sobre la arquitectura Epiphany se explicará en la Sección 4.2.4.

Como ya ha sido previamente mencionado, la política cíclica o *round-robin* es la política de enrutado más común, pero nuestro modelo se podría aplicar a muchas otras políticas. El único requisito que tenemos es que se sepa el retardo máximo que puede suponer la política para un determinado paquete como función del máximo número de paquetes que compiten por el mismo recurso.

3.2.4. Modelado de la NoC con elementos de MAST

Resumiendo lo que ya se ha expuesto en este capítulo, cuando dos actividades consecutivas en un flujo *e2e*, τ_{ij} y $\tau_{i(j+1)}$, están mapeados en diferentes PRs, al final de la ejecución de la actividad τ_{ij} se generarán uno o más mensajes de escritura y lectura para activar la siguiente actividad en el flujo $\tau_{i(j+1)}$. Como se ha dicho anteriormente, el tiempo necesario para generar los paquetes y para inyectarlos en la NoC está incluido en el tiempo de ejecución de la actividad C_{ij}. En el caso de los mensajes de lectura esto también incluye el tiempo de travesía por la NoC de la petición de lectura y del mensaje de respuesta.

Como consecuencia, el tráfico de la NoC afectará al tiempo de ejecución de las actividades. De esta manera, cada mensaje de lectura m_{ijk} aumentará el WCET de τ_{ij} en un valor igual a la máxima interferencia que el último paquete del mensaje pueda sufrir de acuerdo a la Ecuación 3.4, además de la máxima interferencia que pueda sufrir por el correspondiente mensaje de respuesta $m_{ij(k+1)}$.

Siendo C_{ij} el tiempo de ejecución de peor caso de τ_{ij} cuando se mide o estima en aislamiento, podemos calcular C'_{ij} como:

$$C'_{ij} = C_{ij} + \sum_{\forall k: t_{ijk} = read} (I_{ijk} + I_{ij(k+1)}) \tag{3.6}$$

Es importante entender que los mensajes de escritura que no son consecuencia de responder a una petición de lectura nunca incrementarán C_{ij}. Esto solo podría pasar en situaciones que provoquen detención, pero gracias a la restricción de la ratio máxima de generación de paquetes esto nunca sucederá en los sistemas que somos capaces de modelar.

El intervalo que transcurre entre que τ_{ij} inyecta el último paquete en la NoC (acabando entonces su ejecución) y que se activa la siguiente actividad $\tau_{i(j+1)}$ se modela mediante un elemento de retardo de MAST con un tiempo de retardo mínimo y máximo obtenido de las Ecuaciones 3.2 and 3.5 respectivamente.

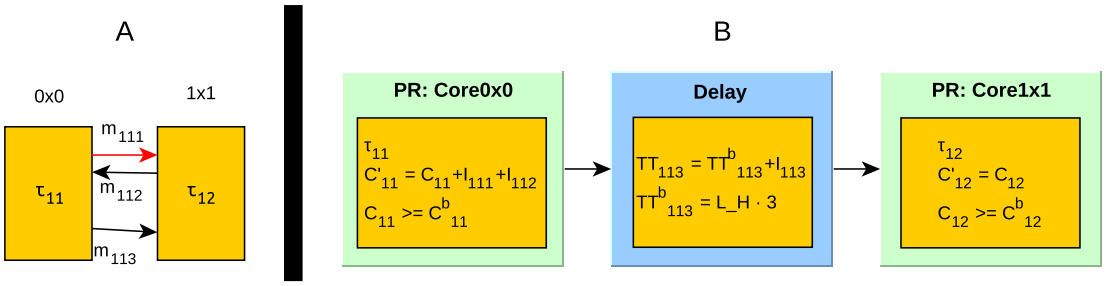

Figura 3.4 (a) Dos tareas mapeadas en dos núcleos diferentes de un procesador muchos núcleos. Una tarea envía un mensaje de lectura y otro de escritura la otra tarea situadas en diferentes recursos de procesamiento. (b) Se muestra como estas tareas son modeladas en MAST.

En la Figura 3.4 se muestra el modelo MAST de dos tareas ejecutando en diferentes núcleos de un procesador muchos núcleos. En la Figura 3.4.a podemos ver que hay dos tareas, τ_{11} ejecutando en el núcleo 0x0 y τ_{12} ejecutando en el núcleo 1x1. La tarea τ_{11} envía un mensaje petición de lectura m_{111}, que recibe el mensaje de de respuesta m_{112}. La tarea τ_{11} acaba después su ejecución enviando un mensaje de escritura m_{113} a la tarea τ_{12}.

En la Figura 3.4.b podemos ver cómo se modelan las tareas de la Figura 3.4.a utilizando elementos MAST:

- La tarea τ_{11} se modela utilizando una actividad que será ejecutada en PR_{00} con la WCET C_{11} y la BCET C^b_{11}. Como la tarea realiza una operación de lectura, el mensaje de petición de lectura m_{111} puede sufrir una interferencia I_{111} por la NoC y el mensaje de respuesta m_{112} puede sufrir una interferencia I_{112}. Esto significa que el escenario de peor caso tiene que incluir todas las interferencias posibles como se formula en C'_{11} en la Figura 3.3.b.

- El tiempo requerido por el último paquete del mensaje m_{113} para viajar desde el núcleo 0x0 hasta el núcleo 0x1 se modela utilizando un bloque de retardo. El intervalo de tiempo mínimo del retardo es el tiempo de mejor caso de travesía de m_{113}, T^b_{113}, calculado utilizando la Ecuación 3.2 con H_{113} igual a tres (el número de enrutadores que atraviesa m_{113} incluyendo el suyo propio). Por otro lado, el intervalo de tiempo máximo del retardo es el tiempo de travesía de peor caso de m_{113}, T_{113}, calculado utilizando la Ecuación 3.5, que incluye la interferencia sufrida por el último paquete de m_{113} debido al resto de paquetes de la NoC.

- La tarea τ_{12} será modelada utilizando una actividad ejecutado en PR_{11}. No es posible tener interferencias de la NoC en esta tarea ya que no envía ningún mensaje por la NoC.

Capítulo 4

Modelado del procesador Epiphany

Para aplicar el modelo descrito en el Capítulo 3 a un sistema real hemos elegido el procesador muchos núcleos Epiphany. Este procesador cumple los requisitos necesarios, es hardware de código abierto y se comercializa una plataforma de desarrollo sobre la que poder desarrollar todo el software que consideremos necesario para la comprobación del modelado. En este capítulo se procede a describir la plataforma de desarrollo *Paralela* que incluye el procesador muchos núcleos Epiphany y, posteriormente, se describe la aplicación del modelo a dicho procesador.

4.1. Procesador muchos núcleos Epiphany

4.1.1. Plataforma Paralela

El procesador Epiphany está integrado en la plataforma de desarrollo Parallella [1] desarrollada por Adapteva que se muestra en la Figura 4.1 por las dos caras. Esta placa es del tamaño de una tarjeta de crédito y tan solo necesita 5W para funcionar. Además del procesador Epiphany, la placa contiene un procesador de doble-núcleo ARM denominado Zynq, 1GB de memoria compartida SDRAM y diversos elementos de entrada salida cuya conectividad se muestra en la Figura 4.2.

El Zynq es el procesador central de la placa de desarrollo Parallella y combina un ARM Cortex-A9 de doble-núcleo con la lógica programable de Xilinx. Como sistema operativo del Zynq se utiliza una adaptación de Ubuntu llamada Parabuntu. El sistema operativo se utiliza para enviar los ejecutables a los diferentes núcleos del procesador Epiphany (eCores) y también se encarga de mandar la señal de comienzo de la ejecución de los eCores.

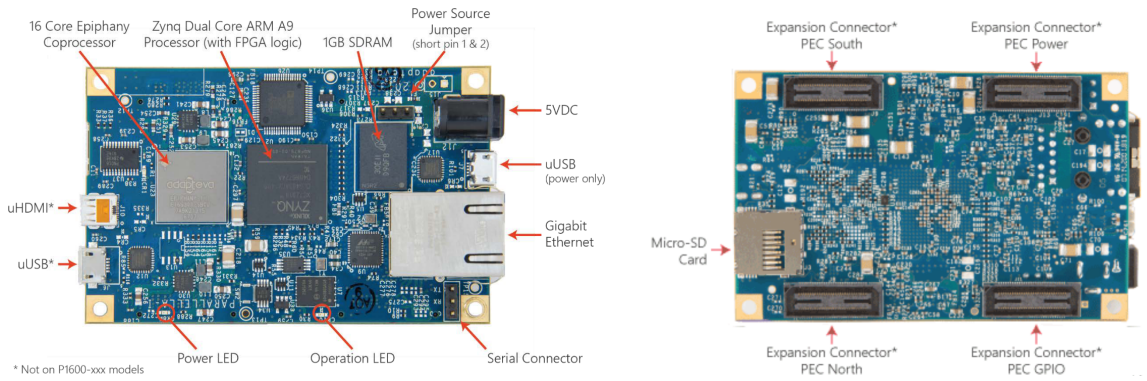

Figura 4.1 Imagen por las dos caras de la placa de desarrollo Parallella. Imágenes obtenidas del manual de referencia de la propia placa [1].

Es a través del procesador Zynq como el procesador muchos núcleos Epiphany puede interactuar con los distintos periféricos, incluida la memoria compartida como podemos ver en la Figura 4.2 donde se muestra el diagrama de conectividad proporcionado por Adapteva.

El sistema operativo Parabuntu que gobernará la placa de desarrollo Parallella tiene que situarse en una tarjeta Micro-SD que se colocará en el slot correspondiente. Es en la misma tarjeta Micro-SD donde tienen que ubicarse los ejecutables que se deseen ejecutar, tanto en el procesador Zynq como en cada núcleo del procesador Epiphany, para poder cargarse allí.

Estas placas ofrecen la oportunidad de conectar varias placas *Parallella* entre sí o con una entrada/salida de propósito general (General Porpoise Input/Output, GPIO) mediante conectores de expansión para crear mallas mayores, pero esta posibilidad no ha sido necesaria para cumplir con los objetivos de esta tesis.

4.1.2. Procesador Epiphany

Es un procesador muchos núcleos diseñado por Adapteva que cuenta con 16 núcleos, que en este procesador se denominan eCore, conectados por una red de interconexión que los distribuye en una malla 2D de dimensiones 4x4 como la mostrada en la Figura 4.3, donde cada cuadrado es una celda cuyo contenido se ve ampliado en la Figura 4.4. Se observa en cada celda un enrutador para que se pueda conectar con las celdas vecinas, la memoria local y un eCore. La conexiones entre enrutadores de celdas vecinas son bidireccionales.

El eCore, cuya arquitectura ha sido diseñada por Adapteva, ejecuta las instrucciones en orden, con una frecuencia de ejecución de 600 MHz. Dicho eCore contiene una

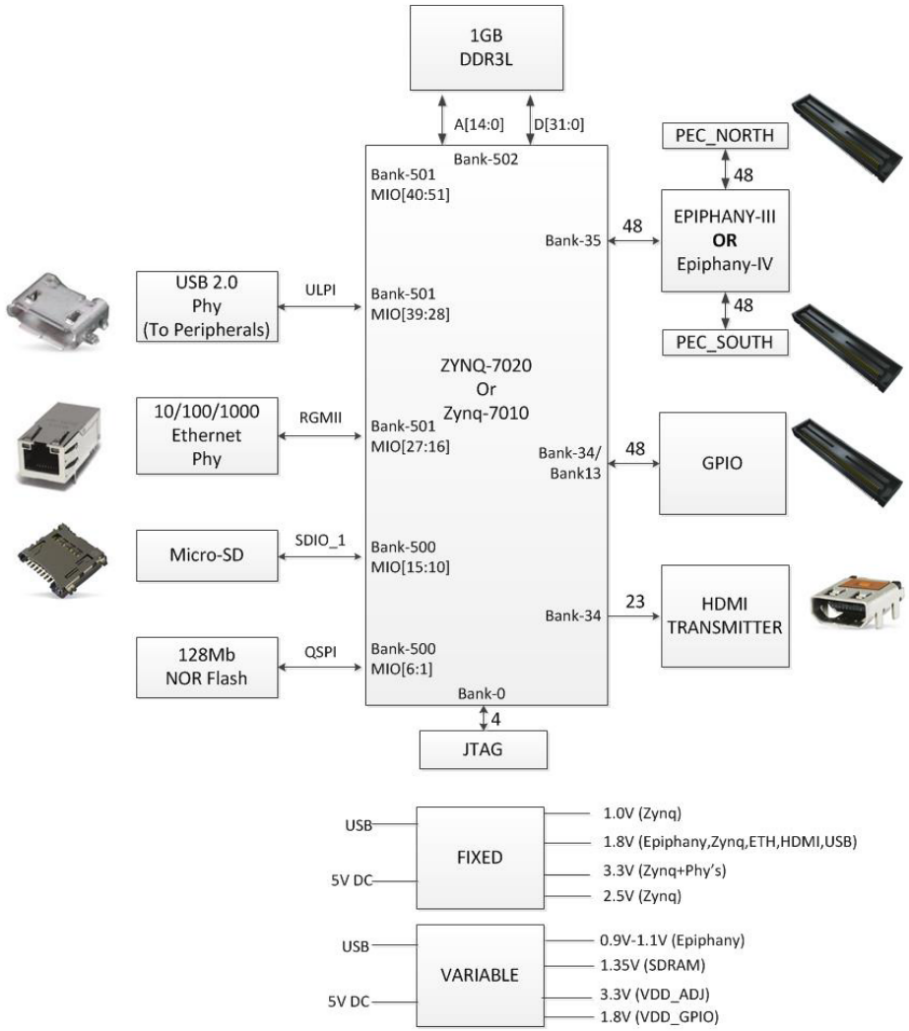

Figura 4.2 Esquema de conectividad de la placa de desarrollo Parallella. Imágenes obtenidas del manual de referencia de la propia placa [1].

ALU de números enteros, una unidad de coma flotante, una unidad de depuración, un controlador de interrupciones, un secuenciador de programa de propósito general y un registro de propósito general de 64-palabras.

Cada eCore tiene dos contadores de 32-bit independientes, la duración de la cuenta completa es de 7,16 segundos aproximadamente. La arquitectura está soportada por el compilador GCC y tiene librerías para OpenMP y MPI. La arquitectura eCore es programable siguiendo los estándares ANSI-C/C++.

Cada celda tiene una memoria local de 32kB. Cualquier eCore puede acceder a la memoria local del resto de eCores del procesador utilizando un rango de direcciones globales especiales. La memoria de un eCore puede ser escrita y leída por otro eCore sin ninguna otra limitación más que el tamaño de la memoria. Los accesos el mismo área

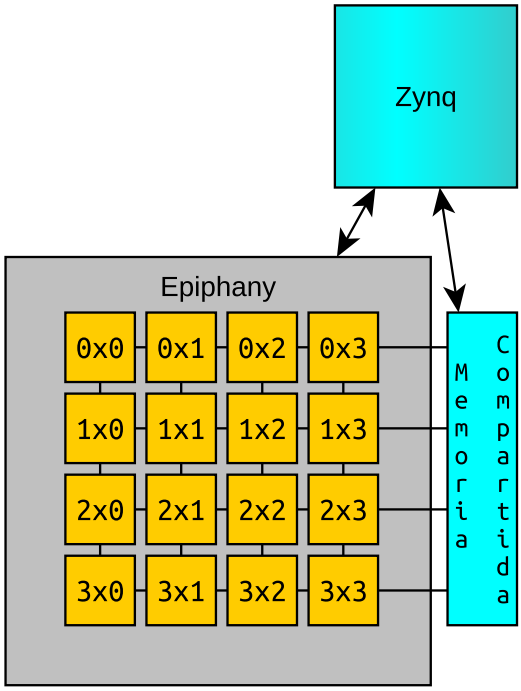

Figura 4.3 Arquitectura de memoria de la plataforma Parallella

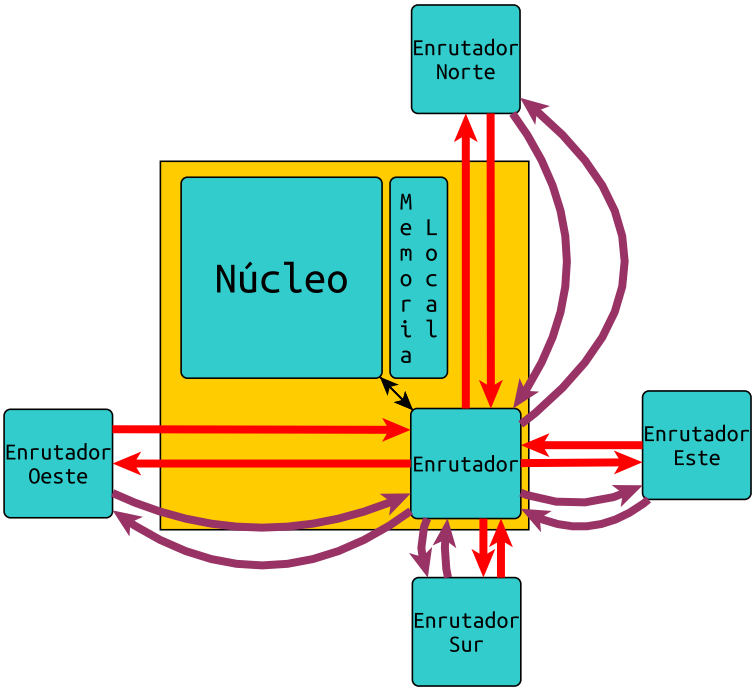

Figura 4.4 Arquitectura de un eCore

de memoria desde diferentes eCores no están sincronizados, aunque el fabricante ofrece una librería que implementa un `mutex` como primitiva de sincronización de acceso mutuamente exclusivo. Lo que sí asegura la red de interconexión del Epiphany es que los paquetes del mensaje llegan ordenados.

La placa de desarrollo Parallella tiene una memoria compartida (ver Figura 4.3) a la que pueden acceder tanto los eCores del procesador Epiphany como el procesador Zynq. El acceso a esta memoria es más lento que el acceso entre memorias locales de los eCores. Hay un orden de magnitud de diferencia (la memoria compartida tarda 468 ns y el acceso remoto a una memoria local 46,67 ns).

El Zynq también puede acceder a la memoria local de los eCores. Pero, de nuevo, es una operación que también requiere mucho más tiempo que el acceso entre eCores.

El diseño es escalable, ya que se ha mostrado una versión del mismo procesador muchos núcleos con 1024 núcleos de procesamiento [59]. Aunque, por desgracia, la versión de Epiphany V no está disponible en ninguna plataforma de desarrollo. Teniendo en cuenta la implementación de la malla del procesador Epiphany, se intuye la limitación del espacio de direcciones pues, dadas las características especiales del direccionamiento de memoria, una arquitectura Epiphany de 32-bit, con un espacio único de direcciones es de 2^{32} bytes, permitiría tener hasta 4.096 celdas mientras que una arquitectura de 64-bit permitiría llegar a los 18000 millones de celdas.

4.1.3. Arquitectura de memoria

La plataforma de desarrollo Parallella tiene un único espacio de direcciones de 2^{32} bytes mapeado en las direcciones entre 0 y hasta $2^{32} - 1$ con un alineamiento a palabras de 4 bytes.

Cada eCore del Epiphany tiene 32kB de memoria local mapeada en el rango de direcciones entre 0x0 y 0x7FFF. Esta dirección también se utilizará para acceder a memoria de otras celdas, aunque añadiendo una parte más significativa que identificará el eCore al que se está accediendo. Estas mismas direcciones extendidas pueden ser utilizadas por el procesador Zynq para acceder a la memoria local de los eCores. Las direcciones extendidas asignadas a la memoria local de cada núcleo que sirven como direcciones de memoria global se pueden ver en el Tabla 4.1.

La memoria local de un eCore está dividida en 4 bancos de 8KB. Durante un ciclo de reloj se pueden realizar cada una de las siguientes operaciones sobre la memoria local:

Dirección externa de memoria local de los eCore

eCore	Dirección inicial (hex)	Dirección final (hex)
0x0	80800000	80807FFF
0x1	80900000	80907FFF
0x2	80A00000	80A07FFF
0x3	80B00000	80B07FFF
1x0	84800000	84807FFF
1x1	84900000	84907FFF
1x2	84A00000	84A07FFF
1x3	84B00000	84B07FFF
2x0	88800000	88807FFF
2x1	88900000	88907FFF
2x2	88A00000	88A07FFF
2x3	88B00000	88B07FFF
3x0	8C800000	8C807FFF
3x1	8C900000	8C907FFF
3x2	8CA00000	8CA07FFF
3x3	8CB00000	8CB07FFF

Tabla 4.1 Mapeado de las diferentes memorias locales de los eCores.

- Operación fetch desde la memoria al secuenciador del programa. Instrucciones de ocho bytes.
- Se pueden transferir ocho bytes de datos entre la memoria local y un registro de la CPU.
- Se pueden escribir ocho bytes en la memoria local desde la interfaz de red. Lo que significa que las escrituras sobre la memoria local de un eCore no afectan el tiempo de ejecución de este.
- Se puede transferir 64 bits desde la memoria local de la red utilizando la memoria de acceso directa (Direct Memory Address, DMA) local.

Las transacciones de lectura y escritura de la memoria local de los núcleos siguen el modelo de ordenación estricta de memoria (*strong memory-order*). Lo cual significa que las transacciones se completan en el mismo orden en el que fueron despachadas por el secuenciador del programa. Sin embargo, cuando las transacciones no acceden a memoria local son más laxas para mejorar el rendimiento. Esto se denomina ordenación débil de lectura y escritura (*weak ordering of load and stores*) y puede implicar que la temporalidad de las operaciones de memoria no sigan la secuencia del código fuente del programa. Por otro lado, en este sentido Epiphany proporciona las siguientes garantías:

- Las operaciones de lectura (load) se competan antes de que el valor retornado sea utilizado en una instrucción subsecuente.
- Las operaciones de lectura utilizarán los valores actualizados que estén secuenciados previamente a la lectura.
- Las operaciones de escritura (store) acabarán por ser propagadas hasta su dirección de destino.

Aunque con estas condiciones las lecturas y escrituras de memoria fuera de las memorias locales del Epiphany tendrán consistencia, al no asegurarse el orden en el que estas operaciones de lecturas y escritura se ejecutarán, esto puede suponer un problema para sistemas de tiempo real estricto. Es por ello que se tiene que tener especial consideración en situaciones en las que el orden no sea determinístico. Estas situaciones son:

- Dos transacciones consecutivas en las que la primera escriba en un núcleo para después leer sobre el mismo núcleo, en una dirección de memoria diferente. Esta situación no influye en los tiempos de respuesta, ya que aunque el orden no sea determinista, se sigue considerando la llegada de la última transacción, sea cual sea.
- Realizar dos transacciones consecutivas en las que se escriba sobre dos núcleos diferentes. Esta situación se evita en esta tesis debido a la utilización de primitivas sincronizadas de escritura y lectura de mensajes.
- Realizar dos transacciones consecutivas en las que se escriba y se lea de dos núcleos diferentes. Esta situación también se evita en esta tesis debido a la utilización de primitivas sincronizadas de escritura y lectura de mensajes.

El procesador Epiphany no ofrece soporte para escrituras o lecturas con direcciones de memoria desalineadas. Para las transacciones se soporta alineación de byte (8 bits), media palabra (16 bits alineados a 2 bytes en memoria), palabra (32 bits alineados a 4 bytes en memoria) y palabra doble (64 bits alineados a 8 bytes en memoria) todo con el alineamiento de bit menos significativo (*least-significant bits*, LSBs). Los diferentes elementos de memoria pueden ser utilizados con diferentes tamaños (que una tarea utilice una variable como entero y otra tarea utiliza esa misma variable como 2 bytes es perfectamente válido).

4.1.4. Funcionamiento de la NoC

La NoC presente en el procesador Epiphany tiene una topología malla 2D (ver Figura 4.3) donde cada celda solo está conectada directamente con las celdas vecinas(ver Figura 4.4), como máximo puede conectarse a una celda por punto cardinal (salvo las celdas de los bordes o esquinas de la malla). Las celdas situadas en los bordes de la malla pueden estar conectadas a módulos de interfaz que se comunican con elementos externos al procesador Epiphany.

La NoC trabaja a la misma frecuencia que los eCores, como consecuencia para la NoC, $F_{NoC} = 600MHz$. A partir de ahora vamos a utilizar el término ciclo para referiros tanto a los ciclos de un núcleo como de la NoC (1 ciclo $= \frac{1}{6{,}0e8}s \simeq 1{,}667ns$).

Existen 3 NoCs ortogonales, cuyo tráfico no se ve influido entre ellas, dentro del procesador Epiphany:

- **cMesh.** Utilizada para las transacciones de escritura cuyo destino es otra celda de la NoC. El arbitrio del enrutado sobre esta NoC puede causar un ciclo de retardo por cada paquete competidor por el mismo enlace ($L_{R_w} = 1ciclo$). El enrutador puede enrutar un paquete de 64 bits por ciclo y enlace de salida.

- **rMesh.** Utilizada para las peticiones de lectura. El arbitrio del enrutado puede causar un retardo de ocho ciclos por cada otro paquete compitiendo por el mismo enlace ($L_{R_r} = 8ciclos$). El enrutador puede enrutar un paquete de 64 bits por ciclo y enlace de salida. Las peticiones de lectura son siempre del tamaño de un paquete.

- **xMesh.** Utilizada para transacciones de escritura destinadas a recursos externos al chip y para que circulen las transacciones dirigidas a otro procesador Epiphany en una configuración multi-chip. Estas transacciones no serán modeladas en esta tesis.

Por lo tanto, puesto que las operaciones de lectura y escritura utilizan dos NoCs ortogonales diferentes, estos dos tipos de tráfico nunca se van a ver afectados entre ellos. Los enlaces bidireccionales de ambas redes se muestran en la Figura 4.4. Las flechas rojas corresponden a los enlaces de la cMesh mientras que las flechas moradas corresponden a los enlaces de la rMesh. Tampoco se verán afectadas las operaciones on-chip por operaciones off-chip.

Cuando varios paquetes situados en los *buffers* de entrada de un enrutador comparten el mismo enlace de salida es preciso decidir entre dichos paquetes para ver cuál poner

en un enlace de salida. Para tomar dicha decisión se utiliza el arbitrio cíclico o *round-robin*, lo que acaba resultando en una división del ancho de banda entre los agentes compitiendo por los recursos compartidos de una forma en la que no se produce el fenómeno de inanición (o *starvation*). Los enlaces de entrada/salida no pueden almacenar más de un paquete, por lo que solo un paquete puede estar esperando en un enlace de entrada a atravesar el enrutador y solo se puede poner un paquete en un enlace de salida.

Todas las NoCs presentes en el procesador Epiphany tienen un sistema de enrutado fijo denominado XY. Con esta política de enrutado el paquete comienza recorriendo la fila X (tomando direcciones Este/Oeste en la malla) hasta llegar a la columna Y, y a partir de ese punto el paquete se desplaza a través de los enrutadores de dicha columna (tomado direcciones Norte/Sur en la malla) hasta alcanzar el enrutador de destino. De esta forma las rutas de los paquetes son fijas y se conocen una vez están determinadas las celdas origen y destino, siendo independiente del tráfico existente.

El tener una política de enrutado fijo XY, junto con tener independizadas las escrituras de las peticiones de lectura y con la política de arbitro *round-robin*, permite garantizar que la red está libre de bloqueos mutuos (o *deadlocks*) para cualquier condición de tráfico existente. Al tener un enrutado fijo se conoce la ruta que van a seguir los paquetes desde el diseño del sistema.

Los enlaces son bidireccionales y los paquetes en un enrutador solo compiten entre sí si comparten el mismo enlace de salida. Se van a exponer a continuación ejemplos sobre cómo diferentes tipos de tráfico se ven afectados en la NoC:

- El tráfico cruzado no produce interferencia alguna. Esto es, el hecho de compartir enrutador, siempre que no se comparta el enlace de salida, no supone ningún tipo de interferencia entre los paquetes. Se muestra en la Figura 4.5.a tráfico entre los eCores 0x1 y 2x1 que no interfiere con el tráfico entre los eCores 1x0 y 1x2.

- El trafico opuesto no produce interferencias. O lo que es lo mismo, utilizar enlaces en direcciones opuestas no genera retardo alguno. Se muestra en la Figura 4.5.a tráfico entre los eCores 0x1 y 2x1 en ambos sentidos, sin interferir entre ellos.

- Compartir el mismo enlace de salida con otro paquete sí puede generar bloqueo cuando se pierda el arbitrio por dicho enlace. Se muestra en la Figura 4.5.b donde el enlace entre las celdas 1x1 y 1x2 es compartido por los mensajes enviados desde las celdas 1x0 y 0x1. En esta situación, el enrutador de la celda 1x1 debe arbitrar el acceso al enlace compartido.

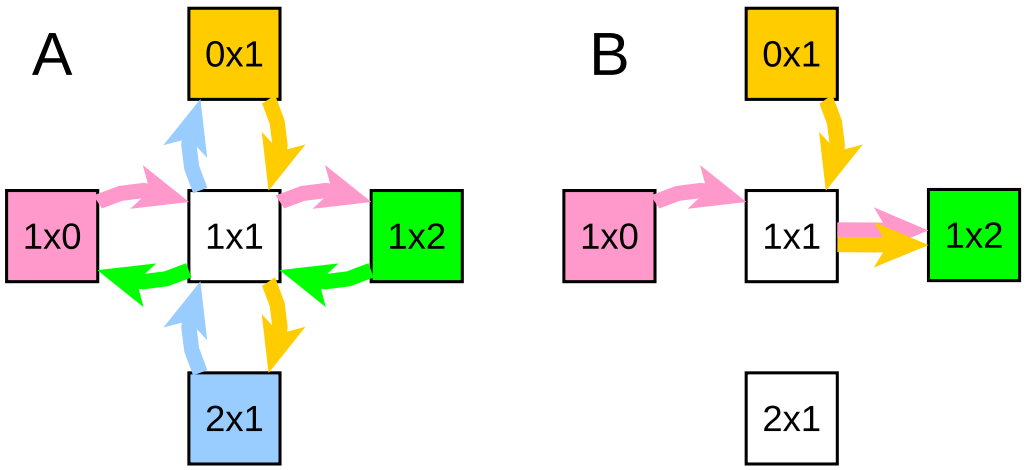

Figura 4.5 Dos situaciones de cuatro eCores realizando envíos de mensajes por la red. (a) En la figura de la izquierda no hay ninguna interferencia mientras que (b) en la figura de la derecha los dos mensajes que hay se interfieren entre ellos.

El número de paquetes que pueden interferir en cada enlace está acotado, pues hay un número máximo de tres enlaces de entrada que pueden contener un paquete con el mismo enlace de salida que el paquete analizado. Por ejemplo, para un paquete llegando a una celda desde el Oeste y queriendo utilizar el enlace de salida situado al Sur, sus posibles paquetes competidores podrían estar situados en el enlace que une el núcleo de la celda con el enrutador y en los enlaces Norte y Este de entrada a la celda.

La travesía entre los enrutadores de la NoC

Independientemente de la red en la que se encuentre un paquete (cMesh o rMesh), este es capaz de atravesar un enrutador pasando de un enlace a otro en 1.5 ciclos ($L_H = 1{,}5 ciclos$).

Este valor de L_H es debido a que un enrutador de una celda del procesador Epiphany siempre se encontrará desfasado medio ciclo con respecto a los enrutadores de las celdas vecinas en la red de interconexión. A nivel hardware, lo que sucede es que unos enrutadores operan sincronizados con el flaco de subida del reloj mientras que otros lo hacen con el flanco de bajada, con la distribución mostrada en la Figura 4.6.

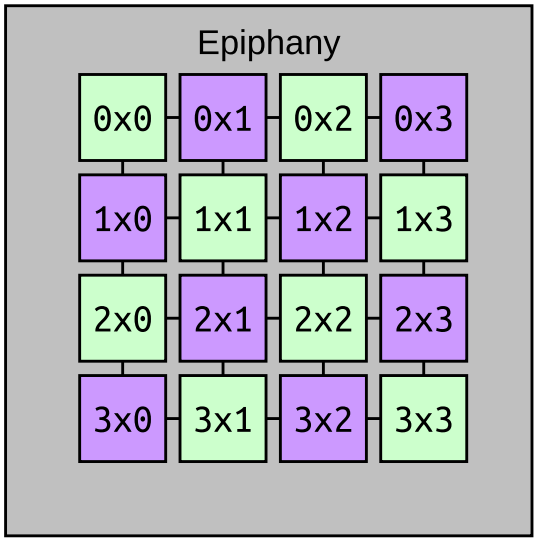

Figura 4.6 Distribuciones de los enrutadores sincronizados con los flancos de subida y bajada del reloj.

Peticiones de lectura

Una petición de lectura se genera cuando un eCore quiere leer el contenido de la memoria local de otro eCore. El mensaje de lectura contiene la dirección remota del dato que se quiere leer y la dirección de memoria local donde escribir el dato leído.

Las peticiones de lectura viajan por la NoC rMesh como ya se ha especificado previamente. En respuesta a la petición de lectura, la NoC genera un mensaje de escritura (que denominaremos mensaje de respuesta) con el dato leído. Todas las peticiones de lectura son de un paquete, y de igual manera, todos los mensajes de respuesta son de un paquete.

Según lo descrito anteriormente, la lectura desde la celda XY de un dato almacenado en la memoria local de la celda X'Y' generará una petición de lectura por cada 64bits (el tamaño de un paquete) del dato a leer. Cada petición de lectura irá de la celda XY hasta la celda X'Y' generará un mensaje de respuesta de un paquete de tamaño 64 bits desde la celda X'Y' hasta la celda XY. En la Figura 4.7 vemos un ejemplo de petición de lectura de un paquete de la celda 00 a la celda 33 (rojo) utilizando la NoC rMesh y la escritura del paquete que se genera como respuesta desde la celda 33 a la celda 00 (azul) utilizando la NoC cMesh.

En esta arquitectura desde que llega una petición de lectura de otro eCore, hasta que se retorna el valor solicitado por la red, las interrupciones son deshabilitas en el

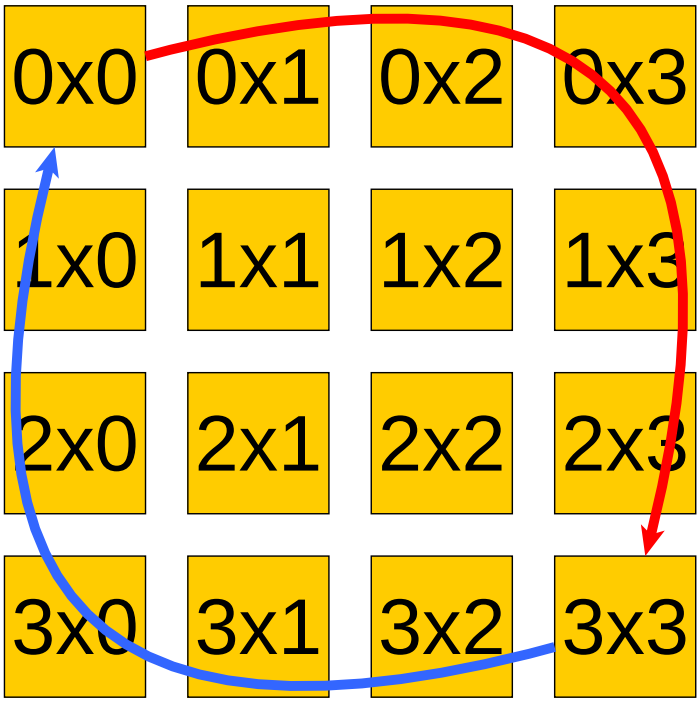

Figura 4.7 Travesía que realiza por la red una petición de lectura de la celda 00 a la celda 33 (rojo) y el paquete de respuesta (azul).

eCore que se está leyendo. Además, el eCore lector no avanza a la siguiente instrucción hasta que no le llegue el valor solicitado.

4.1.5. Memoria compartida

Como se puede ver en la Figura 4.3. La memoria compartida se sitúa en la placa *Parallella* fuera del procesador Epiphany. Concretamente en el lateral Este de la malla que forma la NoC del Epiphany. En caso de la placa de desarrollo *Parallella* tiene una capacidad de 1GB de memoria. Para poder utilizar este espacio de memoria compartida el procesador Zynq tiene que reservar un área dentro de este espacio de memoria al que se le asigna una etiqueta y es a través de esa etiqueta como se puede acceder a dicha memoria compartida.

Para comprobar el funcionamiento de la memoria compartida y poder decidir sobre lo adecuada que sería su utilización en un sistema de tiempo real estricto se realizó una serie de tests que consistían en leer y escribir de manera aislada una zona de la memoria compartida por programas situados en cada uno de los eCores del procesador Epiphany.

Durante el desarrollo de los tests se experimentó cómo en la memoria compartida no se garantizan valores correctos cuando se realiza una escritura tras lectura, Write-After-Read (WAR), al hacer lecturas y escrituras de manera consecutiva desde el mismo eCore.

El test consistió en realizar 5.000 incrementos sobre un espacio de memoria con cuatro ciclos extra de espera entre incrementos, para evitar el WAR. Los resultados temporales de los 5.000 incrementos se pueden observar en la Figura 4.8. Podemos ver que dicha cantidad de incrementos, evitando el WAR, tarda más de 2.339 ms desde cualquiera de los eCores del Epiphany. Esto supone 468 ns por incremento desde el eCore en el que se realiza la operación más rápido, siendo de 489 ns el tiempo más lento de esta operación. En la Figura 4.8 se aprecia claramente el efecto debido a que la memoria compartida está en el lado Este del procesador Epiphany ya que los tiempos se reducen a medida que nos acercamos a la columna 3.

Podemos deducir que no hay un gran impacto debido a la red presente dentro del procesador Epiphany, puesto que alejarse dentro de la red de interconexión del Epiphany apenas incrementa el tiempo necesario para acceder a memoria compartida. Pero sí que resulta costoso a nivel temporal salir del procesador para acceder a la memoria compartida.

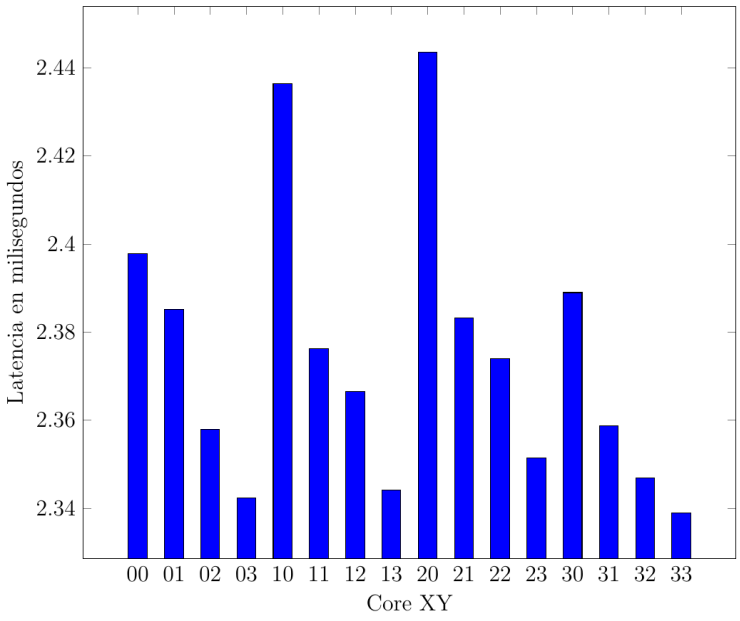

Figura 4.8 Tiempo de acceso a la memoria compartida desde los distintos núcleos

4.1.6. Spinlock para la exclusión mutua

El procesador Epiphany incluye un mecanismo de sincronización de exclusión mutua denominado `mutex`. El `mutex` es un tipo de espera activa (*spinlock*) que permite sincronizar dos o más eCores: un eCore tratando de tomar un `mutex` que ya esté tomado permanecerá parado (en *stall*) hasta que el `mutex` se libere. Un `mutex` se implementa utilizando un número entero localizado en la memoria local de uno de los eCores al que se accederá utilizando las operaciones del `mutex` facilitadas por la librería de herramientas del hardware Epiphany (Epiphany Hardware Utility Library, eLib). La atomicidad de las operaciones de tomar y liberar un `mutex` está garantizada por el hardware de control de la NoC.

Para intentar tomar un `mutex` un eCore genera un mensaje petición de lectura especial que lee el valor del *mutex* y, en caso de que el valor leído sea un cero, lo cambia de manera atómica, utilizando la instrucción de ensamblador `testset`, por el valor del identificador del eCore que está ejecutando la operación de tomar el `mutex`, consiguiendo entonces así tomarlo. Que el valor leído sea diferente de cero significa que el `mutex` ya ha sido tomado por otro eCore. En ese caso el núcleo ejecutando la operación de lectura se queda detenido en espera de que el `mutex` se libere. Una vez que se consigue realizar la petición de lectura especial se genera un mensaje de respuesta para escribir el valor del mutex en la memoria local del eCore que realizó la llamada. En cuanto se obtiene el valor retornado la aplicación que realizó la petición de tomar el `mutex` lo considera tomado por ella misma y puede continuar con su ejecución. En resumen, tomar el `mutex` es una operación bloqueante. Durante el tiempo que la aplicación está bloqueada el *eCore* está parado y, por ende, este tiempo de bloqueo se modelará como tiempo de ejecución ya que el procesador no puede ejecutar ninguna otra instrucción.

La operación de liberar el `mutex` consiste simplemente en escribir un cero en el `mutex`. Como consecuencia, solo requiere escribir un mensaje desde el eCore, liberando así el `mutex`.

La librería eLib proporciona la siguiente interfaz para su utilización:

- `e_mutex_init`. En realidad esta función no hace nada. El requisito de inicializar un `mutex` es que tenga su valor a 0.
- `e_mutex_lock`. Este procedimiento recibe como parámetro de entrada la fila y la columna del núcleo en el que está situado el `mutex` y la dirección del entero que implementa el `mutex`.

- `e_mutex_unlock`. Recibe la fila y columna del eCore en el que está almacenado el `mutex` así como de la dirección del entero que implementa el `mutex`. Este procedimiento escribe el valor cero en la dirección del `mutex`, quedando de esa manera el `mutex` liberado. En el procedimiento de la librería no se hace ninguna comprobación de que el mismo eCore que lo ha bloqueado sea el que lo desbloquee.

4.1.7. Relojes del Epiphany

Desafortunadamente, el procesador muchos núcleos Epiphany no dispone de ningún reloj o temporizador global común a todas las celdas que pueda servir como mecanismo de sincronización temporal entre los núcleos del procesador.

Cada eCore del procesador Epiphany incluye dos temporizadores con un registro de cuenta de 32 bits. Ambos temporizadores pueden ser programados para generar una interrupción cuando su cuenta llegue a cero.

Para iniciar la cuenta del temporizador, se tiene que escribir un número entero en un registro del eCore. Este número será el valor inicial desde el que comenzará la cuenta regresiva una vez se le de la instrucción de inicio de cuenta.

Tanto la operación sobre el eCore para poner el entero inicial como la instrucción de arranque se tienen que realizar de manera local, por lo que los temporizadores no estarán sincronizados per se si no que requerirán de un mecanismo de sincronización explícito entre núcleos en caso de que sea deseable.

Ambos temporizadores locales son independientes entre sí, pudiendo estar en funcionamiento o desactivados de manera independiente además de poder tener diferentes valores de cuenta en cada uno de ellos.

La librería eLib proporciona la siguiente interfaz para su utilización:

- `e_ctimer_get` función que mediante el parámetro de entrada del id del temporizador, retorna el valor del entero que contiene.

- `e_ctimer_set` función que mediante el parámetro de entrada del id del temporizador y un valor entero, asigna dicho valor al temporizador.

- `e_ctimer_start` función que mediante el parámetro de entrada del id del temporizador, inicia la cuenta regresiva del temporizador.

- `e_ctimer_stop` función que mediante el parámetro de entrada del id del temporizador, detiene la cuenta regresiva del temporizador.

- `e_wait` función que mediante el parámetro de entrada del id del temporizador y un número entero de ciclos del reloj realiza una espera activa de esa cantidad de ciclos de temporizador.

4.2. Aplicación del modelo al procesador Epiphany

4.2.1. Parámetros básicos de la red de la NoC

Como se explicó en la Sección 3.2, en nuestro modelo la red se caracteriza por tres parámetros: la frecuencia de la NoC (F_{NoC}), la latencia del salto (L_H) y la latencia de arbitrio del enrutado (L_{R_w} para mensajes de escritura o respuesta y L_{R_r} en caso de mensajes de petición de lectura). Según lo descrito en la Sección 4.1.4, los valores para estos parámetros en el caso de la NoC del Epiphany son $F_{NoC} = 600MHz$, $L_H = 1{,}5ciclos$, $L_{R_r} = 8ciclos$ y $L_{R_w} = 1ciclos$.

4.2.2. Modelado de los mensajes

Como se describió en la Sección 4.1.4 los núcleos del procesador Epiphany pueden comunicarse escribiendo y leyendo la memoria local de un núcleo remoto, lo que provoca dos tipos de operaciones en la NoC. Una operación de escritura consiste en un mensaje de escritura compuesto por uno o varios paquetes que viajan a través de la red cMesh desde la celda que contiene el núcleo que realiza la escritura hasta la celda de destino. Por otro lado, un mensaje de lectura requiere la lectura de uno o varios paquetes. Cada operación de lectura de un paquete requiere dos paquetes: una petición de lectura que viaja a través de la red rMesh desde la celda que contiene el núcleo que realiza la lectura hasta la celda de destino y un paquete de respuesta que retorna por la red cMesh desde la celda que contiene el dato hasta la celda que realizó la petición de lectura. Tanto el paquete de petición de lectura como el de respuesta son de 64 bits. Cuando una tarea de un núcleo quiere leer una sección de memoria externa mayor de 64 bits, la operación se divide en varias lecturas de 64 bits (cada una de ellas con su correspondiente paquete de respuesta de 64 bits) que son ejecutadas secuencialmente (el mensaje de lectura m_{ijk} no será generado hasta que el mensaje de respuesta $m_{ij(k-1)}$ correspondiente a la petición anterior no haya llegado).

A partir del comportamiento anteriormente explicado se puede deducir que, para operaciones de lecturas mayores de un paquete, la ratio de generación de paquetes de peticiones de lectura es el inverso del número de ciclos (c) entre la generación de dos peticiones de lectura consecutivas y el tiempo de travesía de mejor caso del paquete

de petición de lectura y del paquete de respuesta correspondiente (Ecuación 4.1). De la misma manera, la Ecuación 4.2 puede ser utilizada para el cálculo de la ratio de generación de los paquetes de respuesta correspondientes a una operación de lectura de más de un paquete.

$$\forall m_{ijk} : t_{ijk} = read, \rho_{ijk} = \frac{1}{TT^b_{ijk} + TT^b_{ij(k+1)} + c} \tag{4.1}$$

$$\forall m_{ijk} : t_{ijk} = respuesta, \rho_{ijk} = \frac{1}{TT^b_{ijk} + TT^b_{ij(k-1)} + c} \tag{4.2}$$

Respecto a los mensajes de escritura, el compilador transforma las operaciones de más de un paquete en llamadas a la función estándar de C `memcopy`. Analizando el código ensamblador de la función `memcopy` hemos visto que es capaz de generar un paquete en la NoC por cada tres ciclos transcurridos. La forma más rápida de escribir en memoria es mediante asignaciones consecutivas sobre variables. Analizando el código ensamblador generado con la asignación sobre variables de manera consecutiva hemos visto que genera un paquete en la NoC por cada dos ciclos transcurridos. Esta ratio de generación de paquetes de $\frac{1}{2}$ es la mayor que se puede conseguir sobre la cMesh utilizando código de usuario estándar.

Como ya se ha mencionado previamente, el procesador Epiphany utiliza enrutado XY, lo que significa que todo paquete tomará siempre la misma ruta una y otra vez desde el mismo origen y destino, independientemente del tráfico de la red de interconexión. Así que un mensaje m_{ijk} entre los núcleos PR_{xy} y $PR_{x'y'}$ realiza un número determinista de saltos obtenidos mediante la Ecuación 4.3 (valor que coincide con el número de enrutadores atravesado, incluyendo los enrutadores de los núcleos de origen y destino)

$$H_{ijk} = |x - x'| + |y - y'| + 1 \tag{4.3}$$

4.2.3. Restricción de ratio de paquetes máxima

Como las redes del procesador Epiphany son ortogonales, la restricción de ratio máxima se calcula para cada una de ellas de forma independiente. Esto es, cuando la Ecuación 3.3 se aplica a un enlace de la red cMesh solo se consideran los mensajes de escritura (y respuesta). De la misma manera, solo se consideran los mensajes de lectura cuando la Ecuación 3.3 se aplica a un enlace en la red rMesh. Para verificar la restricción de ratio de paquetes máxima todos los enlaces de la red cMesh tienen que

tener una ratio de transmisión acumulada no superior a $\frac{1}{L_{R_w}}$ y todos los enlaces de la red rMesh deben tener una ratio de transmisión acumulada no superior a $\frac{1}{L_{R_r}}$.

Para mostrar que esta limitación no es excesivamente restrictiva en el procesador muchos núcleos Epiphany, vamos a considerar una situación de ejemplo con las siguientes características:

- Por simplicidad solo consideramos mensajes de escritura (es el tipo de mensajes más utilizado por aplicaciones).
- Todos los mensajes se generan con una ratio de $\frac{1}{3}$ utilizando la función estándar de C `memcopy`.
- Los mensajes, de media, atraviesan tres enrutadores de la NoC antes de alcanzar su destino.

Con estas características, y teniendo en cuenta que la malla 4x4 de Epiphany tiene cuarenta y ocho enlaces (tres veces el número de eCores), podrían existir aplicaciones con hasta cuarenta y ocho mensajes entre núcleos sin que se rompieran la limitación de ratio máxima, o lo que es lo mismo, sin que ningún enlace estuviera utilizado por más de tres mensajes.

4.2.4. Tiempos máximos de travesía de paquetes

Como consecuencia del arbitrio *round-robin* del procesador muchos núcleos Epiphany el mayor retardo que puede sufrir un paquete sucede cuando está esperando por causa del resto de enlaces de entrada del enrutador que comparten su mismo enlace de salida. Es decir, tiene que esperar a tres paquetes, como máximo máximo como se vio en la Sección 4.1.4.

Para calcular el tiempo de travesía de peor caso, se utiliza la Ecuación 3.5 con las siguientes variables:

- $L_H = 1{,}5$ ciclos, la latencia por salto del procesador muchos núcleos Epiphany.
- H_{ijk} el número de enrutadores que tiene que atravesar el mensaje entre el núcleo de origen y el de destino obtenido mediante la Ecuación 4.3.
- I_{ijk} la interferencia del paquete calculada con la Ecuación 3.4. Para el procesador Epiphany se convierte en la Ecuación 4.4 si se trata de una petición de lectura ($L_{R_r} = 8 ciclos$) o en la Ecuación 4.5 si se trata de un paquete de escritura o respuesta ($L_{R_w} = 1 ciclo$).

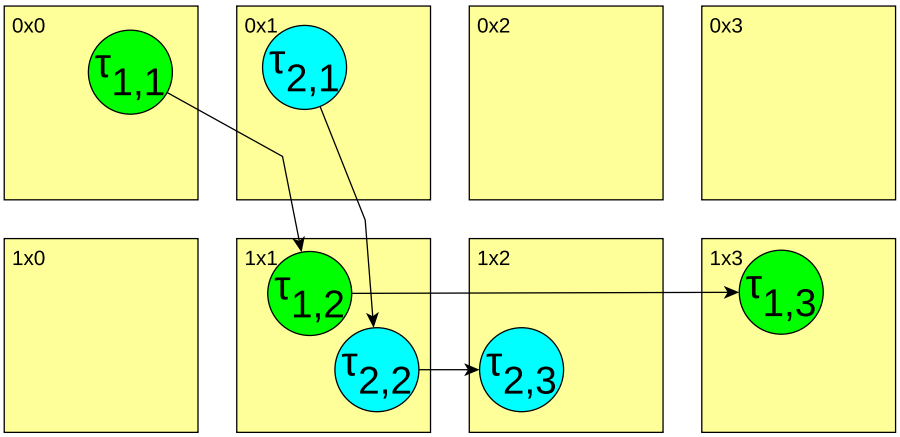

Figura 4.9 Sistema de ejemplo formado por dos flujos de principio a fin con tres tareas cada uno.

$$I_{ijk} = \sum_{r_{xy} \in route_{ij}} nb_{ijkxy} \cdot 8 \tag{4.4}$$

$$I_{ijk} = \sum_{r_{xy} \in route_{ij}} nb_{ijkxy} \cdot 1 \tag{4.5}$$

4.3. Ejemplo simple de modelado MAST sobre Epiphany

Esta sección nos guiará en el proceso de modelado mediante el sistema de ejemplo mostrado en la Figura 4.9, formado por dos flujos e2e con tres tareas cada uno. Las tareas se planifican bajo una política de planificación de prioridades fijas no expulsora con el objetivo de tener un comportamiento lo más parecido posible al que tendrían en M2OS-mc, que será el sistema operativo utilizado en nuestra implementación, aunque el modelo permite especificar también políticas expulsoras. Todos los mensajes del sistema son de escritura al ser esta la forma más sencilla de implementar flujos *e2e* en el procesador muchos núcleos Epiphany. Los parámetros del sistema que se va a utilizar en este ejemplo son los mostrados en la Tabla 4.2. Las ratios de todos los mensajes son $\frac{1}{3} ciclos^{-1}$ puesto que suponemos que están siendo generados con la función estándar de C `memcopy`, que como se ha comentado anteriormente, es capaz de generar un paquete por cada tres ciclos.

Γ_1	$(T_1 = 50\mu s \quad D_1 = 50\mu s)$
τ_{11}	$C_{11} = 5\mu s \quad C^b_{11} = 4\mu s \quad Prio_{11} = 3$
m_{111}	$\mu_{111} = 2 \quad \rho_{111} = 0{,}33 ciclos^{-1}$
τ_{12}	$C_{12} = 3\mu s \quad C^b_{12} = 2\mu s \quad Prio_{12} = 3$
m_{121}	$\mu_{121} = 1 \quad \rho_{121} = 0{,}33 ciclos^{-1}$
τ_{13}	$C_{13} = 7\mu s \quad C^b_{12} = 6\mu s \quad Prio_{13} = 3$

Γ_2	$(T_1 = 160\mu s \quad D_1 = 160\mu s)$
τ_{21}	$C_{21} = 13\mu s \quad C^b_{21} = 12\mu s \quad Prio_{21} = 2$
m_{211}	$\mu_{211} = 4 \quad \rho_{211} = 0{,}33 ciclos^{-1}$
τ_{22}	$C_{22} = 11\mu s \quad C^b_{22} = 10\mu s \quad Prio_{22} = 2$
m_{121}	$\mu_{121} = 5 \quad \rho_{121} = 0{,}33 ciclos^{-1}$
τ_{23}	$C_{23} = 17\mu s \quad C^b_{23} = 16\mu s \quad Prio_{23} = 2$

Tabla 4.2 Parámetros del sistema de ejemplo.

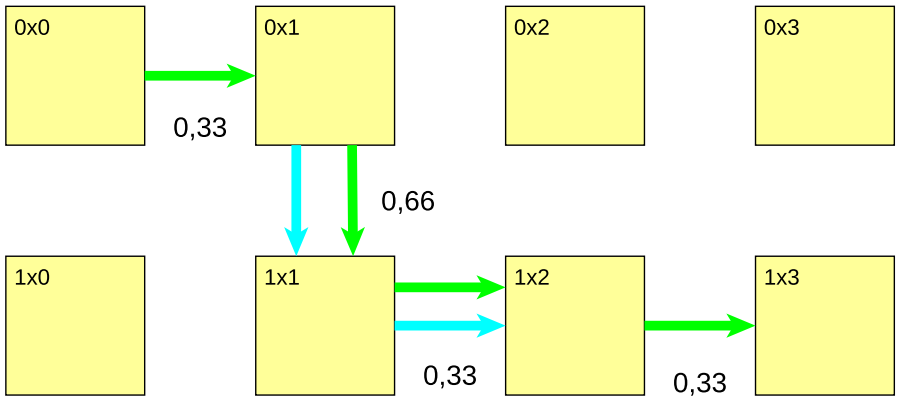

Figura 4.10 Ratio de transmisión acumulada para los enlaces del sistema de ejemplo (unidades en $ciclos^{-1}$).

Antes de ser capaces de ejecutar ningún análisis de MAST necesitamos saber si el sistema cumple con la limitación de ratio máxima ($P_{xy \to x'y'} \leq \frac{1}{L_{R_W}}$, para todo enlace de la NoC). Los resultados del estudio sobre la ratio se muestran en la Figura 4.10, que solo muestra los enlaces con un ratio acumulada superior a cero.

En la Figura 4.10, las ratios de los enlaces que son atravesados por un solo mensaje (con solo una flecha) tienen una ratio acumulada de $0{,}33 ciclos^{-1}$. El enlace desde la celda 0x1 hasta la celda 1x1 tiene una ratio de $0{,}66 ciclos^{-1}$ ya que se utiliza por dos mensajes. En cambio, aunque el enlace entre las celdas 1x1 y 1x2 también es utilizado por dos mensajes, su ratio acumulada es de $0{,}33 ciclos^{-1}$ porque ambos mensajes son generados por tareas del mismo núcleo y, como consecuencia, nunca podrán producirse simultáneamente. Por consiguiente se verifica la limitación de ratio máxima puesto que todos los enlaces de la NoC tienen una ratio acumulada menor que $\frac{1}{L_{R_w}} = 1 cycle^{-1}$.

Para convertir un sistema muchos núcleos en un modelo MAST vamos a enfocarnos en un flujo e2e cada vez. Para cada tarea en el flujo e2e realizamos el siguiente bucle en un orden secuencial (empezando por la primera tarea del flujo, siguiendo el orden del flujo y terminando en el última tarea).

La conversión realizada para cada una de las tareas es:

1. La tarea se transforma en una actividad de MAST incluyendo el tiempo de generación de los mensajes como parte del tiempo de ejecución de la actividad. Esta actividad se ejecuta en el recurso de procesamiento que representa al núcleo en el que está mapeada la tarea.

2. Si hay otra tarea siguiendo el flujo e2e, y esa tarea está en otro PR, se crea un bloque de retardo para modelar el tiempo de travesía del mensaje a través de la NoC. El tiempo de travesía de mejor y de peor caso asignado al retardo se calcula utilizando la Ecuación 3.2 y la Ecuación 3.5 respectivamente.

3. Si no hay ninguna tarea siguiente el flujo e2e esto significa que el modelado del flujo ha terminado y no se requiere ninguna acción más para el flujo e2e.

Una vez generado el fichero MAST del modelo del sistema, se le puede aplicar cualquiera de las herramientas de análisis basadas en *offset*.

La transacción modelada para Γ_1 se muestra en la Figura 4.11 y la transacción modelada para Γ_2 se muestra en la Figura 4.12. En ambas figuras podemos ver los eventos y actividades que modelan ambos flujos. Los eventos internos que conectan cada actividad o retardo con el siguiente se representan con círculos.

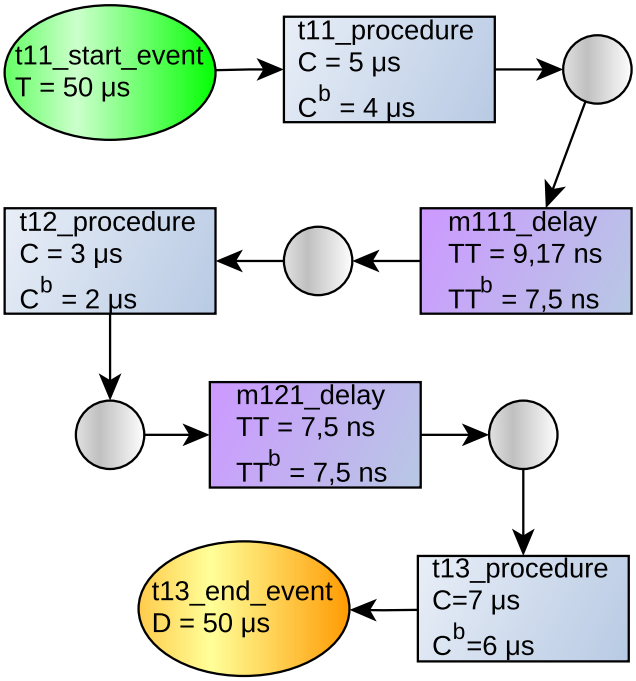

Figura 4.11 Modelo MAST para Γ_1.

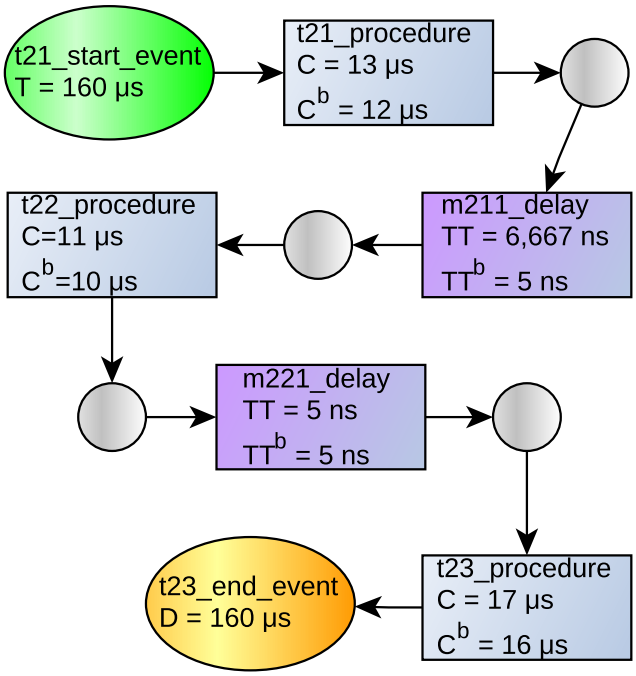

Figura 4.12 Modelo MAST para Γ_2.

El primer elemento de las transacciones modeladas son los eventos que activarán la primera actividad de cada flujo e2e y tienen el periodo de generación correspondiente a cada flujo.

Las actividades, que modelan las tareas, se caracterizan por su tiempo de ejecución de peor caso medido o estimado C además de por el tiempo de ejecución de mejor caso C^b de la ejecución de la tarea. Estos tiempos incluyen el tiempo necesario para la generación del mensaje (si lo hubiera).

El tiempo de travesía de mejor caso utilizado en los bloques de retardo se calcula mediante la Ecuación 3.2. Por ejemplo, en caso del mensaje m_{111} que atraviesa tres enrutadores, este valor se obtiene de la siguiente manera:

$$TT^b_{111} = 1{,}5cycles \cdot 3 = 4{,}5cycles = 7{,}5ns$$

La Ecuación 4.5 se utiliza para calcular la interferencia sufrida por cada paquete de un mensaje. Por ejemplo, el mensaje m_{111} tiene un mensaje competidor, m_{211}, con el que comparte el enlace de salida al Sur del enrutador de la celda 0x1. Como consecuencia el término de interferencia de m_{111} toma el valor:

$$I_{111} = nb_{111_{01}} * L_{R_w} = 1 \cdot 1cycle = 1cycle = 1{,}67ns$$

De acuerdo con la Ecuación 3.5, el tiempo de travesía de peor caso del bloque de retardo utilizado para modelar el mensaje m_{111} se calcula como:

$$TT_{111} = TT^b_{111} + I_{111} = 5{,}5cycles = 9{,}17ns$$

El resto de los cálculos son muy parecios o no tienen interferencias que considerar a la hora de calcular el bloque de retardo.

Los diferentes eventos concatenan las actividades y retardos hasta que se alcanza el último evento. El tiempo de respuesta de peor caso del evento final se compara con el plazo de la última actividad.

Los resultados de la ejecución del análisis de MAST basado en *offsets* para ambos flujos e2e se muestran en la Tabla 4.3. El último evento de ambos flujos tiene un tiempo de respuesta de peor caso menor que el plazo del flujo correspondiente, lo que significa que el sistema es planificable.

Flujo Γ_1 ($T_1 = 50\mu s \quad D_1 = 50\mu s$)		
t1_event_msg	$R^b = 4000ns$	$R = 5000ns$
t2_event_msg	$R^b = 6009ns$	$R = 19010ns$
t3_event_2	$R^b = 12017ns$	$R = 26019ns$
Flujo Γ_2 ($T_2 = 160\mu s \quad D_2 = 160\mu s$)		
t4_event_msg	$R^b = 12000ns$	$R = 13000ns$
t5_event_msg	$R^b = 22005ns$	$R = 27007ns$
t6_event_2	$R^b = 38010ns$	$R = 44012ns$

Tabla 4.3 Resultados del análisis de MAST.

Capítulo 5

Sistema operativo M2OS-mc

En este capítulo presentamos la adaptación del sistema operativo de tiempo real (Real-Time Operating System, RTOS) M2OS para procesadores muchos núcleos basados en malla. Este RTOS, llamado M2OS-mc [17], ha sido implementado para el procesador Epiphany III. Vamos a entrar ahora en más en detalles sobre la implementación de M2OS-mc.

5.1. Descripción general de M2OS

El sistema operativo de tiempo real M2OS [60, 61] es un sistema operativo pequeño y eficiente que permite la ejecución de aplicaciones multi-tarea en pequeños micro-controladores que sufran escasez de recursos de memoria. M2OS se distribuye como software libre bajo licencia GPL y tanto la información como el código del proyecto son accesibles a través de su página web [1].

M2OS implementa una política de planificación no expulsora en la que las tareas se comportan como "tareas de un solo disparo". Cada tarea de un solo disparo está compuesta de dos bloques de instrucciones: las instrucciones de inicialización y las instrucciones del cuerpo de la tarea. Cuando una tarea se ejecuta por primera vez, ejecuta sus instrucciones de inicialización seguidas de las instrucciones correspondientes a su cuerpo, en las siguientes ejecuciones solo se ejecutan las instrucciones del cuerpo de la tarea. Las tareas de un solo disparo no guardan ningún estado local en la pila entre ejecuciones. De esta manera, el área de la pila de una tarea que está esperando a su siguiente ejecución puede ser reutilizada por otras tareas.

[1] http://m2os.unican.es

La combinación de la política no expulsora junto con el uso de tareas de un solo disparo hace posible que todas las tareas puedan compartir el área de memoria utilizado como pila. En consecuencia, el sistema operativo solo necesita reservar el espacio suficiente para la pila más grande, con el consiguiente ahorro me memoria que esto supone.

El código de M2OS está mayormente escrito en Ada, con pequeñas partes en C o ensamblador, especialmente en los componentes del sistema operativo más directamente relacionados con el hardware.

El RTOS M2OS se ha diseñado para poder ser portado a diferentes plataformas requiriendo el mínimo número de cambios posible. Por este motivo los elementos dependientes del hardware se han encapsulado en la capa de abstracción de hardware (Hardware Abstraction Layer o HAL) que constituye la única parte que debe ser modificada para adaptar M2OS a las distintas plataformas hardware.

El núcleo de M2OS es la base del sistema de tiempo de ejecución (Run-Time System, RTS) para el compilador GNAT (GNU Ada Translator) [62] que suporta la planificación no expulsora de tareas de un solo disparo con las mismas restricciones que en el perfil Ravenscar [63].

M2OS proporciona una interfaz de aplicación (Application Programming Interface o API) propia para la creación de tareas de un solo disparo. Además, dispone de una herramienta que permite transformar código Ada que utiliza tareas estándar Ada en código que realiza las llamadas apropiadas la API de M2OS.

En la Figura 5.1 mostramos la arquitectura del sistema operativo de tiempo real M2OS. Se muestran las diferentes capas de lo componen (del más alto nivel a más bajo nivel):

- El código de usuario de la aplicación Ada.
- La librería de tiempo de ejecución de GNAT..
- El pequeño y eficiente núcleo de M2OS. El cual ha sido adaptado en esta tesis a sistemas muchos núcleos basados en malla, con el nombre M2OS-mc.
- La interfaz abstracta con el hardware que encapsula las operaciones relacionadas con la plataforma de ejecución. Para su utilización en este trabajo se ha tenido que adaptar al procesador Epiphany.
- La plataforma de ejecución. En caso de M2OS-mc es un solo eCore, puesto que en cada eCore de Epiphany hay una instancia independiente de M2OS-mc.

Figura 5.1 Arquitectura de M2OS.

Hablaremos ahora específicamente de la versión adaptada para el procesador muchos núcleos Epiphany.

5.2. Compilación, enlazado y carga de las aplicaciones

El primer paso en la adaptación de un sistema operativo a una nueva plataforma consiste en disponer de la batería de herramientas que permita el desarrollo cruzado de las aplicaciones. Como mínimo, estas herramientas incluyen, por un lado, el compilador y enlazador que permite generar las aplicaciones y, por otro, el cargador que permite cargar las aplicaciones en la plataforma e iniciar su ejecución.

La arquitectura de los eCore de Epiphany está soportada directamente en el *GNU compiler collection (gcc)* y el fabricante proporciona una versión binaria del compilador cruzado para Linux y con soporte para el lenguaje C. Dado que M2OS está escrito en Ada, ha sido necesario compilar una versión cruzada de gcc para Epiphany con soporte para los lenguajes C, C++ y Ada (e-gcc v7.4.0). Dicha versión se ofrece en formato binario en la web de M2OS [2].

[2]https://m2os.unican.es/wp-content/uploads/epiphany-gcc-7.4.0.zip

El entorno de desarrollo utilizado es *gnatstudio* y la construcción de las aplicaciones está basada en el uso de ficheros de proyecto del gestor de proyectos de GNAT (GNAT's Project Manager, GPR). El Listado A.1 del Apéndice A, muestra el fichero de proyecto utilizado para construir aplicaciones que ejecuten sobre M2OS-mc.

El guion de enlazado (*linker script*) utilizado por M2OS-mc sitúa todo el código y los datos de la aplicación en la memoria local de los eCores evitando el uso de la memoria compartida proporcionada por la plataforma Parallella. Se genera un fichero ejecutable por cada uno de los eCores. Cada fichero incluye tanto el RTOS M2OS-mc como el código de la aplicación.

El procesador Zynq de la placa Parallella es el encargado de cargar cada ejecutable en el eCore que le corresponda y de iniciar su ejecución. Se ha generado un conjunto de guiones o *scripts* que permiten automatizar el proceso de compilación cruzada y la carga de las aplicaciones en los eCores de la placa Parallella:

- *Script* que realiza una compilación cruzada del código y copia el contenido en la memoria SD del procesador Parallella (Listado A.2).
- *Script* que se ejecutará en el procesador Zynq con el objetivo de cargar, lanzar y ver la salida por consola de un conjunto de ejecutables para Epiphany (Listado A.3).
- *Script* que compila en el procesador Parallella el código para Zynq localizado en el fichero loader.c (Listado A.4).
- *Script* para ejecutar en el procesador Parallella el código para Zynq en el ejecutable loader.elf que cargará los ejecutables en el eCore correspondiente (Listado A.5).

5.3. Interfaz Abstracta con el Hardware

Para la implementación de la interfaz abstracta con el hardware de M2OS-mc ha sido de gran utilidad la librería de soporte proporcionada por el fabricante, *Epiphany Hardware Utility Library* (eLib). Esta librería proporciona funciones para la gestión de bajo nivel de la plataforma, como son las relativas al manejo de las interrupciones y de los temporizadores. Para facilitar su uso desde el núcleo de M2OS-mc se ha desarrollado una interfaz Ada que encapsula las funciones utilizadas.

El HAL de M2OS-mc incluye las siguientes funcionalidades:

- **Cambio de contexto.** Gracias a la simplicidad del planificador de M2OS, el cambio de contexto solo requiere volver a poner el registro de pila del procesador

a la posición base y poner la dirección de comienzo del cuerpo de la tarea a ejecutar en el registro contador de programa.

- **Interrupciones.** Se pueden realizar las operaciones de habilitar, deshabilitar y comprobar el estado de las interrupciones globales del sistema. La implementación de esta funcionalidad está directamente basada en las funciones proporcionadas por la librería eLib.

- **Temporizador del sistema.** Sigue el método *tick* que requiere de una programación periódica del temporizador hardware. Utilizamos uno de los temporizadores que tiene cada eCore para generar una interrupción cada milisegundo. La interrupción se utiliza para llevar la cuenta de la hora del sistema. El temporizador, controlado por un reloj de $600MHz$, solo puede ser programado en modo de disparo único, por lo que debe ser reprogramado en cada ejecución del manejador de la interrupción.

- **Reloj del sistema.** Almacena un contador que es incrementado en cada interrupción del temporizador del sistema. Es un entero de 32 bits con una resolución de milisegundos.

- **Reloj de alta precisión.** Además del reloj del sistema, la implementación de M2OS-mc para Epiphany proporciona un reloj que permite realizar medidas de tiempo con alta precisión. Dicho reloj se implementa sobre el contador utilizado por el temporizador del sistema, por lo que tiene una precisión de 1,67ns (equivalente a un ciclo de reloj). Puesto que la cuenta del temporizador se reinicia cada interrupción, este reloj no sirve para medir intervalos de tiempo superiores a $1ms$.

Además, dadas las peculiaridades de la arquitectura muchos núcleos, ha sido necesario añadir algunas funcionalidades que no existían en el HAL de M2OS:

- **Identificación de núcleos.** La librería eLib proporciona primitivas para identificar en qué núcleo se está ejecutando el código actual. Esto estaba incluido en el HAL de M2OS al ser específico de arquitecturas con más de un núcleo y ahora forma parte de M2OS-mc.

- **Spinlock.** La librería eLib proporciona un tipo de *spinlocks* que permiten implementar sincronización de exclusión mutua ente aplicaciones ejecutando en diferentes eCores del procesador Epiphany. En la terminología utilizada en eLib

se llaman *mutex*. Esta parte del HAL consiste en una interfaz Ada de las funciones de gestión de *mutex* proporcionadas por la librería eLib y descritas en la Sección 4.1.6.

5.4. Sincronización de los relojes

Los ejecutables cargados en los diferentes eCores del procesador Epiphany tienen que recibir una señal concreta del procesador Zynq para comenzar su ejecución. Esta señal se envía de manera secuencial a cada eCore que, en cuanto la reciba, comenzará su ejecución. Debido a esa serialización cada eCore empieza su ejecución en un momento diferente, lo que conlleva que los relojes de las distintas instancias de M2OS-mc que se ejecutan en los diferentes eCores no estén sincronizados.

Para un sistema operativo de tiempo real estricto como M2OS-mc el que los relojes estén sincronizados, o que al menos haya una diferencia pequeña y acotada entre ellos, es algo muy útil para maximizar la sincronía entre tareas que se ejecutan en diferentes eCores. Además, esta sincronización permite medir el tiempo de respuesta de flujos e2e que atraviesen diferentes eCores y por ello utilicen diferentes relojes. Para minimizar la diferencia entre los relojes, M2OS-mc sincroniza todos los relojes durante el arranque de las instancias del RTOS situadas en los diferentes eCores.

La sincronización de relojes está dirigida por el eCore que empieza su ejecución en último lugar, que será el que ocupa la posición 0x0 de la malla, puesto que ese eCore es el último en recibir la señal de arranque desde nuestro *script* de carga y ejecución de aplicaciones (mostrado en el Listado A.3). El resto de eCores que se inicializan antes tienen que esperar a recibir un mensaje del último eCore que les indica el valor que tienen que poner en el contador de su temporizador hardware para estar en sincronía. En el valor enviado a cada eCore se tiene en cuenta el tiempo de transmisión del mensaje desde el eCore 0x0 y el tiempo que la instancia de M2OS destino tarda en cargar el valor en el contador de su temporizador.

En la Figura 5.2 se muestra el desfase de los relojes sin sincronizar. Para su cálculo se ha leído el valor del contador de cada eCore cuando recibe el mensaje proveniente del eCore 0x0. A cada valor leído se le ha restado el tiempo requerido por el eCore 0x0 para generar el correspondiente mensaje más el tiempo de transmisión del mensaje por la NoC. Aparte de ver que hay casi 10.000 ciclos (16.667 ns) de diferencia entre los núcleos con los tiempos más dispares, se puede, además, intuir el orden en el que se han empezado a ejecutar los núcleos siendo 0x1 (el núcleo en el que más tiempo ha

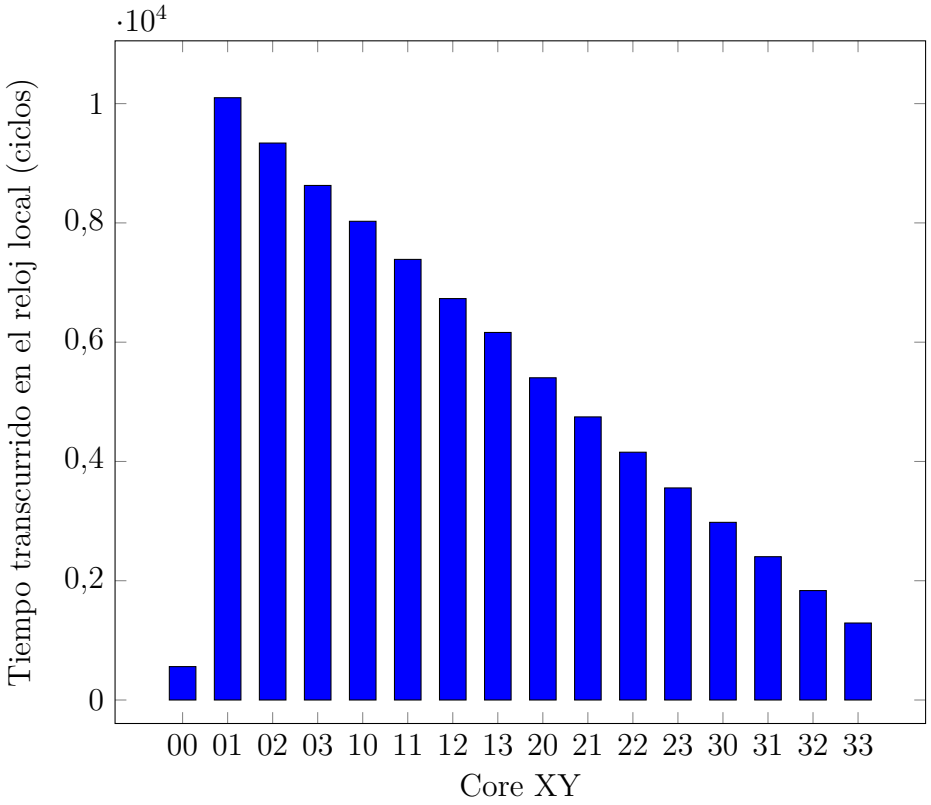

Figura 5.2 Tiempo normalizado marcado por el reloj local a cada núcleo en el momento de recibir el mensaje sin la sincronización de relojes.

transcurrido) el primero y 0x0 (el núcleo en el que menos tiempo ha transcurrido) el último.

En la Figura 5.3 se muestran los tiempos una vez completado el proceso de sincronización. Aunque se puede ver que la sincronización no es perfecta, la mayor diferencia que hay entre el reloj de dos núcleos es de 9 ciclos (15 ns), 1.111 veces menos que con los relojes sin sincronizar. Una mejora considerable y necesaria para el correcto comportamiento temporal un sistema de tiempo real estricto.

Otro dato que puede ser de interés en sistemas en el que el tiempo de arranque del mismo tenga que estar acotado es el intervalo que se tarda en realizar esta sincronización, que se mediría comparando el número de ciclos medidos en el reloj del núcleo de la celda 0x0 con la sincronización y sin ella. que resulta de 231 ciclos (385 ns).

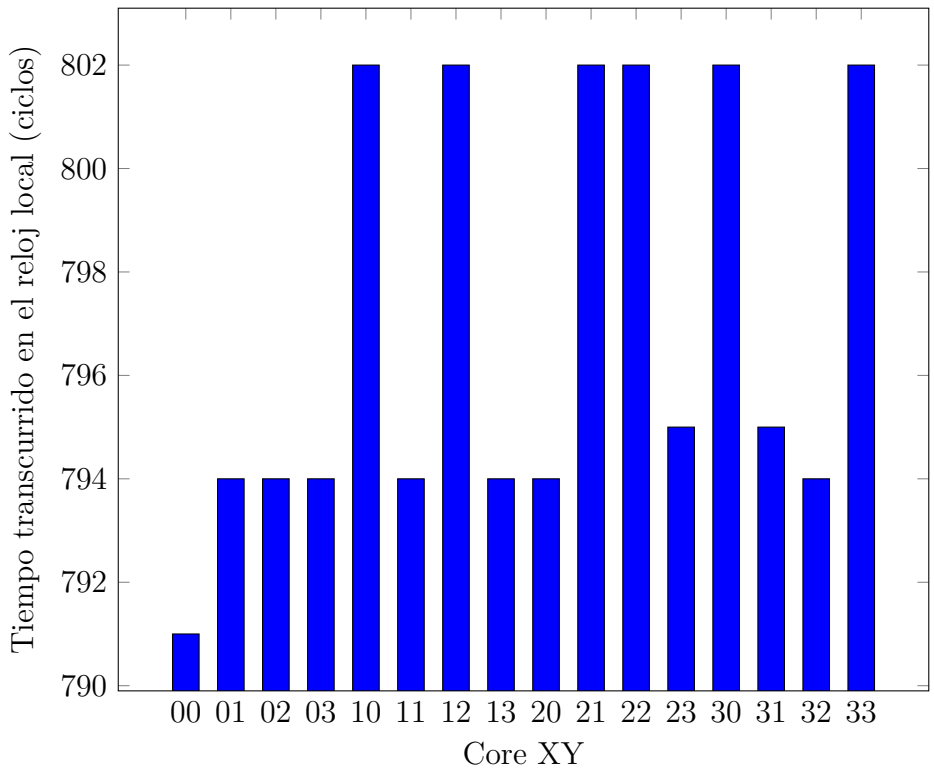

Figura 5.3 Tiempo normalizado marcado por el reloj local a cada núcleo en el momento de recibir el mensaje con la sincronización de relojes de M2OS-mc.

5.5. Rendimiento de la implementación de M2OS-mc

Para evaluar el rendimiento que se puede conseguir utilizando M2OS-mc se han realizado varios tests que involucran la ejecución en un único eCore (en el Capítulo 6 se evaluarán las prestaciones de los mecanismos desarrollados para la sincronización entre núcleos). Se ha medido el coste de la medición del reloj, el tiempo necesario para realizar el cambio de contexto entre tareas, el tiempo requerido por la operaciones de manejo del `mutex` y el tamaño de la aplicación.

Cada test realiza cien medidas consecutivas para observar posibles variaciones entre las diferentes ejecuciones. Con este número de ejecuciones se busca evitar la posible interferencia de la interrupción del reloj del sistema que sucede cada milisegundo ya que esta interrupción afectaría a las mediciones temporales.

- **Lectura del reloj.** Es necesario conocer cuánto tiempo se requiere para leer el reloj y poder así descontar ese tiempo para el resto de mediciones. Se muestra el resultado de estas mediciones en el Tabla 5.1. Viendo que el tiempo mínimo para

leer el reloj son 81 ciclos, para el resto de mediciones temporales se ha restado esa cantidad de ciclos de la medición consiguiendo así aislar la medición.

- **Mutex.** El tiempo que se tarda en tomar o liberar un mutex es constante cuando se ejecuta sin interferencias de otras tareas, como se muestra en el Tabla 5.1.
- **Cambio de contexto.** El tiempo requerido para realizar un cambio de contexto en un eCore se calcula mediante un test ya existente para el sistema operativo de tiempo real M2OS. Los cambios de contexto mostrados en el Tabla 5.2 se han dividido en tests de activaciones y de suspensión. Estos tests son:
  - **Tests de activación.** Latencia que se mide entre el instante en que una tarea se suspende (habiendo previamente desbloqueado una primitiva de suspensión) y el instante en que la tarea que se encontraba bloqueada en la citada primitiva comienza su ejecución. La medida se realiza para las siguientes primitivas de sincronización del lenguaje Ada: *suspension object*, *entry* de objeto protegido sin parámetros o *entry* de objeto protegido con un parámetro.
  - **Tests de suspensión.** Latencia medida desde que una tarea se suspende hasta que otra tarea, que ya se encontraba activa, comienza su ejecución. Se realizan mediciones para varias primitivas de suspensión del lenguaje Ada: *delay until*, *suspension object*, *entry* de objeto protegido sin parámetros y *entry* de objeto protegido con un parámetro.
- **Tamaño de las aplicaciones.** Para medir el tamaño en memoria requerido por las aplicaciones, se ha utilizado una aplicación con un número variable de tareas periódicas. En el Tabla 5.3 se muestran los tamaños de dos aplicaciones, una con dos tareas y otra con seis tareas. Para la medida se ha utilizado el comando `size` de Linux. Cada una de las tareas realiza las mismas operaciones: imprimir un mensaje por consola, poner un booleano a verdadero, calcular el instante de su siguiente activación y, finalmente, realizar el *delay until* con el valor previamente calculado. Se puede observar en el Tabla 5.3 que el incremento del número de tareas no aumenta drásticamente el tamaño del ejecutable.

Gracias a estos tests se puede comprobar que las latencias por utilizar M2OS-mc en un solo núcleo están acotadas y que no suponen un retardo significativo para los tiempos de ejecución. También se ha visto que el tamaño del ejecutable no aumenta enormemente con el incremento del número de tareas, lo cual es un dato muy importante dada la limitada memoria local de que disponen los eCores (32KB).

Tabla 5.1 Latencias medidas en la lectura del reloj y la utilización del mutex.

Test	Max	Min	Medio
Lectura Reloj	81 ciclos	81 ciclos	81 ciclos
Tomar Mutex	211,7 ns	211,7 ns	211,7 ns
Liberar Mutex	133,4 ns	133,4 ns	133,4 ns

Tabla 5.2 Latencias de cambios de contexto.

Tests de activación	Max	Min	Medio
Suspension object	593,5 ns	593,5 ns	593,5 ns
Entry objeto protegido sin parámetro	698,5 ns	698,5 ns	698,5 ns
Entry objeto protegido con un parámetro	736,8 ns	736,8 ns	736,8 ns

Tests de suspensión	Max	Min	Medio
Delay until	596,8 ns	596,8 ns	596,8 ns
Suspension object	345,1 ns	345,1 ns	345,1 ns
Entry objeto protegido sin parámetro	540,1 ns	433,4 ns	453,4 ns
Entry objeto protegido con un parámetro	548,4 ns	548,4 ns	548,4 ns

5.6. Salida por consola

Resulta fundamental disponer de un mecanismo que actúe como consola en la que los eCores puedan escribir los resultados de su ejecución, por ejemplo los tiempos medidos durante la ejecución de un test, etc.

En el caso del portado a la placa Parallella hay que tener en cuenta que la salida por consola es controlada por el procesador Zynq y que a dicha consola no se tiene acceso directo desde los eCores del procesador Epiphany.

Para solventar esta situación y conseguir que desde un eCore de Epiphany se pueda transmitir información a la terminal, sin tener que pasar por el temporalmente costoso trámite de acceder a memoria compartida, se ha reservado un área de memoria en una posición predeterminada de la memoria local de cada eCore para situar allí la información que se quiere mostrar por pantalla. Este espacio de memoria se escribe desde el eCore local como si se tratara de un *buffer* circular y se imprime en pantalla por el procesador Zynq de principio a fin. La escritura respeta la disposición de las líneas y nunca corta la salida de una línea. No tiene una limitación de líneas pero sí una limitación de caracteres (puesto que el tamaño del área de memoria destinado a salida por consola está limitado).

Tabla 5.3 Resultados del comando de Linux `size` para dos ejecutables que contienen 6 y 2 tareas respectivamente.

text	**data**	**bss**	**dec**	**hex**	**filename**
10914	1244	528	12686	318e	seis_tareas
10226	1244	208	11678	2d9e	dos_tareas

Esta salida por consola se puede realizar después del tiempo de ejecución que se considere necesario y se muestra la forma de realizarla en el código para el Zynq proporcionado en el Apéndice A.2 en las líneas 41 a 46.

La salida por consola se imprimirá en el terminal del usuario imprimiendo todas las zonas de memoria de todos los eCores. Si se intenta leer una zona de memoria de un eCore que no ha sido inicializado no se mostrará información de interés, pero la aplicación no fallará. La lectura y escritura de la zona de memoria compartida es asíncrona, lo cual no supone un problema al no constituir una función esencial para M2OS-mc ya que únicamente pretende mostrar el estado de los eCores sin considerarse esto una función crítica del sistema.

No se ha implementado la entrada por consola que, aunque podría tener cierto interés para facilitar la interacción persona-máquina en un sistema de tiempo real, no hemos considerado de vital importancia para el estudio realizado en esta tesis.

5.7. Estado de M2OS-mc

M2OS-mc está disponible en la web de M2OS [3]. La versión de la web es totalmente funcional y el código generado puede probarse en cualquier placa de desarrollo Parallella [4].

Se proporcionan las herramientas necesarias para su funcionamiento como un compilador gcc para eCore que puede compilar Ada, C y C++, unos *scripts* para poder hacer tanto la compilación cruzada como la carga de los ejecutables en los eCores correspondientes (mostrados en el Apéndice A) y también se incluyen las librerías necesarias que se han obtenido del kit de desarrollo de software (Software Development Kit o SDK) de Epiphany.

[3] https://m2os.unican.es/
[4] https://www.parallella.org/

Se ha desarrollado documentación que incluye una guía de usuario para que cualquier persona pueda utilizar M2OS-mc en una placa Parallella de manera satisfactoria incluyendo el uso de los *scripts* generados para facilitar su uso [5].

[5]https://m2os.unican.es/epiphany/

Capítulo 6

Mecanismos de comunicación

La aplicación típica que se ejecuta en un procesador mucho núcleos consiste en un conjunto de flujos de principio a fin (flujos end-to-end o e2e). Cada flujo e2e está formado por una serie de tareas mapeadas en el mismo o en diferentes eCores, una de las cuales responde a un evento externo. Las tareas de un flujo e2e deben disponer de un mecanismo de comunicación sincronizada para despertar a la siguiente tarea del flujo al final de cada ejecución y transmitirle la información pertinente.

Los mecanismos de sincronización desarrollados, que son los puertos de muestreo (Sampling Ports, SP) y los puertos con cola (Queuing Ports, QP) están inspirados en los puertos de comunicación con el mismo nombre definidos en el estándar ARINC-653 [35]. Hemos implementado ambos mecanismos en el sistema operativo M2OS-mc para el procesador muchos núcleos Epiphany [17, 64].

Un puerto de muestreo permite leer y escribir una instancia simple de un dato. Una operación de escritura en un SP sobrescribe el dato existente. Una operación de lectura retorna el último dato escrito. Existe un mecanismo que permite saber al lector si el dato ha sido leído por primera vez. Los SPs pueden ser utilizados cuando los periodos de los lectores y escritores son diferentes, cuando existe un productor y múltiples consumidores o cuando las aplicaciones están interesadas solamente en el dato más reciente.

Un puerto con cola permite la lectura y escritura de un conjunto de datos ordenados. El QP se basa en una cola circular en el que el primer dato en entrar es el primer dato en salir (FIFO). Cuando una tarea intenta escribir en un QP lleno, la operación acaba inmediatamente notificando el error. Cuando una tarea intenta leer de un QP vacío la tarea de lectura es bloqueada y será despertada cuando el dato llegue al QP. Los QPs pueden ser utilizados cuando los periodos de los lectores y escritores sean de media iguales y cuando las aplicaciones no quieran perderse ninguno de los datos. No son

apropiados cuando los periodos de los escritores sean menores que el los periodos de los lectores, ya que esto provocará que el QP pierda mensajes.

La API definida en M2OS-mc para los puertos de muestreo y los puertos con cola está inspirada en la utilizada por [35] en su implementación de SPs y QPs de ARINC-653 en Ada.

A continuación se pasará a describir la implementación de los dos tipos de puertos desarrollados.

6.1. Implementación de los puertos de muestreo

La implementación de los puertos de muestreo, SP, se basa en un *array* de registros, cada uno de ellos con la estructura mostrada en la Sección 6.1.1. La longitud de dicho *array*, o lo que es lo mismo, el número de puertos de muestreo disponibles, es configurable en tiempo de compilación. Para cada eCore, los valores del índice de dicho *array* se utilizan como identificadores de los puertos.

La dirección en memoria local del *array* de puertos de muestreo es la misma en cada uno de los eCores que ejecutan las replicas de M2OS-mc. Esto hace posible que desde cualquier eCore se pueda realizar una operación de lectura o escritura sobre un puerto de muestreo alojado en la memoria local de otro eCore.

El puerto de muestreo almacena la dirección de memoria en la que se encuentra el dato protegido y su tamaño en bytes. El área de memoria del dato es reservada por la aplicación y asignada al puerto al realizar su inicialización. Esto permite que todos los elementos del *array* de puertos de muestreo tengan la misma longitud y, por consiguiente, sean directamente accesibles desde otro eCore sin más que conocer su índice y la dirección de inicio del *array* que, como se ha comentado anteriormente, es la misma en las réplicas de M2OS-mc ejecutadas en cada eCore y es conocida en tiempo de compilación.

El acceso mutuamente exclusivo a cada uno de los puertos de muestreo está protegido por un *spinlock* (denominado `mutex` en la terminología de la librería de Epiphany, eLib).

6.1.1. Estructura de datos

El registro utilizado para implementar un puerto de muestreo incluye los siguientes campos:

- `Init`. Booleano utilizado para saber si el SP ha sido inicializado o no. Toma el valor de falso durante la inicialización de M2OS-mc y se pone a verdadero cuando el puerto es inicializado (ver Sección 6.1.2).
- `Mutex`. El *spinlock* que protege la integridad del puerto.
- `Size`. El tamaño de los datos protegidos.
- `Addr`. Dirección en la que se encuentran los datos protegidos. Utiliza una dirección global de memoria del procesador Epiphany.
- `New`. Booleano que indica si los datos han sido modificados desde la última operación de lectura sobre el SP.
- `Core`. Núcleo en el que está almacenado el SP. La réplica de M2OS-mc que se ejecuta en dicho eCore será la encargada de su inicialización. Este valor es utilizado por las operaciones que se realizan sobre el `mutex`.

El tamaño en memoria del registro correspondiente a un puerto de muestreo es 32 bytes.

6.1.2. Operaciones

Durante la ejecución de una aplicación, un puerto de muestreo puede pasar por los estados mostrados en la Figura 6.1 que se describen a continuación:

- **Inicial**. Estado en el que se encuentra el SP desde que se comienza la ejecución del sistema operativo hasta que es inicializado. Sobre un SP en este estado no se pueden realizar operaciones de escritura/lectura, por lo que estas operaciones retornarán un indicador de error en tal caso.
- **Leído**. Estado en el que se encuentra el SP cuando no tiene un dato nuevo. Las lecturas en este estado retornan el dato al que está apuntando el SP, indicando que no se trata de un dato nuevo. Una escritura renovaría el dato apuntado por el SP, cambiando el estado a **Nuevo dato**.
- **Tomado**. Es el estado en el que se encuentra el SP mientras el `mutex` se mantiene tomado para asegurar el acceso exclusivo a los campos de su registro y a los datos protegidos. A este estado llega el SP de manera transitoria al realizar varias de las operaciones posibles sobre él.

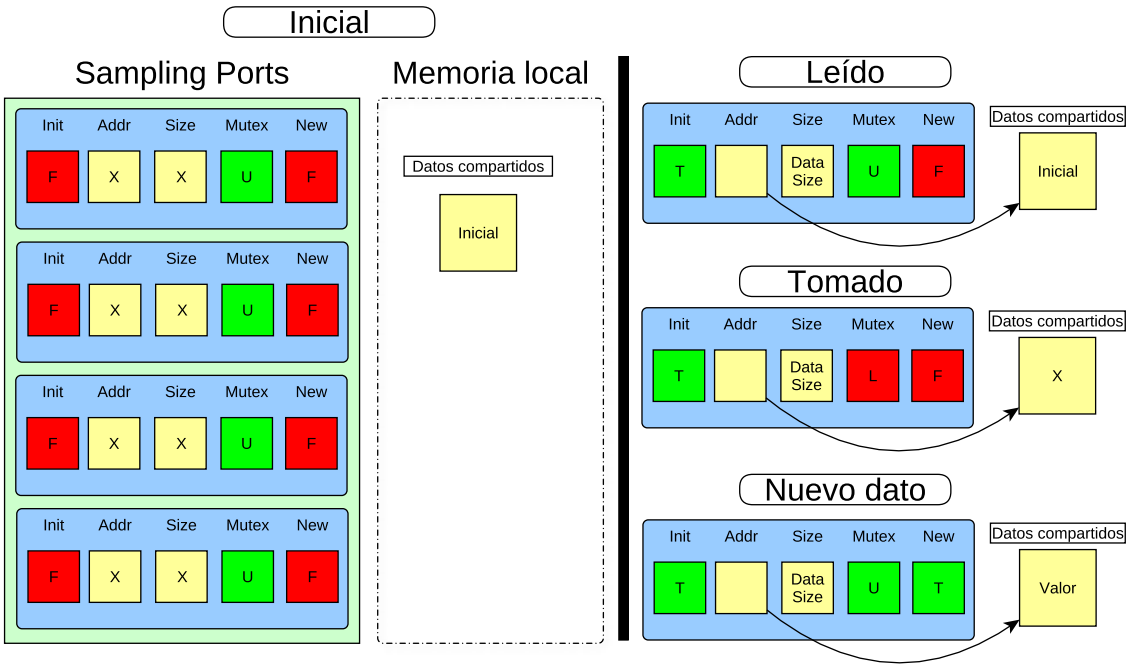

Figura 6.1 Se muestran todos los estados por los que pasa un puerto de muestreo. Se muestran todos los SP en el estado inicial (izquierda). Se muestra un solo SP en el resto de estados (derecha).

- **Nuevo dato**. Estado en el que se encuentra el SP en el momento en el que tiene un dato nuevo, que aún no ha sido leído por ninguna tarea. Las lecturas en este estado retornan el valor al que está apuntando el SP, indicando que se trata de un dato nuevo y transita entonces al estado **Leído**. En caso de que estando en este estado se realice una nueva escritura, el dato protegido será sobrescrito y el SP se mantiene en este mismo estado.

Listado 6.1 Interfaz de los puertos de muestreo

```
type SP_Index is range 0 .. Sampling_Ports_Per_Core-1;
type SP_Id is private;

procedure Init_Sampling_Port (C : in E_Lib.Core;
                              Id : in SP_Index;
                              Addr : in System.Address;
                              Size : in Unsigned_32;
                              SP : out SP_Id);

procedure Get_Sampling_Port (C : in E_Lib.Core;
                             Id : in SP_Index;
                             SP : out SP_Id);

function Is_Initialized (SP : in SP_Id) return Boolean;

procedure Write_Sampling_Port (SP : in SP_Id;
                               Addr : in System.Address;
                               Size : in Unsigned_32;
                               Successful : out Boolean);

procedure Read_Sampling_Port (SP : in SP_Id;
                              Addr : in System.Address;
                              Size : in Unsigned_32;
                              Successful : out Boolean;
                              Is_New : out Boolean);
```

La interfaz diseñada para la implementación de los puertos de muestreo se muestra en el Listado 6.1.

La descripción de las funciones implementadas es la siguiente:

- `Init_Sampling_Port`. El procedimiento inicializa el SP identificado por el eCore y el índice pasados como parámetros. Como consecuencia de la ejecución de este procedimiento el puerto pasa del estado **Inicial** al estado **Leído**, pasando en el proceso por el estado **Tomado**. Si el SP había sido inicializado previamente la función retorna `Null_SP_Id`. Este procedimiento tiene como parámetro de salida el identificador del puerto de muestreo que se ha inicializado.

- `Get_Sampling_Port`. Retorna el identificador del puerto identificado por el eCore y el índice pasados como parámetros. Si el SP no ha sido inicializado aún, retorna `Null_SP_Id`.

- `Is_Initialized`. Función que retorna verdadero si el puerto correspondiente al identificador pasado como parámetro está inicializado o falso en caso contrario.

- `Write_Sampling_Port`. Procedimiento que actualiza el valor del dato protegido en el SP. El dato a copiar se encuentra en la dirección de memoria que la función recibe como parámetro y tiene un tamaño que también se recibe como parámetro. El procedimiento realiza la transición del estado del SP desde **Leído** (o **Nuevo dato**) al estado **Tomado** para finalmente llegar al estado **Nuevo dato** poniendo el campo `New` a verdadero. Si la operación de escritura ha sido ejecutada con éxito, al parámetro de salida `Successful` se le asigna el valor `True`. Este parámetro de salida tomará el valor falso cuando se intente escribir un SP que está en el estado **Inicial** o cuando el tamaño del dato a escribir supere el tamaño definido para el SP.

- `Read_Sampling_Port`. Este procedimiento copia el dato protegido por el puerto de muestreo en la dirección que se proporciona como parámetro de entrada. El campo `New` se pone a falso. Al principio de la operación, el SP se pone en modo **Tomado** para terminar dejando el SP en el estado **Leído** una vez se acaba la operación. Si la lectura ha sido satisfactoria al parámetro de salida `Successful` se le asigna el valor verdadero. Este parámetro tomará el valor falso cuando se trata de leer un SP que no haya sido previamente inicializado, cuando el tamaño definido en el SP es mayor que el tamaño de la variable donde se tiene que volcar el contenido del dato protegido o cuando todavía no se ha escrito ningún dato en este SP.

Listado 6.2 Implementación de las tareas productoras y consumidor utilizando un puerto de muestreo

```
1  procedure Tarea_Consumidor_Init is
2  begin
3    --  Se inicializa la tarea
4    QP := Init_Sampling_Port
5      (Core, Indice_SP, Dato'Address, Dato'Size, SP);
6  end Tarea_Consumidor_Init;
7
8  procedure Tarea_Consumidor_Body is
9  begin
10
11   Read_Sampling_Port
12     (SP, Var'Address, Var'Size, Satisfactorio, Es_Nuevo);
13
14   if Es_Nuevo then
15     Use Data;
16   end if;
17
18   delay until Ahora + Periodo_De_Muestreo;
19 end Tarea_Consumidor_Body;
20
21 --  En el Init del productor no se hace nada sobre el SP
22 procedure Tarea_Productor_Body is
23 begin
24   if not SP_Getted then
25     Get_Sampling_Port (Core, Indice, SP);
26     if not Is_Initialized (SP) then
27       delay until Ahora + Periodo_De_Espera;
28     end if;
29     SP_Getted  := True;
30   end if;
31
32   Generate data;
33   Write_Sampling_Port
34     (SP, Data'Address, Data'Size, Satisfactorio);
35   --  Task work post write
36   delay until Ahora + Periodo_Tarea;
37 end Tarea_Productor_Body;
```

La forma de implementar un sistema productor-consumidor utilizando el mecanismo de comunicación sincronizada de puerto de muestreo se muestra en el Listado 6.2. El SP se encuentra alojado en la memoria local del eCore que ejecuta la tarea consumidora, la cual es la tarea encargada de realizar su inicialización. Si existe un dato nuevo en el SP antes de realizar la lectura del dato (líneas 14 a 16). Si una tarea comprobara constantemente el contenido del SP, podría generar el fenómeno de inanición (*starvation*) al resto de tareas que se ejecutan en el mismo núcleo. Por lo tanto, resulta conveniente utilizar un mecanismo de muestreo periódico que permita liberar liberar el núcleo para que otras tareas puedan ejecutar. Esa es la función del *delay until* de la línea 18.

La tarea productora realiza todas las operaciones relacionadas con el SP en su cuerpo. En su primera activación, la tarea productora debe esperar a que el SP haya sido inicializado por la tarea consumidora (líneas 24 a 30). Puede ser conveniente realizar esta comprobación realizando un periodo de espera para permitir ejecutar a otras tareas (línea 27). Una vez el SP hay sido inicializado, la tarea productora puede comenzar a producir los datos (operaciones representadas por la línea 32) de forma periódica (línea 36).

6.2. Implementación de los puertos con cola

La implementación de los puertos con cola, igual que ocurría con los puertos de muestreo, se basa en un *array* de registros, cada uno de ellos con la estructura mostrada en la Sección 6.2.1. La longitud de dicho *array*, o lo que es lo mismo, el número de puertos con cola disponibles, es configurable en tiempo de compilación. Para cada eCore, los valores del índice de dicho *array* se utilizan como identificadores de los puertos.

La dirección en memoria local del *array* de puertos con cola es la misma en cada uno de los eCores que ejecutan las replicas de M2OS-mc. Esto hace posible que desde cualquier eCore se pueda realizar una operación de lectura o escritura sobre un puerto con cola alojado en la memoria local de otro eCore.

El puerto con cola almacena una cola circular, con una capacidad determinada por el número de elementos y el tamaño de estos (ambos parámetros definidos en tiempo de compilación). El área de memoria de la cola es reservada por la aplicación y asignada al puerto al realizar su inicialización. Esto permite que todos los elementos del *array* de puertos con cola tengan la misma longitud y, por consiguiente, sean directamente accesibles desde otro eCore sin más que conocer su índice y la dirección de inicio

del *array* que, como se ha comentado anteriormente, es la misma en las réplicas de M2OS-mc ejecutadas en cada eCore y es conocida en tiempo de compilación.

El acceso a cada uno de los puertos con cola está protegido por un *spinlock* (denominado `mutex` en la terminología de la librería de Epiphany, eLib).

6.2.1. Estructura de datos

El registro utilizado para implementar un puerto con cola incluye los siguientes campos:

- `Initialized`. Un booleano que indica si el QP ha sido inicializado o no. Toma el valor de falso durante la inicialización de M2OS-mc y se pone a verdadero cuando el puerto es inicializado (ver Sección 6.2.2)
- `Mutex`. El *spinlock* para serializar los accesos al QP.
- `Queue_Size`. El número máximo de elementos que puede almacenar un QP.
- `Tail`. El índice del elemento más reciente escrito en el QP.
- `Head`. El índice del elemento que tocaría leer en el QP.
- `N_Elements`. El número de elementos actualmente almacenados en la cola circular del QP.
- `Element_Size`. El tamaño de cada elemento guardada en el QP. Todos los elementos tienen el mismo tamaño.
- `Elements`. El puntero hacia el área en el que se almacenan los elementos del QP. La estructura de datos es un array de `N_Elements` de tamaño `Element_Size`. Este array es creado por la tarea que inicializa el QP.
- `Core`. Núcleo en el que está almacenado el QP. La réplica de M2OS-mc que ejecuta en dicho eCore será la encargada de su inicialización. Este valor es utilizado por las operaciones que se realizan sobre el `mutex`.

6.2.2. Operaciones

Durante la ejecución de una aplicación, un puerto con cola puede pasar por los estados mostrados en la Figura 6.2 que se describen a continuación:

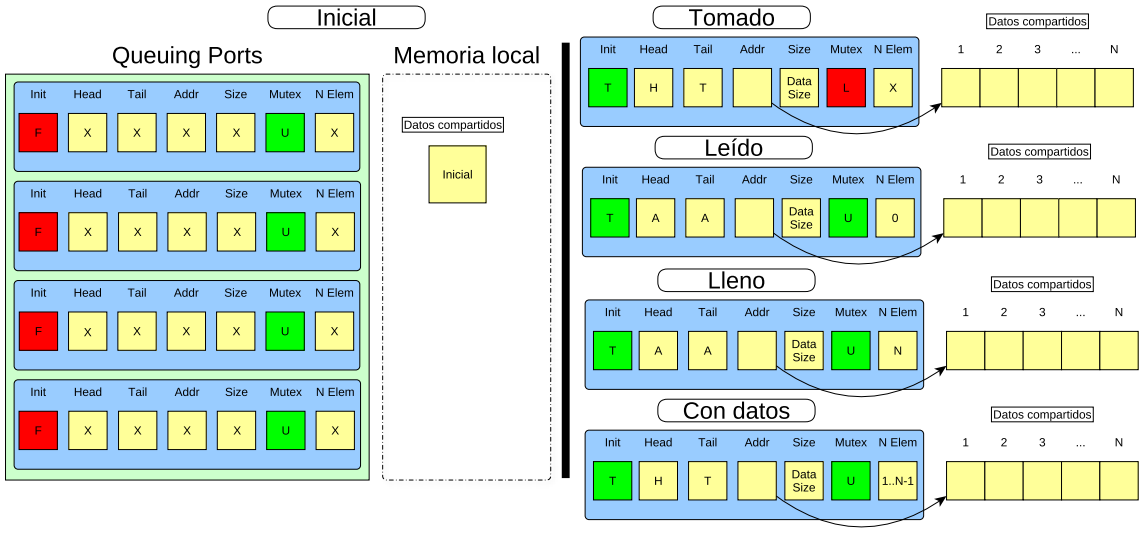

Figura 6.2 Estados por los que pasa un puerto con cola. Se muestran todos los QP en el estado inicial (izquierda). Se muestra un solo QP en el resto de estados (derecha).

- **Inicial**. Estado en el que se encuentra el QP desde que se comienza la ejecución del sistema operativo hasta que se inicializa el QP. Sobre un QP en este estado no se pueden realizar operaciones de escritura/lectura, por lo que la ejecución de dichas operaciones tendrán como parámetro de salida un indicador de error.

- **Tomado**. Es el estado en el que se encuentra el QP mientras el `mutex` se mantiene tomado para asegurar el acceso exclusivo a los campos de su registro y a los datos protegidos. A este estado llega el QP de manera transitoria al realizar varias de las operaciones posibles sobre él.

- **Leído**. Este estado significa que en la cola circular del QP no hay ningún dato válido. En este estado se pueden hacer escrituras (cambiando su estado a **Con datos** o **Lleno** si la escritura llena el *buffer*), pero las lecturas bloquearán a la tarea lectora hasta que haya datos válidos en el QP.

- **Lleno**. Se llega a este estado al tener la cola circular al máximo de capacidad. En este estado no se pueden hacer escrituras, pero sí lecturas (cambiando su estado a **Con datos** o **Leído** si la lectura vacía el *buffer*).

- **Con datos**. Estado al que se llega cuando la cola ni está llena ni está vacía. En este estado se pueden hacer tanto escrituras como lecturas y se cambiará al estado **Vacío** o **Lleno** si la cola se vacía o llena respectivamente.

Listado 6.3 Interfaz del puerto con cola

```
Queuing_Ports_Per_Core : constant Unsigned_32 := 4;
type QP_Index is range 0 ..Queuing_Ports_Per_Core-1;
type QP_Id is private;

procedure Init_Queuing_Port (Index : in QP_Index;
                             Size_Q : in Unsigned_32;
                             Data_Address : in System.Address;
                             Data_Size : in Unsigned_32;
                             Core : in E_Lib.Core;
                             QP : out QP_Id);

procedure Get_Queuing_Port (Core : in E_Lib.Core;
                            Index : in QP_Index;
                            QP : out QP_Id);

function Check_Queuing_Port_Initialized (QP : in QP_Access)
                                        return Boolean;

procedure Write_Queuing_Port (QP : in out QP_Id;
                              Orig : in System.Address;
                              Orig_Size : in Unsigned_32;
                              Successful : out Boolean);

procedure Read_Queuing_Port (QP : in out QP_Id;
                             Dest : in System.Address;
                             Dest_Size : in Unsigned_32;
                             Successful : out Boolean);
```

La interfaz diseñada para la implementación de los puertos con cola se muestra en el Listado 6.3.

La descripción de las funciones implementadas es la siguiente:

- `Init_Queuing_Port`. Este procedimiento inicializa el QP identificado por el eCore y el índice pasados como parámetros. Como consecuencia de la ejecución de este procedimiento el puerto pasa del estado **Inicial** al estado **Leído**, pasando en el

proceso por el estado **Tomado**. Este procedimiento tiene como parámetro de salida el identificador del puerto con cola que se ha inicializado.

- `Get_Queuing_Port`. Retorna el identificador del puerto identificado por el eCore y el índice pasados como parámetros.

- `Check_Queuing_Port_Initialized`. Función que retorna verdadero si el puerto correspondiente al identificador pasado pasado como parámetro está inicializado o falso en caso contrario.

- `Write_Queuing_Port`. Un procedimiento que escribe un dato en la cola del QP. La dirección y el tamaño del dato a copiar se reciben como parámetros de entrada. Se tiene que comprobar que el dato tiene cabida en la cola, ya que no se puede escribir si la cola está en el estado **Llena**. Si la cola no está llena, el dato se copiará en la posición de la cola apuntada por `Tail`, desplazando posteriormente este puntero a la siguiente posición de la cola y aumentando en uno el número de elementos de la cola. El procedimiento realiza inicialmente la transición del estado del QP desde **Leído** (o **Con datos**) al estado **Tomado** para finalmente llegar al estado **Con datos** o **Lleno** en función de cómo acabe de ocupada la cola circular (con huecos pasa al estado **Con datos** y si no tiene huecos libres al estado **Lleno**). Si la operación de escritura ha sido ejecutada con éxito, al parámetro de salida `Successful` se le asigna el valor `True`. Este parámetro de salida tomará el valor falso cuando se intente escribir un QP que está en el estado **Inicial**, cuando el tamaño del dato a escribir supere el tamaño definido en el QP o cuando no se haya podido escribir al encontrarse la cola llena.

- `Read_Queuing_Port`. Si hay elementos en la cola circular protegida, este procedimiento copia el dato apuntado por `Head` en la dirección que se proporciona como parámetro de entrada. Se mueve el puntero `Head` a la siguiente posición de la cola circular y se reduce en uno el número de elementos que hay almacenados en la cola. Al principio de la operación de lectura, el QP se pone en modo **Bloqueado** para terminar en el estado **Leído** o **Con datos** según la cantidad de datos con la que quede el QP. Si la lectura ha sido satisfactoria se asigna el valor verdadero al parámetro de salida `Successful`. Este parámetro tomará el valor falso cuando se trata de leer un QP que no haya sido previamente inicializado o cuando el tamaño definido en el QP es mayor que el tamaño de los elementos de la cola. En caso de que no haya ningún dato disponible en la cola para ser leído, este procedimiento bloquea a la tarea que lo ha invocado. Esta tarea será desbloqueada

por el despachador (*dispatcher*) del sistema operativo cuando se escriba un dato en el QP leído y la tarea sea la de mayor prioridad lista para ser ejecutada.

La implementación de la tarea consumidora que vaya a utilizar puertos con cola tiene que tener en cuenta que M2OS-mc implementa una política de planificación no expulsora, lo que implica que cuando se intente leer un QP vacío la tarea que haya invocado la instrucción de lectura finalizará su activación actual y será bloqueada. Es por ello que el dato leído tiene que ser utilizado al principio del cuerpo de la tarea, antes de la operación de lectura.

Un código tipo de una aplicación productor-consumidor utilizando un puerto con cola se muestra en el Listado 6.4. En él podemos observar como la tarea consumidora tiene que inicializar el QP (líneas 4 a 6), se recomienda hacer esto desde el mismo núcleo del que se va a hacer la lectura. Como se indicó anteriormente, la tarea consumidora debe realizar la primera lectura del QP durante su inicialización (línea 7) para tener un dato disponible en la primera ejecución de su cuerpo. En el cuerpo de la tarea consumidora se utiliza el dato (línea 12) y se realiza la lectura del QP para la siguiente activación (línea 13). La tarea productora también presente en el Listado 6.4 no realiza ninguna operación sobre el QP durante su inicialización. En el cuerpo de esta tarea se comprueba que el puerto con cola está inicializado (líneas 18 a 24) antes de hacer ninguna operación sobre él. Si el QP no está inicializado realizamos un muestreo periódico para no producir el fenómeno de inanición (*starvation*) (línea 21). Con la certeza de que el QP está inicializado, se puede escribir en el QP (línea 27) con la periodicidad deseada. En la línea 29 se muestra la espera al siguiente periodo.

Listado 6.4 Implementación de un sistema productor-consumidor utilizando un puerto con cola

```
1  procedure Tarea_Consumidor_Init is
2  begin
3    -- Inicializacion de la tarea
4    QP := Init_Queuing_Port
5      (Port Index, Size_Q, Data_Q'Address, Unsigned_64'Size,
6       Core);
7    Read_Queuing_Port (QP, Var'Address, Var'Size, Success);
8  end Tarea_Consumidor_Init;
9
10 procedure Tarea_Consumidor_Body is
11 begin
12   Use Data;
13   Read_Queuing_Port (QP, Var'Address, Var'Size, Success);
14 end Read_Task_Body;
15
16 procedure Tarea_Productor_Body is
17 begin
18   if not QP_Getted then
19     Get_Queuing_Port (Core, Indice, QP);
20     if not Check_Queuing_Port_Initialized (QP) then
21       delay until Ahora + Periodo_De_Espera;
22     end if;
23   SP_Getted := True;
24   end if;
25
26   Generate Data;
27   Write_Queuing_Port (QP, Data'Address, Data'Size, Success);
28   -- Codigo post escritura
29   delay until Ahora + Periodo_Tarea;
30 end Tarea_Productor_Body;
```

6.3. Tests de rendimiento

En esta sección se realiza la evaluación temporal de las operaciones definidas en la API de los puertos de muestreo y de los puertos con cola descrita en la sección anterior.

En cada test se han realizado 5.000 medidas de la ejecución de la operación bajo estudio con el objeto de detectar posibles variaciones en la ejecución de las aplicaciones.

Todos los tests se han ejecutado en aislamiento, esto es, sin otras tareas competidoras y sin más tráfico en la red.

Los tiempos incluyen la interrupción del reloj de M2OS, que en caso del mensaje de ida y vuelta esta interrupción puede ocurrir una vez en cada núcleo. Este tiempo es de 268,33 ns y se puede apreciar directamente en los tiempos de peor caso de muchas de las medidas.

6.3.1. Puertos de muestreo

La Tabla 6.1 muestra el tiempo que requiere ejecutar las operaciones descritas en la Sección 6.1.2 (y mostradas en el Listado 6.1). Los procedimientos `Init_Sampling_Port` y `Read_Sampling_Port` se han medido para un SP alojado en la memoria local del eCore. Los procedimientos `Get_Sampling_Port` y `Write_Sampling_Port` se han medido para un SP alojado en la memoria local de un eCore a distancia uno. Los tiempos de las operaciones `Init_Sampling_Port` y `Get_Sampling_Port` no varían con el tamaño del dato contenido en los SP, mientras que los tiempos de las operaciones `Read_Sampling_Port` y `Write_Sampling_Port` varían según el tamaño del dato de forma lineal.

Tabla 6.1 Latencias del uso de puertos de muestreo.

Latencia de	**Max**	**Min**	**Media**
Init_Sampling_Port	518 ns	518 ns	518 ns
Get_Sampling_Port	235 ns	235 ns	235 ns
Write_Sampling_Port (8 bytes)	730 ns	456,67 ns	461,67 ns
Read_Sampling_Port (8 bytes)	896,67 ns	623,33 ns	626,67 ns
Write_Sampling_Port (40 bytes)	726,67 ns	458,33 ns	458,33 ns
Read_Sampling_Port (40 bytes)	951,67 ns	675 ns	683,33 ns

En la Tabla 6.2 se muestra el tiempo requerido por los procedimientos `Write_Sampling_Port` y `Read_Sampling_Port` al acceder a un SP alojado en la memoria local de otro eCore. Se puede observar como la distancia, medida en número de saltos, entre el eCore que ejecuta el procedimiento y el eCore que hospeda el SP tiene

un impacto lineal en los tiempos de ejecución. Los valores mostrados en la Tabla 6.2 corresponden a la escritura/lectura de un dato de ocho bytes (un paquete en la NoC del Epiphany) para ambas operaciones.

Tabla 6.2 Latencias para `Write_Sampling_Port` y para `Read_Sampling_Port` para diferentes distancias de saltos entre eCores.

Write_Sampling_Port	**Max**	**Min**	**Media**
1 salto	730 ns	456,67 ns	461,67 ns
2 saltos	735 ns	456,67 ns	466,67 ns
3 saltos	753,33 ns	471,67 ns	471,67 ns
4 saltos	753,33 ns	468,33 ns	476,67 ns
5 saltos	750 ns	468,33 ns	481,67 ns
6 saltos	753,33 ns	486,67 ns	486,67 ns

Read_Sampling_Port	**Max**	**Min**	**Media**
1 salto	945 ns	675 ns	675 ns
2 saltos	1000 ns	715 ns	725 ns
3 saltos	1068,33 ns	773,33 ns	775 ns
4 saltos	1106,67 ns	825 ns	825 ns
5 saltos	1145 ns	875 ns	875 ns
6 saltos	1195 ns	925 ns	925 ns

Para completar el análisis del rendimiento de los puertos de muestreo, se plantean en la Tabla 6.3 los resultados de las medidas temporales en un escenario de comunicación entre dos tareas situadas en dos eCores diferentes con un SP alojado en la memoria local de cada uno de ellos. En este escenario, una de las tareas es la encargada de iniciar la comunicación, escribiendo un dato en el SP del otro eCore. La tarea alojada en este segundo eCore se encuentra ejecutando un lazo en el comprueba constantemente si hay un nuevo dato en su SP. Al detectar la llegada de un nuevo dato, escribe un dato de vuelta en el SP del eCore inicial. El tiempo es medido por la tarea inicial, entre el instante inmediatamente anterior al envío del primer mensaje y el momento inmediatamente posterior a su activación por parte de M2OS-mc tras la llegada del mensaje de vuelta. Con el objetivo de tener unos tests lo más precisos posibles, ninguna de las tareas se bloquea, permaneciendo en un lazo de consulta del SP correspondiente cuando tiene que esperar la llegada de un mensaje. En los resultados mostrados en la Tabla 6.3 se puede observar cómo existe una dependencia lineal con la distancia entre eCores. Además se produce el fenómeno de bloqueo debido a la exclusión mutua

producida en el acceso al SP, lo que se refleja en un gran incremento de los tiempos de ejecución de peor caso y un menor incremento de los tiempos promedios.

8 bytes	**Max**	**Min**	**Media**
1 salto	2537 ns	1882 ns	2083 ns
2 saltos	3090 ns	1883 ns	2093 ns
3 saltos	3053 ns	1922 ns	2087 ns
4 saltos	3195 ns	1928 ns	2118 ns
5 saltos	3135 ns	1932 ns	2113 ns
6 saltos	3027 ns	1965 ns	2117 ns

40 bytes	**Max**	**Min**	**Media**
1 salto	3608 ns	1952 ns	2430 ns
2 saltos	3582 ns	1957 ns	2393 ns
3 saltos	3488 ns	1993 ns	2448 ns
4 saltos	3483 ns	1997 ns	2422 ns
5 saltos	3495 ns	2003 ns	2442 ns
6 saltos	2948 ns	2042 ns	2468 ns

80 bytes	**Max**	**Min**	**Media**
1 salto	4830 ns	2608 ns	3150 ns
2 saltos	4730 ns	2612 ns	3105 ns
3 saltos	5035 ns	2650 ns	3088 ns
4 saltos	4593 ns	2655 ns	3093 ns
5 saltos	4608 ns	2662 ns	3098 ns
6 saltos	5048 ns	2670 ns	3107 ns

Tabla 6.3 Latencia para mensajes de escritura de ida y vuelta con SPs para datos de 8 bytes (izquierda), 40 bytes (derecha) y 80 bytes (abajo).

6.3.2. Puertos con cola

La Tabla 6.4 muestra el tiempo que requiere ejecutar las operaciones descritas en la Sección 6.2.2 (y mostradas en el Listado 6.3). Los procedimientos `Init_Queuing_Port` y `Read_Queuing_Port` se han medido para un QP alojado en la memoria local del eCore. Los procedimientos `Get_Queuing_Port` y `Write_Queuing_Port` se han medido para un QP alojado en la memoria local de un eCore a distancia uno. Las operaciones `Init_Queuing_Port` y `Get_Queuing_Port` no varían con el tamaño del dato contenido en los QP. Las operaciones `Read_Queuing_Port` y `Write_Queuing_Port` varían según el tamaño del dato de forma lineal.

En la Tabla 6.5 se muestra el tiempo requerido por los procedimientos `Write_Queuing_Port` y `Read_Queuing_Port` al acceder a un SP alojado en la memoria local de otro eCore. Se puede observar cómo la distancia, medida en número de saltos, entre el eCore que ejecuta el procedimiento y el eCore que hospeda el QP tiene un impacto lineal en los tiempos de ejecución. Los valores mostrados en la Tabla 6.5 corresponden a la escritura/lectura de un dato de ocho bytes (un paquete en la NoC del Epiphany) para ambas operaciones.

Tabla 6.4 Latencias del uso de puertos con cola.

Latencia de	**Max**	**Min**	**Media**
Init_Queuing_Port	8670 ns	8670 ns	8670 ns
Get_Queuing_Port	5762 ns	5747 ns	5748 ns
Write_Queuing_Port (8 bytes)	1048 ns	746,67 ns	761,67 ns
Read_Queuing_Port (8 bytes)	2213 ns	1.447 ns	1.463 ns
Write_Queuing_Port (40 bytes)	1037 ns	738,33 ns	755 ns
Read_Queuing_Port (40 bytes)	2272 ns	1505 ns	1527 ns

Tabla 6.5 Latencias para `Write_Queuing_Port` y para `Read_Queuing_Port` para diferentes distancias de saltos entre eCores.

Write_Queuing_Port	**Max**	**Min**	**Media**
1 salto	1017 ns	733,67 ns	750 ns
2 saltos	1048 ns	746,67 ns	761,67 ns
3 saltos	1058 ns	760 ns	778,33 ns
4 saltos	1083 ns	793,33 ns	806,67 ns
5 saltos	1098 ns	808,33 ns	823,33 ns
6 saltos	1108 ns	821,67 ns	836,67 ns

Read_Queuing_Port	**Max**	**Min**	**Media**
1 salto	1760 ns	1453 ns	1473 ns
2 saltos	1818 ns	1508 ns	1528 ns
3 saltos	1873 ns	1563 ns	1583 ns
4 saltos	1923 ns	1622 ns	1630 ns
5 saltos	1972 ns	1673 ns	1692 ns
6 saltos	2058 ns	1727 ns	1758 ns

Para completar el análisis del rendimiento de los puertos con cola, se plantean en la Tabla 6.6 los resultados de las medidas temporales en un escenario de comunicación entre dos tareas situadas en dos eCores diferentes con un QP alojado en la memoria local de cada uno de ellos. En este escenario, una de las tareas es la encargada de iniciar la comunicación, escribiendo un dato en el QP del otro eCore. La tarea alojada en este segundo eCore lee el contenido del QP, quedando bloqueada si no hay un dato en su QP. Al detectar la llegada de un nuevo dato, M2OS-mc activa la tarea que había sido bloqueada previamente, la cual escribe un dato de vuelta en el QP del eCore inicial. El tiempo es medido por la tarea inicial, entre el instante inmediatamente anterior al envío del primer mensaje y el momento en que detecta la llegada del mensaje de vuelta. En los resultados mostrados en la Tabla 6.6 se puede observar cómo existe una

dependencia lineal con la distancia entre eCores. Además se produce el fenómeno de bloqueo debido a la exclusión mutua producida en el acceso al QP, lo que se refleja en un gran incremento de los tiempos de ejecución de peor caso y un menor incremento de los tiempos promedios.

8 bytes	**Max**	**Min**	**Media**
1 salto	4388 ns	3122 ns	3382 ns
2 saltos	4213 ns	3170 ns	3393 n
3 saltos	4250 ns	3212 ns	3435 ns
4 saltos	4283 ns	3232 ns	3482 ns
5 saltos	4327 ns	3280 ns	3510 ns
6 saltos	4347 ns	3302 ns	3542 ns

40 bytes	**Max**	**Min**	**Media**
1 salto	4512 ns	3478 ns	3723 ns
2 saltos	4570 ns	3495 ns	3742 ns
3 saltos	4592 ns	3545 ns	3790 ns
4 saltos	4625 ns	3563 ns	3810 ns
5 saltos	4673 ns	3585 ns	3843 ns
6 saltos	4677 ns	3620 ns	3880 ns

80 bytes	**Max**	**Min**	**Media**
1 salto	5345 ns	3983 ns	4385 ns
2 saltos	5288 ns	4003 ns	4405 ns
3 saltos	5403 ns	4047 ns	4427 ns
4 saltos	5362 ns	4078 ns	4447 ns
5 saltos	5513 ns	4118 ns	4473 ns
6 saltos	5332 ns	4152 ns	4515 ns

Tabla 6.6 Latencia para mensajes de escritura de ida y vuelta con QPs para datos de 8 bytes (izquierda), 40 bytes (derecha) y 80 bytes (abajo).

6.3.3. Formulas de cálculo del coste temporal

Aunque las medidas que se pueden hacer sobre los tiempos de ejecución de las tareas ya tienen en cuenta el coste de la ejecución de los diferentes mecanismos de comunicación sincronizada, es de gran utilidad saber cuánto tiempo llevan las diferentes operaciones que se pueden hacer sobre ellos para tener un mayor conocimiento de en qué se dedica el tiempo o poder simular comportamientos de tareas con ciertas características. No siempre se puede aislar el tiempo de ejecución de una operación sobre un mecanismo de sincronización por lo que resulta de gran interés disponer de una cota superior del tiempo de uso para acotar tiempos de ejecución y, además, estimar los posibles bloqueos que se pueden producir cuando dos tareas intentan acceder a un recurso compartido (en este caso un puerto). Además, es necesario conocer el uso de la NoC por parte de las operaciones sobre los puertos de comunicación, con el fin de poder aplicar las técnicas de modelado y análisis desarrolladas en esta tesis.

Operación sobre un puerto	Lecturas (N_R)	Escrituras (N_W)
Write_Sampling_Port	1	3
Read_Sampling_Port	3	1
Write_Queuing_Port	3	3
Read_Queuing_Port	4	3

Tabla 6.7 Número de operaciones de lectura y escritura que realizan a través de la NoC las operaciones sobre un puerto.

El impacto que produce cualquier operación de un puerto situado en un eCore externo se puede categorizar en operaciones de lectura (con su escritura de retorno) y de escritura. En la Tabla 6.7 podemos observar el número de operaciones de lectura y de escritura que se harían por la red para las diferentes instrucciones sobre un puerto (quitando la inicialización y la obtención de la dirección de acceso del puerto, ya que no son instrucciones que se repetirían en cada iteración de la tarea). En estos números están incluidas la toma y liberación del mutex.

Además, es importante tener en cuenta que una tarea, cuando ejecuta una operación de lectura, para poder ejecutar la siguiente instrucción tiene que acabar de recibir el dato completo solicitado.

Con el objetivo de obtener una cota superior del tiempo de ejecución de las diferentes operaciones sobre un puerto excluyendo las interferencias producidas por el tráfico en la NoC que se analizan aparte, vamos a definir la latencia aportada por una operación sobre un puerto cualquiera como:

$$L_{Port} = P_K + (TT^b + c_{read}) \cdot N_R \cdot 2 + c_{write} \cdot N_W \tag{6.1}$$

Cada elemento de la Ecuación 6.1 se describe a continuación:

- P_K peor caso de la parte del tiempo sin contención con otras tareas ni con el tráfico en la NoC, que está presente en toda la ejecución de la operación sobre el puerto. Cambia en función del tipo de puerto (SP o QP) y del tipo de operación (escritura o lectura). Los valores de este parámetro para las distintas operaciones se muestran en la Tabla 6.8.

- TT^b tiempo de transmisión de mejor caso entre el origen y el destino de la operación del puerto. puerto para los mensajes de escritura y de respuesta. Se pone como 1 en caso de operación local.

- c_{read} tiempo de ejecución que requiere la lectura de un paquete. Este tiempo incluye las instrucciones requeridas porara realizar la lectura en el eCore local

$EscrituraSP$	$450ns$
$LecturaSP$	$650ns$
$EscrituraQP$	$700ns$
$LecturaQP$	$1450ns$

Tabla 6.8 Valor fijo (P_K) según el puerto y la operación.

más el tiempo requerido por el enrutador remoto para generar el mensaje de respuesta correspondiente. Para las operaciones bajo estudio, este valor es de 41,67 ns.

- c_{write} tiempo de ejecución que requiere una escritura de un paquete. Para las operaciones bajo estudio, este valor es 5 ns.
- N_R número de paquetes de lectura en la operación de acceso a un puerto. Los valores de este parámetro para las distintas operaciones se muestran en la Tabla 6.8.
- N_W número de paquetes de escritura en la operación de acceso a un puerto. Los valores de este parámetro para las distintas operaciones se muestran en la 6.8.

6.4. Modelado de los puertos de muestreo

En esta sección se procederá a modelar el comportamiento de los puertos de muestreo utilizando los elementos de modelado descritos en la Capítulo 3 y su aplicación al procesador Epiphany realizada en el Sección 4.2.

Para poder modelar los puertos de muestreo necesitamos identificar los mensajes generados durante la ejecución de las operaciones de escritura y lectura en un SP. Es importante destacar que estos mensajes solo serán generados si el SP está alojado en la memoria local de un núcleo diferente del que está realizando la operación.

Los mensajes identificados afectarán al modelo en los tres aspectos siguientes: (a) cálculo de la ratio acumulada en los enlaces de la NoC, (b) incremento del tiempo de ejecución de peor caso de las tareas por la interferencia sufrida por sus mensajes de lectura y respuesta, y (c) inclusión de un bloque de retardo para modelar el tiempo de travesía del último paquete del mensaje por la NoC.

Listado 6.5 Pseudocódigo de la operación de escritura sobre un SP.

```
Lock the mutex                (R)
Write the data                (W)
Set the data as new           (W)
Unlock the mutex              (W)
```

El pseudocódigo de una operación de escritura en un SP se muestra en el Listado 6.5, indicando además los mensajes de lectura (R) y escritura (W) generados por cada instrucción. Como se describe en la Sección 4.1.6, la toma de un `mutex` requiere un mensaje de lectura (junto al mensaje de respuesta generado). El resto de las instrucciones en el Listado 6.5 generarán mensajes de escritura. De esta manera, para poder considerar la situación de peor caso, el valor medido en aislamiento para el tiempo de ejecución de peor caso de una tarea que escribe en el SP localizado en la memoria local de otro núcleo debe ser incrementado en la duración de las interferencias sufridas por los mensajes de lectura y de respuesta conforme con la Ecuación 3.6 (con sus interferencias calculadas usando la Ecuación 4.4 y la Ecuación 4.5).

Respecto a la limitación de ratio máxima, el mensaje de lectura (de un solo paquete) tendrá una ratio que se calculará mediante la Ecuación 3.1. Entre los mensajes de escritura, seleccionamos el de mayor ratio. En caso de tener un dato con más de un paquete (64 bits), el compilador utilizará la función `memcopy` lo que supone una ratio de generación de paquetes de $\frac{1}{3}ciclos^{-1}$. En caso de que el dato quepa en un solo paquete, hemos medido una ratio de generación para los tres mensajes de escritura mostrados en el Listado 6.5 de $\frac{1}{7}ciclos^{-1}$.

Listado 6.6 Pseudocódigo de la operación de lectura de un SP.

```
Lock the mutex                (R)
Read the data                 (R)
Read if the data is new       (R)
Unlock the mutex              (W)
```

El pseudocódigo de una operación de lectura sobre un SP se muestra en el Listado 6.6. Como ocurría en la operación de escritura, el tiempo de ejecución de peor caso de una tarea que ejecute la operación de lectura tiene que verse incrementado por el tiempo de interferencia sufrido por los mensajes de lectura (y sus correspondientes mensajes de respuesta). La única diferencia en este caso es que el número de mensajes de lectura depende del tamaño del dato a leer.

Respecto a la limitación de ratio máxima, la ratio de cada mensaje de lectura se calcula con la Ecuación 4.1. Viendo el código ensamblador que genera el compilador y

la medición del retardo del enrutador externo al leer el dato y generar el mensaje de respuesta correspondiente se puede deducir que el menor intervalo entre la generación de dos mensajes de lectura consecutivos (parámetro c en la Ecuación 4.1) es de 25 ciclos. El mensaje de escritura utilizado para desbloquear el `mutex` es de un paquete de longitud por lo que su ratio puede ser calculada utilizando la Ecuación 3.1. La ratio de los paquetes de respuesta generados por las peticiones de lectura se calcula utilizando la Ecuación 4.2 con el mismo valor de c que el paquete de petición de lectura (25 ciclos).

El modelado de las operaciones de escritura y lectura en un SP también requiere incluir un elemento de retardo con el tiempo de travesía de mejor y peor caso para el mensaje de escritura generado para la liberación del `mutex` (la última instrucción en el Listado 6.5 y Listado 6.6). Los tiempos de travesía de mejor y peor caso de este mensaje se calcularán utilizando la Ecuación 3.2 y la Ecuación 3.5 respectivamente.

Finalmente, es necesario modelar el tiempo de bloqueo debido a la exclusión mutua producida al utilizar el SP (debido al mutex que protege el acceso a sus operaciones). Cada tarea que utilice un SP puede sufrir un tiempo de bloqueo que, en el peor caso, es igual a la duración de la mayor operación en el SP de entre las realizadas por tareas en otros eCores, ya que modelamos flujos e2e lineales. Este valor se puede medir en aislamiento o utilizar las fórmulas descritas en la Sección 6.3.3 y, para considerar el tiempo de peor caso, se tiene que aumentar con las posibles interferencias producidas por las lecturas y las respuestas de la utilización del SP (calculadas utilizando la Ecuación 4.4 y la Ecuación 4.5). Dado que el núcleo está detenido durante este tiempo de bloqueo, se tienen que sumar el tiempo de bloqueo al tiempo de ejecución de peor caso de las tareas que utilicen el SP (medido en aislamiento) para contemplar así el escenario de peor caso.

Un ejemplo del modelado de dos tareas utilizando un SP se presenta en la Sección 6.6.

6.5. Modelado de puertos con cola

El modelado de las operaciones de escritura y de lectura en un puerto con cola, cuyos pseudocódigos se pueden ver en el Listado 6.7 y Listado 6.8 respectivamente, es muy similar al modelado de los puertos de muestreo expuesto en la sección anterior.

El tiempo de ejecución de peor caso de las tareas que utilicen un QP (típicamente medido en aislamiento) debe verse aumentado en los tiempos de las interferencias de todos los mensajes de lectura y de respuesta conforme con la Ecuación 3.6 (con las interferencias calculadas utilizando las Ecuación 4.4 y la Ecuación 4.5).

Listado 6.7 Pseudocódigo de la operación de escritura de un puerto con cola

```
Lock the mutex                (R)
If QP is not full             (R)
  Allocate the QP slot        (R&W)
  Write the data              (W)
Else
  Set "full QP" error
End if
Unlock the mutex              (W)
```

También necesitamos obtener la ratio de generación de paquetes de los mensajes. Considerando los mensajes de escritura de la operación de escritura del QP (Listado 6.7), la ratio mayor corresponde al mensaje generado para escribir el dato (en caso de que el tamaño del dato sea mayor de un paquete). En esta situación, el compilador utilizará la función `memcopy` que produce una ratio de generación de paquetes de $\frac{1}{3}ciclos^{-1}$. En caso de que el dato cupiera en un solo paquete, hemos medido la ratio de generación de los paquetes de los tres mensajes de escritura como $\frac{1}{8}ciclos^{-1}$. La ratio máxima de generación de los tres mensajes de lectura en el Listado 6.7 se puede obtener aplicando la Ecuación 4.1 con un valor de 25 ciclos para el parámetro c (el valor de c ha sido obtenido mediante la inspección del código ensamblador generado por el compilador y la medición del retardo del enrutador externo al leer el dato y generar el mensaje de respuesta correspondiente). De igual manera se puede calcular la ratio de generación máxima del mensaje de respuesta utilizando la Ecuación 4.2.

Listado 6.8 Pseudocódigo de la operación de lectura de un puerto con cola

```
Lock the mutex                (R)
If QP is not empty            (R)
  Get QP position             (R)
  Read the data               (R)
  Free the  QP slot           (W)
  Unlock the mutex            (W)
Else
  Unlock the mutex            (W)
  Block the task
End if
```

Respecto a las operaciones de lectura de los QP (Listado 6.8), hemos medido la ratio de generación de paquetes de los tres mensajes de escritura (de un paquete cada

uno) siendo de $\frac{1}{4}ciclos^{-1}$. De igual manera, la ratio de generación máxima de los tres mensajes de lectura se calcula utilizando la Ecuación 4.1 con un valor del parámetro c de 25 ciclos cuando el dato a leer es de más de un paquete y de 44 ciclos cuando cabe en un solo paquete. De igual manera, podemos calcular Íla ratio de generación máxima de los mensajes de respuesta utilizando la Ecuación 4.2.

Al igual que con los SPs, se tiene que incluir un retardo para modelar el tiempo de travesía de los mensajes de escritura generados al liberar el `mutex` (al ser la última instrucción tanto en el Listado 6.7 como en Listado 6.8). Los tiempos de travesía de mejor y peor caso de este mensaje se calculan utilizando la Ecuación 3.2 y la Ecuación 3.5 respectivamente.

También tenemos que modelar el tiempo de bloqueo debido a la exclusión mutua producida al utilizar el QP (debido al `mutex` que protege el acceso a sus operaciones). Como sucede con los SPs, la operación más larga sobre un QP puede ser medida en aislamiento, o mediante las fórmulas descritas en la Sección 6.3.3, y debe aumentarse con los tiempos de interferencia de todos los mensajes de lectura y respuesta generados (calculados utilizando la Ecuación 4.4 y la Ecuación 4.5). A los tiempos de ejecución de peor caso de las tareas que utilicen un QP se les tiene que sumar el valor del tiempo de bloqueo calculado para completar así el escenario de peor caso.

Un ejemplo de modelado de un sistema complejo utilizando QPs se expone en la Sección 6.7.

6.6. Ejemplo de modelado con un puerto de muestreo

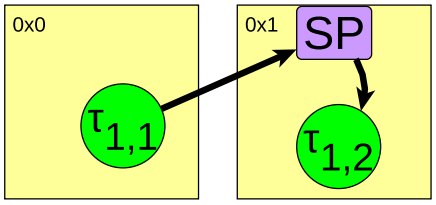

Figura 6.3 Diagrama del ejemplo con un puerto de muestreo

A modo de ejemplo del modelado de un puerto de muestreo, en esta sección se muestra el modelado de una aplicación en la que una tarea, τ_{12}, necesita leer el dato generado por otra tarea, τ_{11}, para realizar unos cálculos. En la Figura 6.3 se muestra

Γ_1	$(T_1 = 1ms \quad D_1 = 1ms)$
τ_{11}	$C_{11} = 5\mu s \quad C^b_{11} = 4\mu s \quad C'_{11} = 5{,}43\mu s \quad Prio_{11} = 3$
m_{111}	$\mu_{111} = 1 \quad t_{ijk} = read \quad \rho_{111} = 1{,}67\mu ciclos^{-1}$
m_{112}	$\mu_{111} = 1 \quad t_{ijk} = respuesta \quad \rho_{112} = 1{,}67\mu ciclos^{-1}$
m_{113}	$\mu_{113} = 2 \quad t_{ijk} = write \quad \rho_{113} = 0{,}33 ciclos^{-1}$
m_{114}	$\mu_{114} = 1 \quad t_{ijk} = write \quad \rho_{111} = 0{,}14 ciclos^{-1}$
m_{115}	$\mu_{114} = 1 \quad t_{ijk} = write \quad \rho_{111} = 0{,}14 ciclos^{-1}$
τ_{12}	$C'_{12} = 3{,}35\mu s \quad C_{12} = 3\mu s \quad C^b_{12} = 2\mu s \quad Prio_{11} = 3$

Tabla 6.9 Parámetros del ejemplo con un puerto de muestreo.

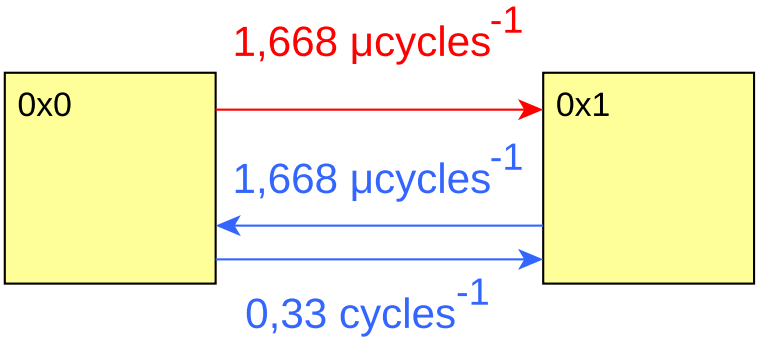

Figura 6.4 Ratios de los enlaces de la NoC en el ejemplo con un puerto de muestreo.

una representación gráfica de la aplicación de ejemplo, mientras que los parámetros del mismo ejemplo pueden verse en la Tabla 6.9.

La tarea τ_{11} actualiza el dato formado por dos paquetes almacenados en el puerto de muestreo localizado en el núcleo 0x1. La tarea τ_{12} ejecutará una operación de lectura sobre el puerto de muestreo con un periodo de muestreo $T_{poll} = 1{,}5\mu s$. Si el dato es nuevo, τ_{12} ejecutará su código durante un tiempo de ejecución de C_{12}.

Los mensajes de la Tabla 6.9 se corresponden con los que se identificaron como parte de una operación de escritura de un SP mostrada en el Listado 6.1. Los mensajes de lectura y respuesta (m_{111} y m_{112} respectivamente) son de un paquete de longitud y, como consecuencia, su ratio de generación de paquetes se calcula utilizando la Ecuación 4.1. El mensaje de escritura que escribe el dato, m_{113}, tiene la ratio de generación de la función `memcopy` ($\frac{1}{3} ciclos^{-1}$). Por otro lado, los mensajes de escritura utilizados para poner el dato como nuevo y liberar el mutex son ambos de un paquete de longitud los cuales, considerándolos un grupo, tienen una ratio de generación de $\frac{1}{8} ciclos^{-1}$. La ratio de los enlaces de la NoC se muestra en la Figura 6.4, donde se puede ver que se verifican todas las restricciones de ratio máximas.

Flujo Γ_1

Actividad	WCRT	BCRT
τ_{11}	5430 ns	5000 ns
τ_{12}	8985 ns	8205 ns

Tabla 6.10 Resultados del modelo MAST para el ejemplo con un puerto de muestreo.

Flujo Γ_1

Actividad	WCRT	Avg RT	BCRT
τ_{11}	5208 ns	5048 ns	4,968 ns
τ_{12}	8703 ns	8225 ns	7,880 ns

Tabla 6.11 Resultados de las ejecuciones en Epiphany del ejemplo con un puerto de muestreo.

En este sencillo ejemplo en el que solo hay un flujo e2e, los mensajes de lectura y escritura (m_{111} y m_{112}) no sufren ninguna interferencia de otros paquetes por lo que el tiempo de ejecución de peor caso de la tarea τ_{11} no tiene que ser aumentado. Sin embargo, el tiempo de ejecución medido para ambas tareas puede verse incrementado por el tiempo de bloqueo debido a la exclusión mutua en el uso del SP. En este caso, ambas tareas ejecutan una operación diferente: τ_{11} escribe dos paquetes en el SP, operación que requiere 346.67 ns, mientras τ_{12} lee dos paquetes del SP, operación que requiere 428.34 ns. Como consecuencia, los tiempos medidos de τ_{11} y τ_{12} (C_{11} y C_{12} respectivamente) han visto incrementados sus valores a $C'_{11} = 5{,}43\mu s$ y $C'_{12} = 3{,}35\mu s$.

El mecanismo de muestreo utilizado por τ_{12} para comprobar si ha llegado el mensaje al SP se modela en MAST especificando una política de planificación de muestreo para la tarea τ_{12}, con un periodo de muestreo de $1{,}5\mu s$ (determinado por el usuario) y una sobrecarga de muestreo que tiene que ser medida para cada implementación. La sobrecarga de muestreo es el tiempo necesario para comprobar la llegada del mensaje al SP.

Además, se añade un retardo para modelar el tiempo de travesía a través de la NoC del último paquete del último mensaje (m_{115}). El tiempo de travesía de mejor caso de este paquete TT^b_{115}, se calcula utilizando la Ecuación 3.2 con un $L_H = 1{,}5 ciclos = 2{,}5 ns$ y un $H_{115} = 2$. Como en este ejemplo simple no se producen interferencias, el tiempo de travesía de peor caso, TT_{115}, es igual a TT^b_{115}.

La Figura 6.5 muestra los elementos resultantes del modelo MAST, la Tabla 6.10 muestra los tiempo de respuesta obtenidos utilizando MAST y la Tabla 6.11 muestra los tiempos de respuesta de la ejecución en el procesador Epiphany de tareas sintéticas con los mismos parámetros. Como se puede ver, los tiempos de respuesta de la aplicación

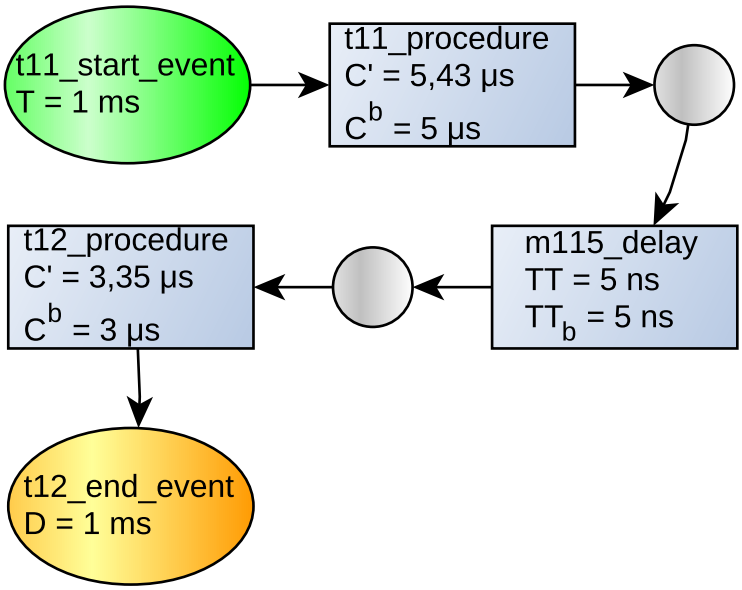

Figura 6.5 Modelo MAST para el ejemplo con un puerto de muestreo.

sintética son menores y muy próximos a los resultados de peor caso del análisis del modelo MAST.

6.7. Ejemplo de modelado de puertos con cola

Utilizar puertos con cola para implementar flujos *e2e* puede resultar más apropiado en la mayoría de los casos debido a que el comportamiento bloqueante de las operaciones de lectura evita la necesidad de utilizar el muestreo periódico.

La Figura 6.6 muestra un sistema con tres flujos *e2e*. Los parámetros de estos tres flujos se muestran en los Tablas 6.12 y 6.13. Para mostrar el proceso de modelado nos vamos a centrar en el flujo *e2e* Γ_2 y, en particular, en la tarea τ_{22}. Los mensajes generados por τ_{22} (de m_{221} a m_{229}) corresponden con los identificados en el Listado 6.7.

Los mensajes de lectura (m_{221}, m_{223} y m_{225}) viajan a través de tres enrutadores teniendo un TT^b de 4,5*ciclos* (calculado usando la Ecuación 3.2) y, puesto que comparten un enlace con el mensaje de lectura generado por τ_{11}, su TT es de 12,5*ciclos* (de acuerdo con la Ecuación 3.5, utilizando la Ecuación 4.4 para obtener el término de la interferencia). El tiempo de travesía de los mensajes de respuesta (m_{222}, m_{224} y m_{226}) se calcula de manera similar (utilizando le Ecuación 4.5, en lugar de la Ecuación 4.4, para obtener el término de la interferencia) obteniendo los valores $TT^b = 4{,}5ciclos$

Γ_1	$(T_1 = 1ms \quad D_1 = 1ms)$
τ_{11}	$C'_{11} = 36{,}35\mu s \quad C_{11} = 35\mu s \quad C^b_{11} = 34\mu s$
m_{111}	$\mu_{111} = 1 \quad t_{111} = read \quad \rho_{111} = 83{,}33mciclos^{-1}$
m_{112}	$\mu_{112} = 1 \quad t_{112} = respuesta \quad \rho_{112} = 83{,}34mciclos^{-1}$
m_{113}	$\mu_{113} = 1 \quad t_{113} = read \quad \rho_{113} = 83{,}33mciclos^{-1}$
m_{114}	$\mu_{114} = 1 \quad t_{114} = respuesta \quad \rho_{114} = 83{,}34mciclos^{-1}$
m_{115}	$\mu_{115} = 1 \quad t_{115} = read \quad \rho_{115} = 83{,}33mciclos^{-1}$
m_{116}	$\mu_{116} = 1 \quad t_{116} = respuesta \quad \rho_{116} = 83{,}34mciclos^{-1}$
m_{117}	$\mu_{117} = 1 \quad t_{117} = write \quad \rho_{117} = 58{,}82mciclos^{-1}$
m_{118}	$\mu_{118} = 3 \quad t_{118} = write \quad \rho_{118} = 0{,}33ciclos^{-1}$
m_{119}	$\mu_{119} = 1 \quad t_{119} = write \quad \rho_{119} = 58{,}82mciclos^{-1}$
τ_{12}	$C'_{12} = 27{,}24\mu s \quad C_{12} = 23\mu s \quad C^b_{12} = 25\mu s$
m_{121}	$\mu_{121} = 1 \quad t_{121} = read \quad \rho_{121} = 0{,}11ciclos^{-1}$
m_{122}	$\mu_{122} = 1 \quad t_{122} = respuesta \quad \rho_{122} = 0{,}11ciclos^{-1}$
m_{123}	$\mu_{123} = 1 \quad t_{123} = read \quad \rho_{123} = 0{,}11ciclos^{-1}$
m_{124}	$\mu_{124} = 1 \quad t_{124} = respuesta \quad \rho_{124} = 0{,}11ciclos^{-1}$
m_{125}	$\mu_{125} = 1 \quad t_{125} = read \quad \rho_{125} = 0{,}11ciclos^{-1}$
m_{126}	$\mu_{126} = 1 \quad t_{126} = respuesta \quad \rho_{126} = 0{,}11ciclos^{-1}$
m_{127}	$\mu_{127} = 1 \quad t_{127} = write \quad \rho_{127} = 71{,}43mciclos^{-1}$
m_{128}	$\mu_{128} = 5 \quad t_{128} = write \quad \rho_{128} = 0{,}33ciclos^{-1}$
m_{129}	$\mu_{129} = 1 \quad t_{129} = write \quad \rho_{129} = 71{,}43mciclos^{-1}$
τ_{13}	$C'_{13} = 15{,}73\mu s \quad C_{13} = 15\mu s \quad C^b_{13} = 14\mu s$

Γ_2	$(T_2 = 1ms \quad D_2 = 1ms)$
τ_{21}	$C'_{21} = 21{,}44\mu s \quad C_{21} = 20\mu s \quad C^b_{21} = 19\mu s$
m_{211}	$\mu_{211} = 1 \quad t_{211} = read \quad \rho_{211} = 0{,}11ciclos^{-1}$
m_{212}	$\mu_{212} = 1 \quad t_{212} = respuesta \quad \rho_{212} = 0{,}11ciclos^{-1}$
m_{213}	$\mu_{213} = 1 \quad t_{213} = read \quad \rho_{213} = 0{,}11ciclos^{-1}$
m_{214}	$\mu_{214} = 1 \quad t_{214} = respuesta \quad \rho_{214} = 0{,}11ciclos^{-1}$
m_{215}	$\mu_{215} = 1 \quad t_{215} = read \quad \rho_{215} = 0{,}11ciclos^{-1}$
m_{216}	$\mu_{216} = 1 \quad t_{216} = respuesta \quad \rho_{216} = 0{,}11ciclos^{-1}$
m_{217}	$\mu_{217} = 1 \quad t_{217} = write \quad \rho_{217} = 71{,}43mciclos^{-1}$
m_{218}	$\mu_{218} = 7 \quad t_{218} = write \quad \rho_{218} = 0{,}33ciclos^{-1}$
m_{219}	$\mu_{219} = 1 \quad t_{219} = write \quad \rho_{219} = 71{,}43mciclos^{-1}$
τ_{22}	$C'_{22} = 27{,}47\mu s \quad C_{22} = 25\mu s \quad C^b_{22} = 23\mu s$
m_{221}	$\mu_{221} = 1 \quad t_{221} = read \quad \rho_{221} = 83{,}33mciclos^{-1}$
m_{222}	$\mu_{212} = 1 \quad t_{222} = respuesta \quad \rho_{222} = 83{,}34mciclos^{-1}$
m_{223}	$\mu_{223} = 1 \quad t_{223} = read \quad \rho_{223} = 83{,}33mciclos^{-1}$
m_{224}	$\mu_{224} = 1 \quad t_{224} = respuesta \quad \rho_{224} = 83{,}34mciclos^{-1}$
m_{225}	$\mu_{225} = 1 \quad t_{225} = read \quad \rho_{225} = 83{,}33mciclos^{-1}$
m_{226}	$\mu_{226} = 1 \quad t_{226} = respuesta \quad \rho_{226} = 83{,}34mciclos^{-1}$
m_{227}	$\mu_{227} = 1 \quad t_{227} = write \quad \rho_{227} = 58{,}82mciclos^{-1}$
m_{228}	$\mu_{228} = 11 \quad t_{228} = write \quad \rho_{228} = 0{,}33ciclos^{-1}$
m_{229}	$\mu_{229} = 1 \quad t_{229} = write \quad \rho_{229} = 58{,}82mciclos^{-1}$
τ_{23}	$C'_{23} = 10{,}94\mu s \quad C_{23} = 10\mu s \quad C^b_{23} = 9\mu s$

Tabla 6.12 Parámetros de los flujos Γ_1 y Γ_2 del ejemplo de los puertos con cola con tres flujos *e2e*.

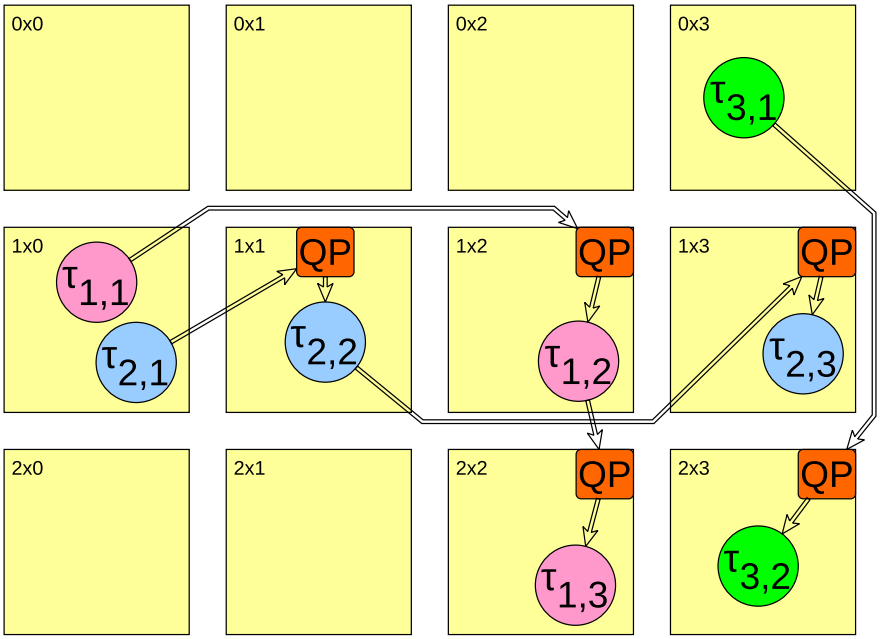

Figura 6.6 Sistema del ejemplo de los puertos con cola.

y $TT = 5{,}5 ciclos$. Los tiempos de travesía de los mensajes de escritura (m_{227}, m_{228} y m_{229}) también se calculan utilizando la Ecuación 3.5 y la Ecuación 4.5 obteniendo así los valores $TT^b = 4{,}5 ciclos$ y $TT = 5{,}5 ciclos$. A modo de ejemplo, se muestran a continuación los cálculos de los tiempos de travesía de un paquete correspondiente a un mensaje de lectura (m_{221}), a un mensaje de respuesta (m_{222}) y a un mensaje de escritura (m_{227}):

$$TT^b_{221} = L_H \cdot H = 1{,}5 \cdot 3 = 4{,}5 ciclos$$

$$TT_{221} = TT^b_{221} + I_{221} = 4{,}5 + 8 = 12{,}5 ciclos$$

$$TT^b_{222} = L_H \cdot H = 1{,}5 \cdot 3 = 4{,}5 ciclos$$

$$TT_{222} = TT^b_{222} + I_{222} = 4{,}5 + 1 = 5{,}5 ciclos$$

$$TT^b_{227} = L_H \cdot H = 1{,}5 \cdot 3 = 4{,}5 ciclos$$

$$TT_{227} = TT^b_{227} + I_{227} = 4{,}5 + 1 = 5{,}5 ciclos$$

Una vez hemos calculado los tiempos de travesía de los paquetes podemos obtener la ratio de generación de paquetes, que para los mensajes de lectura y de respuesta se calcula utilizando la Ecuación 4.1 con un valor de 25 ciclos para el parámetro c (como se ha descrito en la Sección 6.5) y el tiempo de travesía obtenido en el párrafo anterior.

Γ_3	$T_3 = 1ms \quad D_3 = 1ms$
τ_{31}	$C'_{31} = 21{,}69\mu s \quad C_{31} = 20\mu s \quad C^b_{31} = 19\mu s$
m_{311}	$\mu_{311} = 1 \quad t_{311} = read \quad \rho_{311} = 83{,}33mciclos^{-1}$
m_{312}	$\mu_{312} = 1 \quad t_{312} = respuesta \quad \rho_{312} = 83{,}34mciclos^{-1}$
m_{313}	$\mu_{313} = 1 \quad t_{313} = read \quad \rho_{313} = 83{,}33mciclos^{-1}$
m_{314}	$\mu_{314} = 1 \quad t_{314} = respuesta \quad \rho_{314} = 83{,}34mciclos^{-1}$
m_{315}	$\mu_{315} = 1 \quad t_{315} = read \quad \rho_{315} = 83{,}33mciclos^{-1}$
m_{316}	$\mu_{316} = 1 \quad t_{316} = respuesta \quad \rho_{316} = 83{,}34mciclos^{-1}$
m_{317}	$\mu_{317} = 1 \quad t_{317} = write \quad \rho_{317} = 58{,}82mciclos^{-1}$
m_{318}	$\mu_{318} = 13 \quad t_{318} = write \quad \rho_{318} = 0{,}33ciclos^{-1}$
m_{319}	$\mu_{319} = 1 \quad t_{319} = write \quad \rho_{319} = 58{,}82mciclos^{-1}$
τ_{32}	$C'_{32} = 50{,}9\mu s \quad C_{32} = 50\mu s \quad C^b_{32} = 47\mu s$

Tabla 6.13 Parámetros de los flujos Γ_3 del ejemplo de los puertos con cola con tres flujos *e2e*.

El cálculo de las ratios de generación de paquetes de un mensaje de lectura (m_{221}) y de un mensaje de respuesta (m_{222}) se muestran a continuación:

$$\rho_{221} = \frac{1}{TT^b_{221} + TT^b_{222} + c} = \frac{1}{4{,}5 + 4{,}5 + 25} = \frac{1}{34} ciclos^{-1}$$

$$\rho_{222} = \frac{1}{TT^b_{222} + TT^b_{221} + c} = \frac{1}{4{,}5 + 4{,}5 + 25} = \frac{1}{34} ciclos^{-1}$$

Respecto a la ratio de generación de paquetes de un mensaje de escritura, el mensaje más restrictivo es m_{113} (el mensaje que escribe el dato en el QP) ya que tiene la ratio de generación de la función `memcopy` ($\frac{1}{3} ciclos^{-1}$). Por otro lado, los mensajes de escritura para indicar que el dato es nuevo y para liberar el `mutex` son de un paquete de longitud y, de manera conjunta, tienen una ratio de generación de $\frac{1}{8} ciclos^{-1}$.

Las ratios de los enlaces de la NoC se muestran en la Figura 6.7, la cual incluye la ratio de generación de paquetes de todos los mensajes del sistema. Como se puede ver, todos los enlaces cumplen la restricción de ratio máxima, ya que los enlaces de la red *cMesh* (flechas azules) deben tener una ratio menor o igual que uno mientras que los enlaces de la red *rMesh* (flechas rojas) deben tener una ratio menor o igual que $\frac{1}{8}$.

El tiempo de ejecución de peor caso de la tarea τ_{22} tiene que aumentarse en el tiempo de bloqueo máximo debido al uso compartido de QPs con las tareas τ_{21} y τ_{23}. El tiempo de ejecución de la operación de lectura ejecutada por τ_{23} en su QP local, se ha medido en su peor caso (añadiendo la interrupción del reloj) y conlleva un tiempo $B_{23} = 1602ns$. Por otro lado, la operación de escritura ejecutada por la tarea τ_{21} en el QP del núcleo 1x1 se ha medido en su peor caso (añadiendo la interrupción del reloj) y

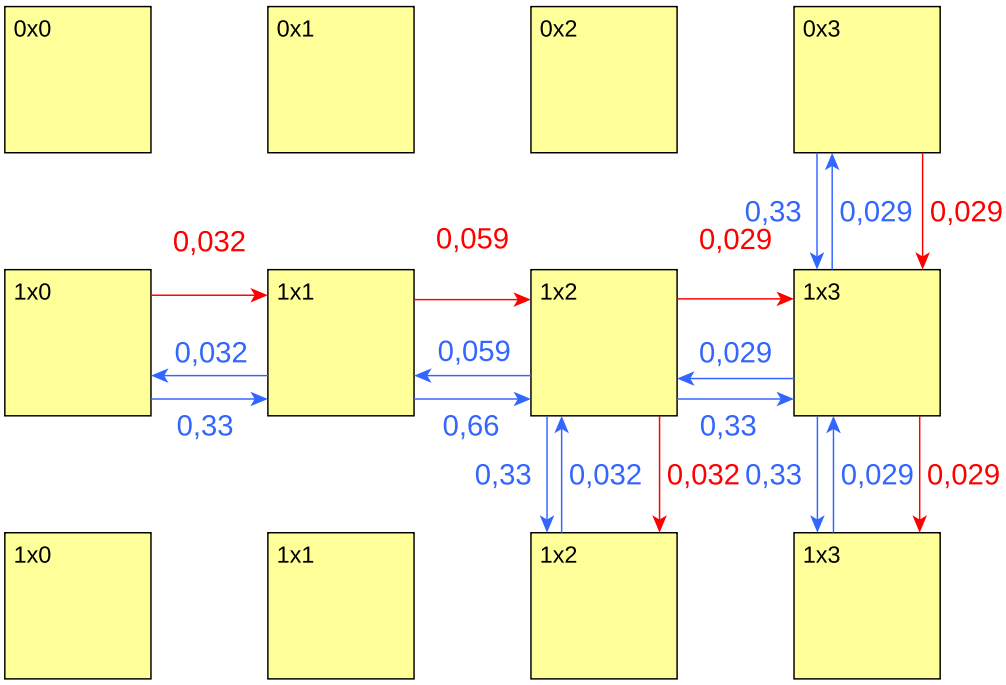

Figura 6.7 Ratios de los enlaces de la NoC en el ejemplo utilizado para los puertos con cola (mensajes de lectura en rojo y mensajes de escritura y respuesta en azul)

necesita un tiempo $B_{21} = 780ns$. Con el objetivo de tener en cuenta la peor situación posible, a B_{21} se le tendrían que sumar los tiempos de interferencia de los paquetes de lectura y respuesta, aunque, en este caso concreto, este valor de interferencia es cero puesto que los mensajes que viajan desde τ_{21} hasta τ_{22} no comparten ningún enlace con otros paquetes de la NoC:

$$B'_{21} = B_{21} + I_{211} + I_{212} + I_{213} + I_{214} + I_{215} + I_{216}$$

$$B'_{21} = 780ns + 0ns = 780ns$$

Además, para llegar a considerar la situación de peor caso, se tiene que sumar los tiempos de interferencia de los paquetes tanto de lectura como de respuesta al tiempo de ejecución de peor caso de la tarea τ_{22} como se muestra a continuación:

$$C'_{22} = C_{22} + B_{23} + B'_{21} + I_{221} + I_{222} + I_{223} + I_{224} + I_{225} + I_{226}$$

$$C'_{22} = 25\mu s + 1602ns + 780ns + (25{,}33ns + 2{,}5ns) \cdot 3ns$$

$$C'_{22} = 27{,}47\mu s$$

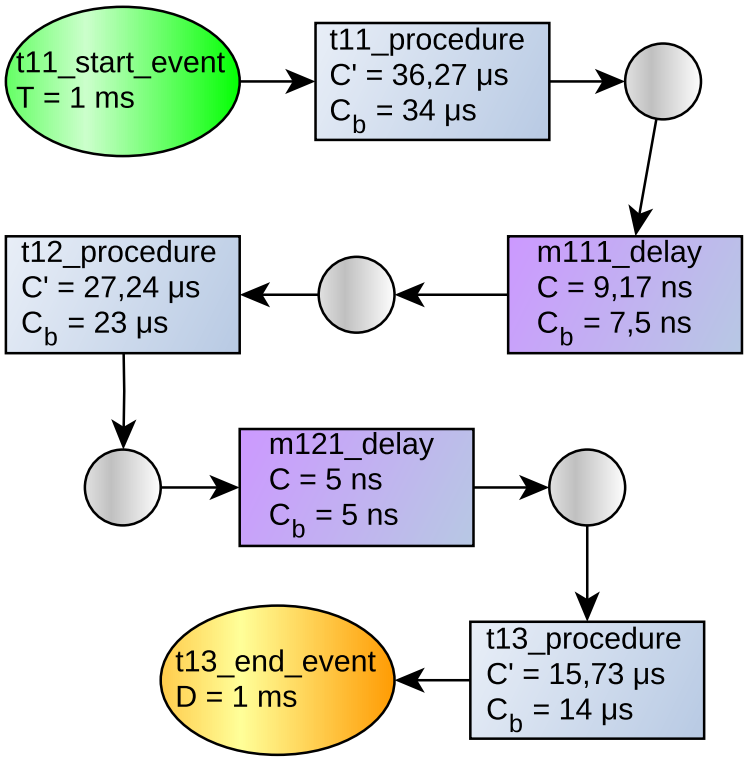

Figura 6.8 Primera transacción modelada en MAST.

El mismo proceso se sigue con el resto de mensajes y tareas dando como resultado los valores mostrados como C'_{ij} en la Tabla 6.12.

Las Figuras 6.8, 6.9 y 6.10 muestran los elementos MAST correspondientes al modelo del ejemplo planteado. La Tabla 6.14 muestra los tiempos de respuesta del procesador muchos núcleos Epiphany ejecutando tareas sintéticas con la misma configuración del ejemplo. Puede verse que los tiempos de respuesta de la aplicación sintética son menores y muy próximos a los obtenidos con el análisis del modelo MAST.

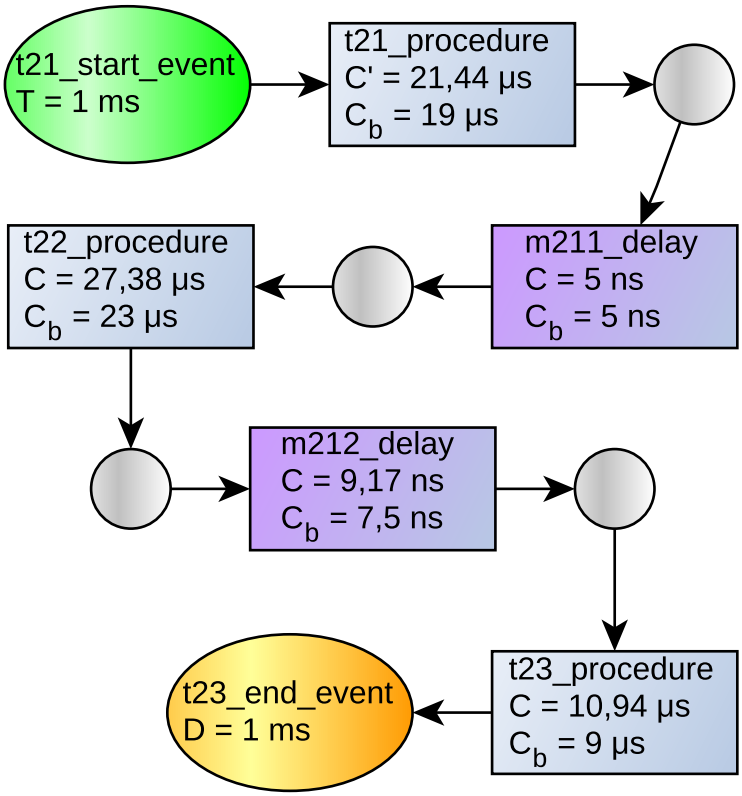

Figura 6.9 Segunda transacción modelada en MAST.

Flujo Γ_1

Actividad	WCRT	BCRT
τ_{11}	57779 ns	34000 ns
τ_{12}	85039 ns	57008 ns
τ_{13}	100774 ns	71013 ns

Flujo Γ_2

Actividad	WCRT	BCRT
τ_{21}	57790 ns	19000 ns
τ_{22}	85235 ns	42005 ns
τ_{23}	96184 ns	51013 ns

Flujo Γ_3

Actividad	WCRT	BCRT
τ_{31}	21690 ns	19000 ns
τ_{32}	72598 ns	66008 ns

Tabla 6.14 Resultados del modelo MAST para el ejemplo de puertos con cola.

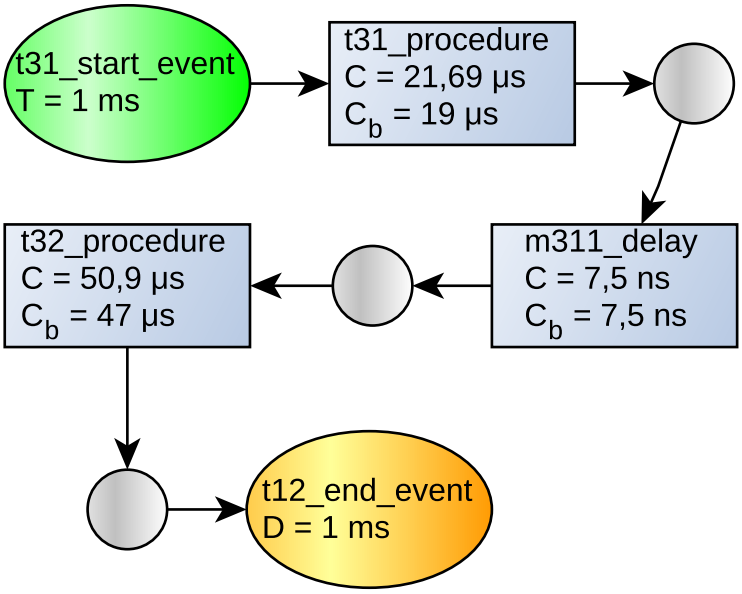

Figura 6.10 Tercera transacción modelada en MAST.

Flujo Γ_1

Actividad	Max RT	Avg RT	Min RT
τ_{11}	35305 ns	35247 ns	35218 ns
τ_{12}	71015 ns	63343 ns	59458 ns
τ_{13}	84990 ns	77308 ns	73410 ns

Flujo Γ_2

Actividad	Max RT	Avg RT	Min RT
τ_{21}	20172 ns	20170 ns	20170 ns
τ_{22}	55698 ns	47943 ns	43967 ns
τ_{23}	64113 ns	56350 ns	52377 ns

Flujo Γ_3

Actividad	Max RT	Avg RT	Min RT
τ_{31}	20007 ns	20003 ns	20003 ns
τ_{32}	67868 ns	67865 ns	67862 ns

Tabla 6.15 Resultados de las ejecuciones del ejemplo de puertos con cola en el Epiphany.

Capítulo 7

Generador automático

Durante el desarrollo del sistema operativo M2OS-mc expuesto en el Capítulo 5 y de los mecanismos de sincronización presentados en el Capítulo 6 ha sido necesario realizar múltiples aplicaciones de prueba para comprobar el correcto funcionamiento del código desarrollado, medir los tiempos de respuesta de las aplicaciones y medir los tiempos de ejecución de los servicios implementados.

También ha sido necesario desarrollar aplicaciones complejas para validar el modelo desarrollado en el Capítulo 3 y adaptado al procesador Epiphany en el Capítulo 4. Una parte de la validación del modelo ha consistido en la comparación de los resultados obtenidos por el análisis MAST con los tiempos de respuesta medidos en la ejecución de una aplicación sintética con la misma configuración que el sistema modelado.

Para facilitar la generación de aplicaciones de prueba se ha desarrollado una herramienta que, partiendo de una definición de alto nivel de una aplicación, genera el fichero MAST que la modela y una implementación de la misma basada en tareas con carga de ejecución sintética.

La citada herramienta también podría ser utilizada para generar el esqueleto de una aplicación, liberando a los desarrolladores de la tediosa y propensa a errores labor de creación e inicialización de tareas y puertos de sincronización.

7.1. Fichero de entrada

Como ya hemos dicho con anterioridad el generador utiliza un fichero de entrada que define la aplicación en términos de flujos e2e, tareas y puertos. El formato de este fichero se muestra en el Listado 7.1.

Listado 7.1 Formato del fichero de entrada del generador

```
Ammount_of_tasks: N_Tasks
Number_Of_Iterations: N_Iterations
Precision: ns/ms

Task (
  Name              => name,
  Core_Name         => e_X_Y,
  WCET              => C,
  BCET              => Cb,
  Period            => P,
  Deadline          => D,
  Wait_Period       => Wp,
  Num_Packets       => N,
  Send_Ratio        => r,
  Port              => sampling/queuing,
  Port_Size         => N_Messages_Port,
  Next_Task         => dest,
  Priority          => task_priority,
  Role              => init/mid/last)
```

A continuación se procede a describir cada uno de los parámetros incluidos en el fichero. En las tres primeras líneas nos encontramos los parámetros generales del sistema, que solo se deben poner una vez por cada fichero. En las siguientes líneas podemos ver la definición de una tarea. Ese bloque se debe repetir por cada tarea que se quiera incluir en el sistema.

Como parámetros generales del sistema se debe especificar:

- **Ammount_of_tasks** El número de tareas que componen la aplicación.
- **Number_Of_Iterations** El número de iteraciones que deben realizar las tareas. Al final de la ejecución, la aplicación mostrará para cada tarea sus tiempos de mejor caso, peor caso y promedio en relación a la activación de la primera tarea del flujo de tareas al que pertenece.
- **Precision** Indica si las medidas temporales se quieren hacer en nanosegundos o en microsegundos.

Por cada tarea del sistema se tiene que definir:

- **Name** El nombre de la tarea, que tiene que ser unívoco.

- **Core_Name** El nombre del núcleo: se utilizan las coordenadas empezando por 0. Ejemplo: e_2x1 es el eCore situado en la tercera fila segunda columna.
- **WCET** Tiempo de ejecución de peor caso en ns.
- **BCET** Tiempo de ejecución de mejor caso en ns.
- **Period** Periodo de la tarea en ns. Solo se usará en la primera tarea del flujo e2e.
- **Deadline** Plazo de la tarea en ns. Sólo se pone en la primera tarea del flujo e2e, pero es el plazo de todo el flujo y se compara con el tiempo de respuesta de la última tarea del flujo.
- **Wait_Period** Periodo de muestreo en nanosegundos. Este periodo se utiliza en el bucle de espera a que se inicialice el puerto al que enviar los datos o en el bucle de espera de espera a que llegue un dato nuevo a un puerto de muestreo.
- **Num_Packets** Tamaño del dato enviado a la tarea destino (medido en número de paquetes de 8 bytes).
- **Send_Ratio** Ratio de envío de paquetes (medido en $ciclos^{-1}$).
- **Port** Indica el tipo de puerto sobre el que se va a escribir.
- **Port_Size** Número de datos que es capaz de almacenar el puerto utilizado. Solo utilizado para puertos con cola.
- **Next_Task** Nombre de la tarea a la que se va a enviar el mensaje. Toma el valor `end` si la tarea no envía ningún mensaje.
- **Priority** Prioridad de la tarea: cuanto mayor el número, mayor prioridad.
- **Role** La posición de la tarea en el e2e flow: si es la primera tarea y solo envía mensajes (init), si es una tarea intermedia que tanto envía como recibe mensajes (mid) o si es una tarea que solo recibe mensajes (last).

Los puertos necesarios para la comunicación entre tareas se generarán en la memoria local del core que aloja la tarea lectora ya que es más eficiente la escritura en un núcleo externo que la lectura.

7.2. Funcionamiento del generador

El esquema de funcionamiento del generador se muestra en la Figura 7.1. El generador toma como entrada un fichero de texto, con el formato descrito en la Sección 7.1, que describe la aplicación formada por uno o más flujos e2e y genera el modelo MAST de la citada aplicación y una implementación de la misma basada en tareas con carga de ejecución sintética, puertos de muestreo y puertos con cola.

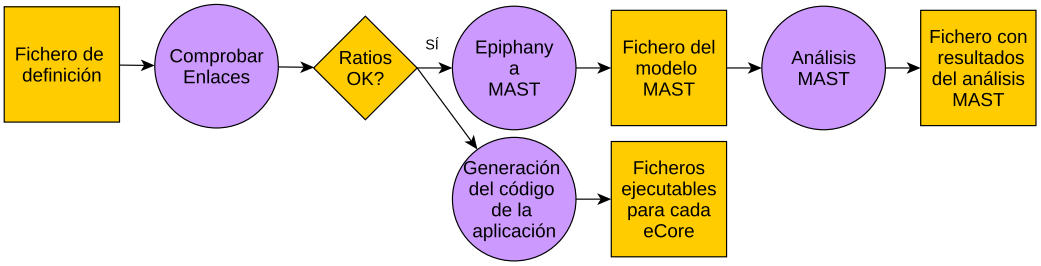

Figura 7.1 Esquema de funcionamiento del generador.

El proceso seguido por el generador consiste en:

1. El generador toma como entrada el fichero de definición del sistema generado por el usuario siguiendo el formato expuesto en la Sección 7.1.

2. Se comprueba que se verifica la restricción de ratio máxima de todos los enlaces, descrita en la Sección 3.2.2. En esta parte se podría incluir la verificación de más restricciones como la superación de la utilización máxima por eCore o el uso de un número de puertos mayor al configurado.

3. Solo si las ratios de los enlaces están dentro de los límites establecidos, se continúa con la generación.

4. Se generan los ejecutables para el procesador Epiphany que podrán ser ejecutados en la plataforma de desarrollo Parallella. Cada ejecutable está destinado a ser ejecutado en un eCore en concreto.

5. Se genera el modelo MAST descrito en el Capítulo 3 y adaptado a Epiphany en el Capítulo 4.

6. Con el fichero generado previamente realizamos el análisis MAST con la herramienta basada en *offsets*, obteniendo los resultados en un fichero.

El tiempo de ejecución de las tareas con carga de ejecución sintética se simula mediante la inclusión de una espera activa en el código generado para el cuerpo de cada una de ellas. Hay que tener en cuenta que el tiempo de ejecución asignado a una tarea en el fichero de entrada incluye tanto el trabajo útil que realiza como el tiempo de acceso a los puertos de comunicación de entrada y salida. Por consiguiente, la duración de la espera activa en el cuerpo de la tarea debe ser igual al valor asignado en el fichero de entrada menos el tiempo de acceso a los puertos. El generador incluye las fórmulas descritas en la sección 6.3.3 para calcular la duración de las operaciones de escritura y lectura en puertos de muestreo y puertos con cola en función del tamaño del dato y de la distancia entre los eCores.

Otro aspecto destacable del código generado es que la tarea inicial de cada flujo incluye en el dato a escribir en su puerto de salida a modo de marca temporal (*timestamp*) el valor de su reloj al comienzo de cada una de sus activaciones. Este valor es propagado en los datos escritos por el resto de tareas en sus respectivos puertos de salida. De esta forma todas las tareas pueden calcular sus tiempos de respuesta respecto al instante de llegada del evento inicial del flujo. El error de las medidas es pequeño gracias al mecanismo de sincronización de relojes implementado en M2OS-mc el cual fue descrito en la Sección 5.4.

Tanto los tiempos del fichero con los resultados de análisis MAST como los resultados temporales de la ejecución del sistema en el procesador Epiphany son directamente comparables.

7.3. Ejemplos de generación

En esta sección vamos a ilustrar, mediante ejemplos, el uso del generador automático y las salidas (código y modelo MAST) generadas.

En primer lugar mostraremos la generación de un sistema muy sencillo formado por dos tareas que se comunican utilizando un puerto de muestreo y posteriormente mostraremos otro sistema algo más complejo que consiste en un flujo e2e formado por tres tareas que se sincronizan utilizando puertos con cola.

Para cada ejemplo mostraremos el fichero de entrada correspondiente al sistema, una explicación del código generado y cómo se ha generado el modelo para MAST.

7.3.1. Ejemplo sencillo con un puerto de muestreo

El ejemplo más sencillo posible utilizando un puerto de muestreo, consiste en una tarea que escribe en un puerto de muestreo y otra tarea que lee de dicho puerto. El puerto de muestreo esta alojado en el núcleo de la tarea lectora.

```
Task (                                  Task (
Name                => t1,              Name                => t2,
Core_Name           => e_0x1,           Core_Name           => e_1x1,
WCET                => 10000,           WCET                => 50000,
BCET                => 9000,            BCET                => 45000,
Period              => 1000000,         Period              => 0,
Deadline            => 1000000,         Deadline            => 0,
Wait_Period         => 2000,            Wait_Period         => 2000,
Num_Packets         => 10,              Num_Packets         => 0,
Send_Ratio          => 0.34,            Send_Ratio          => 0,
Port                => sampling,        Port                => sampling,
Port_Size           => 0,               Port_Size           => 0,
Next_Task           => t2,              Next_Task           => end,
Priority            => 5,               Priority            => 5,
Role                => init)            Role                => last)
```

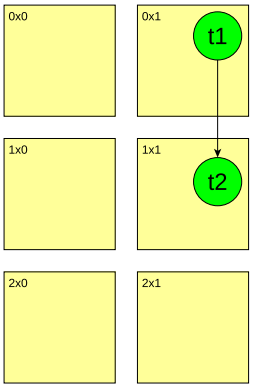

Figura 7.2 Flujo e2e de ejemplo utilizando exclusivamente un puerto muestreo para la comunicación entre núcleos

La Figura 7.2 muestra el fichero que describe este sistema, junto a una representación gráfica del mismo. El código generado se muestra en las Figuras 7.2 y 7.3:

La tarea t1 ejecuta repetidamente en el núcleo 0x1 el código que se muestra en el Listado 7.2. Para la primera ejecución del código se espera a que el puerto de muestreo de la tarea t2 esté inicializado (líneas 9 a 22). Una vez ha comprobado que el puerto de muestreo destinatario de su mensaje esta inicializado, simula la ejecución de un trabajo útil mediante una carga de ejecución sintética implementada con una espera activa con la duración de su WCET menos el tiempo de escritura en el SP de 296 ciclos (líneas 23 a 27). A continuación envía un mensaje de tamaño igual a 10 paquetes al puerto (líneas 28 y 29). En el mensaje se incluye su instante de activación. Seguidamente, se calcula el tiempo de respuesta de la tarea (líneas 30 y 31), y con ese valor se actualizan los tiempos de respuesta máximo, mínimo y medio (líneas 32 a 39) que se mostrarán una vez concluido el número de ejecuciones predeterminado (líneas 40 a 51). Finalmente se espera mediante una instrucción `delay until` (líneas 52 y 53) hasta el inicio del siguiente periodo del flujo e2e.

Listado 7.2 Código simplificado generado para t1

```
1 package body E_Core_1x0_Task1 is
2    Active_Wait_Cycles : constant Timer_Ticks := 296;
```

```
3     Period : constant Time_Span := Nanoseconds (20000);
4     Get_Delay : Time_Span := Nanoseconds (200);
5     procedure Task_Body is
6     begin
7        Data_1_1 (1) := Unsigned_64 (M2.HAL.Timer.Get_Counter);
8        if not SP_1_1_Get then
9           Get_Sampling_Port (Target_Core, SP_Index_1_1, SP_1_1);
10          if SP_1_1 /= Null_SP_Id then
11             if Is_Initialized (SP_1_1) then
12                SP_1_1_Get := True;
13             else
14                Next_Time := Next_time + Get_Delay;
15                delay until Next_Time;
16             end if;
17          else
18             Next_Time := Next_time + Get_Delay;
19             delay until Next_Time;
20          end if;
21       end if;
22       -- Carga de ejecucion sintetica
23       Active_Delay := M2.HAL.Timer.Get_Counter
24        - Active_Wait_Cycles;
25       while (M2.HAL.Timer.Get_Counter > Active_Delay) loop
26          null;
27       end loop;
28       Write_Sampling_Port
29         (SP_1_1, Data_1_1'Address, Data_1_1'Size, Success);
30       Timer_Value := Unsigned_32 (Data_1_1 (1))
31         - Unsigned_32 (M2.HAL.Timer.Get_Counter);
32       if Min_Measured > Timer_Value then
33          Min_Measured := Timer_Value;
34       end if;
35       if Max_Measured < Timer_Value then
36          Max_Measured := Timer_Value;
37       end if;
38       Mean_Measured := Mean_Measured + Timer_Value;
```

```
39          N_Executions := N_Executions + 1;
40          if N_Executions = Max_Executions then
41             DIO.Put_Line ("Task: " & Task_Text);
42             DIO.Put ("Max:");
43             DIO.Put (Integer(Max_Measured));
44             DIO.Put_Line ("");
45             DIO.Put ("Mean:");
46             DIO.Put (Integer(Mean_Measured / N_Executions));
47             DIO.Put_Line ("");
48             DIO.Put ("Min:");
49             DIO.Put (Integer(Min_Measured));
50             DIO.Put_Line ("");
51          end if;
52          Next_Time := Next_Time + Period;
53          delay until Next_Time;
54       end Task_Body;
55    end E_Core_1x0_Task1;
```

Listado 7.3 Código simplificado generado para t2

```
1  package body E_Core_1x1_Task1 is
2     Active_Wait_Cycles : constant Timer_Ticks := 16700;
3     SP_Period : Time_Span := Nanoseconds (200);
4     procedure Task_Init is
5     begin
6        Init_Sampling_Port
7          (Core, SP_Index_1_1, Data_1_1'Address, Data_1_1'Size, SP_1_1);
8        Next_Time := Ada.Real_Time.Clock;
9     end Task_Init;
10
11    procedure Task_Body is
12    begin
13       Read_Sampling_Port
14         (SP_1_1, Data_1_1'Address, Data_1_1'Size, Success, Is_New);
15       if Is_New then
16         Data_1_2 (1) := Data_1_1 (1);
17         -- Carga de ejecucion sintetica
```

```
18            Active_Delay := M2.HAL.Timer.Get_Counter
19              - Active_Wait_Cycles;
20            while (M2.HAL.Timer.Get_Counter > Active_Delay) loop
21               null;
22            end loop;
23            Timer_Value := Unsigned_32 (Data_1_1 (1))
24              - Unsigned_32 (M2.HAL.Timer.Get_Counter);
25
26            if Min_Measured > Timer_Value then
27               Min_Measured := Timer_Value;
28            end if;
29            if Max_Measured < Timer_Value then
30               Max_Measured := Timer_Value;
31            end if;
32            Mean_Measured := Mean_Measured + Timer_Value;
33            N_Executions := N_Executions + 1;
34            if N_Executions = Max_Executions then
35               DIO.Put_Line ("Task: " & Task_Text);
36               DIO.Put ("Max:");
37               DIO.Put (Integer(Max_Measured));
38               DIO.Put_Line ("");
39               DIO.Put ("Mean:");
40               DIO.Put (Integer(Mean_Measured / N_Executions));
41               DIO.Put_Line ("");
42               DIO.Put ("Min:");
43               DIO.Put (Integer(Min_Measured));
44               DIO.Put_Line ("");
45            end if;
46         end if;
47         Next_Time := Next_Time + SP_Period;
48         delay until Next_Time;
49      end Task_Body;
50   end E_Core_1x1_Task1;
```

La tarea t2 ejecuta en el núcleo 1x1 el código que se muestra en la Figura 7.3. En su primera activación, la tarea inicializa el puerto de muestreo donde va a recibir los mensajes (líneas 7 y 8). Posteriormente lee un mensaje del puerto de muestreo (líneas

14 y 15) y si hay un mensaje nuevo simula un trabajo útil mediante una carga de ejecución sintética de una duración igual a su WCET menos el tiempo de lectura del SP de 16.700 ciclos (líneas 18 a 22). Seguidamente, se calcula el tiempo de respuesta de la tarea utilizando el instante de activación del flujo que había sido enviado por la tarea t1 como parte de su mensaje de escritura (líneas 23 y 24), y con ese valor se actualizan los tiempos de respuesta máximo, mínimo y medio (líneas 26 a 33) que se mostrarán una vez concluido el número de ejecuciones predeterminado (líneas 34 a 45). Finalmente se espera al próximo periodo de muestreo mediante una instrucción `delay until` (líneas 47 y 48), operación que realiza independientemente de que haya un mensaje nuevo o no.

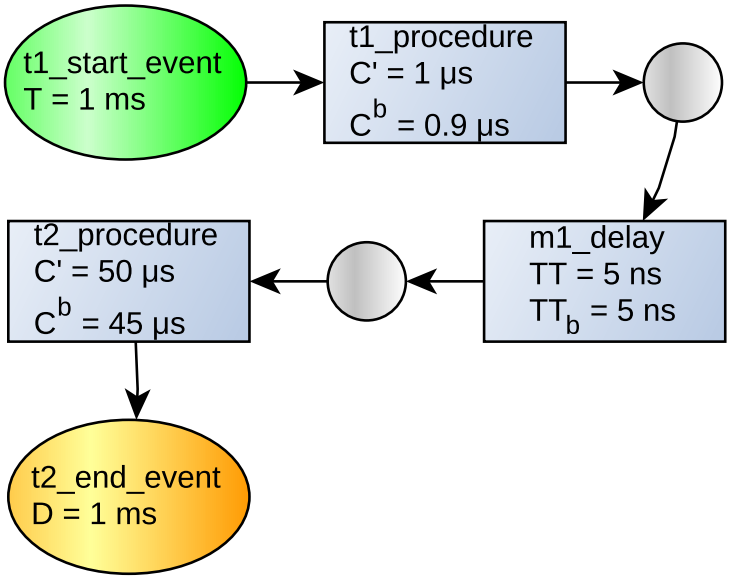

Figura 7.3 Modelo MAST generado automáticamente por el generador para el ejemplo con un puerto de muestreo

El modelo MAST generado correspondiente al fichero de definición de sistema se muestra en la Figura 7.3. El *Scheduling Server* o hilo de ejecución de la actividad `t2_procedure` tiene asignada la política de planificación de muestreo periódico *Polling Policy*.

```
Task (
Name            => t3,
Core_Name       => e_0x0,
WCET            => 1000,
BCET            => 900,
Period          => 1000000,
Deadline        => 1000000,
Wait_Period     => 200,
Num_Packets     => 10,
Send_Ratio      => 0.34,
Port            => queuing,
Port_Size       => 4,
Next_Task       => t4,
Priority        => 5,
Role            => init)
```

```
Task (
Name            => t4,
Core_Name       => e_1x1,
WCET            => 20000,
BCET            => 15000,
Period          => 0,
Deadline        => 0,
Wait_Period     => 2000,
Num_Packets     => 5,
Send_Ratio      => 0.34,
Port            => queuing,
Port_Size       => 4,
Next_Task       => t5,
Priority        => 5,
Role            => mid)
```

```
Task (
Name            => t5,
Core_Name       => e_2x1,
WCET            => 5000,
BCET            => 4500,
Period          => 0,
Deadline        => 0,
Wait_Period     => 200,
Num_Packets     => 0,
Send_Ratio      => 0,
Port            => queuing,
Port_Size       => 4,
Next_Task       => end,
Priority        => 5,
Role            => last)
```

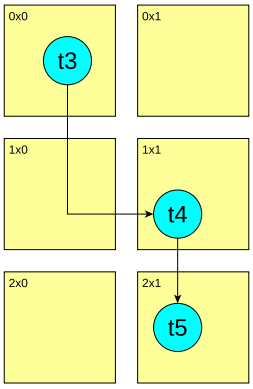

Figura 7.4 Sistema de ejemplo que incluye puertos con cola para la comunicación entre núcleos

7.3.2. Ejemplo de flujo e2e con puertos con cola

Este ejemplo consiste en un flujo e2e de tres tareas utilizando puertos con cola. Tanto la configuración de las tareas como una representación gráfica del sistema se muestran en la Figura 7.4.

La tarea t3, cuyo código se muestra en el Listado 7.4, se ejecuta de manera periódica en el núcleo 0x0. En su primera activación, la tarea espera a que a que el puerto con cola de la tarea t4 esté inicializado (líneas 9 a 19). Una vez confirmada dicha inicialización, se realiza la ejecución sintética implementada con una espera activa de una duración igual a su WCET menos el tiempo de escritura en el QP de 593 ciclos (líneas 21 a 23). A continuación se envía un mensaje de tamaño igual a diez paquetes al puerto con cola (líneas 25 y 26). En el mensaje se incluye su instante de activación. Seguidamente, se calcula el tiempo de respuesta de la tarea (líneas 27 y 28), y con dicho valor se actualizan los tiempos de respuesta máximo, mínimo y promedio (líneas 30 a 37), los

cuales se mostrarán cuando se llegue a un número de ejecuciones determinado (líneas 39 a 50). Finalmente la tarea espera a que vuelva a pasar su periodo de ejecución mediante una instrucción `delay until` (líneas 51 y 52).

Listado 7.4 Código simplificado generado para t3

```
1  package body E_Core_0x0_Task1 is
2     Active_Wait_Cycles : constant Timer_Ticks := 593;
3     Period : constant Time_Span := Nanoseconds (50000);
4     Get_Delay : Time_Span := Nanoseconds (200);
5
6     procedure Task_Body is
7     begin
8       Data_2_1 (1) := Unsigned_64 (M2.HAL.Timer.Get_Counter);
9       -- Carga de la ejecucion sintetica
10      Active_Delay := Data_2_1 (1) - Active_Wait_Cycles;
11       if not QP_2_1_Get then
12          Get_Queuing_Port (Target_Core, QP_Index_2_1, QP_2_1);
13          if Check_Queuing_Port_Initialized (QP_2_1) then
14             QP_2_1_Get := True;
15          else
16             Next_Time := Next_time + Get_Delay;
17             delay until Next_Time;
18          end if;
19       end if;
20       -- Ejecucion sintetica
21       while (M2.HAL.Timer.Get_Counter > Active_Delay) loop
22          null;
23       end loop;
24
25       Epiphany_Queuing_Ports.Write_Queuing_Port
26         (QP_2_1, Data_2_1'Address, Data_2_1'Size, Success);
27       Timer_Value := Unsigned_32 (Data_2_1 (1))
28         - Unsigned_32 (M2.HAL.Timer.Get_Counter);
29
30       if Min_Measured > Timer_Value then
31          Min_Measured := Timer_Value;
32       end if;
```

```
33          if Max_Measured < Timer_Value then
34             Max_Measured := Timer_Value;
35          end if;
36          Mean_Measured := Mean_Measured + Timer_Value;
37          N_Executions := N_Executions + 1;
38
39          if N_Executions = Max_Executions then
40             DIO.Put_Line ("Task: " & Task_Text);
41             DIO.Put ("Max:");
42             DIO.Put (Integer(Max_Measured));
43             DIO.Put_Line ("");
44             DIO.Put ("Mean:");
45             DIO.Put (Integer(Mean_Measured / N_Executions));
46             DIO.Put_Line ("");
47             DIO.Put ("Min:");
48             DIO.Put (Integer(Min_Measured));
49             DIO.Put_Line ("");
50          end if;
51          Next_Time := Next_Time + Period;
52          delay until Next_Time;
53       end Task_Body;
54    end E_Core_0x0_Task1;
```

Listado 7.5 Código simplificado generado para t4

```
1  package body E_Core_1x1_Task1 is
2     Active_Wait_Cycles : constant Timer_Ticks := 32348;
3     Get_Delay : Time_Span := Nanoseconds (200);
4     procedure Task_Init is begin
5        Init_Queuing_Port
6          (QP_Index_2_1, Size_Q_2_1, Data_Q_2_1'Address,
7           Data_Q_2_1'Size, Core, QP_2_1);
8        Read_Queuing_Port
9          (QP_2_1, Data_2_1'Address, Data_2_1'Size, Success);
10       Next_Time := Ada.Real_Time.Clock;
11    end Task_Init;
12
```

```
13    procedure Task_Body is
14    begin
15      -- Carga de la ejecucion sintetinca
16        Active_Delay := M2.HAL.Timer.Get_Counter
17          - Active_Wait_Cycles;
18        if not QP_2_2_Get then
19           Get_Queuing_Port  (Target_Core, QP_Index_2_2, QP_2_2);
20           if (Check_Queuing_Port_Initialized (QP_2_2))
21           then
22              QP_2_2_Get := True;
23           else
24              Next_Time := Next_time + Get_Delay;
25              delay until Next_Time;
26           end if;
27        end if;
28        Data_2_2 (1) := Data_2_1 (1);
29        -- Ejecucion sintetica
30        while (M2.HAL.Timer.Get_Counter > Active_Delay) loop
31           null;
32        end loop;
33        Epiphany_Queuing_Ports.Write_Queuing_Port
34          (QP_2_2, Data_2_2'Address, Data_2_2'Size, Success);
35        Timer_Value := Unsigned_32 (Data_2_1 (1))
36          - Unsigned_32 (M2.HAL.Timer.Get_Counter);
37        if Min_Measured > Timer_Value then
38           Min_Measured := Timer_Value;
39        end if;
40        if Max_Measured < Timer_Value then
41           Max_Measured := Timer_Value;
42        end if;
43        Mean_Measured := Mean_Measured + Timer_Value;
44        N_Executions := N_Executions + 1;
45
46        if N_Executions = Max_Executions then
47           DIO.Put_Line ("Task: " & Task_Text);
48           DIO.Put ("Max:");
```

```
49              DIO.Put (Integer(Max_Measured));
50              DIO.Put_Line ("");
51              DIO.Put ("Mean:");
52              DIO.Put (Integer(Mean_Measured / N_Executions));
53              DIO.Put_Line ("");
54              DIO.Put ("Min:");
55              DIO.Put (Integer(Min_Measured));
56              DIO.Put_Line ("");
57          end if;
58
59          Read_Queuing_Port
60            (QP_2_1, Data_2_1'Address, Data_2_1'Size, Success);
61      end Task_Body;
62  end E_Core_1x1_Task1;
```

La tarea t4, cuyo código se muestra en el Listado 7.5, se ejecuta en el núcleo 1x2. En su primera activación inicializa el puerto con cola donde va a recibir los paquetes (líneas 6 a 8), y seguidamente lee el puerto para poder utilizar el dato en el cuerpo de la tarea (líneas 9 y 10). En el cuerpo de la tarea espera a que el puerto con cola de la tarea t5 donde quiere escribir esté activo (líneas 16 a 27). Una vez confirmada dicha inicialización, se realiza la ejecución sintética implementada con una espera activa de una duración igual a su WCET menos el tiempo de lectura y escritura en los QPs de 32348 ciclos (líneas 29 a 31). A continuación envía el mensaje de cinco paquetes al puerto con cola de la siguiente tarea (líneas 32 y 33). En el mensaje se incluye el tiempo de activación del flujo, que fue recibido de la tarea t3. Seguidamente, se calcula el tiempo de respuesta de la tarea utilizando para ello el tiempo de inicio del flujo (líneas 35 y 36), y con dicho valor se actualizan los tiempos de respuesta máximo, mínimo y promedio (líneas 37 a 44), los cuales se mostrarán cuando se llegue a un número de ejecuciones determinado (líneas 46 a 57). Finalmente se lee el mensaje que activa esta tarea (líneas 59 y 60). La lectura del puerto con cola bloqueará la tarea hasta que haya algún dato disponible.

Listado 7.6 Código simplificado generado para t5

```
1  package body E_Core_2x1_Task1 is
2      Active_Wait_Cycles : constant Timer_Ticks := 8350;
3
4      procedure Task_Init is
```

```
5         Core_Id : Interfaces.Unsigned_32;
6      begin
7         Init_Queuing_Port
8           (QP_Index_2_2, Size_Q_2_2, Data_Q_2_2'Address,
9            Data_Q_2_2'Size, Core, QP_2_2);
10        Epiphany_Queuing_Ports.Read_Queuing_Port
11          (QP_2_2, Data_2_2'Address, Data_2_2'Size, Success);
12        Next_Time := Ada.Real_Time.Clock;
13     end Task_Init;
14
15     procedure Task_Body is
16     begin
17        -- Carga de la ejecucion sintetica
18        Active_Delay := M2.HAL.Timer.Get_Counter
19          - Active_Wait_Cycles;
20        while (M2.HAL.Timer.Get_Counter > Active_Delay) loop
21           null;
22        end loop;
23
24        Timer_Value := Unsigned_32 (Data_2_2 (1))
25          - Unsigned_32 (M2.HAL.Timer.Get_Counter);
26        if Min_Measured > Timer_Value then
27           Min_Measured := Timer_Value;
28        end if;
29        if Max_Measured < Timer_Value then
30           Max_Measured := Timer_Value;
31        end if;
32        Mean_Measured := Mean_Measured + Timer_Value;
33        N_Executions := N_Executions + 1;
34        if N_Executions = Max_Executions then
35           DIO.Put_Line ("Task: " & Task_Text);
36           DIO.Put ("Max:");
37           DIO.Put (Integer(Max_Measured));
38           DIO.Put_Line ("");
39           DIO.Put ("Mean:");
40           DIO.Put (Integer(Mean_Measured / N_Executions));
```

```
41              DIO.Put_Line ("");
42              DIO.Put ("Min:");
43              DIO.Put (Integer(Min_Measured));
44              DIO.Put_Line ("");
45           end if;
46           Epiphany_Queuing_Ports.Read_Queuing_Port
47             (QP_2_2, Data_2_2'Address, Data_2_2'Size, Success);
48       end Task_Body;
49    end E_Core_2x1_Task1;
```

La tarea t5, cuyo código se muestra en el Listado 7.6, es ejecutada en el núcleo 1x3. En su primera activación inicializa el puerto con cola donde va a recibir los paquetes (líneas 7 a 9), y seguidamente lee el puerto para poder utilizarlo en el cuerpo de la tarea (líneas 10 y 11). Repetidamente ejecuta la carga de tiempo de ejecución sintética implementada con una espera activa de una duración igual a su WCET menos el tiempo de lectura en el QP de 8350 ciclos (líneas 18 a 22). Seguidamente, calcula el tiempo de respuesta de la tarea utilizando para ello el tiempo de inicio del flujo obtenido del dato escrito por t4 (líneas 24 y 25), con dicho valor se actualizan los tiempos de respuesta máximo, mínimo y promedio (líneas 26 a 45), los cuales se mostrarán cuando se llegue a un número de ejecuciones determinado (líneas 46 y 47). Finalmente realiza la lectura del puerto con cola de entrada que bloqueará la tarea hasta que haya algún dato disponible.

El modelo MAST generado correspondiente al fichero de definición del sistema se muestra en la Figura 7.5.

Las tareas generadas son ejecutadas sin problema alguno en el procesador Epiphany y analizadas por la herramienta MAST. Los tiempo de respuesta de la ejecución real de los flujos e2e generados están acotados por los tiempos de respuesta de peor caso obtenidos tras analizar el modelo generado, sin que estos últimos introduzcan un gran pesimismo.

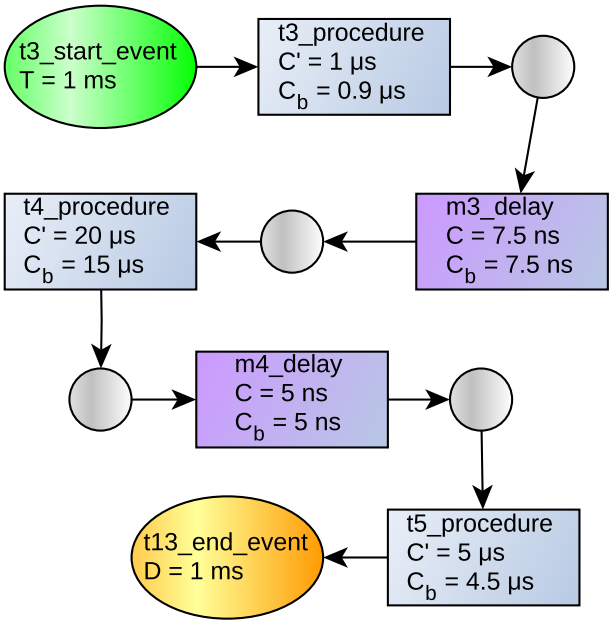

Figura 7.5 Modelo MAST generado automáticamente por el generador para el ejemplo de los puertos con cola

Capítulo 8

Conclusiones y trabajo futuro

8.1. Conclusiones

En esta tesis se ha alcanzado el objetivo general que nos habíamos marcado, consistente en demostrar que es posible implementar aplicaciones con requisitos de tiempo real en procesadores muchos núcleos basados en malla y obtener modelos fiables de su comportamiento temporal encaminados al análisis de planificabilidad. A continuación describimos las principales aportaciones que nos han permitido alcanzar este objetivo general.

Se ha desarrollado una técnica de modelado y análisis de las tareas que se ejecutan en un procesador muchos núcleos basado en malla junto con los mensajes enviados por estas tareas a través de la NoC. El modelo MAST original para el análisis de planificabilidad en sistemas distribuidos de tiempo real no dispone de soporte para procesadores muchos núcleos. En este trabajo hemos mostrado cómo se puede crear un modelo del sistema que, utilizando elementos de modelado preexistentes en MAST nos permite realizar el análisis de planificabilidad de aplicaciones ejecutadas en procesadores muchos núcleos. La técnica de modelado desarrollada nos permite reducir la complejidad del análisis de la interferencia entre los mensajes que circulan por la red de interconexión gracias a la introducción de una limitación sobre la ratio máxima de utilización de los enlaces.

Nuestra técnica de modelado y análisis se ha implementado y probado en una plataforma real: el procesador muchos núcleos Epiphany, que tiene 16 núcleos conectados por una malla 4x4 2D. Este hecho constituye una importante diferencia con la mayor parte del trabajo existente en la bibliografía.

Se ha realizado un análisis del comportamiento temporal del procesador Epiphany y de su red de interconexión. De este análisis se han deducido los parámetros necesarios

para el modelo de tiempo real del comportamiento de una aplicación que se ejecuta sobre esta plataforma.

Se ha desarrollado el sistema operativo de tiempo real para sistemas muchos núcleos M2OS-mc, con una filosofía *micro kernel* donde cada procesador tiene su propia instancia de sistema operativo. Este sistema operativo ha pasado satisfactoriamente todos los tests.

Se ha conseguido dotar a M2OS-mc para Epiphany de unas primitivas de comunicación sincronizada entre núcleos basadas en ARINC-653, estando perfectamente integradas en M2OS-mc.

Se ha desarrollado el modelo de comunicación de las primitivas de sincronización, integrado en el modelo general de análisis del comportamiento temporal de sistemas basados en la plataforma Epiphany.

Se ha desarrollado una herramienta que genera, a partir de un único fichero de descripción, el código de una aplicación con cargas de ejecución sintéticas a ejecutar en los diferentes núcleos del procesador Epiphany. Gracias a esta herramienta se ha podido comprobar el correcto funcionamiento de los puertos de sincronización (puertos de muestreo y puertos con cola) en M2OS-mc para diversos escenarios y se ha podido evaluar la respuesta temporal de la NoC y del sistema completo.

La herramienta también es capaz de generar el modelo MAST de la aplicación, incluyendo las comunicaciones por la NoC, de modo que se facilita la comparación de los resultados de la ejecución real con el análisis de planificabilidad.

Esta herramienta de generación se ha utilizado para evaluar numerosos ejemplos de sistemas, comprobándose así la concordancia entre la ejecución real y el modelo desarrollado para caracterizar el funcionamiento de la aplicación y de los mensajes enviados por la NoC en el procesador muchos núcleos Epiphany.

8.2. Trabajo futuro

Como trabajo futuro se plantean diversos objetivos. En primer lugar, ampliar el estudio de procesadores muchos núcleos para que se puedan modelar memorias compartidas y dispositivos externos a través de los que se puedan hacer operaciones de entrada/salida. A partir de este estudio se desarrollarán modelos de comportamiento temporal que se puedan integrar en el modelo desarrollado en esta tesis. Asimismo, se integrará el uso de estos dispositivos en nuevos servicios del sistema operativo M2OS-mc.

Un segundo objetivo consistiría en ampliar el catálogo de procesadores muchos núcleos soportados, estudiando el comportamiento temporal de todo lo desarrollado para esta tesis en diversas plataformas.

En tercer lugar se podría ampliar la herramienta de generación de código para que funcione también con flujos no lineales, mediante la utilización de bifurcaciones (*forks*) e uniones (*joins*), ya que la implementación de los puertos de comunicación en M2OS-mc lo permite.

Más adelante se podrían añadir al sistema operativo servicios de monitorización de la salud del sistema, identificando los efectos de los fallos que pueden ocurrir dentro del procesador muchos núcleos, diseñar, implementar y verificar medidas con objetivos de seguridad y detectar y manejar los fallos conteniendo los efectos de cualquier fallo detectado.

En otra vertiente se podría estudiar el comportamiento del procesador cuando coexistan aplicaciones basadas en M2OS-mc con otras programadas mediante interfaces OpenMP.

Otra posible línea de trabajo futuro consiste en desarrollar algoritmos de asignación de tareas a núcleos de un procesador muchos núcleos para minimizar los tiempos de respuesta de las aplicaciones.

Bibliografía

[1] A. Olofsson, "Parallella reference manual." [Online]. Available: http://www.parallella.org/docs/parallella_manual.pdf

[2] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." pp. 33–35, 2006.

[3] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 309–318.

[4] T. Kelter, T. Harde, P. Marwedel, and H. Falk, "Evaluation of resource arbitration methods for multi-core real-time systems," in *13th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), C. Maiza, Ed., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 1–10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4117

[5] A. Stockman and P. Moens, "On-state gate stress induced threshold voltage instabilities in p-gan gate algan/gan hemts," in *2020 IEEE International Integrated Reliability Workshop (IIRW)*. IEEE, 2020, pp. 1–4.

[6] G. Gracioli and A. A. Fröhlich, "On the design and evaluation of a real-time operating system for cache-coherent multicore architectures," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 2, p. 2–16, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2883591.2883594

[7] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

[8] EASA, "Easa: Certification memorandum subject development assurance of airborne electronic hardware," 2011. [Online]. Available: https://www.easa.europa.eu/document-library/product-certification-consultations/easa-cm-swceh-001

[9] X. Jean, M. Gatti, G. Berthon, and M. Fumey, "Mulcors–use of multicore processors in airborne systems. research project easa. 2011/6. dossier ref," CCC/12/006898–Rev. 07. Thales Avionics, Tech. Rep., 2012.

[10] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[11] S. K. Baruah, "Dynamic-and static-priority scheduling of recurring real-time tasks," *Real-Time Systems*, vol. 24, no. 1, pp. 93–128, 2003.

[12] S. Davari and L. Sha, "Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions," *ACM SIGOPS Operating Systems Review*, vol. 26, no. 2, pp. 110–120, 1992.

[13] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems," *Real-Time Systems*, vol. 55, no. 4, pp. 850–888, 2019.

[14] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2012, pp. 145–155.

[15] A. Vajda, "Multi-core and many-core processor architectures," in *Programming Many-Core Chips*. Springer, 2011, pp. 9–43.

[16] M. González Harbour, J.L. Medina, J.J. Gutiérrez, J.C. Palencia, and J.M. Drake, "MAST: An open environment for modeling, analysis, and design of real-time systems," October 2002.

[17] D. G. Villaescusa, M. A. Rivas, and M. G. Harbour, "M2OS-Mc: An RTOS for Many-Core Processors," in *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*, ser. OpenAccess Series in Informatics (OASIcs), M. Bertogna and F. Terraneo, Eds., vol. 87. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 5:1–5:13. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13481

[18] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, ser. NOCS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 161–170. [Online]. Available: http://dl.acm.org/citation.cfm?id=1397757.1397996

[19] L. S. Indrusiak, "End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration," *Journal of Systems Architecture*, vol. 60, no. 7, pp. 553 – 561, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762114000800

[20] L. S. Indrusiak, A. Burns, and B. Nikolic, "Analysis of buffering effects on hard real-time priority-preemptive wormhole networks," *ArXiv*, vol. abs/1606.02942, 2016.

[21] Q. Xiong, Z. Lu, F. Wu, and C. Xie, "Real-time analysis for wormhole noc: Revisited and revised," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016, pp. 75–80.

[22] M. Dridi, F. Singhoff, S. Rubini, and J.-P. Diguet, "Ectm: A network-on-chip communication model to combine task and message schedulability analysis," *Journal of Systems Architecture*, p. 101931, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762120301909

[23] S. Gopalakrishnan, L. Sha, and M. Caccamo, "Hard real-time communication in bus-based networks," in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 405–414.

[24] B. Nikolic, S. Tobuschat, L. Indrusiak, R. Ernst, and A. Burns, "Real-time analysis of priority-preemptive nocs with arbitrary buffer sizes and router delays," *Real-Time Systems*, vol. 55, 06 2018.

[25] S. Metzlaff, J. Mische, and T. Ungerer, "A real-time capable many-core model," in *Proceedings of 32nd IEEE real-time systems symposium: work-in-progress session*, 2011, pp. 21–24.

[26] M. Zhang, J. Shi, T. Zhang, and Y. Hu, "Hard real-time communication over multi-hop switched ethernet," in *2008 International Conference on Networking, Architecture, and Storage*, 2008, pp. 121–128.

[27] A. Yiming and T. Eisaka, "A switched ethernet protocol for hard real-time embedded system applications," *Journal of Interconnection Networks*, vol. 6, no. 3, pp. 345–360, 2005.

[28] S. Lee, K. C. Lee, and H. H. Kim, "Maximum communication delay of a real-time industrial switched ethernet with multiple switching hubs," in *30th Annual Conference of IEEE Industrial Electronics Society, 2004. IECON 2004*, vol. 3, 2004, pp. 2327–2332 Vol. 3.

[29] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2-3, pp. 117–134, 1994.

[30] J. C. Palencia and M. G. Harbour, "Offset-based response time analysis of distributed systems scheduled under edf," in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.* IEEE, 2003, pp. 3–12.

[31] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Syst.*, vol. 54, no. 3, p. 607–661, Jul. 2018. [Online]. Available: https://doi.org/10.1007/s11241-017-9285-4

[32] L. Benini and G. Micheli, "Networks on chips: A new soc paradigm," *Computer*, vol. 35, pp. 70–78, 02 2002.

[33] S. Hesham, J. Rettkowski, D. Goehringer, and M. A. Abd El Ghany, "Survey on real-time networks-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, p. 1500–1517, May 2017. [Online]. Available: https://doi.org/10.1109/TPDS.2016.2623619

[34] A. Burns, J. Harbin, and L. Indrusiak, "A wormhole noc protocol for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*. IEEE, Dec. 2014, pp. 184–195.

[35] J. G. Balaguer, J. R. Z. Flores, and J. A. de la Puente Alfaro, "Arinc-653 inter-partition communications and the ravenscar profile," *Ada Letters*, vol. 35, no. 1, pp. 38–45, 2015. [Online]. Available: http://oa.upm.es/42418/

[36] D. García Villaescusa, M. González Harbour, and M. Aldea Rivas, "Selección de una arquitectura many-core comercial como plataforma de tiempo real," in *V Simposio de Sistemas de Tiempo Real, CEDI*. Ediciones Universidad de Salamanca, 2016. [Online]. Available: http://hdl.handle.net/10902/17870

[37] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[38] Corporation, "Tile processor architecture overview for the tile-gx series," 2012. [Online]. Available: https://cdn.manesht.ir/17871_____210769647-UG130-ArchOverview-TILE-Gx.pdf

[39] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[40] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 14–24.

[41] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.

[42] S. Skalistis and A. Simalatsar, "Worst-case execution time analysis for many-core architectures with noc," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2016, pp. 211–227.

[43] B. Nikolić, P. M. Yomsi, and S. M. Petters, "Worst-case memory traffic analysis for many-cores using a limited migrative model," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013, pp. 42–51.

[44] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 67–76. [Online]. Available: https://doi.org/10.1145/2997465.2997472

[45] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling mixed-criticality in soc-based real-time embedded systems," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 235–244. [Online]. Available: https://doi.org/10.1145/1629335.1629367

[46] P. H. Feiler, B. A. Lewis, and S. Vestal, "The sae architecture analysis amp; design language (aadl) a standard for engineering performance critical systems," in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, 2006, pp. 1206–1211.

[47] J. Zamorano and A. Juan, "Memory isolation in many-core embedded systems," *Highperformance and Real-time Embedded System (HiRES)*, 2014.

[48] X. Gu, P. Liu, M. Yang, J. Yang, C. Li, and Q. Yao, "An efficient scheduler of rtos for multi/many-core system," *Computers Electrical Engineering*, vol. 38, no. 3, pp. 785–800, 2012, the Design and Analysis of Wireless Systems and Emerging Computing Architectures and Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790611001340

[49] F. Kluge, B. Triquet, C. Rochange, T. Ungerer, and S. Airbus Operations, "Operating systems for manycore processors from the perspective of safety-critical systems," in *Proceedings of 8th annual workshop on Operating Systems for Embedded Real-Time applications (OSPERT 2012)*. Citeseer, 2012, pp. 16–20.

[50] X. Gu, P. Liu, M. Yang, J. Yang, C. Li, and Q. Yao, "An efficient scheduler of rtos for multi/many-core system," *Computers & Electrical Engineering*, vol. 38, no. 3, pp. 785 – 800, 2012, the Design and Analysis of Wireless Systems and Emerging Computing Architectures and Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790611001340

[51] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[52] E. Enterprise, "Erika3," [Online; accessed 29-January-2020]. [Online]. Available: https://www.erika-enterprise.com

[53] eSol, "Scalable and High-performance Real-Time OS available for various types of processors," [Online; accessed 29-January-2020]. [Online]. Available: https://www.esol.com/embedded/emcos.html

[54] H. Almatary, "Operating system kernels on multi-core architectures," Ph.D. dissertation, University of York, January 2016. [Online]. Available: http://etheses.whiterose.ac.uk/12959/

[55] P. Mesidis and L. S. Indrusiak, "Genetic mapping of hard real-time applications onto noc-based mpsocs—a first approach," in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2011, pp. 1–6.

[56] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, vol. 52, no. 4, pp. 399–449, 2016.

[57] L. S. Indrusiak, J. Harbin, and A. Burns, "Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 47–56.

[58] M. N. S. M. Sayuti and L. S. Indrusiak, "A function for hard real-time system search-based task mapping optimisation," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 2015, pp. 66–73.

[59] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[60] M. Aldea-Rivas and H. Pérez-Tijero, "Proposal for a new ada profile for small microcontrollers," *Ada Lett.*, vol. 38, no. 1, p. 34–39, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3241950.3241955

[61] M. A. Rivas and H. P. Tijero, "Leveraging real-time and multitasking ada capabilities to small microcontrollers," *Journal of Systems Architecture*, vol. 94, pp. 32 – 41, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762118302212

[62] E. Schonberg and B. Banner, "The gnat project: A gnu-ada 9x compiler," in *Proceedings of the Conference on TRI-Ada '94*, ser. TRI-Ada '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 48–57. [Online]. Available: https://doi.org/10.1145/197694.197706

[63] A. Burns, "The ravenscar profile," *Ada Lett.*, vol. XIX, no. 4, p. 49–52, dec 1999. [Online]. Available: https://doi.org/10.1145/340396.340450

[64] D. G. Villaescusa, M. A. Rivas, and M. G. Harbour, "Queuing Ports for mesh based many-core porcessors," in *25th International Conference on Reliable Software Technologies, Ada-Europe 2021, Work-in-Progress*.

Apéndice A

Scripts y códigos de ejemplo

A.1. Scripts de compilación cruzada

```
1  with "../global_switches.gpr";
2  with "../arch/epiphany/shared_switches_epiphany.gpr";
3  with "../arch/epiphany/m2os_epiphany.gpr";
4
5  project Tests_API_M2OS_Epiphany is
6
7     type Code_Processing is ("No_Use_Tool");
8     Use_Code_Processing_Tool : Code_Processing:= external ("use_tool",
9                                                            "
                                                                No_Use_Tool
                                                                ");
10
11    for Languages use ("Ada");
12    for Main use ("e_core_0x0.adb",
13                  "e_core_0x1.adb",
14                  "e_core_0x2.adb",
15                  "e_core_0x3.adb",
16                  "e_core_1x0.adb",
17                  "e_core_1x1.adb",
18                  "e_core_1x2.adb",
19                  "e_core_1x3.adb",
20                  "e_core_2x0.adb",
21                  "e_core_2x1.adb",
22                  "e_core_2x2.adb",
23                  "e_core_2x3.adb",
24                  "e_core_3x0.adb",
25                  "e_core_3x1.adb",
```

```
26                 "e_core_3x2.adb",
27                 "e_core_3x3.adb");
28    for Source_Dirs use (".",
29                         "../tests/reports",
30                         Global_Switches'Project_Dir & "adax");
31    for Exec_Dir use ".";
32
33    for Runtime ("ada") use Shared_Switches_Epiphany.Runtime;
34    for Target use Shared_Switches_Epiphany.Target;
35
36    package Compiler is
37       for Switches ("Ada") use
38         Shared_Switches_Epiphany.Compiler'Switches ("ada")
39         & Global_Switches.App_Ada_Flags;
40       for Switches ("c") use
41         Shared_Switches_Epiphany.Compiler'Switches ("c")
42         & Global_Switches.App_C_Flags;
43    end Compiler;
44
45    package Binder renames Shared_Switches_Epiphany.Binder;
46
47    package Linker renames Shared_Switches_Epiphany.Linker;
48
49    package Ide renames Shared_Switches_Epiphany.Ide;
50
51 end Tests_API_M2OS_Epiphany;
```

Listado A.1 Proyecto GPR para construir aplicaciones que ejecuten sobre M2OS-mc

```
1  #!/bin/bash
2  M2OS_DIR=$HOME/Dev/M2OS/
3  API_TEST_DIR=tests/api_m2os
4
5  PARALLELLA_DIR=/home/parallella/propio/generated/Debug
6  PARALLELLA_USER=parallella
7  PARALLELLA_PASS="parallella"
8  PARALLELLA_IP=$ENV_PARALLELLA_IP
9
10 cd $M2OS_DIR
11 #Compile M2OS
12 #make clean
13 #make install
14
15 cd $API_TEST_DIR
16
17 gprbuild -v -P tests_api_m2os_epiphany.gpr
18
19 sshpass -p $PARALLELLA_PASS scp e_core_0x0
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_0x0.elf
20
21 sshpass -p $PARALLELLA_PASS scp e_core_0x1
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_0x1.elf
22
23 sshpass -p $PARALLELLA_PASS scp e_core_0x2
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_0x2.elf
24
25 sshpass -p $PARALLELLA_PASS scp e_core_0x3
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_0x3.elf
26
27 sshpass -p $PARALLELLA_PASS scp e_core_1x0
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_1x0.elf
28
29 sshpass -p $PARALLELLA_PASS scp e_core_1x1
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_1x1.elf
30
31 sshpass -p $PARALLELLA_PASS scp e_core_1x2
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_1x2.elf
32
33 sshpass -p $PARALLELLA_PASS scp e_core_1x3
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_1x3.elf
34
35 sshpass -p $PARALLELLA_PASS scp e_core_2x0
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_2x0.elf
```

```
36
37 sshpass -p $PARALLELLA_PASS scp e_core_2x1
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_2x1.elf
38
39 sshpass -p $PARALLELLA_PASS scp e_core_2x2
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_2x2.elf
40
41 sshpass -p $PARALLELLA_PASS scp e_core_2x3
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_2x3.elf
42
43 sshpass -p $PARALLELLA_PASS scp e_core_3x0
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_3x0.elf
44
45 sshpass -p $PARALLELLA_PASS scp e_core_3x1
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_3x1.elf
46
47 sshpass -p $PARALLELLA_PASS scp e_core_3x2
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_3x2.elf
48
49 sshpass -p $PARALLELLA_PASS scp e_core_3x3
      $PARALLELLA_USER@$PARALLELLA_IP:$PARALLELLA_DIR/e_core_3x3.elf
50
51 #Copy sources for Debugging
52 #cd $M2OS_DIR
53 #sshpass -p "parallella" scp -r * parallella@193.144.198.97:/home/
      parallella/propio/status_monitor/debug_src
```

Listado A.2 Script que realiza una compilación cruzada del código y copia el contenido en la memoria SD del procesador Parallella

A.2. Código de ejemplo para Zynq

```
1  #define CONSOLE_SIZE 256
2  #define CONSOLE_ADDR 0x7500
3  #define N_CORES 16
4  #define INIT_CORE 0
5
6  int main()
7  {
8    int i,j;
9    char buf[CONSOLE_SIZE*sizeof(char)];
10
11   //System initialization
12   e_init(NULL);
13   e_reset_system();
14
15   //Workgroup definition
16   e_epiphany_t dev;
17   e_open(&dev, 0, 0, 4, 4);
18   e_reset_group(&dev);
19
20   char code_name[14] = "e_core_0x0.elf";
21
22   //Loading the executables
23   for (i = INIT_CORE; i < N_CORES; i++) {
24     code_name[7] = i/4+'0'; // ASCII
25     code_name[9] = i%4+'0'; // ASCII
26     if ( E_OK != e_load(code_name, &dev, i/4, i%4, E_FALSE) ) {
27       fprintf(stderr, "Failed to load %s\n", code_name);
28       return EXIT_FAILURE;
29     }
30   }
31
32   //Starting the execution
33   for (i = INIT_CORE; i < N_CORES; i++) {
34     e_start (&dev, i/4, i%4);
35   }
36
37   //Waiting for the tests execution
38   usleep(1000000);
39
40   //Console reading
41   for (i = INIT_CORE; i < N_CORES; i++) {
42     e_read(&dev, i/4, i%4, CONSOLE_ADDR, &buf, sizeof(buf));
```

```
43      printf("Core ***** %dx%d *****\n", i/4, i%4);
44      printf("%s\n", buf);
45      printf("------------------------\n");
46    }
47
48    //End execution
49    e_close(&dev);
50    e_finalize();
51
52    return 0;
53  }
```

Listado A.3 Código para el procesador Zynq para cargar, lanzar y ver la salida por consola de un conjunto de ejecutables para Epiphany

A.3. Script de compilación en Parallella del código para Zynq

```
1  #!/bin/bash
2
3  set -e
4
5  ESDK=${EPIPHANY_HOME}
6  ELIBS="-L ${ESDK}/tools/host/lib"
7  EINCS="-I ${ESDK}/tools/host/include"
8  ELDF=${ESDK}/bsps/current/fast.ldf
9
10 SCRIPT=$(readlink -f "$0")
11 EXEPATH=$(dirname "$SCRIPT")
12 cd $EXEPATH
13
14 # Build HOST side application
15 ${CROSS_COMPILE}gcc src/loader.c -o Debug/loader.elf ${EINCS} ${ELIBS
       }  -le-hal -le-loader -lpthread -O0 -g
```

Listado A.4 Script para compilar en el procesador Parallella el código para Zynq localizado en loader.c

A.4. Script para ejecutar en Parallella el código para Zynq

```
1  #!/bin/bash
2
3  set -e
4
5
6  cd Debug
7
8  ./loader.elf
```

Listado A.5 Script para ejecutar en el procesador Parallella el código para Zynq en el ejecutable loader.elf