Proyecto de fin de Máster en Ingeniería de Computadores
Curso 2007 - 2008

Planificación de grafos de tareas para sistemas multi-proceso dinámicamente reconfigurables

Juan Antonio Clemente Barreira

Dirigido por:
Jesús Javier Resano Ezcaray
Dpto: Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

AGRADECIMIENTOS

Soy consciente de que esta será la sección más leída de esta memoria, por lo que intentaré redactar un prólogo a la altura de quienes me ayudaron y apoyaron durante este largo camino y, en consecuencia, merecen mis más sinceros agradecimientos. Espero y deseo no olvidarme de nadie importante; en todo caso, te pido perdón si la memoria me falla y no apareces aquí, prometo no olvidarme de ti en los últimos instantes de la redacción de mi futura tesis (cruzo los dedos y toco madera, en este orden). Debo agradecer y agradezco:

A **Javier Resano**, mi director de proyecto, mi mentor, por todo el apoyo y ayuda prestados. Por haberme guiado y aconsejado durante este tiempo de una manera inmejorable y con la máxima abnegación. Aunque una parte egoísta de mí se siente un poquito triste de que te vayas a Zaragoza a partir de la temporada que viene, te deseo lo mejor y que nuestros caminos profesionales sigan teniendo una intersección no vacía.

A **Daniel Mozos**, por tus sabios consejos y a que tu pasión por la investigación y la docencia esté dejando huella en mi trayectoria profesional. Por tu inestimable ayuda, apoyo y confianza que en cada momento depositas en mí. Por proporcionarme una visión humana y crítica de la vida, desde tu experiencia y sabiduría. Y por saber mostrar una sonrisa en los momentos difíciles o menos buenos, que me ha ayudado a afrontar los problemas con paciencia y pragmatismo.

A **Carlos González**, compañero inseparable e infatigable de prácticas desde hace casi 7 años; y espero que dentro de poco, también compañero de Departamento. Esas tardes interminables de prácticas, partidas de mus en la cafetería, viajes a congresos… que no me canso de recordar; y lo que nos queda. Sin tu ayuda, llegar hasta aquí habría sido poco menos que un imposible.

A mis, desde hace 10 meses, compañeros de trabajo y también amigos. En particular, los moradores del despacho 347 que, a mi llegada a la ayudantía en la facultad, hicieron más fácil mi incorporación y adaptación a este "mundillo". Sin ningún orden en particular, a **José Luis Vázquez** (¡enhorabuena, disfruta de tus, espero, últimos 12 meses de soltero!)**, Jesús Fernández, Ángel González, Miguel Peón** y **Pablo García** (memorables esos penalties contra Italia vividos en tu casa). Por otro lado, no me olvido de **Francisco Rincón, Alberto del Barrio, Antonio Artés, José Luis Ayala, Carlos Roa, Javier Sánchez Jurado, David Sigüenza** y **Guillermo Botella**; todos ellos grandes profesionales y mejores personas. Espero seguir compartiendo con vosotros mi vida profesional y, cómo no, esos ratos de ocio que tanto nos han unido: *in-jokes, coffee breaks*, comidas, viajes, partidos, visitas al Delic, etc, etc. Y por supuesto, a **Adolfo Domínguez**. No te creas que por irte de Erasmus te vas a librar tan fácilmente de mí.

Y, por supuesto, a mi familia, especialmente **a mis padres y a mi hermano**. Todo en lo que me he convertido os lo debo en gran parte a vosotros por proporcionarme la mejor educación con la máxima dedicación y sin esperar nada a cambio. Esto es sólo una pequeña parte de todo lo que os debo.

Este trabajo es también vuestro. Mil gracias a todos.

ÍNDICE

ÍNDICE DE IMÁGENES Y TABLAS

RESUMEN DEL PROYECTO Y PALABRAS CLAVE

RESUMEN DEL PROYECTO

La computación reconfigurable es una tecnología prometedora que permite ejecutar con gran eficiencia aplicaciones con una alta carga computacional y/o un comportamiento dinámico difícil o imposible de predecir, a la vez que reutilizar los mismos recursos HW para distintas tareas. En sistemas empotrados, las aplicaciones se representan frecuentemente como grafos de tareas dirigidos acíclicos. Toda tarea HW necesita un proceso de reconfiguración previo a su ejecución que puede producir penalizaciones significativas (del orden de ms) que, a su vez, pueden reducir en gran medida el rendimiento del sistema. Para intentar superar este problema, es imprescindible contar con una buena estrategia de planificación que reduzca en gran medida estas penalizaciones, solapándolas con ejecuciones de otras tareas. No obstante, esta labor puede derivar en una gran carga computacional si un procesador empotrado se encarga de ello, al involucrarse en el uso de estructuras de datos complejas y numerosas comunicaciones HW-SW. Como consecuencia, esta gestión puede producir penalizaciones adicionales. Por tanto, para superar este nuevo problema se ha desarrollado un planificador HW utilizando recursos reconfigurables (aproximadamente el 13% de una FPGA Virtex-II PRO xc2vp30). Mediante técnicas de reutilización, prebúsqueda y reemplazo de tareas, este planificador consigue reducir las penalizaciones por reconfiguración del 42% del tiempo total de ejecución de las tareas a aproximadamente el 9%. Además, esta gestión se produce en unos pocos ciclos de reloj, por lo que las penalizaciones que produce en su gestión son insignificantes. Por otro lado, se ha comparado esta implementación con una versión SW equivalente que no tiene coste HW pero, de acuerdo a nuestros experimentos, produce unas penalizaciones que oscilan entre el 1% y el 3% del tiempo total de ejecución de los grafos de tareas.

ABSTRACT

Reconfigurable computing is a promising technology that allows executing efficiently applications with a very high workload and/or dynamic and unpredictable behaviour. In addition, with this technology, HW resources can be reused for different tasks. In embedded systems, applications are frequently represented as direct acyclic task graphs. Every HW task needs to perform a reconfiguration process before starting its execution. This process can generate significant overheads (of the order of milliseconds) which, in turn, can greatly reduce the system's performance. To try to overcome this problem, it's essential to use a good scheduling strategy that greatly reduces these overheads, overlapping these reconfigurations with the execution of previous tasks. However, this management involves a hith workload since the processor must deal with complex complex data structures and HW-SW communications. As a result, it may produce additional time penalties. Hence, to overcome this new problem we have developed a HW scheduler using reconfigurable resources (approximately 13% of a Virtex-II PRO xc2vp30). Applying reuse, prefetch and replacement techniques; this scheduler can reduce reconfiguration overheads from 42% to 9% of the total execution time of task graphs. Moreover, this management is performed in just a few clock cycles, so it generates a negligible overnead. Moreover, we have compared this implementation with an equivalent SW version, which has no HW cost but, according to our experiments, generates overheads from 1% to 3% of the execution time of a task graph.

PALABRAS CLAVE

Hardware multitarea, reconfiguración parcial dinámica, planificador de tareas, SoC, unidad reconfigurable (UR), FPGA, ISE, EDK, Virtex II-PRO.

AUTORIZACIÓN

El ponente Juan Antonio Clemente Barreira, con DNI 52887871-S, autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Juan Antonio Clemente Barreira

En Madrid, a 15 de septiembre de 2008

INTRODUCCIÓN

La reconfiguración parcial [Lys06] abre la posibilidad de desarrollar un sistema multitarea HW dividiendo la totalidad del área reconfigurable en unidades reconfigurables (URs) que incluyan un interfaz fijo conocido por el diseñador de tareas [Mares02]. Estas URs se pueden reconfigurar para ejecutar tareas de las aplicaciones que se estén ejecutando; por tanto, con las configuraciones correctas, la misma plataforma se puede personalizar en tiempo de ejecución para acelerar varias aplicaciones específicas, o para adaptarse a los requisitos variables de una aplicación dinámica compleja. Además, las reconfiguraciones se pueden actualizar en tiempo de ejecución para extender la funcionalidad del sistema, mejorar las prestaciones o arreglar errores. Actualmente, algunas plataformas comerciales, como XILINX Virtex™ series [Xil08] o Altera® Stratix [Altera08], se pueden utilizar para implementar un sistema multitarea HW en el que varias URs colaboren con uno o más procesadores empotrados. Es más, algunos vendedores han desarrollado entornos de diseño especiales para proporcionar soporte a este tipo de sistemas, como el XILINX Embedded Development Kit (EDK) [EDK08] o de Altera SOPC builder [SOPC08].

Sin embargo, algunos trabajos previos ([Res05b], [Li02]) han demostrado que la eficiencia de un sistema HW multi-tarea basado en FPGAs puede reducir drásticamente debido a las latencias de reconfiguración a menos que se utilicen en tiempo de ejecución ciertas técnicas de optimización.

En sistemas empotrados, las tareas se representan frecuentemente como grafos dirigidos acíclicos y lo más común es que un procesador empotrado deba encargarse de su ejecución teniendo en cuenta sus dependencias (de control o de datos) internas. Este procesador debe aplicar también las técnicas de optimización que, para tratar eventos dinámicos, deben ser tratados como mínimo parcialmente en tiempo de ejecución. Esto implica tratar con estructuras de datos complejas y, para un sistema HW multi-tarea, realizar cuantiosas comunicaciones HW/SW, que pueden introducir importantes penalizaciones en la ejecución del sistema.

Como estamos trabajando con sistemas reconfigurables, mi propuesta es utilizar parte de estos recursos para implementar un planificador HW que se realice estas operaciones de planificación, incluyendo las técnicas de optimización citadas. Nuestro sistema objetivo contará con una serie de unidades reconfigurables (*URs*) en las que se ejecutan las tareas HW. Nuestro planificador debe comunicarse directamente con ellas, para dar las órdenes adecuadas de inicio de reconfiguración e inicio de ejecución, entre otras. Estas comunicaciones son necesarias, y si se realizasen SW/HW serían altamente costosas; luego un planificador HW puede resultar altamente eficiente en este sentido. Por otro lado, una implementación HW introduce muchas menos penalizaciones en tiempo de ejecución que una implementación SW equivalente. Estos son los motivos que nos han llevado a considerar un desarrollo HW de un planificador de tareas para sistemas multi-tarea dinámicamente reconfigurables.

Al margen del tipo de implementación que se realice, en todo planificador de tareas la estrategia de planificación es un punto clave que hay que considerar en gran medida. Una buena estrategia permite reducir las penalizaciones que se producen debido a los tiempos de reconfiguración de las tareas en las URs. Y una reducción significativa en estas penalizaciones es imprescindible para que los sistemas multi-tarea HW sean una alternativa a otro tipo de sistemas multi-proceso. En este sentido, hemos desarrollado una técnica de reemplazo e integrado en el planificador.

En esta memoria se presenta un planificador de tareas HW para sistemas dinámicamente reconfigurables. Presentamos sus detalles de implementación, así como su coste en términos de área y penalizaciones en el rendimiento cuando se utiliza. En cuanto a la técnica de reemplazo desarrollada, evaluamos su funcionamiento comparándola con otras estrategias sencillas y de similar coste. Finalmente, evaluamos el impacto en el rendimiento que tiene su utilización y lo comparamos con una implementación SW equivalente.

CONTEXTO DE LA MEMORIA

El trabajo de investigación que ha dado como resultado esta memoria se ha desarrollado en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid. Este departamento está compuesto por los siguientes grupos de investigación:

- ***Arquitectura de sistemas distribuidos**:* El grupo DSA lleva a cabo varias actividades de investiación relacionadas con la asignación dinámica y la planificación de tareas (*GridWay Metascheduler*), la penalización de máquinas virtuales respecto a recursos grid, gestión dinámica de máquinas virtuales (*OpenNebula Virtual Infrastructure Engine*) y federación de infraestructuras grid (*Grid4Utility*).

- ***Arquitectura y tecnología de sistemas informáticos***: La actividad investigadora del Grupo de Arquitectura y Tecnología de Sistemas Informáticos (*ArTeCS*) de la Universidad Complutense de Madrid se centra en la concepción y construción de sistemas informáticos digitales, y su eficiencia en cuanto a rendimiento, consumo de energía y coste. Dentro de esta amplia área, el grupo presta mayor atención a procesadores de alto rendimiento, computación distibuida, sistemas empotrados y systems-on-chip.

- ***Ingeniería de sistemas, control, automatización y robótica*** (ISCAR): El grupo centra su investigación en las siguientes líneas: Estabilización de barcos rápidos, Robótica terrestre y aérea, Visión por computador, Avionica y temas del espacio, Algoritmos Genéticos, Logística y optimización, Sistemas Borrosos, Robots marinos cooperantes, E-learning y laboratorios virtuales, Simulación y Control, modelado y optimización de procesos.

- ***Grupo de Hardware Dinámicamente Reconfigurable**:* Grupo de investigadores que vienen del campo de Síntesis de Alto Nivel y co-diseño HW/SW. Actualmente, centran su interés en la gestión de recursos de hardware dinámicamente reconfigurable, en sistemas de propósito general que ejecutan una serie continua de tareas. El problema de la gestión de hardware para sistemas hardware multitarea es similar al problema de gestión de memoria y tiempo de CPU para sistemas software multitarea.

El trabajo de investigación presentado en esta memoria, se enmarca en esta última línea de investigación, *Hardware Dinámicamente Reconfigurable*.

TRABAJO RELACIONADO

Recientemente muchos grupos de investigación han propuesto crear sistemas multitarea HW usando recursos parcialmente reconfigurables. Por ejemplo, Marescaux et al. presentaron en [Mares02] la primera implementación de un sistema de esas características en una FPGA comercial que estaba particionada en un conjunto de URs idénticas. Otras aproximaciones recientes que presentan sistemas HW multitarea son [Gar06], [Wal03], [Qu06], [Nogue04a], [Vik06a] y [Fu05]. El planificador desarrollado en este trabajo es compatible con los sistemas HW multi-tarea presentados en [Mares02], [Gar06], [Wal03], [Qu06] y [Nogue04a], pero no con [Vik06a] y [Fu05] porque [Vik06a] únicamente se centra en extraer paralelismo de datos, y [Fu05] propone utilizar un Sistema Operativo (SO) multi-hilo de propósito general, con algunas optimizaciones para la planificación, que gestione el sistema.

En todos estos sistemas, normalmente las URs están fuertemente acopladas a un procesador que guía la ejecución de las tareas. Este procesador debe monitorizar la ejecución HW y encargarse tanto del SO como de las gestiones de los grafos en ejecución. Adicionalmente, este procesador podría tener que ejecutar también otras tareas; por lo que, si se encuentra demasiado ocupado gestionando la ejecución de las URs, se pueden producir retardos importantes no sólo en la ejecución de sus propias tareas, sino que también en el

rendimiento general del sistema. Una forma de reducir el impacto de este problema es distribuir el SO y las computaciones de las planificaciones entre los elementos de proceso. Esta aproximación reduce tanto la carga computacional del procesador como las costosas comunicaciones HW/SW. Algunos ejemplos interesantes de sistemas HW multitarea son [Nollet04a], [Nogue04a] y [So06]. En [Nollet04a] los autores proponen un SO distribuido con soporte para comunicaciones entre tareas. En [Nogue04a] los autores proponen un planificador dinámico basado en HW para arquitecturas reconfigurables que aplica una heurística de planificación en listas. Sin embargo, los autores no implementaron su diseño, sino que sólo la incluyeron en su entorno de simulación específico. Finalmente, en [So06] los autores proponen incluir módulos HW distribuidos que colaboran con un procesador para soportar llamadas de SO de tipo UNIX para tareas HW.

En un sistema HW basado en una FPGA multi-tarea, optimizar el proceso de reconfiguración es una cuestión clave. Es por ello por lo que varios autores han propuesto incluir soporte para reducir las penalizaciones que producen estas reconfiguraciones. En [Res05b], [Li02], [Qu06], [Nogue04a] y [Shang02a] los autores proponen técnicas de planificación que intentan reducir estas latencias. [Qu06] y [Shang02a] centran su trabajo en aplicaciones estáticas mientras que las técnicas restantes se aplican en tiempo de ejecución. Estas técnicas asumen que un procesador empotrado se encarga de realizar todas las computaciones. Sin embargo, en ninguno de estos trabajos se implementan estas técnicas en un procesador empotrado para, de esta manera, evaluar el impacto en el rendimiento que tiene aplicarlas. Por otro lado, [ZLi02] trata el problema de asignar tareas en la FPGA tratando de maximizar su uso. Sin embargo, los autores asumen que una tarea se puede colocar en cualquier lugar de la FPGA, lo que no es una suposición demasiado realista a menos que cada vez que se cargue una nueva tarea se llame a un proceso muy costoso. Finalmente, [Res05b] propone una técnica de reemplazo muy eficiente, que utiliza listas enlazadas para obtener resultados cercanos a los óptimos.

En el planificador desarrollado se ha incluido una optimización de prebúsqueda muy similar a la presentada en [Res05b], pero adaptada a una implementación HW. Esta técnica genera penalizaciones insignificantes en tiempo de ejecución y, de acuerdo con los experimentos realizados, oculta la mayor parte de las latencias de reconfiguración. Además, este planificador aplica una técnica de reutilización, especialmente diseñada para una implementación HW eficiente, que no sólo mejora el porcentaje de tareas reutilizadas cuando éstas se ejecutan recurrentemente, sino que también mejora los resultados de la técnica de prebúsqueda desarrollada.

CONTRIBUCIÓN DEL PROYECTO

La principal contribución de esta memoria es la descripción y la evaluación de las prestaciones de un planificador de grafos de tareas para sistemas multitarea dinámicamente reconfigurables. Este planificador controla la ejecución de una serie de grafos de tareas en un conjunto de unidades reconfigurables teniendo en cuenta las dependencias de control y de datos de los grafos de tareas dados. Asimismo, cada vez que se debe realizar la reconfiguración de una tarea, el planficador decide en tiempo de ejecución en qué UR debe ejecutarse. La información acerca de los grafos de tareas que maneja el sistema se guarda en una tabla asociativa.

Para tomar estas decisiones, y debido a que la latencia de reconfiguración puede generar retardos significativos en la ejecución (para las FPGAs estos retardos son del orden de milisegundos [Shang02a]), el planificador aplica dos técnicas para reducir estas penalizaciones. En primer lugar, antes de reconfigurar una UR para cargar una tarea en ella, el planificador comprueba si esa tarea había sido cargada allí previamente. Si esto ocurriese, la configuración puede ser reutilizada directamente sin necesidad de cargarla; y en caso contrario, selecciona una de las URs disponibles utilizando un algoritmo de reemplazo especialmente diseñado para reducir la latencia de reconfiguración. En segundo lugar, el planificador aplica una técnica de prebúsqueda [ZLi02] para ocultar las latencias de reconfiguración en la medida de lo posible.

Además, hemos diseñado ambas técnicas para que colaboren entre ellas. Para ello nuestro algoritmo de reemplazo tiene en cuenta cómo afectan sus decisiones a la estrategia de prebúsqueda, y trata de no reemplazar aquellas tareas que vayan a reutilizarse próximamente o que no puedan ser prebuscadas sin generar penalizaciones. Por último, se ha realizado una implementación HW de todas las técnicas desarrolladas y se han integrado en un simulador multi-tarea reconfigurable que se ejecuta en una FPGA.

OBJETIVOS Y LOGROS REALIZADOS

Entre los objetivos iniciales del proyecto se incluyó el diseño e implementación en una FPGA de un módulo gestor básico, que con la información de un grafo de tareas y una planificación dada, fuese capaz de guiar su ejecución en una serie de unidades reconfigurables. Asimismo, también se pretendía la realización de una comparativa con un gestor SW equivalente.

El lenguaje escogido para la implementación HW del gestor fue VHDL, al tratarse de uno de los más extendidos en el diseño HW. Una vez que ésta estuvo completa, todos los módulos fueron sintetizados y comprobados desarrollando bancos de pruebas para las herramientas de simulación Post Place & Route en el ISE 9.1i. Estas simulaciones garantizan el mismo resultado que una ejecución en la FPGA final, pero resulta mucho más cómodo trabajar con ellas al poder definirse de forma sencilla todo tipo de bancos de pruebas. Finalmente, se realizó una comparativa con un gestor SW equivalente implementado en C y que se ejecutó en un POWERPC con ayuda de la herramienta EDK 9.1. Los resultados obtenidos nos confirmaron la rapidez del sistema HW frente al SW.

En un siguiente paso, procedimos a integrar este módulo como un periférico más de una arquitectura diseñada mediante la herramienta EDK 9.1. De este modo pudimos realizar comunicaciones punto a punto entre el procesador y este primer gestor en ambos sentidos. No obstante, el avance más importante de esta integración supuso comprobar el correcto funcionamiento de este gestor ejecutando grafos de tareas en una micro-arquitectura real. También procedimos a mejorar la versión SW del sistema, creando un completo programa en C que se comunicaba con contadores programables (que representaban las URs) utilizando órdenes proporcionadas por la interfaz de comunicación HW/SW proporcionada por EDK. En este caso, la gestión dinámica de los eventos se producía por medio de interrupciones que los contadores generaban cada vez que querían generar un evento. En este paso sirvió para que tuviéramos datos más exactos acerca de la evaluación de las prestaciones de ambas implementaciones.

Finalmente, el último avance realizado ha sido ampliar la funcionalidad de este gestor convirtiéndolo en un planificador; es decir, dotándole de la capacidad para decidir en tiempo de ejecución en qué UR efectuar cada reconfiguración. Para ello, se ha desarrollado una política de reemplazo, a la que hemos llamado LF+C (*Look Forward* + *Critical*). Ambas versiones (HW y SW) han experimentado esta mejora, por lo que conocemos (y exponemos) en detalle qué impacto en el rendimiento tiene su utilización en un sistema multi-tarea. Por tanto, el trabajo expuesto en esta memoria es el resultado de la implementación y el testeo de todas estas versiones incrementales (tanto en HW como en SW) del planificador, cada una construida sobre la anterior.

ESTRUCTURA DEL DOCUMENTO

El presente documento está dividido en dos partes bien diferenciadas. En la primera se realiza una aproximación tecnológica de todos los dispositivos, técnicas de reconfiguración y software involucrados en la realización del proyecto. Se describen en detalle las características y funcionalidades de los sitemas reconfigurables, y las herramientas de diseño empleadas para la realización del proyecto: *Xilinx ISE 9.1i, ModelSim 6.0a* y *Xilinx EDK 9.1i*.

Finalmente, se ofrece una visión general de la reconfiguración dinámica, sus alternativas de uso, y cómo y por qué es interesante utilizala para implementar un sistema con varias unidades reconfigurables.

En la segunda parte de la memoria, se describe todo el diseño del proyecto. En primer lugar, se ofrece una visión general (abstracta) del problema que queremos abordar. En segundo lugar, se ofrece una visión general del planificador que hemos desarrollado e implementado; para finalmente, explicar todas las unidades en que se divide el mismo: describimos las FIFOs utilizadas, la tabla de tareas (entrada de la tabla, red iterativa, HW de control y finalmente, su esquemático a alto nivel), los módulos que caracterizan a las unidades reconfigurables, las FIFOs de reconfiguraciones y de eventos, el árbitro que controla las escrituras en la FIFO de eventos y la unidad de control. Asimismo, para comprender mejor el funcionamiento del sistema, explicamos la ejecución de un grafo de tareas paso a paso. Por otro lado, se realiza un análisis comparativo para conocer las ventajas que proporciona este diseño hardware sobre un control meramente software. Finalmente, exponemos algunas conclusiones y posibles ampliaciones del proyecto que se pueden realizar como trabajo futuro.

Como apéndice, se muestra el código VHDL desarrollado, código C utilizado para la versión software comparativa y un glosario de términos.

APROXIMACIÓN TECNOLÓGICA

SISTEMAS RECONFIGURABLES

Introducción

Los métodos convencionales para realizar computación son dos. El primero de ellos consiste en el procesamiento hardware utilizando circuitos cableados de forma fija, bien mediante la integración en un circuito de aplicación específica (Application Specific Integrated Circuit -ASIC-) o bien conectando componentes individuales en una placa [Com02a]. El segundo método, denominado procesamiento software, se basa en el uso de microprocesadores que ejecutan un conjunto de instrucciones para realizar la computación.

El primer sistema se caracteriza por su rapidez y eficiencia para la aplicación concreta para la que ha sido diseñado, pero el circuito no puede ser alterado después de la fabricación, lo que le resta flexibilidad. Usando un microprocesador se incrementa la flexibilidad del sistema para poder cambiar la funcionalidad empleando otro software, pero se reduce la eficiencia debido a las secuencias necesarias de lectura, decodificación y ejecución de instrucciones.

Los dispositivos reconfigurables vienen a cubrir el espacio existente entre estos dos métodos, de forma que se disponga de la eficiencia del procesamiento hardware y de un alto grado de flexibilidad [Tuley97]. La adaptabilidad de las arquitecturas reconfigurables permite explotar el paralelismo existente en muchas aplicaciones de forma que se realice computación específica. En la figura 1 se muestra cómo los sistemas configurables, basados en dispositivos reconfigurables, se sitúan en una zona intermedia en la relación flexibilidad-especialización. No son tan flexibles como un procesador de propósito general ni tan específicos, ni tan óptimos, como un ASIC. Sin embargo se benefician de las características positivas de ambos.

Las diferencias más importantes entre la lógica reconfigurable y el procesamiento convencional se pueden resumir en los siguientes aspectos según K. Bondalapati y V. K. Prasanna [Bon02]:

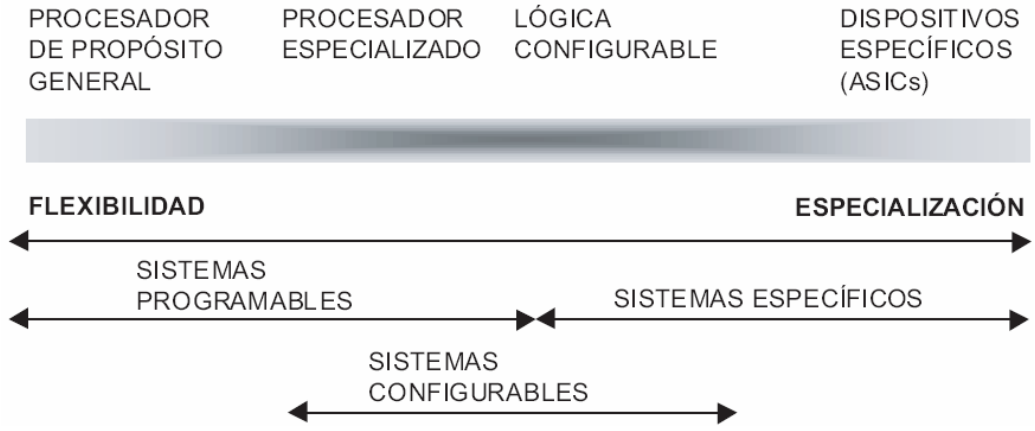


Figura 1. Sistemas de computación según el grado de flexibilidad

- **Computación espacial**: El procesamiento de los datos se realiza distribuyendo las computaciones de forma espacial, en contraste con el procesamiento secuencial.

- **Ruta de datos (datapath) configurable**: Empleando un mecanismo de configuración es posible cambiar la funcionalidad de las unidades de computación y de la red de interconexión.

- **Control distribuido**: Las unidades de computación procesan datos de forma local en

vez de estar gobernados por una única instrucción.

- **Recursos distribuidos**: Los elementos requeridos para la computación se encuentran distribuidos por todo el dispositivo, en contraste con una única localización.

Otras características importantes de estos sistemas destacadas por A. DeHon [DeHon96a] y R. Tessier [Tes01] son la capacidad de adaptación, la posibilidad de configuración en tiempo de ejecución y la especialización. Beneficiándose de estas características específicas se han desarrollado sistemas reconfigurables eficientes para aplicaciones como la programación genética [Sid99a], detección de patrones de texto [Sid99b, Sid01], criptografía [Hau97, Leung00, Laz04], compresión de datos [Huang00] o procesamiento de imágenes [Burns97, Gau02] entre otras.

En este apartado se realizará un repaso de varias arquitecturas reconfigurables que han dado soporte al desarrollo de la disciplina de la Computación Reconfigurable. A lo largo del mismo se presenta la evolución sufrida desde los primeros dispositivos FPGAs hasta las plataformas híbridas que integran microprocesadores de propósito general y ASICs en el mismo chip entre otros elementos. De esta forma la distribución de la computación puede ser repartida entre los distintos componentes del sistema. También se ha incluido una sección donde se presentan los distintos modos de configuración de los dispositivos reconfigurables, siendo estas distintas modalidades uno de los factores más característicos de los mismos.

Tecnología Reconfigurable: FPGAs

Las FPGAs consisten en una matriz de bloques lógicos (Logic Blocks -LBs-) y una red de interconexión. La funcionalidad de los LBs y las conexiones de la red de interconexión pueden modificarse mediante la descarga de los bits de configuración en el hardware [Bon02]. La configuración del dispositivo se realiza empleando dispositivos anti-fuse [Ahr90] o bits de memoria SRAM que controlan la configuración de los transistores [Hsieh90]. El primer modo de configuración tiene menor capacidad de reprogramación mientras que la configuración mediante elementos de memoria SRAM es más versátil admitiendo incluso reconfiguración dinámica y/o parcial.

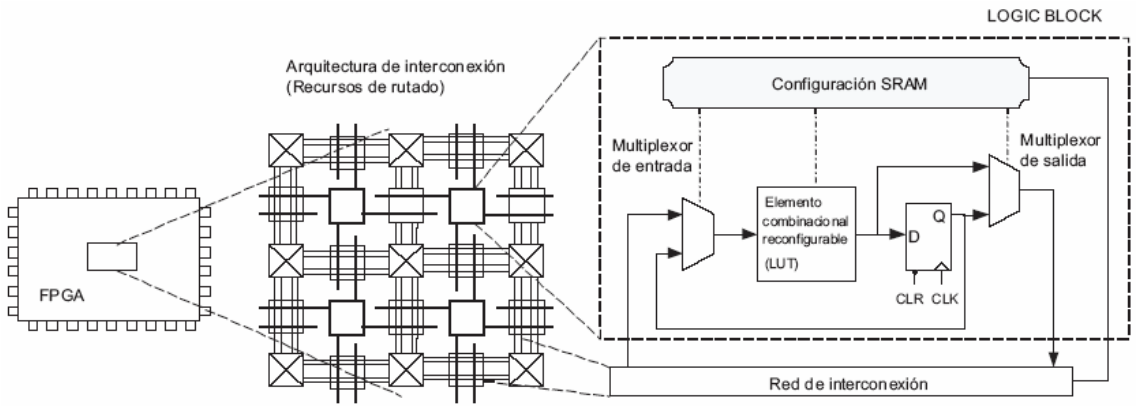


Figura 2. Modelo genérico de una FPGA

La figura 2 muestra la estructura interna simplificada de una FPGA. A modo ilustrativo se ha elegido para la representación una distribución de tipo isla empleada en varias familias de Xilinx. Existen otras arquitecturas de interconexión como la basada en filas [Actel97], sea-of-gates [Actel01], jerárquica o estructuras en una única dimensión como las empleadas en Garp [Hau97], Chimaera [Hauck04] o NAPA [Rupp98].

Los LBs interconectados mediante esa red contienen típicamente un circuito combinacional programable Look-Up Table (LUT), un biestable, lógica adicional y las celdas de memoria

SRAM requeridas para la configuración de todos los elementos. Las tareas de entrada-salida se realizan en la periferia del dispositivo, bien mediante LBs o disponiendo de bloques específicos denominados Input-Output-Blocks (IOBs). Actualmente se integran habitualmente otros elementos como son los bloques de memoria RAM dedicada [Xil00], multiplicadores e incluso microprocesadores.

Una de las clasificaciones más habituales del HW reconfigurable se realiza atendiendo a su granularidad. La granularidad de la lógica reconfigurable se define como el tamaño de la menor unidad funcional que es tratada por las herramientas de emplazamiento y rutado [Bon02]. Las FPGAs de grano fino disponen de elementos funcionales de pequeño tamaño, lo que las dota de una mayor flexibilidad. Sin embargo, sufren de retardos elevados cuando se componen circuitos complejos. De forma típica son unidades funcionales de 2 a 4 entradas. El HW reconfigurable con unidades funcionales grandes se denominan de grano grueso, existiendo arquitecturas como la Chameleon [CS00] con elementos aritméticos de 32 bits o la Morphosys compuesta por un componente reconfigurable, un procesador RISC y una interfaz de memoria con un gran ancho de banda.

Con la integración de múltiples elementos arquitecturales (entendidos como procesadores, memoria e interfaces para periféricos) en FPGAs que disponen de una sección de lógica programable por el usuario surge el concepto de Arquitecturas Híbridas [Bon02].

El avance tecnológico que ha permitido la integración de sistemas en un único dispositivo ha ido acompañado de diversa terminología, todavía no normalizada, para designar a estos sistemas. A continuación se detallan algunos de estos términos:

- **System-on-Chip (SoC)**: Circuito integrado formado por diversos módulos VLSI con distinta funcionalidad que interconectados entre sí ofrecen toda o casi toda la funcionalidad específica para una aplicación.

- **System-on-Programable-Chip (SoPC)**: Se aplica este término específicamente cuando el dispositivo utilizado para realizar el sistema en un chip es reconfigurable. En los SoPC no se utiliza la capacidad de reconfiguración dinámica que puedan disponer estos integrados, sino únicamente las facilidades que ofrecen estos dispositivos en la fase de desarrollo y posteriores actualizaciones del sistema.

- **Configurable-System-On-Programable-Chip (CSoPC)**: Mediante este término se definen los sistemas SoPC en los que se hace uso de la capacidad de reconfiguración de los mismos para aplicaciones de Computación Reconfigurable. Pueden incluirse bajo la denominación CSoPC tanto los sistemas que admiten diferente configuración estática según ciertas condicionantes, como los que utilizan la reconfiguración parcial dinámica para modificar en tiempo de ejecución una sección hardware.

- **Multiprocessor-Configurable-System-On-Programable-Chip (MCSoPC)**: Se aplica esta definición a los sistemas CSoPC que incluyen varios procesadores que ejecutan software, funcionando de forma simultánea.

Dentro de las distintas arquitecturas dinámicamente reconfigurables actuales, las FPGAs dominan ampliamente el mercado. La razón principal es que se basan en una tecnología madura con muchos años de desarrollo y sustentada por un amplio conjunto de herramientas de diseño. Por esta razón, el trabajo realizado en este proyecto ha tomado a las FPGAs como punto de referencia a la hora de desarrollar los prototipos y evaluar los resultados.

Tipos de configuración

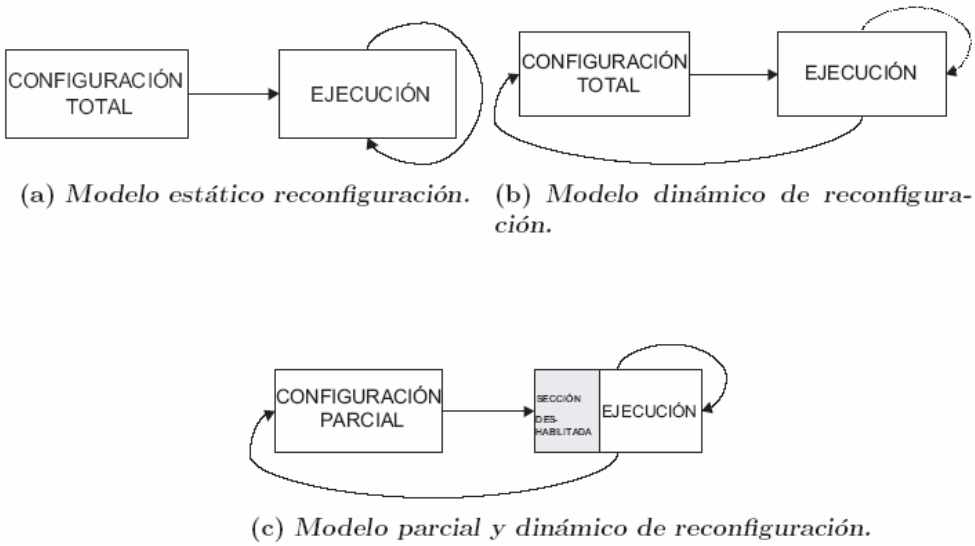
De forma general un dispositivo reconfigurable se configura cargando en el mismo una secuencia de bits denominada bitstream. El modo de carga varía según la interfaz que éste disponga. Las interfaces de configuración son de tipo serie o paralelo. El tiempo de configuración es directamente proporcional al tamaño del bitstream. Las FPGAs de grano fino tienen, en general, tiempos de configuración mayores que las de grano grueso debido al mayor

tamaño de sus bitstreams. Esto es debido a que tienen muchos elementos para ser configurados lo que implica ficheros de configuración grandes.

Las FPGAs tradicionalmente se han utilizado para realizar una determinada función en un único contexto, realizándose una configuración de todo el dispositivo. En el caso de que se desee reconfigurar en tiempo de funcionamiento, la reconfiguración de todo el dispositivo es un proceso lento y limitado. La lógica que se va a reconfigurar debe parar la computación y continuar tras la nueva configuración. La penalización impuesta por el tiempo de reconfiguración es importante [Li00], haciendo en muchas aplicaciones inviable la aplicación de la reconfiguración. A continuación se presentan brevemente los modelos de reconfiguración más representativos:

- **Reconfiguración estática**: Implica parar el sistema y reiniciarlo con una nueva configuración. Su utilidad se centra en los procesos de diseño (depuración) y en la actualización de sistemas. Cada aplicación dispone de una configuración que se carga una vez tras el encendido. La mayoría de los sistemas realizados en la actualidad con lógica reconfigurable disponen de este tipo de reconfiguración, también denominada reconfiguración en tiempo de compilación (en el proceso de diseño). La figura 3(a) sintetiza este modo de operación en el que tras la configuración comienza la ejecución de la lógica configurada sin posibilidad de una nueva carga.

- **Reconfiguración dinámica**: Con el objetivo de obtener un equilibrio entre velocidad de ejecución y área de silicio surge el modo de configuración dinámico mediante el cual se modifica la lógica configurable (incluyendo el rutado) en tiempo de ejecución. La reconfiguración dinámica se basa en el concepto de Hardware Virtual [And99] de forma similar a la memoria virtual. Utilizando la capacidad de reprogramación del dispositivo se cambian las configuraciones según se requieren distintas computaciones, reduciendo de esta manera el área de circuito necesaria.



(a) *Modelo estático reconfiguración.* (b) *Modelo dinámico de reconfiguración.*

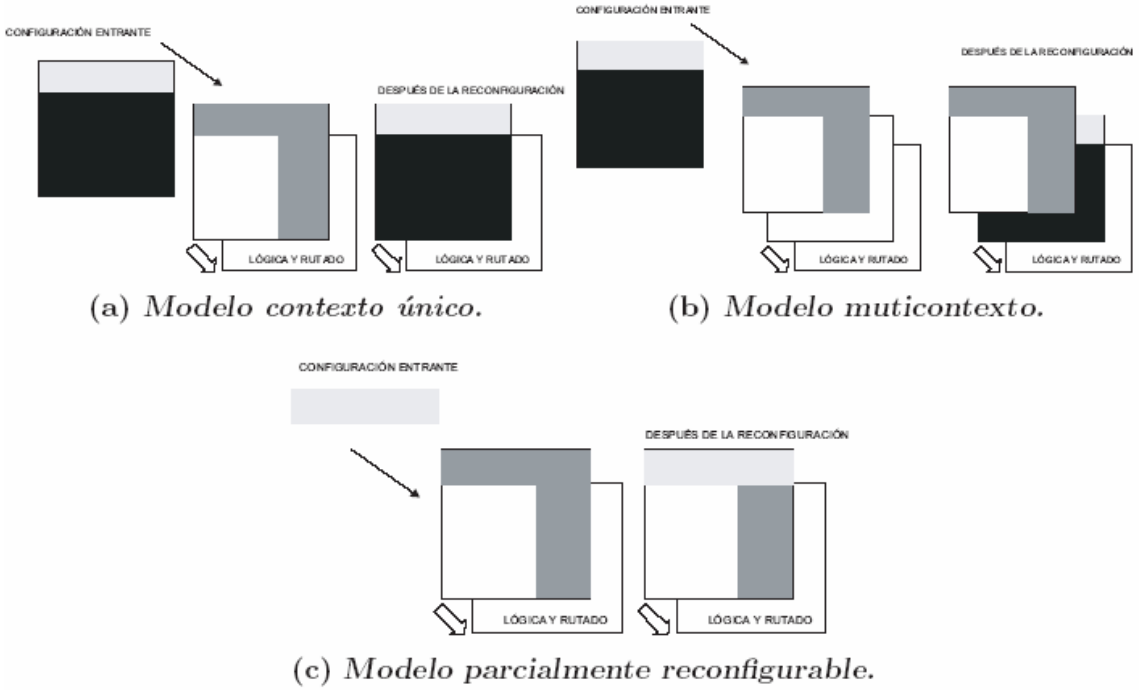
(c) *Modelo parcial y dinámico de reconfiguración.*

Figura 3. Modelos de configuración

La figura 3(b) representa el modelo de reconfiguración dinámico. De la computación realizada por la lógica configurada (ejecución) se obtiene información que sirve para determinar la nueva configuración. En el caso de que el sistema lo permita, se podría realizar la aplicación de la nueva configuración mientras se mantiene la ejecución. En la práctica, la solución más habitual es la de mantener deshabilitada la sección que se va a reconfigurar mientras continúa la ejecución en la otra sección del dispositivo. Este modo de reconfiguración parcial dinámico se representa en la figura 3(c).

Por tanto, la reconfiguración dinámica tiene diversas variantes según el modo en el que

ésta se aplique. Las tres más representativas son:



(a) *Modelo contexto único.* (b) *Modelo muticontexto.*

(c) *Modelo parcialmente reconfigurable.*

Figura 4. Modelos de reconfiguración dinámica

- **Reconfiguración contexto único**: Corresponde con el modo de configuración de las FPGAs que únicamente soportan un acceso secuencial a la memoria de configuración. En el caso de realizar una reconfiguración dinámica con estos dispositivos se sufren unas penalizaciones temporales importantes debido a que cada intercambio de funcionalidad requiere una reprogramación completa de los mismos. El modelo de este modo se ha representado en las figuras 4(a) y 4(b). La configuración entrante sustituye completamente a la que estaba aplicada sobre la lógica configurable.

- **Reconfiguración multi-contexto**: Los dispositivos que soportan este tipo de reconfiguración tienen varios bits de memoria de configuración para cada bit de los elementos configurables [Tri97, DeHon96b]. En la figura 4(b) se representa un modelo de estos dispositivos donde los bits de memoria pueden considerarse como múltiples planos de información de configuración. Cada plano debe configurarse totalmente, de igual forma que los de contexto único. Sin embargo, el cambio entre contextos se realiza de forma muy rápida, admitiéndose además la carga de una nueva configuración en un plano no activo mientras otro lo está.

- **Reconfiguración parcial**: Uno de los avances tecnológicos más importantes en el área de la reconfiguración consiste en la capacidad de algunos dispositivos para admitir la modificación de parte de la configuración mientras el resto del hardware sigue realizando la computación de forma ininterrumpida.

  La figura 4(c) muestra este modelo de reconfiguración. En este caso el plano de configuración funciona como una memoria RAM. De este modo se pueden emplear las direcciones para especificar una determinada localización que de desea reconfigurar. La reconfiguración parcial dinámica también permite que se carguen configuraciones diferentes en áreas no usadas del dispositivo con el fin de reducir la latencia en el cambio de contexto, tal y como se propone en el trabajo de K. Danne [Dan03].

  Existen varias plataformas reconfigurables que soportan la reconfiguración parcial como Chimaera [Hauck04], PipeRench [Chou00], NAPA [Rupp98], Xilinx 6200 [Xil02a] y Virtex [Xil03a]. Una variante de la reconfiguración parcial realizada con el fin de

reducir la penalización por el tiempo necesario para la carga de los bitstreams parciales es la reconfiguración pipeline. En este caso, la reconfiguración ocurre en incrementos de etapas de pipeline [Sch97]. Este sistema está orientado a computaciones de estilo datapath, donde se emplean más etapas pipeline que las que caben simultáneamente en el hardware. El caso más avanzado sería una situación donde comenzaría la computación de cada etapa tras el instante de ser programada. En este caso la configuración de cada etapa se situaría un paso a la cabeza del flujo de datos [Com02b].

Para aliviar la problemática de la penalización en tiempo impuesta por el proceso de reconfiguración, se han investigado diversas tácticas. La precarga de configuraciones [Hauck98a] es una de ellas. El objetivo en este caso es cargar la configuración en el dispositivo con anticipación a que se requiera ésta. El carácter especulativo de esta técnica fija la complejidad de la misma en determinar con suficiente antelación cuál va a ser la siguiente configuración requerida. También la compresión de la configuración se ha estudiado como una de las alternativas para la reducción del tiempo de configuración [Hauck98b].

Otra alternativa abordada con el mismo fin es la del uso de caché de configuraciones en el dispositivo [Com02b]. Al retener configuraciones en el integrado, se reduce la cantidad de información de configuración transferida. Al igual que en las cachés de los procesadores de propósito general, se aplican los conceptos de localidad temporal y espacial con el fin de decidir qué configuraciones se mantienen y cuáles se eliminan cuando se produce la reconfiguración. Decisiones incorrectas pueden producir el efecto contrario al deseado, aumentando la penalización temporal producida por la reconfiguración.

Xilinx Virtex-II Pro

Xilinx es una empresa especializada en el desarrollo de dispositivos programables (FPGAs, CPLDs, etc.) desde 1985. El modelo de FPGA utilizado en este proyecto es el Virtex-II Pro, desarrollado por Xilinx en 2004. A continuación se enumeran algunas de sus características principales:

- Arquitectura lógica programable
  - Tecnología de 130nm y 9 capas de cobre
  - Desde 3000 hasta 99000 celdas lógicas
  - Velocidad de reloj superior a los 400MHz
  - Alto rendimiento y bajo consumo
- Escalabilidad. Hasta 11 formatos.
- Características avanzadas
  - Memoria distribuida y empotrada (Flash)
  - Gestión de relojes digitales
  - Reconfiguración de la FPGA total/parcial en función de la actualización de los productos del mercado
  - Tecnología XCITE que permite mejorar la integridad de la señal y reducir espacio
- Conectividad
  - Hasta 20 *serial transceivers* (RocketIO) *full-duplex* (de 622Mbps hasta 3.125Gbps)
  - Conexiones a 100MHz mediante LVDS
- Procesamiento avanzado
  - Dos procesadores empotrados IBM PowerPC 405 a 400MHz
- Herramientas de desarrollo
  - Herramientas para la programación de la FPGA

La FPGA con la que se ha trabajado es la XC2VP30-5 FG676C de familia Virtex-II PRO de Xilinx. La familia Virtex-II se caracteriza por cargar los datos de configuración en celdas internas de memoria estática. Los valores guardados en estas celdas determinan las funciones lógicas y las interconexiones implementadas en la FPGA. El hecho de guardar los valores dentro de estas celdas permite infinitas reprogramaciones del dispositivo. Además en esta

FPGA la configuración puede cambiarse en tiempo de ejecución (reconfigurción dinámica) lo que posibilita la realización de un sistema multiprocesador hardware con varias unidades reconfigurables y uno o varios procesadores.

El nombre de la FPGA, XC2VP30-5 FG676C, indica lo siguiente:

- **Tipo de dispositivo:** XC2VP30, dispositivo 30 de la familia Virtex-II PRO.
- **Grado de velocidad:** 5, estándar.
- **Tipo de embalaje:** FG, Fine Pitch BGA 27 x 27 mm, 1.0 mm ball pitch.
- **Número de pines:** 676.
- **Rango de temperatura:** C, comercial (entre 0º y 85º).

VHDL

El VHDL (*Very High Speed Integrate Circuit Hardware Description Language*), es un lenguaje de descripción y modelado, diseñado para describir la funcionalidad y la organización de sistemas *hardware* digitales, placas de circuitos y componentes.

La finalidad del modelado es la simulación. La sintaxis amplia y flexible del lenguaje VHDL permite tanto el modelado estructural como el modelado funcional de circuitos. En el primer caso, se describe el circuito indicando los componentes y las conexiones que lo componen (lo cual requiere un conocimiento detallado del circuito). En el segundo caso, se describe el circuito indicando lo que hace y cómo funciona, es decir, describiendo su comportamiento (sin necesidad de conocer su estructura interna).

Esta segunda metodología de modelado resulta muy interesente desde el punto de vista del diseño de sistemas digitales. Más aún teniendo en cuenta que hoy en día otra de las aplicaciones del lenguaje VHDL, con una gran demanda de uso, es la síntesis automática de circuitos. En el proceso de síntesis, se parte de una especificación de entrada con un determinado nivel de abstracción, y se desciende verticalmente por los niveles de la jerarquía de diseño hasta llegar a una implementación más detallada, menos abstracta. Puesto que el VHDL fue inicialmente concebido para el modelado de sistemas digitales, su utilización en síntesis no es inmediata. Sin embargo, la sofisticación de las actuales herramientas de síntesis es tal que permiten implementar diseños especificados en un alto nivel de abstracción.

Así pues, VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción hasta el nivel de definición estructural de puertas lógicas. Además, siguiendo ciertas guías para síntesis, permite la implementación de diseños a nivel de puertas lógicas. Al estar basado en un estándar (IEEE Std. 1076-1987) reduce errores de comunicación y problemas de compatibilidad. Finalmente, dada su característica de modularidad, permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.
Los componentes de un proyecto VHDL son los siguientes:

- **Entity**: Es el más básico de los bloques de construcción en un diseño. Una entidad VHDL especifica el nombre de la entidad, sus puertos, e información relacionada con ella. Todos los diseños son creados usando una o varias entidades. La entidad describe la interfaz en el modelo VHDL.

- **Architecture**: La arquitectura describe la funcionalidad esencial de la entidad y contiene los estados que modelan el funcionamiento de ésta. Una entidad puede tener varias arquitecturas.

- **Configuration**: Permite unir la instancia de un componente a la pareja entidad-arquitectura. Describe el comportamiento a utilizar para cada entidad.

- **Package**: Es una colección de los tipos de datos y subprogramas usados comúnmente en un diseño. Las librerías forman parte de los *packages*.

PAQUETES SOFTWARE UTILIZADOS

Para el desarrollo de la implementación en VHDL del planificador y su posterior testeo, se utilizaron los programas Xilinx ISE 9.1i y ModelSim 6.0a respectivamente. Posteriormente, se utilizó el programa Xilinx EDK 9.1i, que es un *wrapper* para ISE que proporciona una serie de herramientas que permiten diseñar un sistema empotrado completo para su implementación posterior en una FPGA de Xilinx. Se utilizó esta herramienta para integrar el sistema en una arquitectura real y para desarrollar un sistema SW equivalente sobre PowerPC.

Xilinx ISE 9.1i

El entorno de desarrollo ISE de Xilinx posee un aspecto similar al de los entornos de programación actuales como puede ser Visual Basic o Visual C, es decir, posee diversas ventanas para la visualización de tareas específicas sobre cada una de ellas. En este caso existen cuatro tipos de ventanas (figura 5):

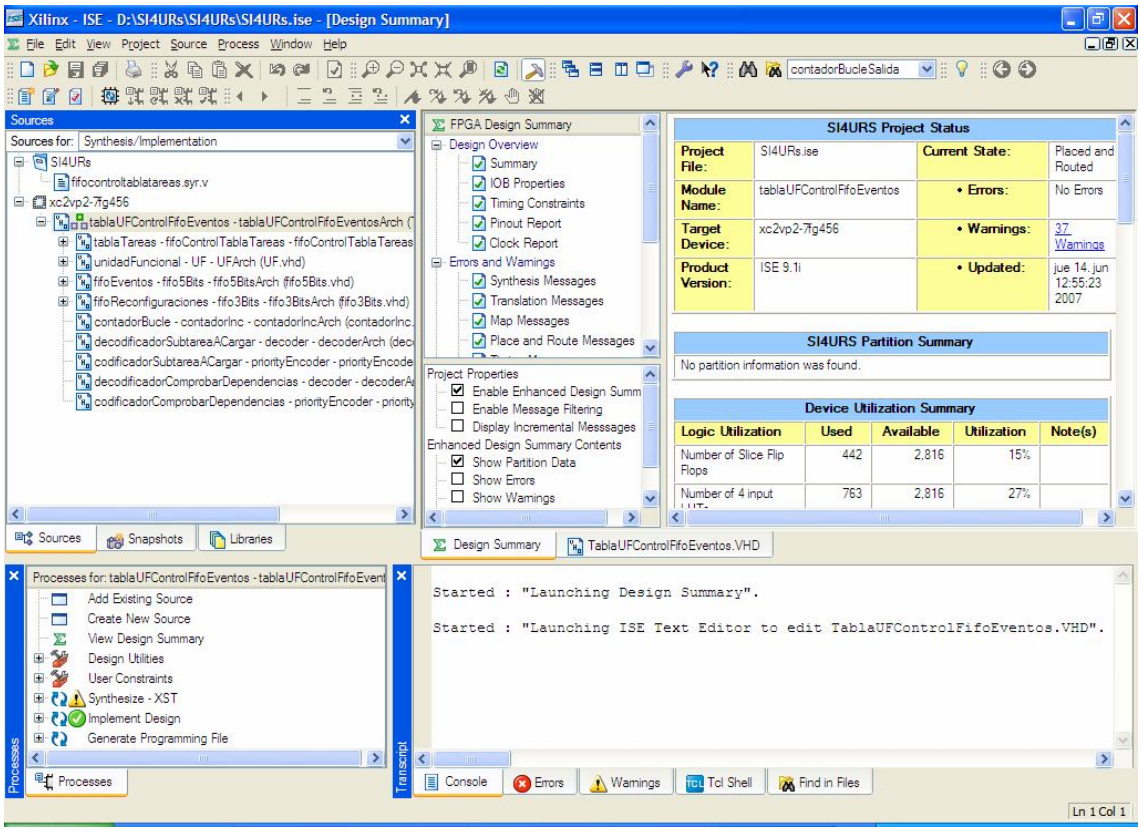


Figura 5. Pantalla principal del entorno Xilinx ISE

1. Ventana de ficheros fuente. En esta ventana se muestran los ficheros fuentes utilizados en el diseño y las dependencias entre ellos. También es aquí donde se elige el tipo de dispositivo donde se desea almacenar el diseño. Esta ventana posee diversas solapas para visualizar diferentes tipos de información relativa a las fuentes de diseño empleadas.
2. Ventana de Procesos. Esta ventana muestra todos los procesos necesarios para la ejecución de cada etapa de diseño. La lista de procesos se modifica dinámicamente dependiendo del tipo de fuente seleccionado en la ventana de ficheros fuente.
3. Ventanas de edición. Al hacer doble clic sobre un fichero fuente de la ventana de ficheros fuente se abre una ventana de edición para modificar el fichero (en caso de

lenguaje VHDL), o bien se ejecuta el programa que permite editar el diseño (en caso de diseños esquemáticos o máquinas de estado).

4. Ventana de información, situada en la parte inferior. Muestra mensajes de error, aviso o información emitidos por la ejecución de los programas de compilación, implementación, etc.

Tanto en la ventana de procesos como en la de ficheros fuente es posible modificar las opciones de cada elemento a través del botón derecho del ratón, o bien a través de los menús del entorno de diseño; estos menús se modifican dependiendo del tipo de selección realizada en las ventanas de ficheros fuente y de procesos. Cada elemento mostrado en la ventana posee un icono diferente dependiendo del tipo de acción o fichero de que se trate, por ejemplo, nos indica si un elemento es un documento de texto, una acción para ejecutar en el entorno ISE o una acción para ejecutar por un programa adicional como puede ser ModelSim a la hora de simular. También muestra información sobre el estado que ha dado resultado tras la ejecución del proceso, es decir, si ha sido satisfactorio, si ha tenido errores, o ha tenido avisos. Las imágenes de la figura 6 muestran un ejemplo de los diferentes tipos de información mostrados en estas dos ventanas. Para la ventana de ficheros fuente, se indica si un fichero es de código, de vectores de test, si es un paquete, o una selección de dispositivo.

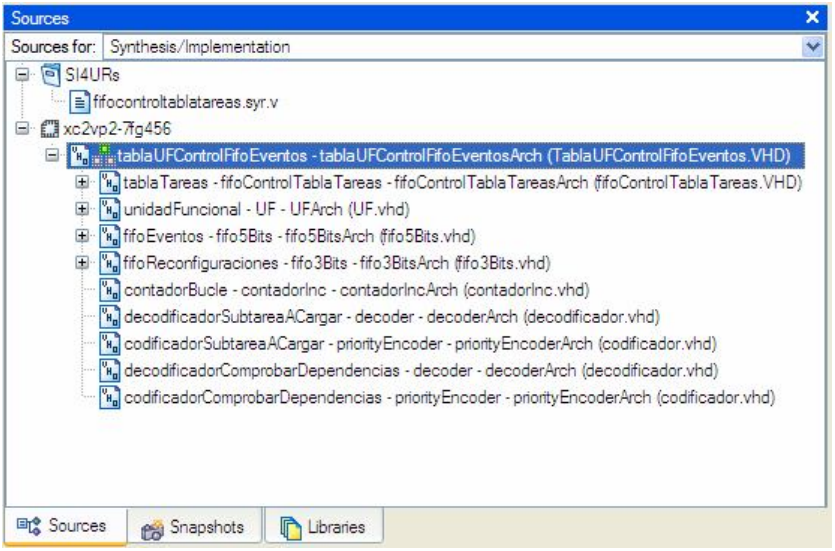


Figura 6. Detalle de las ventanas de ficheros fuente y de procesos

En concreto, la ventana de procesos incorpora todas las opciones necesarias para realizar todos los pasos de implementación de sistemas en lógica programable, incluyendo la edición y verificación. La figura 7 muestra el diagrama de flujo de diseño en Xilinx ISE, y la figura 8 muestra las diversas partes en que se divide la ventana de procesos dependiendo de la tarea a realizar.

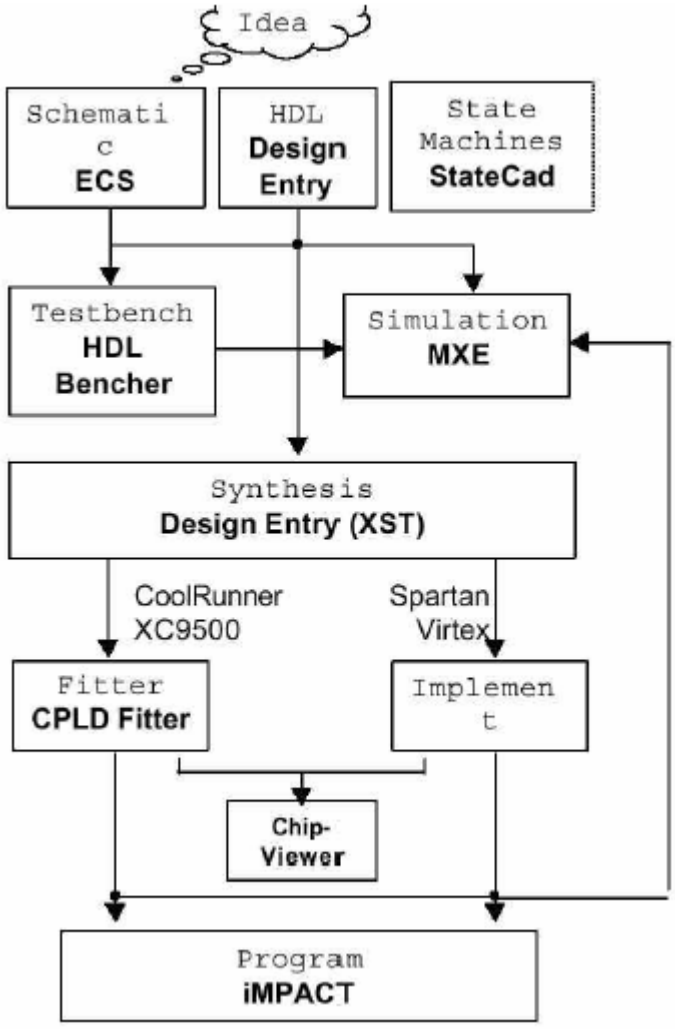


Figura 7. Proceso de diseño en Xilinx ISE

De manera resumida, el proceso de diseño resulta sencillo y se realiza en tres pasos, el primero consiste en añadir los ficheros fuente, en el segundo paso se selecciona el fichero de más alto nivel que se quiere implementar, y finalmente se hace doble clic sobre el último proceso al que se desea llegar, de este modo se ejecutarán todos los procesos intermedios necesarios para llegar al proceso seleccionado en último lugar (figura 9).

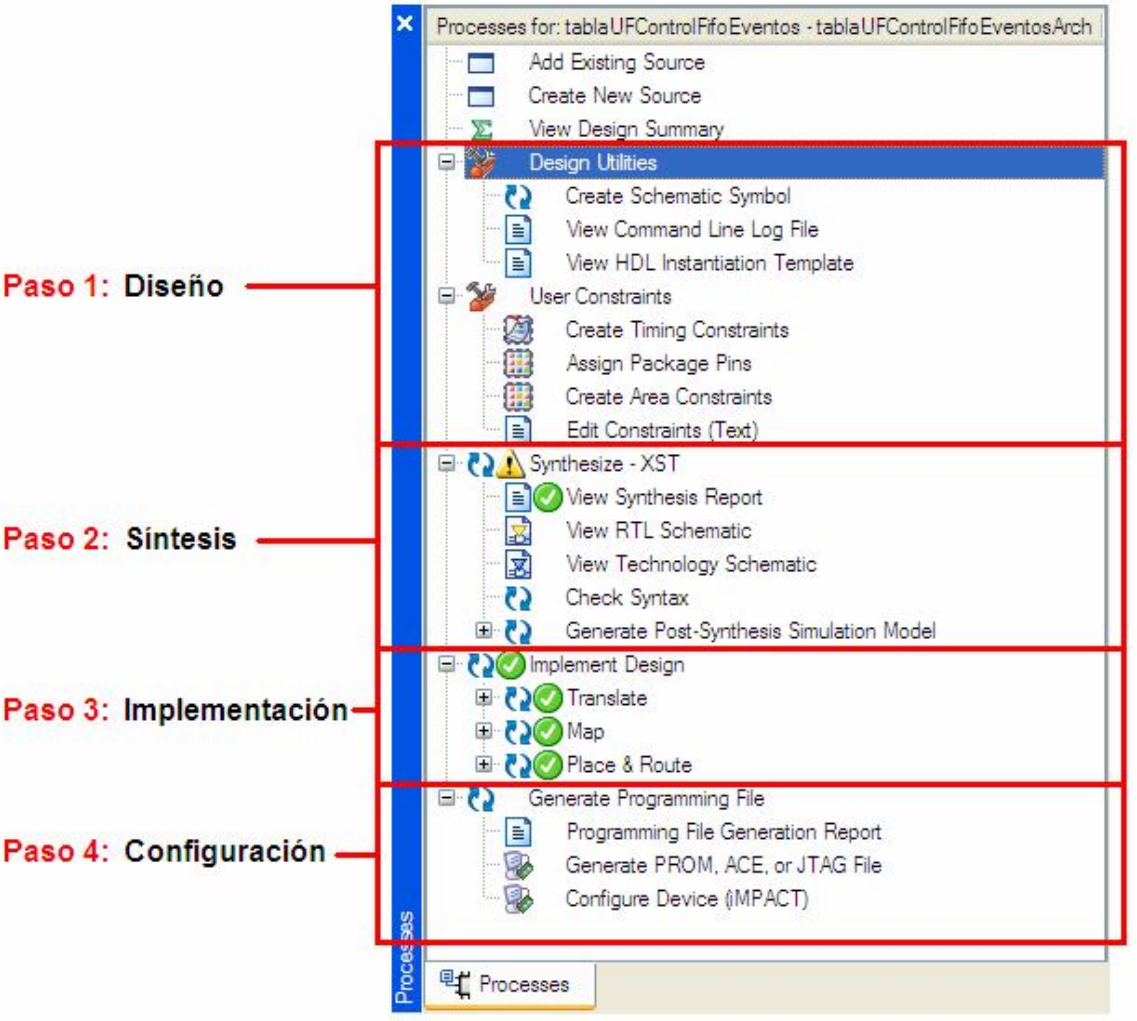


Figura 8. División de tareas dentro de la ventana de procesos

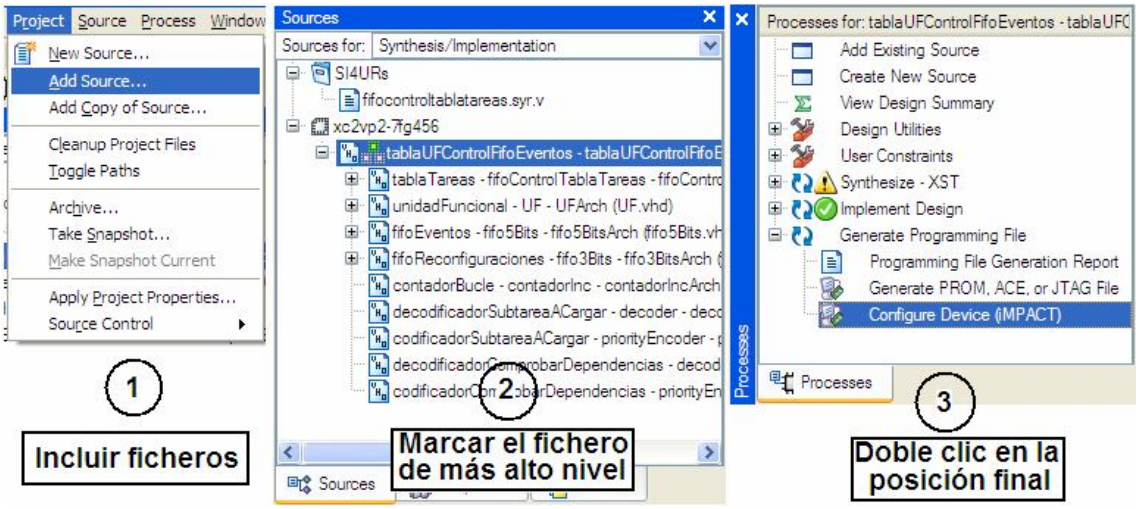


Figura 9. Proceso simplificado para el desarrollo de un diseño en Xilinx ISE

En el proyecto, este software se utilizó para la implementación en VHDL del gestor y del planificador y para el testeo de los módulos básicos, con el propio simulador que incorpora ISE.

ModelSim 6.0a

Los bancos de prueba (o *testbenchs*) son una parte esencial del proceso de diseño, ya que permiten comprobar su correcto funcionamiento y ayudan en la automatización del proceso de verificación del diseño. Recoger estadísticas de la cobertura del código durante la simulación ayuda a asegurar la calidad y la minuciosidad de las pruebas.

El software ModelSim SE 6.0 utilizado en este proyecto, proporciona un entorno integrado de depuración (*Integrated Debug Environment*) que facilita el depurado eficiente de diseños basados en FPGAs, programados en cualquiera de los tres lenguajes siguientes: VHDL, Verilog y SystemC.

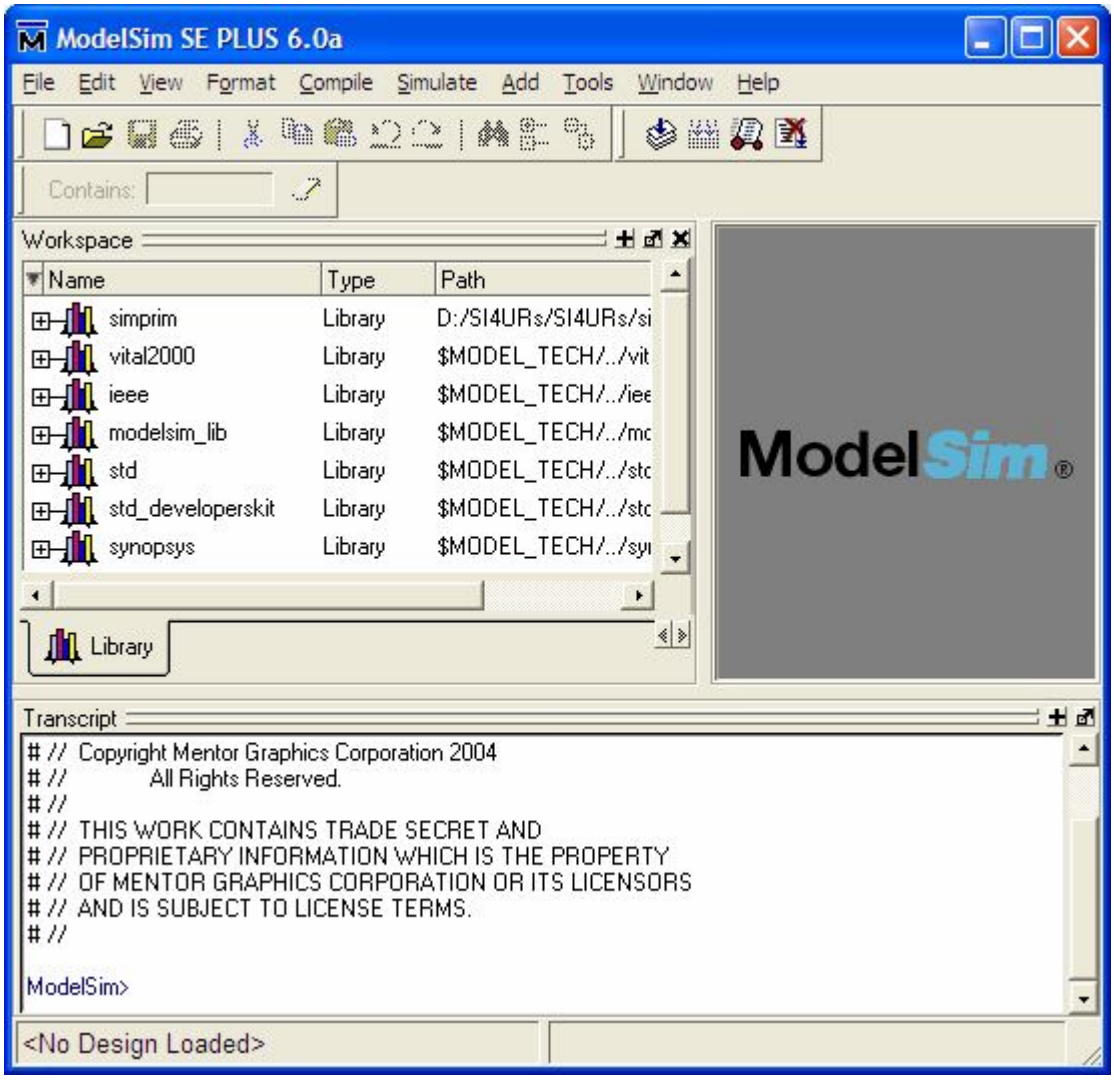


Figura 10. Ventana principal del simulador ModelSim SE 6.0

Además de simular el funcionamiento del sistema diseñado, el ModelSim SE 6.0 permite visualizar las señales de cada puerto del bloque simulado y realizar las modificaciones necesarias del código en función del resultado obtenido.

Partiendo del código proporcionado por el programa Xilinx ISE 9.1i, la simulación se puede realizar a distintos niveles: desde el nivel más alto en el que se utilizan modelos “ideales” de los componentes hasta el nivel Post Place & Route donde cada componente ha sido mapeado a un elemento concreto de la FPGA y posteiormente han sido conectadas entre sí, de forma que la simulación se basa en modelos muy precisos de los componentes que incluso incluyen los retardos de las conexiones.

Xilinx EDK 9.1i

Xilinx EDK es la abreviatura de Xilinx Embedded Development Kit y como indica la palabra es un conjunto de herramientas para el diseño y desarrollo de sistemas embebidos. Esta herramienta de diseño se encuentra dividida en tres subbloques o subpestañas principales. En el primer de ellos, encontramos un catálogo con todos los elementos de que consta nuestro diseño “IP Catalog”; el segundo subbloque consta de la información del projecto “Project” y el tercer y último subbloque “Applications”, muestra las aplicaciones (en código C o C++) asociadas a nuestro diseño.

Asimismo contamos con la barra de herramientas y de iconos para generar el fichero .bit del proyecto, guardar, cargar, descargar dicho fichero a la fpga y muchas más opciones.

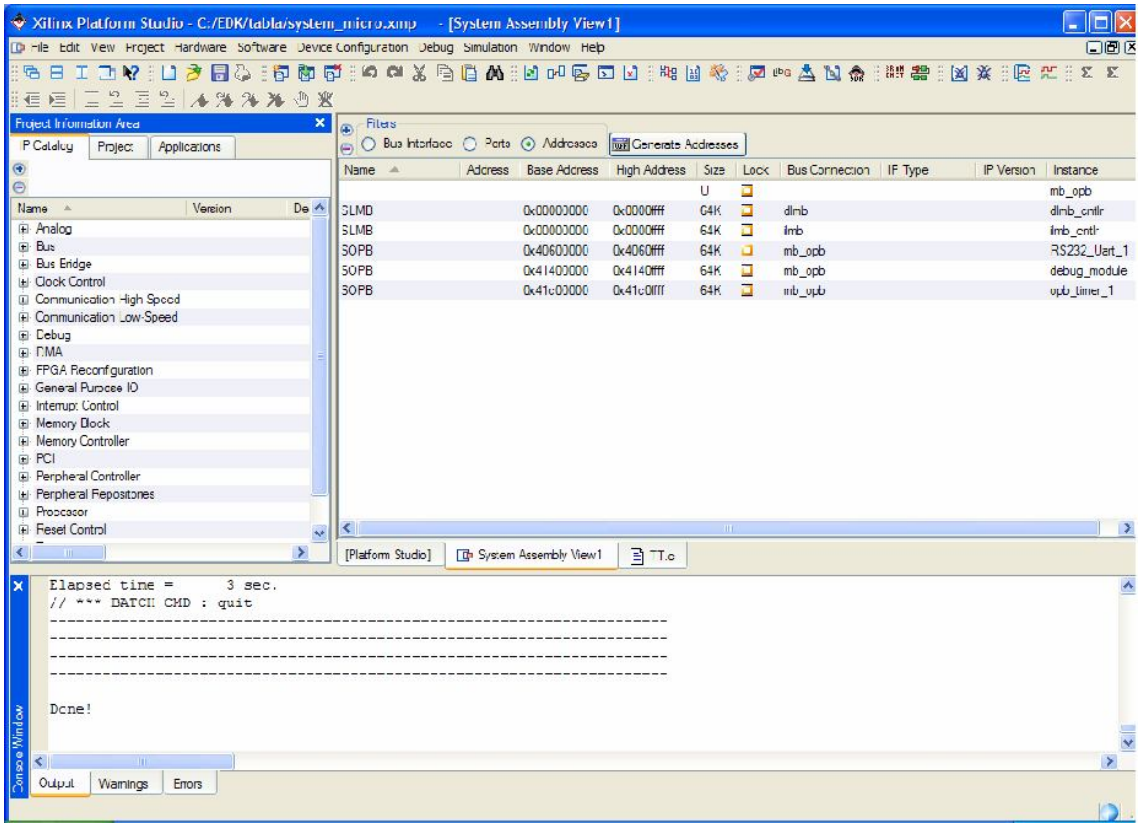


Figura 11. Xilinx EDK

RECONFIGURACIÓN DINÁMICA

Opciones arquitectónicas

El HW reconfigurable ha captado la atención de un amplio número de grupos de investigación desde su aparición. Está ganando popularidad tanto a nivel académico, donde cada vez son más los grupos de investigación dedicados a su estudio, como a nivel comercial. Este considerable desarrollo ha propiciado que existan numerosas opciones arquitectónicas por las que hay que decantarse a la hora de realizar un diseño HW que conviene comentar brevemente. En particular, podemos optar por los siguientes criterios:

1) ***Grano fino o grano grueso.*** Las arquitecturas de grano fino permiten cambiar la configuración para alterar el comportamiento de la placa a nivel de bit. Por otro lado, las arquitecturas de grano grueso trabajan con palabras de varios bits; partiendo del hecho de que las operaciones se realizan a nivel de palabra. El grano fino proporciona máxima flexibilidad para crear una configuración óptima de un algoritmo, a cambio de pagar un precio en área, longitud y cantidad de conexiones existentes, tiempo de reconfiguración, consumo de energía e incluso el rendimiento. El grano grueso representa el extremo opuesto.

2) ***Utilización de uno o múltiples contextos***. Una FPGA de un contexto es aquélla que sólo guarda la información de una reconfiguración en un instante de tiempo dado. Por otro lado, la utilización de múltiples contextos implica que la plataforma dispone de la memoria suficiente como para guardar múltiples configuraciones, así como la capacidad de cambiar de uno a otro en tiempo de ejecución cada vez que sea necesario. La inmensa mayoría de las FPGAs del mercado sólo soportan un único contexto, debido al sobrecoste de área de almacenamiento de los diferentes contextos, así como la lógica necesaria para la selección de los mismos.

3) ***Reconfiguración parcial o global.*** Hablamos de reconfiguración global cuando las reconfiguraciones de la FPGA se han de realizar simultáneamente en toda la placa. Por otro lado, cuando se permite que una parte de la plataforma pueda cambiar su funcionalidad mientras el resto permanece inalterado se habla de reconfiguración parcial. Esta última tiene dos ventajas importantes: el tiempo de reconfiguración es menor (al ser sólo parte del HW de la placa el involucrado) y permite gestionar de forma independiente la ejecución de múltiples tareas.

Para el desarrollo de este proyecto se ha optado por un modelo de grano fino combinado con algunos componentes empotrados dentro de la FPGA, al representar un compromiso entre consumo de recursos y rendimiento. Utilizamos un único contexto, ya que las FPGAs de que disponíamos sólo operaban de ese modo. Finalmente, hacemos uso de reconfiguración parcial, elección clave para que se pueda desarrollar un sistema que gestione la reconfiguración y la ejecución de tareas en varias unidades reconfigurables y/o varios procesadores.

Hacia un sistema con varias unidades reconfigurables

En la planificación típica de un sistema operativo, la existencia de muchos procesos y/o *threads* puede llegar a degradar considerablemente el rendimiento del sistema. Como posibles soluciones a este problema, se pueden plantear, fundamentalmente, dos mejoras: aumentar el número de procesadores y migrar tareas SW a HW. Ambas propuestas buscan paralelizar a nivel de datos y a nivel de cómputo.

El HW dinámicamente reconfigurable permite llevar a cabo estas dos mejoras (ejecución SW/HW), cuyas ventajas e inconvenientes son:

Ventajas: Se realiza una ejecución totalmente en paralelo y permite combinar de manera razonable una ejecución rápida con consumo de energía aceptable.

Inconvenientes: No es trivial pasar de SW a HW, así como tampoco lo es reconfigurar parcialmente y en tiempo de ejecución parte de la FPGA. Asimismo, las latencias de reconfiguración pueden degradar el rendimiento del sistema. Esto exite una buena planificación que las oculte; y estas planificaciones suelen ser costosas: A menudo hay que encontrar una planificación casi óptima para que merezca la pena llevarla a cabo y se manejan estructuras de datos complejas, como listas, grafos…

Quizá las características más interesantes del HW dinámicamente reconfigurable sean su versatilidad y el hecho de que se puedan ejecutar tareas totalmente en paralelo manteniendo un compromiso entre tiempo de reconfiguración y/o ejecución y consumo de energía. Unido a que queda aún por realizar en este campo un amplio trabajo de investigación (la gran parte de las posibilidades que ofrece el HW dinámicamente reconfigurable están todavía desaprovechadas debido a la falta de soporte que proporcionan los fabricantes) nos ha parecido suficiente como para habernos decantado por la realización de una labor de investigación propia en este campo desarrollando un planificador que permita la ejecución de tareas en una plataforma en HW dinámicamente reconfigurable siguiendo una planificación dada.

Reconfiguración parcial dinámica en dispositivos Virtex

La figura 12 muestra la distribución de dos tipos de *frames* en la memoria de configuración de la FPGA. Se distinguen tres zonas: a la izquierda y a la derecha, BRAM0 y BRAM1 incluyen los *frames* para la configuración de los datos de los bloques de RAM dedicados (BRAM *frames*) y en el centro de la matriz se sitúan los *frames* de configuración de los CLBs (CLB *frames*).

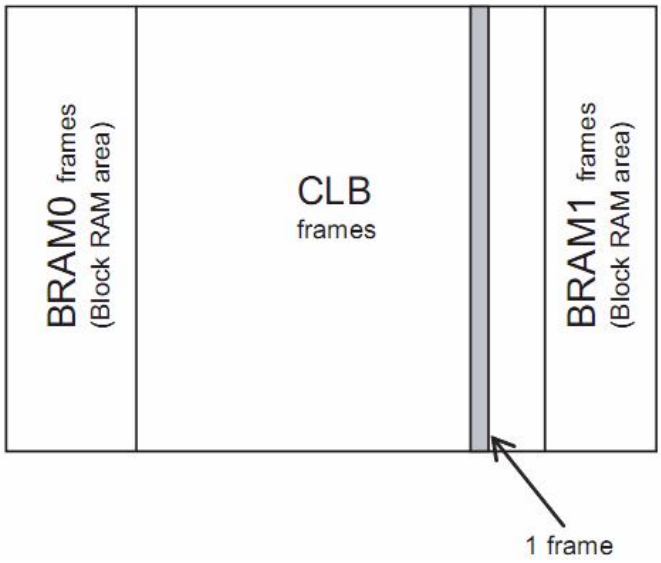


Figura 12. Distribución de los diferentes tipos de frames en un dispositivo Virtex

Los CLB *frames* contienen todos los datos de configuración para todos los elementos programables de la FPGA incluyendo los valores de las LUTs, lógica auxiliar, IOBs, elementos de control de los BRAMs y las interconexiones. De esta forma, se puede acceder a la configuración de estos recursos mediante una operación de escritura sobre el puerto de configuración de la FPGA, sin interrupción en el funcionamiento del circuito.

Además, como el contenido inicial de los biestables de los CLBs no se especifica en el *bitstream* de configuración, y por tanto, tampoco en el *frame*, los datos almacenados en los mismos no se ven modificados en el proceso de reconfiguración parcial.

En cuanto a los bloques de memoria RAM dedicada, se debe tener en cuenta que al escribir los BRAM *frames* en la memoria de configuración se está alterando su contenido. Por ello, si la aplicación accede a los mismos durante este periodo transitorio puede leer valores no válidos. Esta situación debe ser tenida en cuenta por el sistema de control de la reconfiguración que se esté utilizando en el circuito.

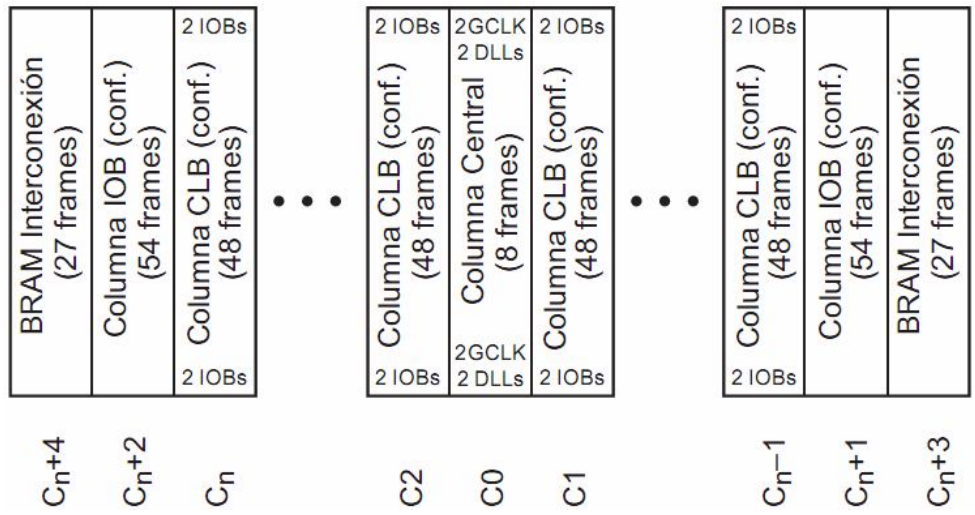


Figura 13. Distribución de las frame columns en la memoria de configuración

Los *frames* están agrupados en unidades superiores denominadas *frame columns* [Xil02b, Xil02b]. El área de la memoria de configuración de los CLB frames tiene cuatro categorías de *frame columns*: una columna central de 8 *frames*, columnas de 48 *frames* para la configuración de los CLBs, dos columnas de 27 *frames* que especifican el conexionado de los BRAMs y dos columnas para la configuración de los IOBs (54 *frames* por columna). En la figura 13 se muestra la distribución y numeración de las *frame columns* en la memoria de configuración, siguiendo una distribución de *ping-pong*.

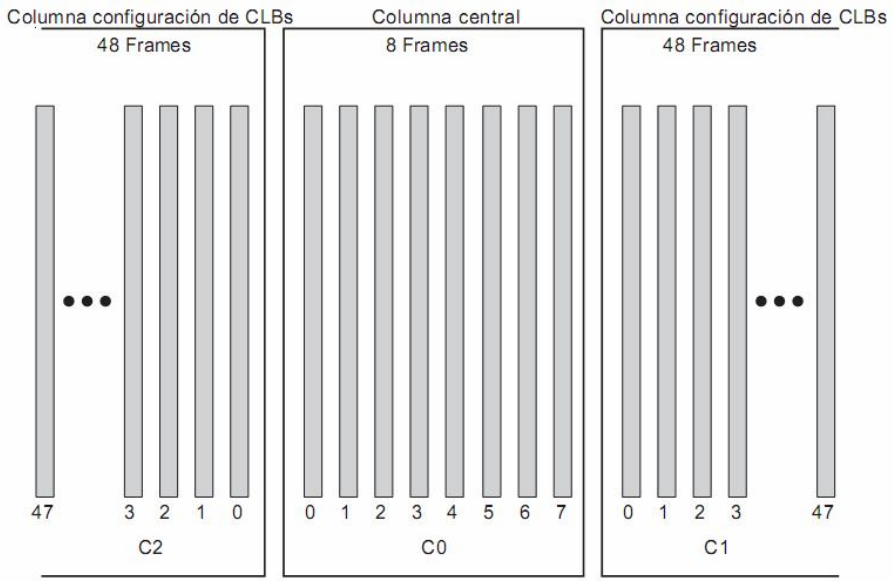


Figura 14. Distribución de los frames en los frame columns

Dentro de cada *frame column*, los *frames* se numeran secuencialmente desde el frame que se encuentre más cercano al centro hasta el más alejado, tal y como se muestra en la figura 14.

La dirección de un *frame* en la memoria de configuración está compuesta por dos números: el *major address*, que es el número de *column frame*; y el *minor address*, que es el número del *frame* en esa columna. Por tanto, los *bitstream* parciales de configuración incluyen las direcciones y los contenidos de los frames que se deben modificar, además de comandos específicos para el puerto de configuración.

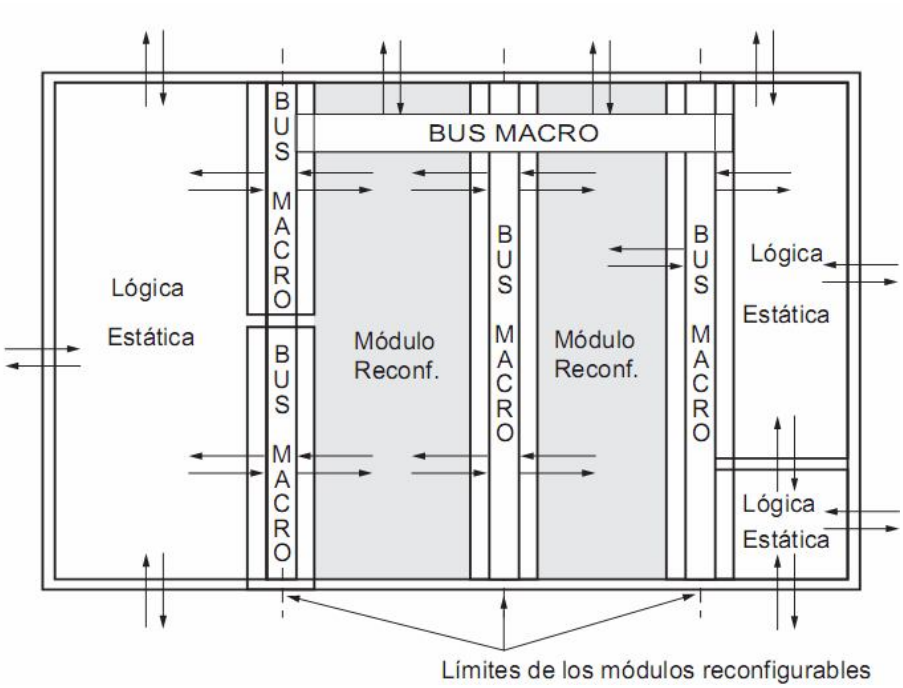


Figura 15. Conexión entre módulos reconfigurables para aplicar reconfiguración *inter-task* en dispositivos Virtex de Xilinx

Pero para poder aplicar la reconfiguración parcial dinámica sobre estos dispositivos, se deben tener en cuenta las siguientes restricciones generales [Xil02c]:

- La altura de una sección reconfigurable deberá ser la de toda la matriz de recursos lógicos.
- La anchura ocupada por un módulo reconfigurable deber ser múltiplo de 4 slices (2 CLBs). Por ejemplo, un módulo reconfigurable podría comenzar en coordenadas X=0, 4, 8, 10, etc. con anchuras de w=4, 8, 12, etc. slices.
- Toda la lógica que se defina en la anchura ocupada por ese módulo reconfigurable pertenece a ese módulo. Esto incluye tanto a los elementos que forman parte del CLB como a los *buffers* triestado (*Tbufs*), IOBs, multiplicadores y rutado.
- La lógica de reloj (*buffers globales*, DCM o DLL, etc.) tiene *frames* separados de los utilizados para configurar los CLBs y BRAMs.
- La configuración de los IOBs situados en el lateral derecho y en el izquierdo de la matriz reconfigurable se realiza mediante los CLB *frames* correspondientes a las columnas extremas.

- Los límites definidos para un determinado módulo reconfigurable no pueden ser cambiados dinámicamente, siendo fija la posición y el área que ocupa.

- La comunicación entre distintos módulos reconfigurables se debe realizar utilizando unos elementos fijos que definan rutas únicas de configuración. Estas herramientas se denominan *Bus-Macros* y se encuentran pre-rutadas antes de realizar la fase de emplazamiento y rutado de todo el diseño de la FPGA (figura 15).

Esta restricción se debe tener en cuenta cuando el tipo de reconfiguración parcial dinámica que se pretende aplicar es de tipo *inter-task*, lo que obliga al cambio completo del módulo reconfigurable.

- Las señales globales de *reset* y *set* no pueden utilizarse de manera local sobre un módulo reconfigurable, lo que implica que sea necesario disponer de señales de inicialización independiente para cada módulo.

DESARROLLO DEL PROYECTO

VISIÓN GENERAL DEL PROBLEMA Y EJEMPLO DE MOTIVACIÓN

El planificador desarrollado en este trabajo se enmarca en un sistema en el que una serie de eventos dinámicos dispara la ejecución de grafos de tareas en tiempo de ejecución. Los nodos de estos grafos pueden ejecutarse como tareas HW (que irán asignadas a las URs) o como tareas SW; y queda fuera de los objetivos de este trabajo decidir este particionamiento. El planificador que hemos desarrollado recibe tan sólo la información de los grafos que se ejecutan en recursos reconfigurables y garantiza su correcta ejecución teniendo en cuenta sus dependencias internas. Los grafos de tareas se representan como grafos dirigidos acíclicos en los que cada nodo representa una tarea computacional, y cada arista entre dos nodos representa una dependencia, que obliga, según el sentido de la misma, a ejecutar una tarea tras otra en el orden especificado. Estamos asumiendo que se dispone en tiempo de diseño de una estimación del tiempo de ejecución de cada tarea y que los grafos de tareas se ejecutan secuencialmente; es decir, el planificador no comenzará la ejecución de un grafo hasta que el grafo previo haya finalizado.

La función del módulo desarrollado es decidir en tiempo de ejecución en qué UR ubicar cada tarea cada vez que se solicite una reconfiguración y guiar la ejecución de los grafos de tareas garantizando que se respetan las dependencias entre las tareas y entre los grafos. Es decir, se debe garantizar que un grafo de tareas se ejecuta correctamente; y cuando sea necesario realizar una reconfiguración para cargar una nueva tarea, el planificador debe elegir una de las URs que esté libre (que no tenga tareas actualmente en ejecución) y reemplazarla por la nueva tarea. Obviamente, si todas las URs estuvieran ocupadas, este reemplazo no se podría efectuar.

Asimismo, al margen de esta la estrategia de reemplazo, se aplican las siguientes técnicas:

- **Prebúsqueda:** mediante esta técnica se persigue realizar las reconfiguraciones tan pronto como sea posible, teniendo en cuenta que las reconfiguraciones se deben realizar secuencialmente (sólo una en cada instante de tiempo).

- **Reutilización:** mediante esta técnica se persigue que, cada vez que una tarea deba ser cargada, si ya se encontrase en alguna UR debido a una ejecución previa, no se vuelva a reconfigurar en otra; sino que simplemente se vuelva a ejecutar en la misma UR.

Además, se han diseñado las técnicas de reemplazo, prebúsqueda y reutilización para que colaboren entre sí; de forma que al trabajar juntas mejoren sus resultados individuales.

Como ya se ha comentado, para tomar estas decisiones correctamente, el planificador debe recibir la información de los grafos de tareas: sus relaciones de precedencia y el tiempo de ejecución de cada nodo. A continuación, se realiza el proceso de planificación. Hemos dividido este proceso en tres fases: *Cálculo de pesos*, *identificación de tareas críticas* y *gestión y reemplazo en tiempo de ejecución*.

- **Cálculo de pesos:** en esta primera fase se asigna a cada nodo un peso, que no es más que un valor positivo. Estos pesos servirán para determinar el orden de la secuencia de reconfiguraciones que deben producirse.

- **Identificación de tareas críticas:** en esta segunda fase se identifican, para cada grafo de tareas, aquéllas que son críticas. Se trata del mínimo conjunto de tareas tales que, si todas ellas reutilizan, mientras que las demás (las no críticas) se deben reconfigurar, la heurística de planificación ocultará todas las latencias de reconfiguración. Por tanto, no generarían ninguna penalización temporal. Evidentemente, el subconjunto de tareas

críticas dependerá de la estrategia de planificación utilizada.

- **Gestión y reemplazo en tiempo de ejecución:** finalmente, con la información obtenida en las dos fases anteriores, se puede llevar a cabo la ejecución del grafo. Esta es la única etapa que se realiza íntegramente en tiempo de ejecución.

Para la realización de estos cálculos, es interesante que la mayor parte los mismos se realice en tiempo de diseño, porque así se reduce significativamente la penalización en tiempo de ejecución que se produce en el sistema. Por ello, las dos primeras etapas se realizan en tiempo de diseño, ya que sus resultados dependen únicamente de los datos de entrada (el grafo de tareas). Por el contrario, la última etapa se realiza en tiempo de ejecución; ya que, para que nuestro planificador tome las decisiones, necesita saber cuál es el estado del sistema, y esto sólo se puede saber en tiempo de ejecución.

El planificador desarrollado supone que las dos primeras fases ya se han efectuado y utiliza la información que proporcionan para ocuparse íntegramente de la última y decisiva etapa.

A continuación, procedemos a detallar cada uno de estas etapas. En el último apartado, incluimos un ejemplo de motivación que detalla a alto nivel el funcionamiento de nuestra estrategia de planificación.

Cálculo de pesos

El objetivo de esta fase es asignar un peso a cada tarea del grafo de tareas. Este peso representa cómo de crítica es la ejecución de cada tarea y se utilizará más adelante para decidir entre varias tareas cuál debe cargarse antes.

Para ello se calcula el camino máximo (en tiempo de ejecución) que separa el comienzo de la ejecución de dicha tarea del final de la ejecución de la última tarea del grafo. Se puede calcular este camino aplicando una planificación ALAP (*As Late As Possible*), según muestra el ejemplo de la figura 16.

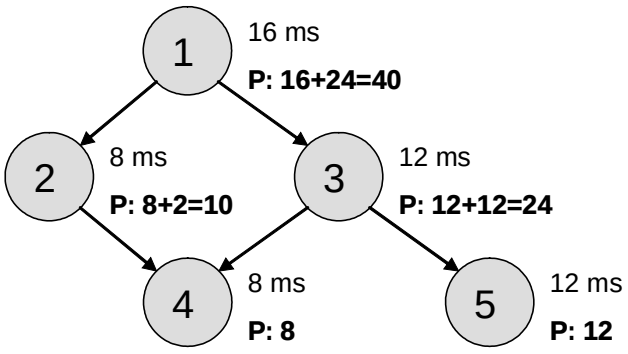


Figura 16. Ejemplo del cálculo de pesos para un grafo de tareas

Este cálculo se realiza de la siguiente manera: primero se empieza con los pesos de los nodos hoja (es decir, tareas que no tienen sucesores). Su peso será igual a su tiempo de ejecución. En el ejemplo vemos que los nodos 4 y 5 tienen de peso 8 y 12, que coinciden con sus tiempos de ejecución: 8 y 12 ms, respectivamente. A continuación, para el resto de los nodos, el peso asignado se calcula sumando a su tiempo de ejecución el peso del nodo sucesor con mayor peso. Por ejemplo, en el grafo de la figura, el nodo 3 tiene de peso: **12** (su tiempo de ejecución) **+ 12** (el máximo de los pesos de sus sucesores, que en este caso son 8 y 12) **= 24**. Siguiendo esta metodogía se calculan los pesos de todos los nodos recursivamente.

Una vez que se tienen etiquetados todos los nodos, éstos se ordenan según sus pesos, de mayor a menor valor. Ésta será la secuencia de reconfiguraciones que se efectuarán en el sistema, de manera que los nodos con mayor peso se reconfiguran antes que los de menor peso. En el ejemplo de la figura 16 esta secuencia sería: 1-3-2-5-4. La idea que hay detrás de determinar así la secuencia de reconfiguraciones es asegurarnos de dos cosas:

1) Si el nodo A es predecesor de B; entonces en la secuencia de reconfiguraciones, A debe aparecer antes que B. Efectivamente, esto ocurre, al ser P(A) > P(B)

2) Si dos nodos no tienen relaciones de precedencia entre ellos, en la secuencia de reconfiguraciones, aparecerá antes aquél cuyo peso sea mayor. Como el peso es una medida de cómo de crítica es una tarea, es lógico que las reconfiguraciones más críticas se realicen antes que las menos críticas, para retrasar lo menos posible la ejecución del grafo de tareas.

Por último, recordar que esta fase se efectúa en tiempo de diseño, al igual que la identificación de tareas críticas, detallada a continuación.

Identificación de tareas críticas

El objetivo de esta etapa es determinar las tareas críticas de un grafo de tareas. Para ello, la única información que se necesita es el conjunto de relaciones de precedencia entre los nodos, el tiempo de ejecución de los mismos y sus pesos. No necesitamos saber dichos tiempos de ejecución con una precisión del 100%; bastaría con que, gracias a esta información, se pudiera identificar qué nodos son más críticos para la ejecución del grafo. Por tanto, se pueden utilizar tiempos de ejecución medios después de haber ejecutado el mismo grafo con diferentes datos de entrada. No obstante, si se identificase algún nodo cuyo tiempo de ejecución varíe drásticamente para datos de entrada diferentes, se recomienda utilizar más de un grafo de tareas para representar estas variaciones, seleccionando en tiempo de ejecución el más apropiado para las entradas que se proporcionen en ese momento, tal y como se propone en [Yang02].

En cualquier caso, el resultado final será un conjunto de tareas críticas (*TC*). Lo definimos como el mínimo subconjunto de tareas de un grafo tales que, si se reutilizasen; reconfigurándose las demás, no se producirán penalizaciones debido a las reconfiguraciones. Para ello, utilizamos el algoritmo cuyo pseudo-código aparece en la figura 17.

```
Cálculo de tareas críticas (TC):
 1. TC := Ø
 2. Mientras (calcula_penalizaciones(TC) ≠ 0){
       S := tareas que generan retardos;
       S1 := MAX_peso(S);
       Añadir_tarea(S1, CS);
    }
```

Figura 17. Pseudo-código para la selección de las tareas críticas

Como se puede ver en el la figura, inicialmente el conjunto *TC* está vacío. A continuación, se ejecuta el bucle: En primer lugar, se aplica una estrategia de planificación que trata de ocultar las latencias de las reconfiguraciones de todas las tareas que no pertenezcan a *TC*. Mientras esta estrategia no consiga ocultar todas las reconfiguraciones, se identifican qué tareas generan los retardos y aquélla con mayor peso se añade a *TC*. Este bucle se sigue ejecutando hasta que las reconfiguraciones de las tareas que pertenezcan a TC no generen ningún retardo. La función *calcula_penalizaciones(TC)* asume que las tareas de *TC* se reutilizan.

Por tanto, vemos que las tareas críticas son aquéllas tales que, si no se reutilizan, generan retardos en sus reconfiguraciones. Este es el motivo que hace que sea especialmente interesante reutilizarlas.

Por último, comentar que este cálculo de tareas críticas depende de la estrategia de planificación, ya que la función *`calcula_penalizaciones(TC)`* utiliza dicha estrategia para decidir si una tarea genera penalizaciones o no. Por ejemplo, si se utilizase una estrategia de planificación que únicamente se limitase a cargar todas las tareas en la misma UR, todas ellas generarían retardos, y todas ellas serían críticas. En nuestro caso utilizamos la estrategia que hemos desarrollado para nuestro planificador y que se desarrollará más adelante.

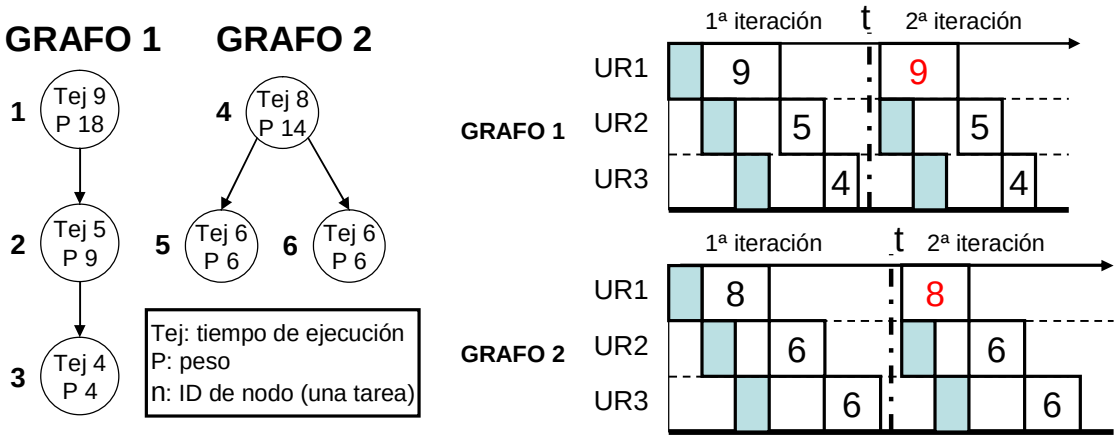


Figura 18. Ejemplo de cálculo de tareas críticas.

Vamos a ilustrar el funcionamiento de este algoritmo con un ejemplo. La figura 18 muestra dos grafos para los cuales se calcularán las tareas críticas. Estos dos mismos grafos se utilizarán en la siguiente sección para ilustrar el funcionamiento de nuestro planificador. Lo primero que se hace es calcular los pesos de los nodos. Siguiendo el algoritmo explicado anteriormente, tendríamos: P(3) = 4; P(2) = 5 + MAX {4} = 9; P(1) = 9 + MAX {9} = 18; P(5) = 6; P(6) = 6; P(4) = 8 + MAX{6, 6} = 14.

A continuación, se ejecuta el bucle *mientras*. Para el grafo 1, se supone inicialmente que todas sus tareas no son críticas y se ejecuta nuestro planificador (todo esto en tiempo de diseño, figura 18). Se han producido penalizaciones por reconfiguración (la primera tarea); por tanto, se selecciona la de mayor peso y se añade al conjunto de tareas críticas. Esto ocurre en la tarea 1, que pasaría a ser crítica. En la segunda iteración del bucle, se volvería a repetir el proceso asumiendo que la tarea 1 se reutiliza. En este caso, ya no se producen penalizaciones, al ocultarse las reconfiguraciones de 2 y 3 en el tiempo de ejecución de 1. Por tanto, el grafo 1 no tendría más tareas críticas. Análogamente, para el grafo 2, la tarea 4 sería crítica, mientras que 5 y 6 serían no críticas.

Una vez que se han calculado los pesos de las tareas del grafo, determinado la secuencia de reconfiguraciones del sistema e identificado las tareas críticas, se procede a la etapa de gestión y reemplazo en tiempo de ejecución, detallada a continuación.

Gestión y reemplazo en tiempo de ejecución

El objetivo de esta etapa es guiar la ejecución de un conjunto de grafos de tareas secuencialmente, utilizando la información proporcionada por las dos etapas anteriores. Esta es la etapa de la que se encarga íntegramente el planificador desarrollado en este trabajo.

Por tanto, nuestro planificador HW recibe como datos de entrada la información acerca de un grafo de tareas (las relaciones entre sus nodos y qué tareas son críticas) y la secuencia de reconfiguraciones, y planifica las reconfiguraciones y las ejecuciones necesarias, garantizando que se respetan las dependencias entre nodos. Asimismo, aplica una técnica de prebúsqueda para realizar las reconfiguraciones tan pronto como sea posible y la técnica de reemplazo, para decidir dónde cargar una nueva tarea en el caso de que no se pueda reutilizar.

Se pueden utilizar muchas técnicas de reemplazo, algunas tan simples como *First Free* (reemplaza en la primera UR que encuentre libre) o LRU (*Least Recently Used*, *Menos Recientemente Usada*), o tan complejas como LFD (*Longest Forward Distance*) [Bela66], que garantiza el resultado óptimo en cuanto a reutilización pero necesita conocer anticipadamente qué eventos van a suceder en el futuro. En este caso hemos optado por desarrollar y testear una técnica propia, a la que hemos llamado LF+C (*Look Forward* + *Critical*). Se trata de una estrategia que no obtiene resultados óptimos, ya que queremos trabajar con entornos dinámicos, no podemos conocer los eventos que se producirán en el futuro. No obstante, como demostraremos en la sección de "*Evaluación del rendimiento*", se obtienen resultados cercanos a los óptimos en cuanto a tareas reutilizadas y resultados incluso mejores que la estrategia LFD en cuanto a la reducción de la latencia de reconfiguración, como mostraremos en la sección de resultados experimentales. Además, no se penaliza casi en absoluto el rendimiento del sistema en tiempo de ejecución.

El funcionamiento de nuestra estrategia es el siguiente: cuando se deba efectuar la reconfiguración de una tarea, primero se comprueba si se puede reutilizar en alguna UR. Si esto no es posible, se aplica el reemplazo. Para ello, todas las URs que estén libres (no tengan ninguna tarea actualmente en ejecución ni en reconfiguración), son candidatas; y para cada una de ellas, se comprueba si la tarea que esté actualmente cargada en la unidad:

- está etiquetada como crítica.
- va a volver a ser ejecutada pronto; es decir, si sabemos que está actualmente esperando para su ejecución. Como los grafos se ejecutan secuencialmente, las únicas tareas que sabemos que se van a ejecutar en el futuro son las del grafo que está actualmente en el sistema.

En función de si hay alguna tarea en la UR; y si, en caso de existir, ésta cumple o no estas dos condiciones, cada unidad se clasifica en una de las siguientes cuatro categorías:

- **Candidatos libres (CL):** Aquéllos que no tienen ninguna tarea cargada.
- **Candidatos perfectos (CP):** Aquéllos que a corto plazo no se van a volver a ejecutar y tampoco son críticos.
- **Candidatos críticos (CC):** Aquéllos que no se van a volver a ejecutar y sí son críticos.
- **Candidatos reutilizables (CR):** Aquéllos que a corto plazo se van a volver a ejecutar, independientemente de si son críticos o no.

A continuación, se selecciona el candidato: si fuera posible, se reemplaza en un CL. Si no hubiera ninguno, se seleccionaría un CP. Si no hay CPs, entonces intentará reemplazar un CC; y sólo si esto no es posible, seleccionaría un CR. En el caso de poder elegir entre varios candidatos del mismo tipo, se selecciona el primero que encuentre.

En la figura 19 se describe una secuencia de ejecución en la que los dos grafos de tareas se ejecutan dos veces consecutivas en nuestro sistema siguiendo nuestra estrategia de reemplazo. Comparamos dicha ejecución con una estrategia más sencilla, LRU; y con la que obtiene resultados óptimos, LFD.

De momento fijémonos en la columna LF+C, que servirá para mostrar el funcionamiento de dicha estrategia. La primera reconfiguración que se solicita es la de la tarea 1. En ese momento, no hay ninguna tarea cargada en ninguna UR, y todas están libres, luego todas ellas son CLs. Se selecciona la UR1 porque es la primera que encuentra. A continuación, se efectúa la reconfiguración de 2, y ocurre lo mismo, con la diferencia de que esta vez la UR1 está en uso, y no es candidata para reemplazo. Las demás son CLs, y se selecciona la UR2. Análogamente, para la reconfiguración de 3, se selecciona la UR3.

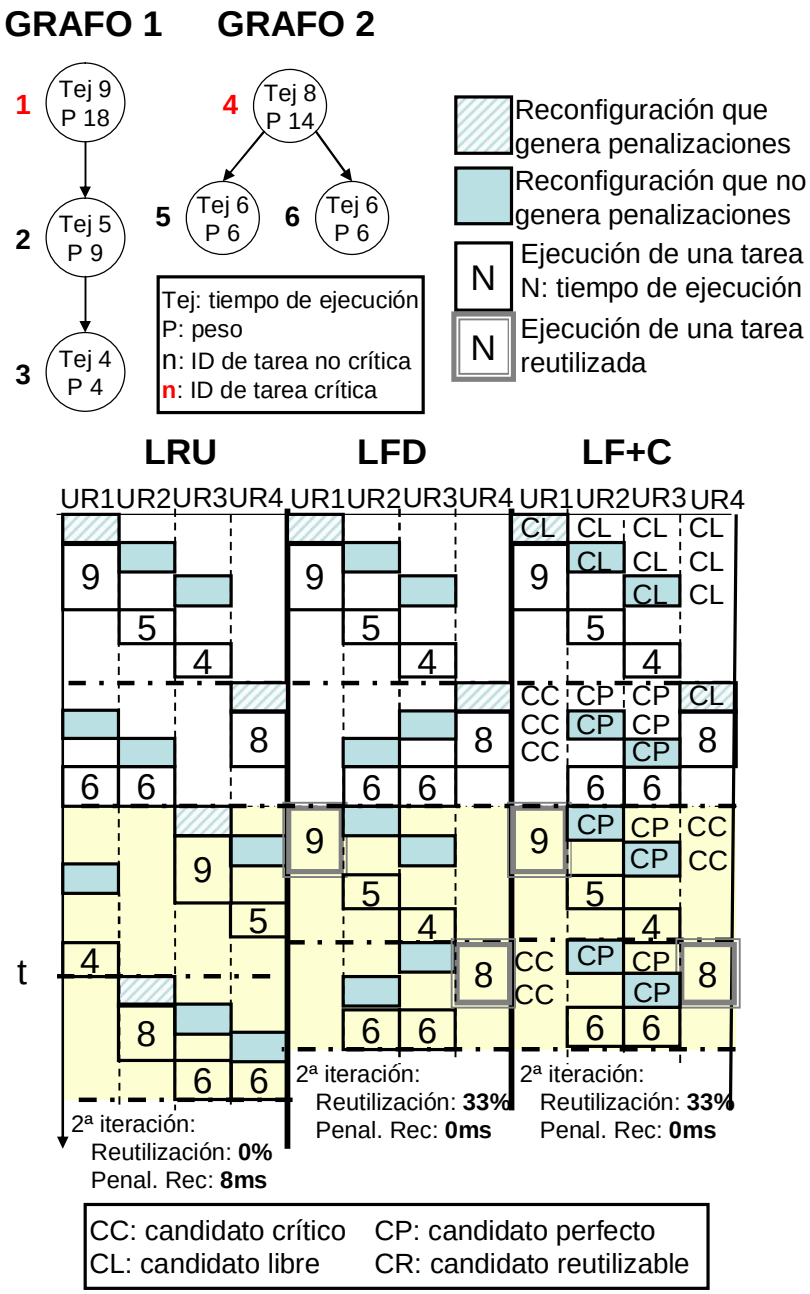


Figura 19. Ejemplo de motivación para un sistema con 4 URs y 4 ms de latencia de reconfiguración

A continuación, se ejecuta el grafo 2. Nótese que su primera reconfiguración (tarea 4) no se solapa con la ejecución de ninguna tarea del grafo anterior, ya que estamos asumiendo que nuestro sistema no comienza la ejecución de un grafo hasta que no haya terminado totalmente con el anterior. Cuando se intenta efectuar esta reconfiguración, todas las URs son candidatas a reemplazo, porque 4 no se puede reutilizar en ninguna y todas están ociosas. La UR1 es un CC, ya que tiene cargada 1, que se tiene etiquetada como crítica, y no está en el grafo actualmente en ejecución (2º grafo). Las URs 2 y 3 son CPs, porque las tareas 2 y 3 no son críticas ni están en el grafo actual. Finalmente, la UR4 es un CL, la única que aún no se ha utilizado. Siguiendo nuestro criterio de selección de candidatos, esta última es seleccionada para el reemplazo. Para la siguiente reconfiguración (tarea 5), ocurre exactamente lo mismo que en el caso anterior, con la diferencia de que la UR4 está en uso (ejecución de 4), con lo

que hay dos CPs y un CC. Se selecciona un CP, concretamente la UR2. Y para la reconfiguración de 6, sólo hay dos candidatos (UR1 y UR3), que son CC y CP respectivamente. Se realiza el reemplazo en la UR3.

El siguiente grafo a ejecutar es, nuevamente, el grafo 1. La primera reconfiguración es el nodo 1. Como ya está cargada en la UR1, se reutiliza y no es necesario cargarla, ahorrándonos así su tiempo de reconfiguración. Nótese que de no haber tomado las decisiones adecuadas en la ejecución del grafo anterior, (no reemplazar en la UR1 porque era un CC y había CPs) esta reutilización no habría sido posible. A continuación se cargan 2 y 3, en las URs 2 y 3 respectivamente, dejando la UR4 sin reemplazar por ser un CC. Esta decisión hace posible que, cuando entra a ejecutarse el grafo 2 otra vez, la tarea 4 pueda reutilizarse en la UR4. En consecuencia, hemos vuelto a reducir la penalización por reconfiguraciones. De este modo, conseguimos reutilizar el 33% de las tareas y eliminamos las penalizaciones por reconfiguración.

Por su parte, la columna LFD muestra que esta heurística es capaz de reutilizar también el 33% de las tareas y dejar las penalizaciones por reconfiguración a 0 ms, pero se debe recordar que sólo se puede aplicar LFD si se conocen con exactitud los eventos futuros. Por otro lado, la columna LRU muestra que, en este caso, esta estrategia no es capaz de reutilizar nada, al existir 6 tareas compitiendo por 4 URs. La penalización por reconfiguración asciende a 8 ms.

VISIÓN GENERAL DEL SISTEMA

La implementación se ha desarrollado considerando el planificador como un componente HW incluido como un periférico on-chip conectado con el procesador a través de un bus y con las URs a través de comunicaciones punto a punto. Se ha implementado utilizando el entorno de desarrollo Xilinx EDK 9.1i porque proporciona soporte para desarrollar e integrar periféricos; así como compilar, simular y depurar proyectos SW escritos en C o en C++. Además, facilita el manejo de interrupciones, comunicaciones HW/SW (usando la interfaz del periférico y la jerarquía proporcionada por el bus), transacciones por DMA y medidas de tiempos de ejecución (con una precisión de ciclos de reloj).

La figura 20 muestra una visión general del sistema completo con un planificador HW incluido como un periférico en el sistema, en la que se puede distinguir la estructura típica de un sistema objetivo; en nuestro caso, incluye un procesador Embebido Power PC, algunas RAM on-Chip que pueden usarse como caches de nivel 1 o *scratchpads*, una DDR *off-Chip* que puede usarse como memoria principal y una jerarquía de buses que puede usarse para extender la funcionalidad del sistema incluyendo diferentes periféricos. Por su parte, el planificador y las URs se incluyen como periféricos en el sistema conectados a traves de la línea de interrupciones y la jerarquía de buses. En este caso las URs no se comunican directamente con el procesador, sino que interactúan directamente con el planificador. Este esquema reduce drásticamente el número de interrupciones que el procesador debe gestionar; al incluir un planificador HW que se comunica directamente con las URs se evita que éstas deban generar una interrupción cada vez que se produzca un evento. Estas interrupciones serían necesarias si el planificador fuese un proceso SW ejecutándose en el procesador. Por el contrario, en esta arquitectura, cuando la plataforma ejecuta un nuevo grafo de tareas, el procesador envía la información al planificador incluyendo la descripción del grafo de tareas. Con esta información se guía la ejecución del grafo, teniendo en cuenta los eventos generados por las URs, y finalmente genera una interrupción para informar al procesador. El entorno EDK soporta dos opciones fáciles de utilizar para enviar datos al planificador: una FIFO incluida en su interfaz y una transacción por DMA. La segunda opción es óptima en prestaciones, ya que el procesador no tiene que escribir directamente la información en la FIFO, sino que sólo necesita enviar la dirección inicial y el tamaño de los datos. Por otro lado, el controlador DMA genera una penalización en términos de área (que evaluaremos); sin embargo, de acuerdo con los experimentos realizados, una transacción por DMA es casi 3.5 veces más rápida que una transacción de datos equivalente entre el procesador y la FIFO local al periférico.

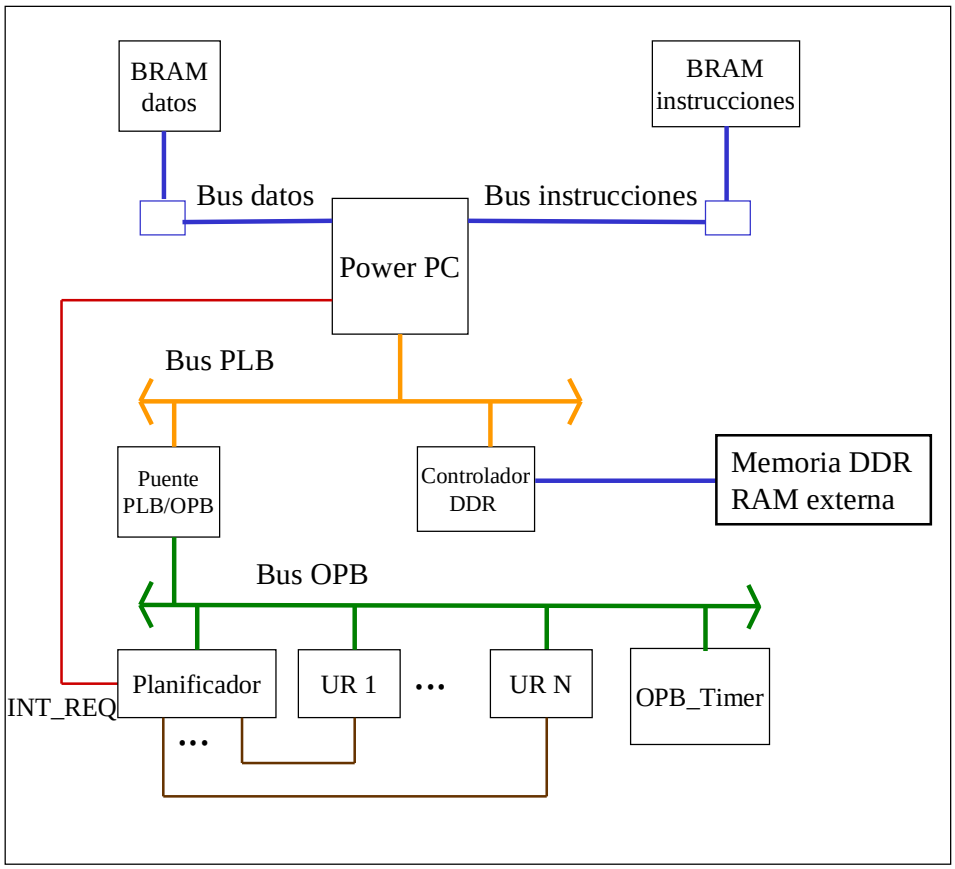


Figura 20. Esquema de la arquitectura con del planificador HW

Dado que no existen en la actualidad sistemas multitarea HW, hemos desarrollado una plataforma que simula estos sistemas utilizando contadores. Estos contadores se incluyen en el sistema y se utilizan para simular las latencias de reconfiguración y el tiempo de ejecución de cada tarea. Cada UR incluye dos contadores (uno para las reconfiguraciones y otro para las ejecuciones) Cuando un contador termina su cuenta, informa al sistema de que ha finalizado. De esta forma se generan los eventos de fin de ejecución y de fin de reconfiguración en tiempo de ejecución.

Por último disponemos de un contador programable (un *OPB_Timer*) para contar el número de ciclos que han transcurrido durante la ejecución de un grafo de tareas. Como se puede ver en la figura, este contador se incluye como un periférico más del sistema y se gestiona desde el procesador. Estos contadores son muy fáciles de utilizar gracias a los drivers que proporciona la herramienta Xilinx EDK, de modo que se pueden dar órdenes de reseteo, inicio/detención/reanudación y configuración de interrupciones usando llamadas a funciones en C.

VISIÓN GENERAL DEL PLANIFICADOR

El sistema es un planificador de tareas basado en HW reconfigurable. En la figura 21 se puede observar la organización del mismo y seguidamente se procederá a la descripción de cada una de sus componentes.

- **FIFO de reconfiguraciones:** Esta FIFO almacena la secuencia de tareas que deben ser reconfiguradas.

- **Tabla de dependencias entre tareas:** Cuando se debe ejecutar un nuevo grafo de tareas, sus dependencias se almacenan en una tabla asociativa. Cada vez que una tarea finaliza su ejecución, todos sus sucesores se actualizan, y esa tarea se elimina de la tabla. Esta tabla se consulta antes de comenzar a ejecutar una tarea para comprobar si todas las dependencias están resueltas, es decir, si las tareas predecesoras se han ejecutado ya. Esta operación se puede realizar en un solo ciclo de reloj.

- **Cola de eventos:** Almacena los eventos que se generan en tiempo de ejecución, como el fin de la ejecución de una tarea o el fin de una reconfiguración.

- **Unidad de control:** Esta unidad lee los eventos para identificar si una tarea puede comenzar su ejecución o si es posible anticipar una configuración.

- **Módulo de reemplazo:** Este módulo es el que dota al gestor la capacidad de decidir en tiempo de ejecución en qué UR se efectuarán las reconfiguraciones, en función de el criterio o la estrategia de reemplazo que se especifique. Como se explicará más adelante, hemos implementado varias versiones para las siguientes estrategias:

    - First Free (FF): Una tarea se reemplaza en la primera UR que encuentre libre, siguiendo un criterio de prioridad por órdenes muy simple.
    - Least Recently Used (LRU): Una tarea se reemplaza en la UR cuyo último uso fue el más lejano en el tiempo respecto a las demás.
    - LF+C: Se trata de la estrategia de reemplazo que hemos diseñado y en cuyo desarrollo y testeo se ha centrado gran parte del presente trabajo. Su funcionamiento se ha explicado anteriormente en la sección "*Visión general del problema y ejemplo de motivación*".

  El motivo del desarrollo de estas estrategias de reemplazo es hacer un análisis comparativo de la eficacia de cada una de ellas al ejecutar varias tareas consecutivas, así como evaluar comparativamente el impacto en el rendimiento que tiene utilizar cada una de estas estrategias en un planificador de tareas en un sistema HW multitarea. Los resultados de estas comparativas se muestran en la sección "*Evaluación del rendimiento*".

- **Información de la unidad:** El planificador asigna dos registros a cada unidad reconfigurable; que guardan el estado de la unidad y la tarea que en ese momento allí esté cargada, respectivamente. También existe un pequeño controlador que gestiona su funcionamiento.

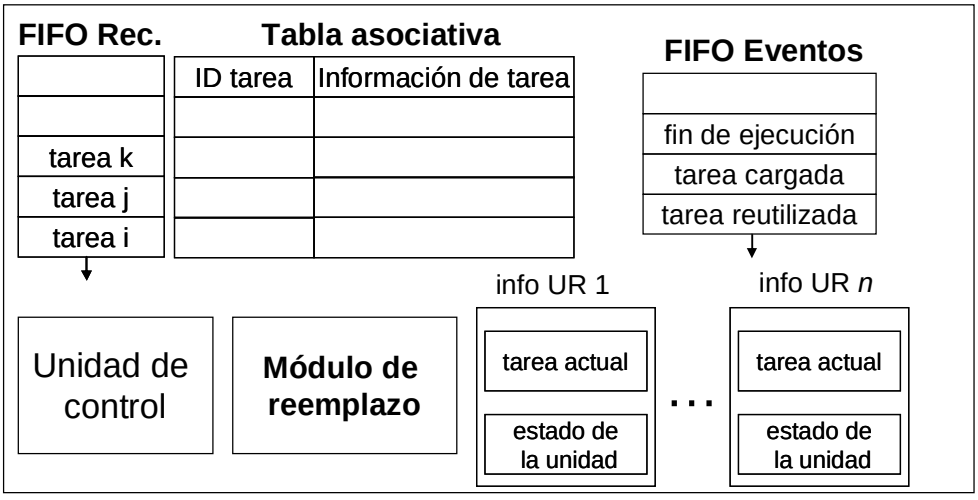


Figura 21. Esquema del planificador

El sistema es escalable en cuanto a número de unidades reconfigurables, número máximo de predecesores y/o sucesores que puede tener una tarea y longitud de los identificadores de

las tareas. Esto ha permitido hacer pruebas con diferentes valores para estos parámetros y así determinar cuáles de estos valores son los más adecuados para un funcionamiento óptimo.

ARQUITECTURA HW DEL PLANIFICADOR

FIFOs

1) Las BLOCK RAMs

En primer lugar, mencionar que, al ser las FIFOs dispositivos de almacenamiento masivo de datos, es muy probable que si no se hace un uso eficiente de los recursos disponibles en la FPGA, el diseño ocupará demasiada área de integración y será poco eficiente.

Para subsanar este problema se ha utilizado un componente de memoria de Xilinx llamado *RAMB16Sn*, en donde n puede ser: 1, 2, 4, 9, 18 y 36. No todas las FPGAs los tienen; sin embargo, sólo hay que consultar la documentación para asegurarnos de que en una Virtex XC2VP30-5 FG676C, que es la placa que se utilizará, este componente está disponible:

Componente	Celdas de datos		Celdas de paridad		Bus de direcciones	Bus de datos	Bus de paridad
	Profundidad	Anchura	Profundidad	Anchura			
RAMB16_S1	16384	1	-	-	(13:0)	(0:0)	-
RAMB16_S2	8192	2	-	-	(12:0)	(1:0)	-
RAMB16_S4	4096	4	-	-	(11:0)	(3:0)	-
RAMB16_S9	2048	8	2048	1	(10:0)	(7:0)	(0:0)
RAMB16_S18	1024	16	1024	2	(9:0)	(15:0)	(1:0)
RAMB16_S36	512	32	512	4	(8:0)	(31:0)	(3:0)

Tabla 1. Posibles configuraciones del módulo RAMB16Sn

Los componentes *RAMB16_S1, RAMB16_S2, RAMB16_S4, RAMB16_S9, RAMB16_S18* y *RAMB16_S36* son bloques de memoria de acceso aleatorio con escritura asíncrona. Tienen 16384 bits de memoria para datos. Por su parte, el *RAMB16_S9, RAMB16_S18* y el *RAMB16_S36* tienen además 2048 bits adicionales de paridad que no utilizaremos. Las posibles configuraciones del *RAMB16_Sn* son las que se pueden observar en la tabla 1, mientras que su comportamiento es el que se puede observar en la tabla 2.

Se pueden utilizar los atributos *INIT_XX* para especificar valores iniciales de los contenidos en memoria de una *RAMB16*. Esta inicialización se establece para cada *RAMB16_Sn* por medio de 64 valores hexadecimales (desde *INIT_00* hasta *INIT_3F*), cada uno de los cuales representa 16384 bits. Paralelamente, se pueden usar también los atributos *INITP_XX* para especificar valores iniciales en la memoria de paridad. Esta nueva inicialización se puede llevar a cabo mediante 8 atributos iniciales (desde *INITP_00* hasta *INITP_07*), compuestos por 64 valores hexadecimales, que conforman un total de 2048 bits; a través de los puertos configurados para 9, 18 o 36 bits. Si estos atributos (*INIT_XX* o *INITP_XX*) se quedan sin especificar, se les asigna como valor por defecto a ceros. A las cadenas de bits incompletas se les añaden ceros a la izquierda hasta completar el total del bus.

En las *placas Spartan-3, Virtex-II, Virtex-II Pro, y Virtex-II Pro X*, cada bit del registro de salida se puede inicializar a 0 o a 1. Esta inicialización se puede realizar por medio de dos atributos: *INIT* y *SRVAL*. El atributo *INIT* especifica la salida de este registro de salida cuando

la FPGA se enciende (*power on*), mientras que el atributo *SRVAL* especifica la salida cuando se activa la señal de entrada SSR. Sus valores por defecto son a ceros.

Entradas								Salidas			
GSR	**EN**	**SSR**	**WE**	**CLK**	**ADDR**	**DI**	**DIP**	**DO**	**DOP**	**Contenido de la RAM**	
										Datos en la RAM	**Paridad en la RAM**
1	X	X	X	X	X	X	X	INIT	INIT	S/C	S/C
0	0	X	X	X	X	X	X	S/C	S/C	S/C	S/C
0	1	1	0	↑	X	X	X	SRVAL	SRVAL	S/C	S/C
0	1	1	1	↑	dir	datos	Pdatos	SRVAL	SRVAL	RAM(dir) =>datos	RAM(dir) =>pdatos
0	1	0	0	↑	dir	X	X	RAM(dir)	RAM(dir)	S/C	S/C
0	1	0	1	↑	dir	datos	Pdatos	S/C [a] RAM (dir)[b] datos[c]	S/C [a] RAM(dir)[b] pdatos[c]	RAM(addr) =>datos	RAM(dir) =>pdatos

GSR=Señal para Set y Reset globales
INIT=Valor especificado en el atributo INIT para los datos en memoria. Su valor por defecto es a ceros
SRVAL= Valor de salida tras activarse SSR, que coincide con el valor del atributo SRVAL, especificado por el usuario.
S/C= Sin cambios
dir=dirección de la RAM
RAM(dir)=Contenidos de la RAM en la dirección DIR
data=Entrada de datos en la RAM
pdata=Datos de paridad en la RAM
[a]WRITE_MODE=NO_CHANGE
[b]WRITE_MODE=READ_FIRST
[c]WRITE_MODE=WRITE_FIRST

Tabla 2. Comportamiento del módulo RAMB16Sn

En cuanto al modo de escritura, sólo comentar brevemente que el atributo *WRITE_MODE* controla los contenidos y la salida de la *RAMB16_SX*. Por defecto, su valor es *WRITE_FIRST*. Esto implica que si se realiza una escritura, en la salida se muestran los datos de la entrada. También se puede configurar el *WRITE_MODE* a *READ_FIRST* para que, cuando se realice una escritura, mostrar en la salida el contenido en memoria que había antes de que se realizase dicha escritura. Asimismo, se puede configurar el *WRITE_MODE* a *NO_CHANGE* para que, cuando se realice una escritura, no se observen cambios en la salida de datos de la RAM.

Por último, comentar que se utilizará un tipo de *RAMB16_SX* u otro en función de las necesidades de la FIFO en cuestión, en términos de longitudes de palabra y/o número de entradas. Se ha observado que cuanto mayor tamaño de palabra tenga la memoria que se utilice, más recursos consume (en todos los casos se utiliza una BLOCK RAM, pero a mayor tamaño de palabra, se realiza un mayor número de interconexiones, con el consecuente consumo de recursos). Por ello, se han implementado FIFOs de diferentes tamaños, que utilizaremos en función de la longitud de palabra de los datos que queramos almacenar tanto en la FIFO de reconfiguraciones, la FIFO de eventos, o en la tabla de tareas. Así se consigue hacer un uso muy eficiente de los recursos disponibles. Por otro lado, la configuración de los parámetros de inicialización de las BLOCK RAMs es la siguiente:

- ***WRITE_MODE*** = Configurado a NO_CHANGE.

- ***Datos de paridad***: No utilizados.
- ***Contenido inicial de las FIFOs***: a ceros.
- ***GSR***: No utilizada
- ***SSR***: Reset activo a baja.

2) Implementación de las FIFOs

El manejo de una planificación de tareas dada exige por parte del sistema el uso de complejas estructuras de datos, como listas enlazadas o indexadas, como ya hemos comentado.

Esto ha obligado a implementar un HW que proporcione el soporte de almacenamiento necesario para llevar a cabo un manejo eficiente y transparente de operaciones de inserción, borrado y actualización de tareas en listas enlazadas. Este HW implementa la política de actualización FIFO (*First in, First out*), por lo que conceptualmente representa una cola cuyos elementos se insertan al final de la misma. Por otro lado, al realizar una operación de extracción de un elemento, se extrae el que esté en primer lugar en la cola. El número de entradas de las FIFOs es constante.

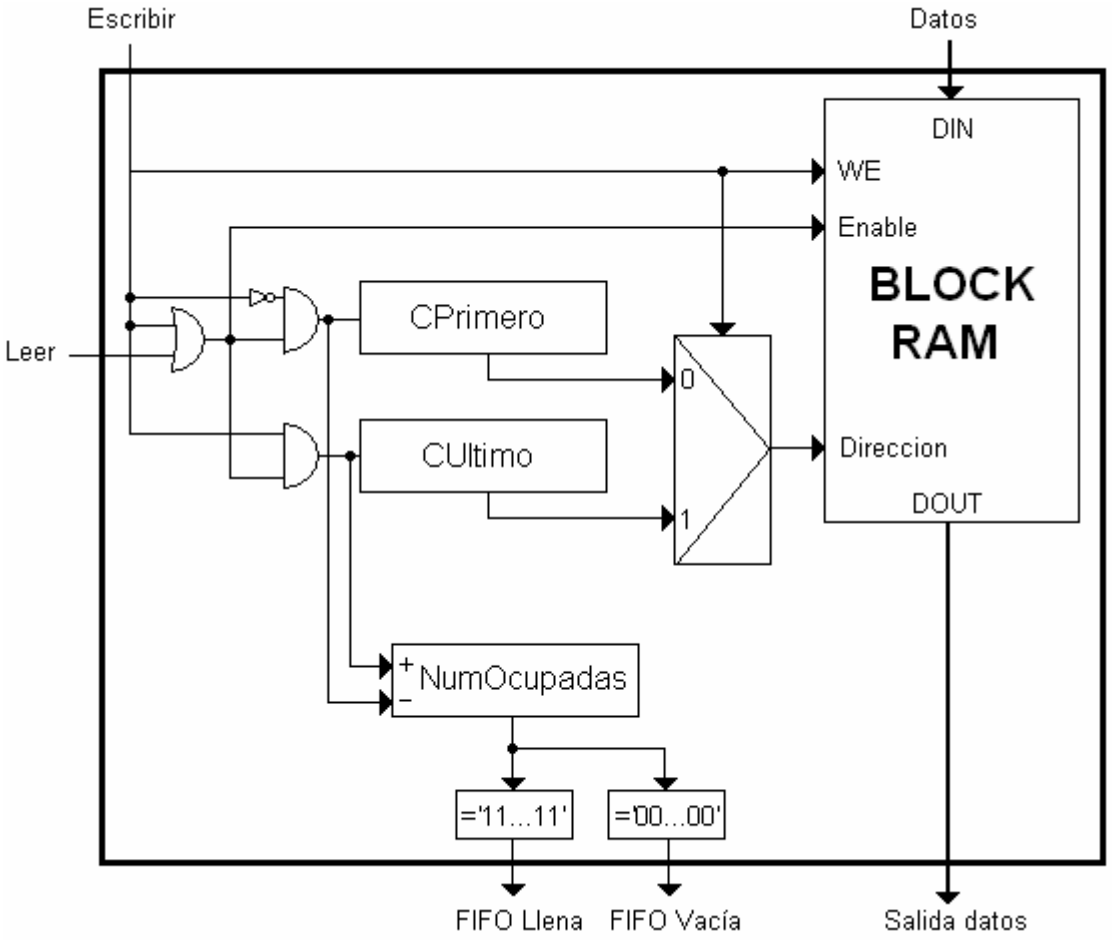


Figura 22. Esquema de las FIFOs desarrolladas

Las operaciones que soporta son las siguientes:

a) ***Inserción***: Se inserta un elemento al final de la cola. Si la FIFO estuviera llena, no se llevaría a cabo esta operación y se encontraría activa la señal "*FIFO llena*".

b) ***Extracción***: Se extrae el primer elemento de la lista. Si la FIFO se vaciara como consecuencia de una extracción de su único elemento en la misma, se activaría la señal "*FIFO vacía*". Si la FIFO ya estuviera vacía y se intentase realizar una extracción, ésta no se lleva a cabo con éxito y la señal "*FIFO vacía*" permanece activa.

Por otro lado, las FIFOs implementadas sólo tienen una entrada de datos, por lo que solamente se pueden realizar una lectura o escritura a la vez. No se permiten lecturas y escrituras simultáneas; y en el caso de que se intentase realizar dicha operación, se daría prioridad a las escrituras. (Se ha configurado el *WRITE_MODE* de la RAM a *NO_CHANGE*).

En cuanto a la implementación, su esquemático es el que aparece en la figura 22. Se utilizan dos contadores (*CPrimero* y *CUltimo*) para guardar la dirección de la primera y de la última posición de los datos en la tabla, respectivamente. Cuando se hace una operación de escritura, se escribe en la posición que indique el contador *CUltimo* (que apunta a la posición siguiente a la última ocupada) y éste se incrementa. Por otro lado, cuando se hace una operación de extracción, se realiza una lectura en la *BLOCK RAM* seleccionando como dirección la que indique *CPrimero* y se decrementa este contador. Así nos aseguramos que los datos están siempre en posiciones contiguas en la *BLOCK RAM*. También utilizamos un contador ascendente/descendente para saber cuántas posiciones hay ocupadas en la FIFO. (*NumOcupadas*). Si este contador está a cero, la FIFO estaría vacía, mientras que si este contador está a su máximo valor posible, la FIFO estaría llena.

La tabla asociativa

Es la tabla asociativa en donde se guarda la información de la(s) tarea(s) presentes en el sistema. Esta tabla se utiliza para supervisar las dependencias entre las tareas de un grafo de tareas. Su tamaño es escalable en cuanto a número de entradas y número máximo de sucesores. Se proponen dos implementaciones distintas: como una sola tabla totalmente asociativa o como un conjunto de subtablas independientes totalmente asociativas.

Las operaciones que soporta son: *inserción, borrado y actualización,* y *comprobación*, que pasamos a describir a continuación:

a) ***Inserción***: Esta operación escribe la información de la tarea en la tabla.

Con la implementación como una única tabla totalmente asociativa, la ubicación donde se escriben los datos no es relevante. Para seleccionar dónde se guarda una tarea se utiliza la red iterativa, que siempre indica la primera entrada libre de la tabla; y se que genera un error si la tabla está llena. Con este enfoque la operación puede realizarse en un ciclo de reloj, pero a medida que el tamaño de la tabla crece, el ciclo de reloj dismuye.

Con la imlementación como un conjunto de subtablas asociativas, al igual que en la anterior, la ubicación donde se escriben los datos no es relevante. Para seleccionar dónde se guarda una tarea se pregunta secuencialmente a cada subtabla si puede ubicarse en ella hasta que se encuentre una ubicación disponible. Para ello nos ayudaremos de la red iterativa descrita anteriormente, que nos dirá que la subtabla se encuentra llena o en caso contrario, indica la primera entrada libre de la subtabla. Con este enfoque la operación puede llegar a tardar N ciclos de reloj, donde N es número de subtablas existentes, pero sin embargo, se consigue que al aumentar el número de subtablas disponibles, el ciclo de reloj permanezca constante.

b) ***Borrado y actualización***: Esta operación se realiza cuando una tarea termina su ejecución. La operación elimina esta tarea de la tabla, con lo que ésta puede ser usada en el futuro para ubicar otras tareas. A continuación, se realiza una actualización de todas sus dependencias, consistente en disminuir en uno el número de sucesores de cada uno de sus predecesores. Esto es posible gracias a que en la entrada de la tabla guardamos la información acerca de todos los identificadores de todos predecesores;

con lo que la actualización consistirá en buscar cada uno de estos predecesores en la tabla y disminuir en uno su número de sucesores. Queda claro, por tanto, que el objetivo de la actualización es garantizar en todo momento que la información que contiene la tabla sea correcta. Esta operación tiene una complejidad de Θ(N), donde N es el número de sucesores a actualizar.

c) ***Comprobación***: Esta operación consulta si una tarea dada está lista para comenzar su ejecución, es decir, si están resueltas todas sus dependencias. Esto se realiza en un sólo ciclo de reloj, introduciendo la etiqueta de la tarea como entrada de la tabla y leyendo la salida *tarea preparada* en el siguiente ciclo. Una tarea está preparada si su contador de predecesores es cero

Estructuralmente, la tabla está compuesta por una serie de entradas unidas entre sí mediante una red iterativa (que sirve para decidir en qué entrada se guardará una nueva tarea que entre en la tabla). Alternativamente, existe la posibilidad de implementarlo como un conjunto de subtablas más pequeñas, de forma que las inserciones en cada subtabla se gestionen por separado. Esta modificación reduce el retardo del sistema aunque aumenta el número de ciclos necesarios para cada inserción. Asimismo, existe también un controlador que gestiona las distintas operaciones que la tabla permite realizar.

1) Entrada de la tabla

Consiste en un módulo básico en el que se guarda toda la información relacionada con una misma tarea: su identificador, el número de predecesores, el número de sucesores, así como los identificadores de estos sucesores. En la figura 23 se puede observar su esquemático.

El registro "*etiqueta*" guarda el identificador de la tarea que se encuentra en la entrada. El contador de predecesores y el registro de sucesores guardan la información acerca del número de predecesores y sucesores, respectivamente; y que será muy útil para gestionar las dependencias correctamente. Por otro lado, los registros "*Sucesor 1*", …, "*Sucesor n*" guardan los identificadores de los sucesores de la tarea que esté en esa entrada. El resto del HW sirve para generar señales de control útiles para la correcta gestión de las dependencias: "*Tarea lista para ejecución*" indica si la tarea presente en la entrada está lista para su ejecución (equivalente a que su número de predecesores sea 0), y la señal "*acierto*" indica si la tarea de la entrada es la que se pide desde la entrada "*Etiqueta*".

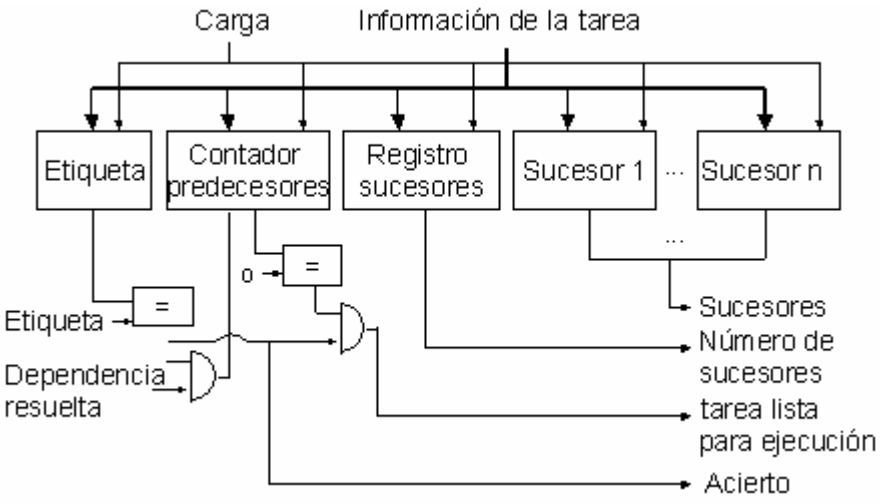


Figura 23. Entrada de la tabla de dependencias de tareas

2) Red iterativa

Consiste en una red formada por una serie de celdas individuales conectadas entre sí y cada una de las cuales está a su vez conectada con una entrada de la tabla asociativa. En la figura 24 se puede observar su esquemático.

Esta red se utiliza para saber cuál es la primera entrada de la tabla que se encuentra libre ("*load_tarea*$_i$"); y esta información es necesaria cuando se hace una inserción de una tarea en la tabla.

Se trata de una red iterativa porque existe una señal (*libre_it*) que se van pasando unas celdas a otras. Su significado es el siguiente:

$$libre\_it_{i+1} = \begin{cases} 1 & si \;\; libre\_it_i = 0 \;\; AND \;\; \forall x \in [0..i-1] \, libre_x = 0 \\ 0 & en \;\; caso \;\; contrario \end{cases}$$

De esta forma, si hay varias posiciones libres en la tabla, sólo en la primera de estas celdas (empezando desde el subíndice 0) se activará *load_tarea*$_i$, por lo que nos aseguramos que una tarea se cargará en la primera posición libre de la tabla.

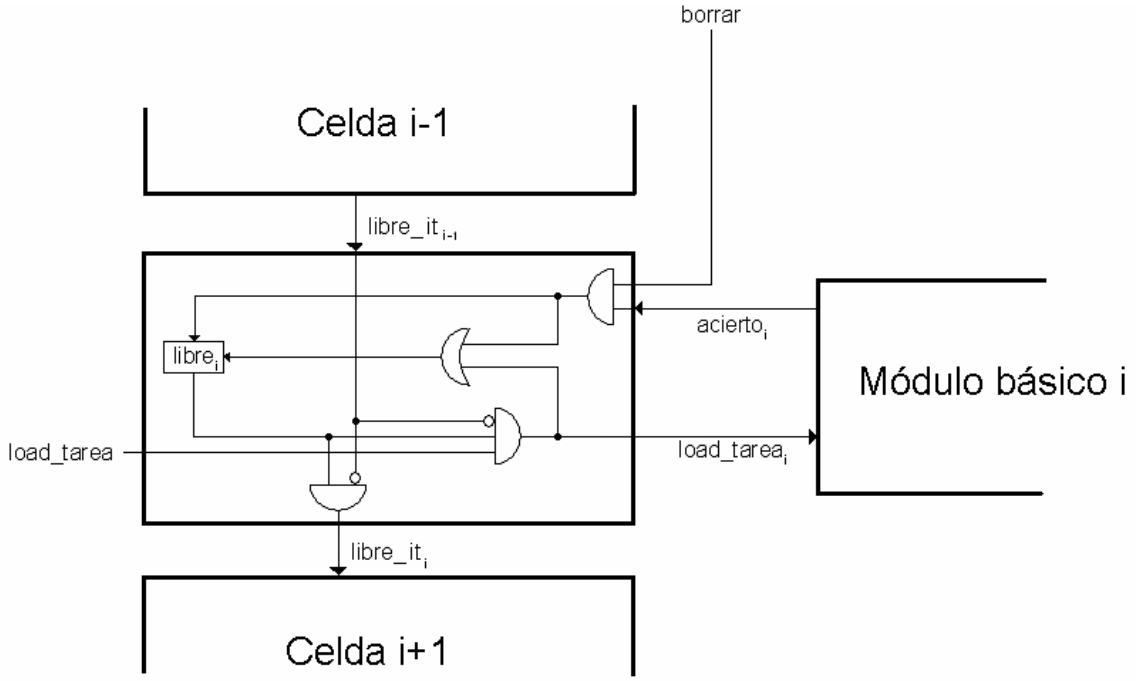


Figura 24. Red iterativa

Cada celda de esta red contiene un registro de un bit que indica si la entrada asociada a dicha celda está ocupada por una tarea (1) o no (0). Por tanto, la actualización de su contenido sólo tendrá lugar en dos situaciones:

1) Cuando se cargue una nueva tarea en el módulo básico asociado. Esto ocurre cuando la señal *load_tarea*$_i$ está activa, la celda está libre y la celda anterior no lo está. Si *load_tarea*$_i$ se activa, se desencadenaría la carga de la tarea en el módulo básico correspondiente y se activaría la señal de carga del registro. Lo que se carga en este caso en el registro es un 0, ya que la señal *borrar* no está activa.

2) Cuando se pretenda eliminar esa tarea. Esto ocurre cuando la señal *borrar* está activa y cuando la tarea que se quiere borrar está en el módulo básico asociado ($acierto_i$ es 1). Se pondrían a 1 simultáneamente la señal de carga del registro y su entrada de datos, y se cargaría un 1.

Las operaciones que hacen uso del HW de la red iterativa tienen un tiempo de ejecución de un solo ciclo de reloj. Recordar que en el caso de la implementación con un conjunto de subtablas, es un ciclo por cada consulta (cada uso de la red iterativa de cada subtabla).

3) HW de control

Para que las operaciones que soporta la tabla se puedan llevar a cabo correctamente es necesario cierto HW de control. Es especialmente destacable el HW desarrollado para la operación de *borrado y actualización* se lleve a cabo correctamente. En la figura 25 se puede observar su estructura.

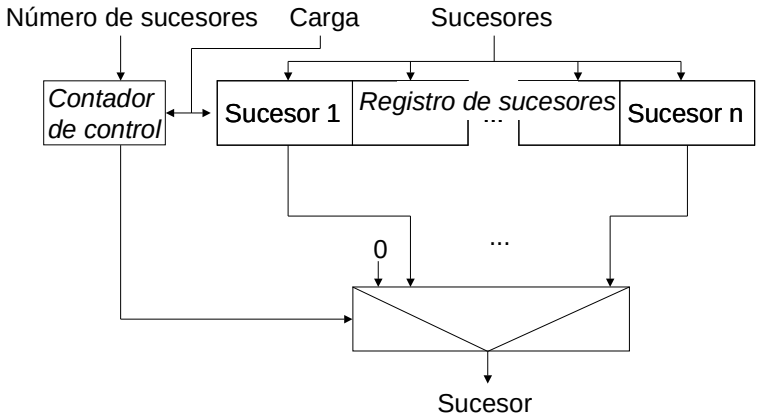


Figura 25. Soporte HW para la operación borrado y actualización

Disponemos de un contador, un registro de sucesores y un multiplexor. En el contador y en el registro de sucesores se guardan el número de sucesores y los identificadores de los sucesores que tiene la tarea cuyas dependencias queremos actualizar, respectivamente.

Su funcionamiento es el siguiente: se empieza cargando la información correspondiente en el contador y en el registro de sucesores. Después, se actualiza secuencialmente cada sucesor, operación que se realiza en un ciclo de reloj para cada uno. En cada ciclo se realizan tres operaciones: uno de los sucesores se selecciona en el registro de sucesores usando el *contador de control* y el multiplexor, se activa la entrada de la tabla asociativa *dependencia resuelta* y el *contador de control* se decrementa. Cuando el contador de control toma el valor cero la operación finaliza. Por su parte, cuando la tabla detecta que la señal *dependencia resuelta* está activa, decrementa el contador de predecesores de la tarea correspondiente.

4) Esquemático a alto nivel

1) Primera implementación – Una única tabla totalmente asocitativa

Todos los módulos anteriormente explicados y debidamente interconectados componen la tabla de tareas, cuyo esquemático a alto nivel es el que indica la figura 26.

Vemos que la red iterativa está estrechamente relacionada con las celdas básicas de la tabla asociativa, pues cada celda de la red está conectada con una entrada de la tabla. Las entradas "*Entrada_datos*", "*Dep_resuelta*" y "*Etiq_in*" entran directamente en las celdas básicas de la tabla asociativa, mientras que "*Load_tarea*" entra *también* en la red iterativa.

Por otro lado, el codificador sirve para determinar la posición en que se encuentra la celda cuyo contenido coincide con la entrada "*Etiq_in*". Posteriormente, mediante un multiplexor se selecciona la salida de datos correspondiente a la tarea seleccionada y estos datos se utilizan en el siguiente HW de control, cuya función es garantizar que se realice la actualización de las dependencias en cada uno de sus sucesores. Esta actualización se realizará en la operación "*borrado y actualización*", explicada con detalle en el apartado "*Operaciones*".

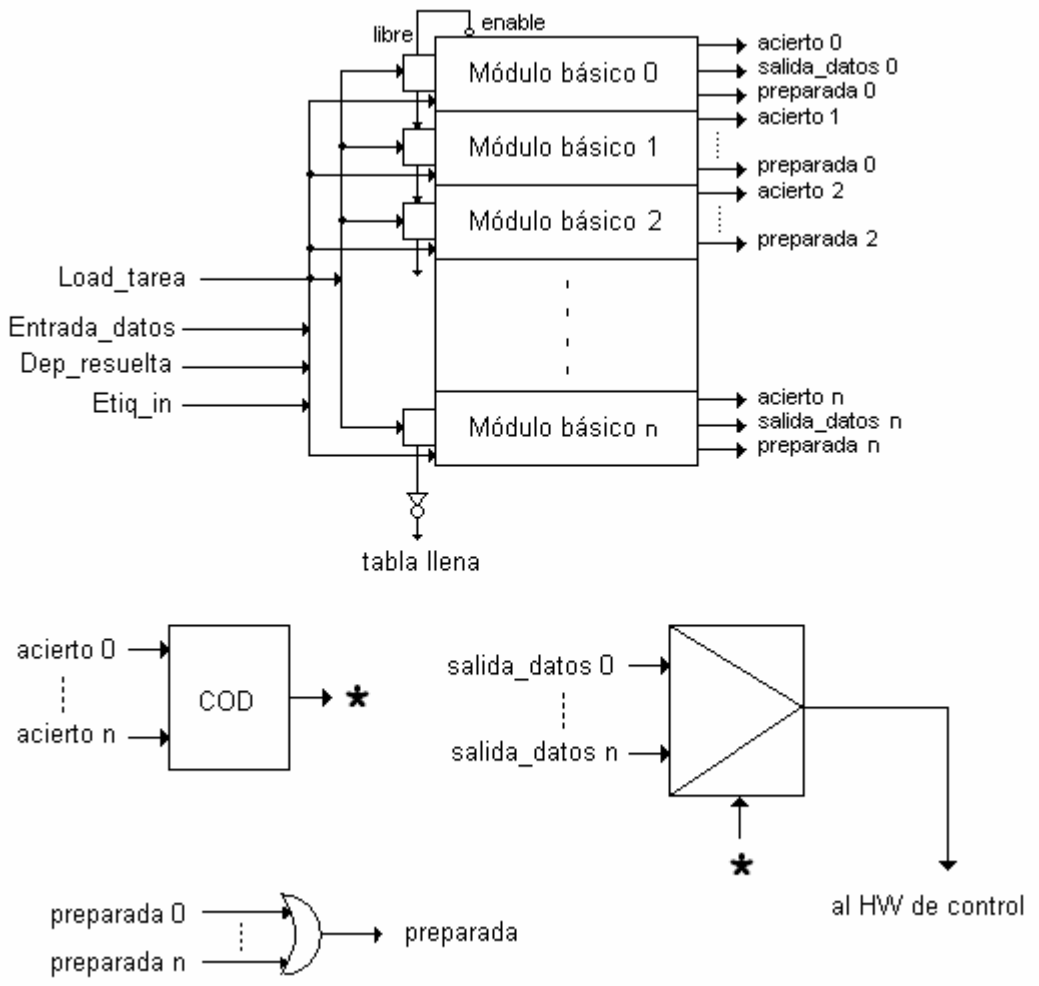


Figura 26. Tabla de tareas

2) Segunda implementación – Conjunto de subtablas asociativas

Como puede verse en la figura 27, se trata de un conjunto de subtablas asociativas como las explicadas anteriormente y una lógica combinacional para seleccionar las señales de salida correspondientes a la subtabla donde está ubicada la tarea que se esté tratando.

Esta implementación resulta ventajosa, ya que el módulo crítico del sistema es la red iterativa que nos dice cuál es la primera entrada libre de la tabla o si ésta se encuentra llena.

De esta forma, al quedar particionada la red iterativa, se consigue que el ciclo de reloj permanezca constante a media que aumenta el número de subtablas asociativas del planificador. Sin embargo, mientras las operaciones de *comprobación* y *borrado y actualización* tienen el mismo comportamiento que anteriormente, se van a sufrir penalizaciones en la operación de *inserción*. De forma consecutiva, comprobamos si la tarea cabe en la subtabla o si ésta se encuentra llena, hasta que finalmente se encuentre una subtabla con una entrada libre o si todas se encuentran llenas, en este caso, generamos una señal de error. Con este enfoque la operación puede llegar a tadar N ciclos de reloj, donde N es número de subtablas existentes, pero sin embargo, conseguimos que al aumentar el número de subtablas disponibles, el ciclo de reloj permanezca constante.

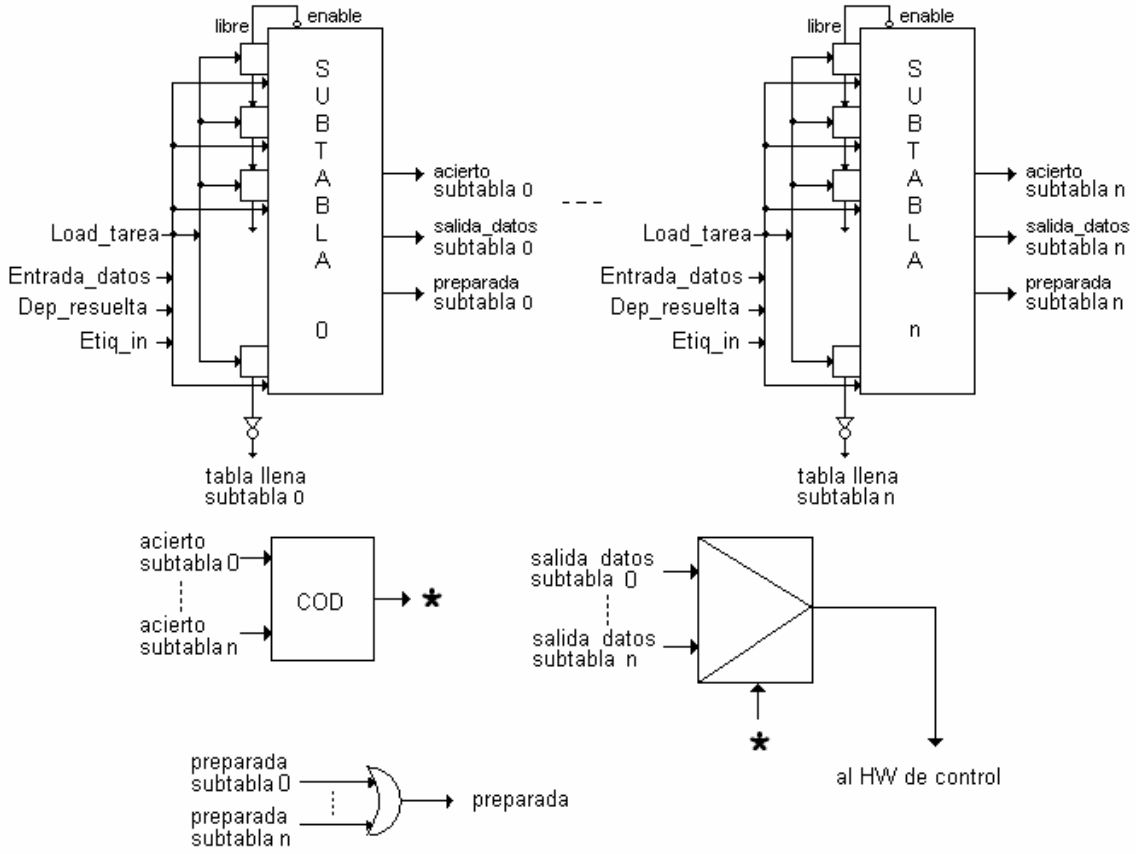


Figura 27. Tabla de subtablas de tareas

El módulo para las URs

Este módulo se encarga de caracterizar cada una de las unidades reconfigurables del planificador. Su misión es proporcionar información a la unidad de control sobre el estado en el que se encuentra cada una de las unidades reconfigurables. (Para más información acerca de este módulo consultar el apartado dedicado a la *unidad de control.*)

En el marco global del sistema, este módulo está continuamente funcionando y sincronizándose únicamente con el árbitro para gestionar las escrituras a la FIFO de eventos. A grandes rasgos, lo que hace es simular las latencias de reconfiguración y de ejecución que se soliciten, por medio de unas señales que permiten establecer la comunicación entre estos módulos y la unidad de control. Su esquemático es el que se puede ver en la figura 28.

Sus funciones son:

1- Responder a las órdenes de comienzo de carga y comienzo de ejecución. En esos casos, este módulo simula la carga y/o la ejecución de la tarea que se encuentre actualmente en la unidad.

2- Generar los eventos apropiados:
   a. Cuando se termina de ejecutar una tarea se genera el evento "*fin de ejecución de la tarea i*". El código de evento es: "10"
   b. Cuando se termina de reconfigurar una tarea o cuando una tarea es reutilizada, el evento que se genera es "*fin de reconfiguración de la tarea i*". El código de evento es: "01".

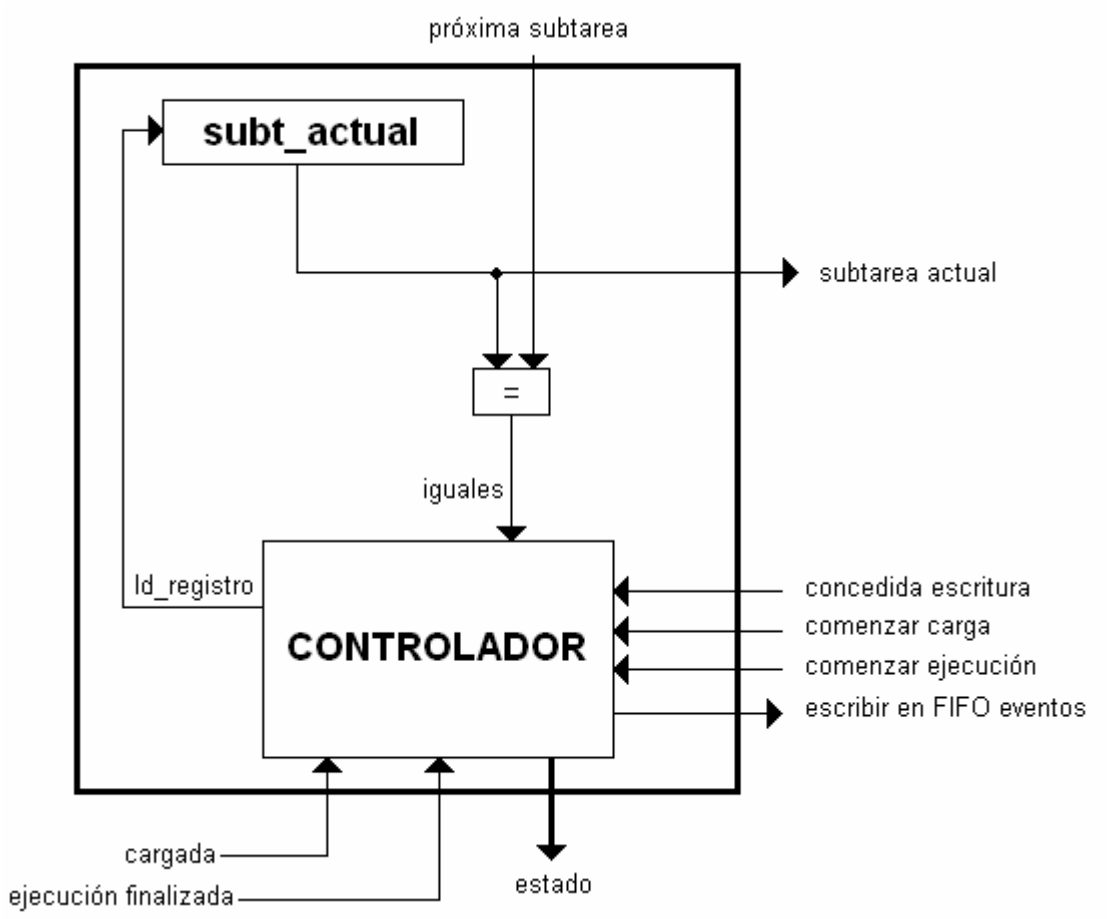


Figura 28. Módulo para las unidades reconfigurables

Estos eventos se escriben en la FIFO de eventos. Una situación problemática que podría suceder es que varias unidades reconfigurables intentasen escribir simultáneamente en dicha FIFO. Para que esa situación no represente problema alguno, existe un árbitro externo que garantiza el acceso exclusivo a las operaciones de escritura en la FIFO, a la vez que organiza las peticiones para que ninguna de ellas se quede sin atender. Para más información acerca del árbitro consultar el apartado dedicado al *árbitro*.

Vemos que este módulo consta, a grandes rasgos, de los siguientes componentes HW:

1- Un registro en el que se guarda la tarea actual de la unidad.

2- Un HW de control que gestiona los cambios de estado activando las señales correspondientes, así como las señales de salida del módulo. Este HW de control está

compuesto por el comparador y el controlador, cuyo diagrama de estados se puede observar en la figura 29.

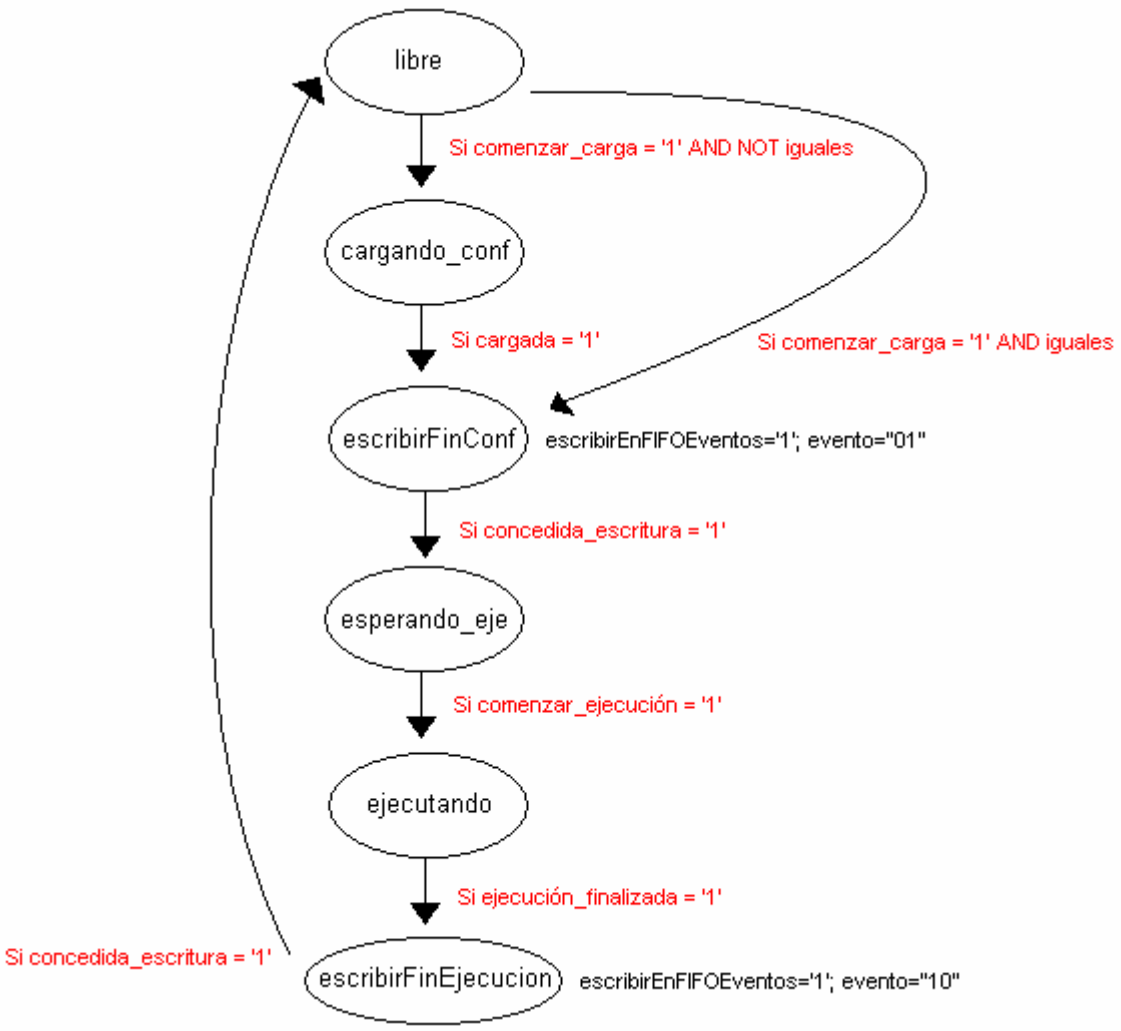


Figura 29. Diagrama de estados para el control de los módulos de las URs

Este diagrama de estados es una máquina de Moore. Sus estados son los siguientes:

"***libre***": La UR está en este estado cuando no está simulando el comportamiento de ninguna tarea. Permanece en este estado mientras no se solicite el comienzo de una carga. En caso contrario, en función del valor de la señal *iguales*, pasa al estado *cargandoConf* (se simula la reconfiguración cuando *iguales* = '0'; es decir, la tarea que debo cargar no está en este momento en la UR) o *escribirConf* (no se simula ninguna reconfiguración cuando *iguales* = '1' porque la tarea a cargar ya está en la UR).

"***cargando_conf***": La unidad reconfigurable está en este estado mientras se esté llevando a cabo la reconfiguración de la tarea. Es en este estado cuando se carga la tarea en el registro de la unidad reconfigurable. El número de ciclos durante los que la unidad se encuentre en este estado dependerá del tiempo de reconfiguración, tiempo que viene incluido entre la información que se proporciona de la tarea. Tan pronto como se recibe la confirmación de que la tarea ha sido cargada (*cargada*=='1') pasa al estado "*escribirFinConf*".

"***escribirFinConf***": La unidad reconfigurable está en este estado cuando se ha finalizado la reconfiguración de una tarea o cuando se acaba de decidir que una tarea debe

ser reutilizada. En este estado se genera el evento "*tarea reconfigurada*". Este evento se genera tanto si una tarea es reconfigurada como si es simplemente reutilizada. El motivo por el que se genere el mismo evento en dos situaciones diferentes es que ambas situaciones son equivalentes desde el punto de vista del gestor de los eventos. La unidad permanece en este estado el tiempo que haga falta hasta que se le concede la escritura en la FIFO de eventos; porque, como ya se dijo anteriormente, puede ocurrir que varias unidades estén funcionando en paralelo y quieran escribir simultáneamente en dicha FIFO, con el consecuente problema de una escritura concurrente que la FIFO no soporta. Para garantizar el acceso exclusivo a estas operaciones de escritura debe existir un árbitro y es precisamente ese árbitro el que informa a los módulos UR acerca de las concesiones.

"***esperando_eje***": La unidad reconfigurable está en este estado mientras se encuentre a la espera de comenzar la ejecución de su tarea. Cuando se reciba desde el exterior la señal de que dicha ejecución debe comenzar, pasa al estado "*ejecutando*".

"***ejecutando***": La unidad reconfigurable está en este estado mientras la tarea que contiene se esté ejecutando. Permanece en este estado tantos ciclos como tarde en llevarse a cabo dicha ejecución; y, cuando finaliza, activa la señal "*ejecutada*" y pasa al estado "*escribirFinEjecucion*".

"escribirFinEjecucion": La unidad reconfigurable está en este estado cuando se ha finalizado la ejecución de una tarea. En este estado se genera el evento "*tarea ejecutada*". Al igual que ocurría con el estado "*EscribirFinConf*", la unidad permanece en este estado el tiempo que haga falta hasta que se le concede la escritura del evento ahora generado en la FIFO de eventos. Cuando se recibe una confirmación de que dicho evento ha sido escrito con éxito (*concedidaEscritura*= ='1'), pasa al estado "*lee_fifo*".

Como última aclaración, cabe mencionar que para implementar la estrategia LRU es necesario hacer una pequeña modificación en estos módulos, que se explicará más adelante cuando se detalle las distintas implementaciones del módulo de reemplazo.

La FIFO de reconfiguraciones

Disponemos de una FIFO en la que se guardan qué tareas deben ser reconfiguradas en el sistema y en qué orden. Como ya se ha comentado anteriormente, dicho orden se realiza en la fase previa de cálculo de los pesos de las tareas: para cada grafo se etiquetan todas sus tareas con un peso y se ordenan de manera decreciente, de modo que se reconfiguren antes aquéllas tareas con más peso.

Es, por tanto, en esta FIFO donde que se guarda la información acerca de la planificación que debe seguir el sistema. Esta FIFO se inicializa cuando entra una nueva tarea, momento en el cual se escriben en ella (en un determinado orden) todas las reconfiguraciones que han de producirse. Posteriormente, se irán extrayendo tareas de esta FIFO y se les dará un tratamiento adecuado: buscar en qué unidad está; y cuando esta unidad esté en estado "finalizada", ordenar una ejecución (si tiene lugar una reutilización de esta tarea en la UR) o una reconfiguración (en caso contrario). Este tratamiento tiene lugar en la unidad de control.

Sus características tecnológicas son las de una FIFO ya explicada anteriormente.

La FIFO de eventos

Disponemos de una FIFO de eventos para guardar y así poder gestionar los eventos que se producen en el sistema en tiempo de ejecución.

Los eventos que se pueden producir en el sistema y, que, por tanto, son susceptibles de ser guardados aquí, se pueden generar desde los módulos que caracterizan a las URs o desde el cargador de tareas; este último caso sólo ocurre cuando entre una nueva tarea en el sistema.

"nueva tarea": Este evento se genera cuando entra una nueva tarea en el sistema. Es el evento que menos veces se genera en el sistema; sólo una vez por tarea recibida. Su código binario es "11".

"fin de ejecución de la tarea i": Este evento se genera cuando se termina de ejecutar una tarea. Como una tarea se ejecuta dentro de una UR, será el módulo que caracteriza a su UR el que genere este evento. Su código binario es "10".

"fin de reconfiguración de la tarea i": Este evento se genera cuando se termina de reconfigurar una tarea. Se genera desde los módulos que caracterizan a las URs. Su código binario es "01".

"tarea i reutilizada": Este evento se genera cuando se termina de ejecutar una tarea. Se genera desde los módulos que caracterizan a las URs. A efectos prácticos, este evento y "*fin de ejecución de la tarea i*" son equivalentes debido a que a ambos se les da el mismo tratamiento en la unidad de control. Por tanto, en las situaciones en las que se deba generar este evento, se generaría "*fin de reconfiguración de la tarea i*", con la consecuente simplificación de la gestión de los eventos.

El árbitro

Se acaba de mencionar que en el sistema propuesto se lleva a cabo una gestión de eventos, consistente en su generación, almacenamiento en una FIFO y posterior tratamiento. Debido a la naturaleza imprevisible (al menos desde el punto de vista del sistema) de estos eventos, es posible que se generen simultáneamente dos o más eventos y que, por tanto, se intenten escribir simultáneamente en la FIFO de eventos (entendemos "simultáneamente" por "en el mismo ciclo de reloj"). Esta es una situación problemática porque las FIFOs utilizadas no soportan dos o más escrituras simultáneas, al tener un solo puerto de escritura. Por tanto, es necesario garantizar la exclusión mutua en los accesos este puerto de escritura para que no quede ningún evento sin capturar. Para ello, se ha diseñado un árbitro (figura 30) que recibe todos los eventos que se generan en el sistema y establece una política de prioridades fijas de forma que se garantiza que sólo se intenta escribir en esta FIFO en un instante de tiempo dado.

Este árbitro recibe todos los eventos en *eventos[1..n]*, (en el que *eventos[i]* es un código de dos bits) que se pueden producir en el sistema y *n* señales de peticiones (*peticiones[1..n]*) procedentes de cada uno de los módulos que los generan. Si *peticiones[i]* está activa, se hace una petición para que *eventos[i]* se escriba en la FIFO. Por otro lado, la señal de salida *concesiones[1..n]* indica las concesiones que se realizan: Si *concesiones[i]* está activa, se permite la escritura de *eventos[i]* en la FIFO; en caso contrario, se prohíbe dicha escritura. Obviamente, sólo una de las *concesiones[i]* posibles estará activa en el mismo instante de tiempo.

El árbitro da la mayor prioridad a la escritura del evento "*nueva_tarea*"; seguido de los generados por las URs, de más significativa a menos significativa. (Se supone que las URs están identificadas por el árbitro de la forma *UR[1..n]*). Por tanto, su comportamiento es similar a un codificador de prioridad.

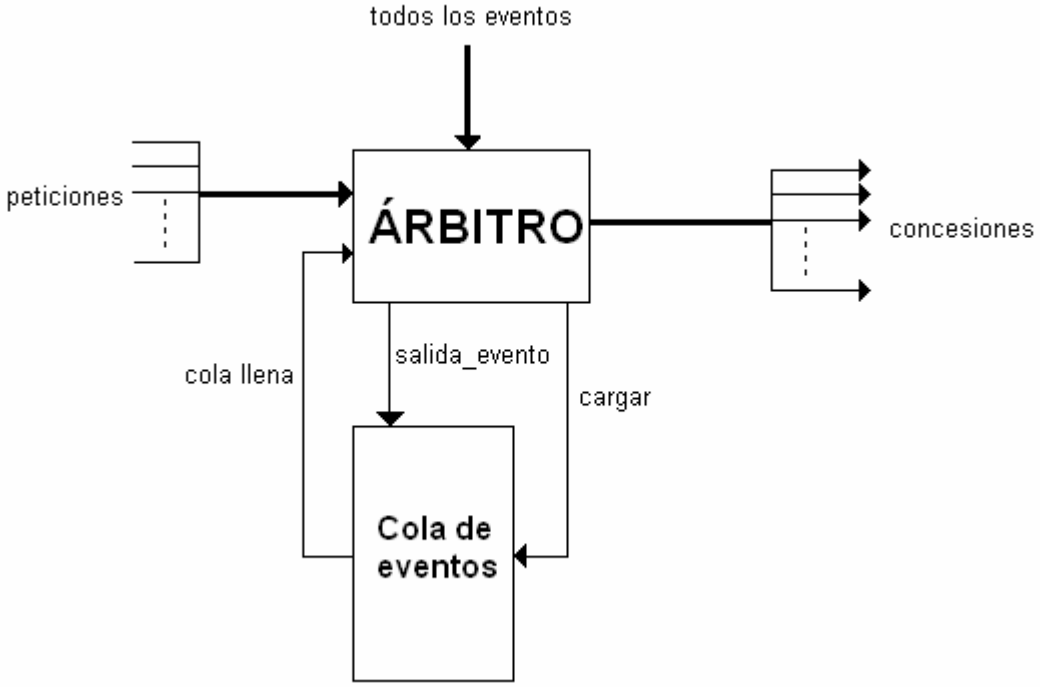


Figura 30. Árbitro de interconexión con la FIFO de eventos

Además, el árbitro también controla las lecturas de la FIFO de eventos, dándoles mayor prioridad que a las escrituras.

La unidad de control

La unidad de control extrae los eventos de la cola y lleva a cabo las acciones apropiadas. Su pseudocódigo se describe en la figura 31.

```
Gestión de eventos:
 CASE evento IS:
  fin_de_ejecucion:
    actualizar_tabla
    buscar_una_reconfiguración()
    FOR i=0 hasta número de URs
       IF tarea_preparada
             comenzar_ejecución()
  fin_de_reconfiguración;
    IF tarea_preparada
        comenzar_ejecución()
        buscar_una_reconfiguración()
  nuevo_grafo:
    actualizar_tabla
    actualizar_FIFOs_UR
    IF politica_precarga=basado_planifica
            actualizar_FIFO_reconfig
            buscar_una_reconfiguración()
```

Figura 31. Pseudo-código de la unidad de control

Cuando un evento "*fin de ejecución*" se procesa, este módulo actualiza las dependencias guardadas en la tabla asociativa. Entonces, si el circuito de reconfiguración está libre, trata de empezar una reconfiguración. Si el sistema usa la opción basada en una planificación, leerá la FIFO de reconfiguración, y comprobará si es posible comenzar la reconfiguración correspondiente. Finalmente, la unidad de control comprobará si cualquiera de las tareas que actualmente están cargadas puede comenzar su ejecución.

Para los eventos "*fin de reconfiguración*" y "*tarea reutilizada*" la unidad de control comprobará si la tarea que ha sido cargada puede comenzar su ejecución.

Finalmente, para el evento "*nuevo grafo*", la unidad de control leerá los datos desde el buffer de entrada y los almacenará en la tabla asociativa y en las respectivas FIFOs. Tras esto, si el circuito de reconfiguración está libre, comprobará si es posible comenzar a cargar una de las nuevas tareas.

El módulo de reemplazo

Este módulo se encarga de decidir en tiempo de ejecución en qué unidad reconfigurable se efectuarán las cargas de las tareas, en función de la técnica o estrategia de reemplazo que se especifique. Como ya se ha comentado, se han implementado varias estrategias de reemplazo que pasamos a detallar a continuación:

1) First Free (FF)

Esta estrategia de reemplazo es la más sencilla y menos eficiente. Cada vez que se desea hacer un reemplazo, ésta se realiza en la primera UR que se encuentre libre. Para ello, únicamente hace falta un codificador de prioridad y, en un solo ciclo, se obtiene el índice de la UR en la que se debe realizar el reemplazo (*señal UR a reemplazar*). La figura 32 muestra el HW que hemos desarrollado para esta estrategia.

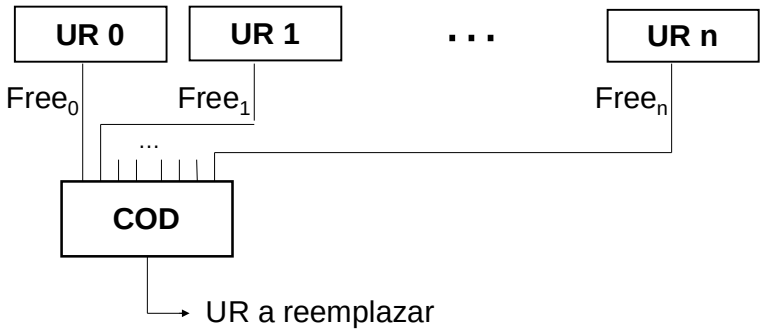


Figura 32. Estrategia First Free. HW que selecciona la UR a reemplazar

2) LRU

Hemos implementado una versión HW de la estrategia LRU para aplicarla al reemplazo de URs en nuestro planificador. Dicha implementación puede verse en detalle en la figura 33. Como se puede ver, consta básicamente de dos codificadores de prioridad, dos multiplexores y

un decodificador; además de un HW de control en los módulos de que caracterizan las unidades reconfigurables.

En primer lugar, se comprueba si la tarea a cargar está ya en alguna UR y si esa UR está libre. (Esto no aparece en la figura por simplicidad). Esa información va al codificador de prioridad 1, cuya salida A informa si ha habido alguna UR que cumpla esta condición, y cuya salida *reutilizacion* es el identificador de esa UR. El control del multiplexor de dos entradas lo gobierna la señal A, por lo que si se puede hacer una reutilización, ésta será prioritaria.

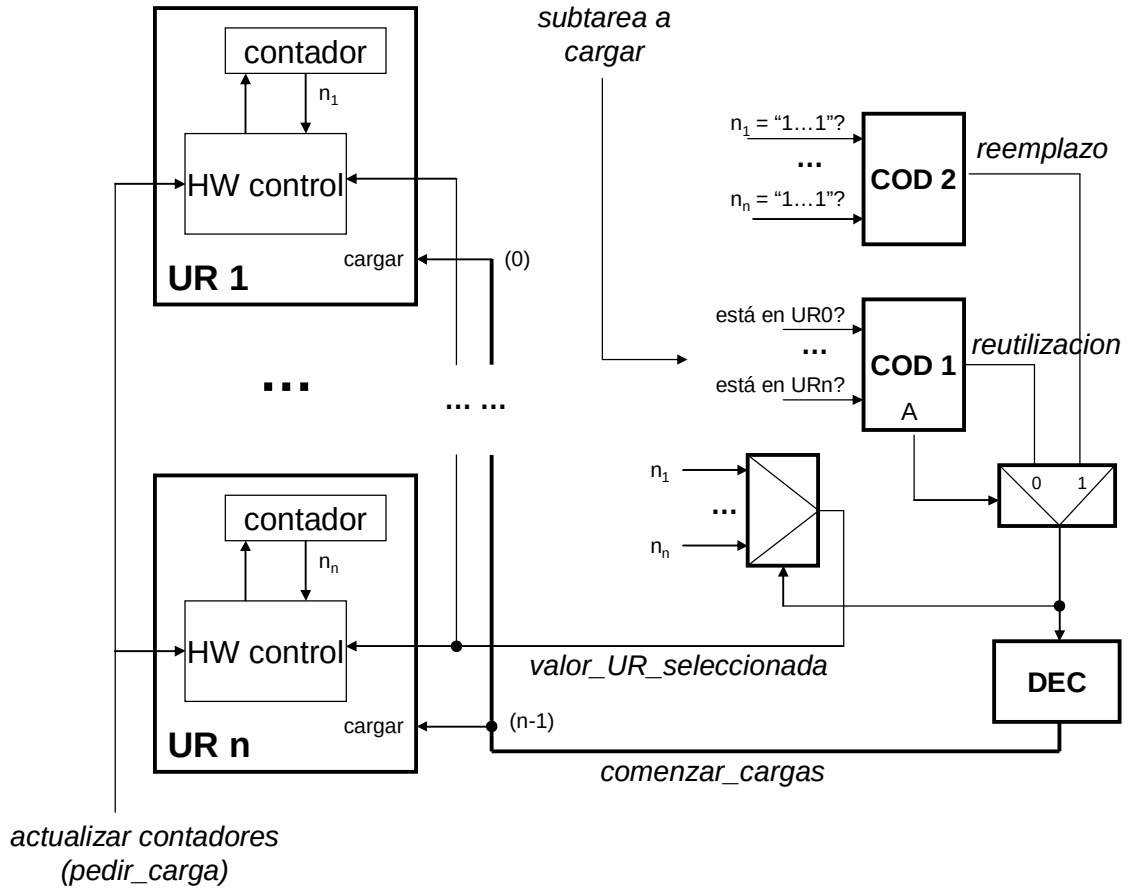


Figura 33. Estrategia LRU. HW que selecciona la UR a reemplazar

Sólo si la tarea a cargar no se puede reutilizar, se aplica LRU. Para ello, en cada UR hay un contador, cuyo valor representa el tiempo desde el último uso de esa UR respecto a las demás. De esta forma, cuanto más alto sea el valor del contador, más lejano en el tiempo ha sido el último uso de esa UR. Por tanto, LRU siempre buscará la UR cuyo contador tenga el valor más alto. Esto se hace gracias a *NUM_URs* puertas AND, que comparan el valor de los contadores con el máximo posible ("111…1", esto tampoco aparece en la figura por simplicidad). Sus salidas van al codificador de prioridad 2; cuya salida informa qué UR tiene el valor más alto. Como se puede ver, esta señal sólo se seleccionará en el multiplexor de dos entradas si la tarea a cargar no se puede reutilizar (señal *A* desactivada). La salida de este multiplexor informa en qué UR se efectuará el reemplazo, y se conecta a la entrada de un decodificador (DEC). La salida de este decodificador se conecta a la entrada *cargar* de las *NUM_URs* URs, de forma que la sólo la UR codificada en la entrada de DEC recibirá la orden de carga. Obviamente, una UR sólo obedecerá a una orden de carga si la señal *pedir_carga* también está a 1 (esto se tiene en cuenta en el controlador de las URs). Para que todas ellas sepan cuál ha sido el valor del contador de la UR seleccionada, existe otro multiplexor de *NUM_URs* entradas que selecciona este valor de entre todos, y cuya señal de selección es la

misma entrada del decodificador anterior. El HW de control de todas las URs necesita saber este valor para poder actualizar convenientemente los contadores después de cada reemplazo.

En cuanto a la gestión de estos contadores, mencionar que se inicializan a los valores 0, 1, 2, … *NUM_URs*-1, siguiendo un orden establecido. Cada vez que se carga una tarea, sus valores deben actualizarse convenientemente. Para ello tenemos un HW de control en el interior de los módulos de las URs, y detallado en la figura 34. En cada una de ellas, se necesita saber el valor de la UR que ha sido seleccionada (*valor_UR_seleccionada*). Entonces se comprueba si el valor del contador coincide con *valor_UR_seleccionada* o si es menor. Si coincide, el contador se resetea a 0. Si es menor, se le suma 1. Y si es mayor, no se hace nada. De esta forma, cada vez que se realiza una carga (la señal *actualizar_contadores* se pone a 1), los contadores de la estrategia LRU quedan actualizados correctamente; y nos aseguramos que siempre que todas las URs tengan un valor único entre 0 y *NUM_URs*-1.

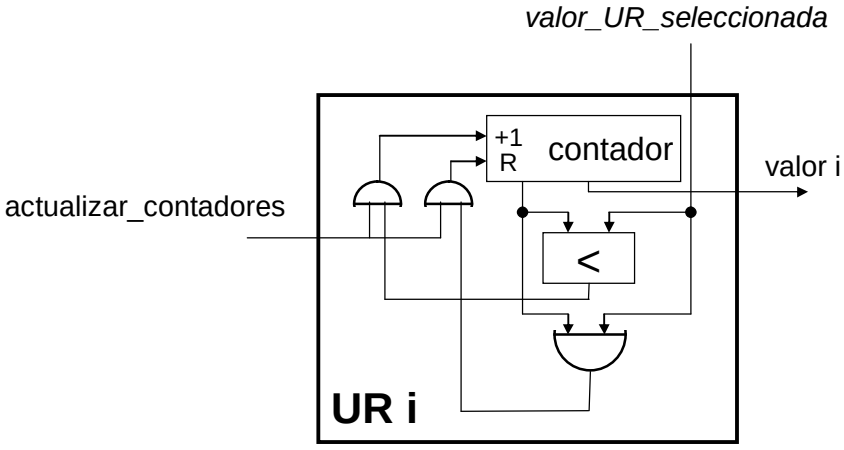


Figura 34. Estrategia LRU. HW de control necesario en los módulos de las URs

Tal y como está implementado actualmente, estas decisiones se toman en un solo ciclo de reloj, por lo que no hay que hacer cambios en la unidad de control con respecto a la estrategia *First Free*.

3) LF+C

Como ya se ha explicado en la sección "*Visión general del problema y ejemplo de motivación*", esta estrategia de reemplazo consiste en la identificación y etiquetado de las URs libres en tres categorías (un candidato no puede ser de dos tipos a la vez): Candidatos Perfectos, (CP), Candidatos Críticos (CC) y Candidatos Reutilizables (CR), en función de la tarea cargada en esa UR. Este etiquetado se realiza en función de si la tarea es crítica y si sabemos que va a ser ejecutada próximamente:

1) Es crítica si su carga tiene un impacto especialmente negativo en el rendimiento del sistema. El planificador no tiene que ocuparse de decidir si una tarea es crítica o no: dicha información ya viene proporcionada junto con el resto de la información de la tarea, y se calcula en tiempo de diseño. Por tanto, se puede diseñar un HW específico que lo decida o suponer que viene dado. En este caso, suponemos que esta información viene dada. Ya se comentó el algoritmo que decidía qué nodos de un grafo de tareas eran críticos en la sección de "*Cálculo de pesos*".

2) Si sabemos que esa tarea se va a ejecutar próximamente, esto significaría que, si reemplazamos la tarea en esa unidad, habrá que volver a cargarla en el futuro; por

lo que muy posiblemente se producirán penalizaciones en el rendimiento del sistema debido a la penalización temporal que supone esa carga.

Como ya se comentó anteriormente, para que un candidato sea CP, no debe estar en la tabla asociativa ni ser crítico. Para que sea un CC, no debe estar en la tabla y debe ser crítico. Finalmente, para que sea un CR, basta con que esté en la tabla. El criterio de selección busca en primer lugar los CP; si no encuentra ninguno, busca un CC; y como último recurso, buscaría un CR. Si no hubiese candidatos libres, la reconfiguración no se puede efectuar. Como se puede ver, consideramos más problemático reemplazar un candidato que se encuentre actualmente en la tabla asociativa antes que un candidato simplemente crítico; porque al primero habrá que volver a cargarlo, mientras que el segundo quizá no se vuelva a usar más.

La figura 35 muestra el HW de selección de los candidatos. Como se puede ver, consta únicamente de 3 multiplexores, 3 puertas lógicas y un contador. El valor del contador oscila entre 0 y *NUM_URs*-1, y selecciona en cada ciclo y a través de los multiplexores la información acerca de cada una de las URs. Necesitamos un contador porque no se pueden realizar varios accesos concurrentes a la tabla asociativa. Finalmente, las puertas lógicas deciden el tipo de candidato del que se trata.

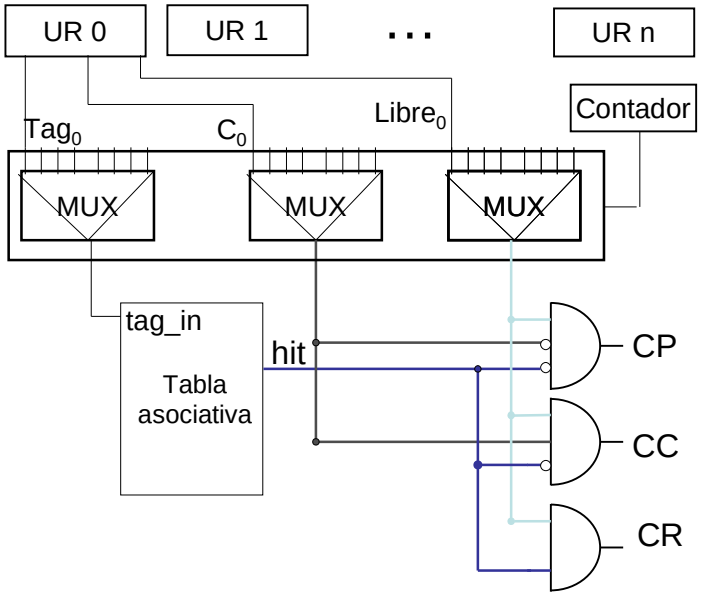
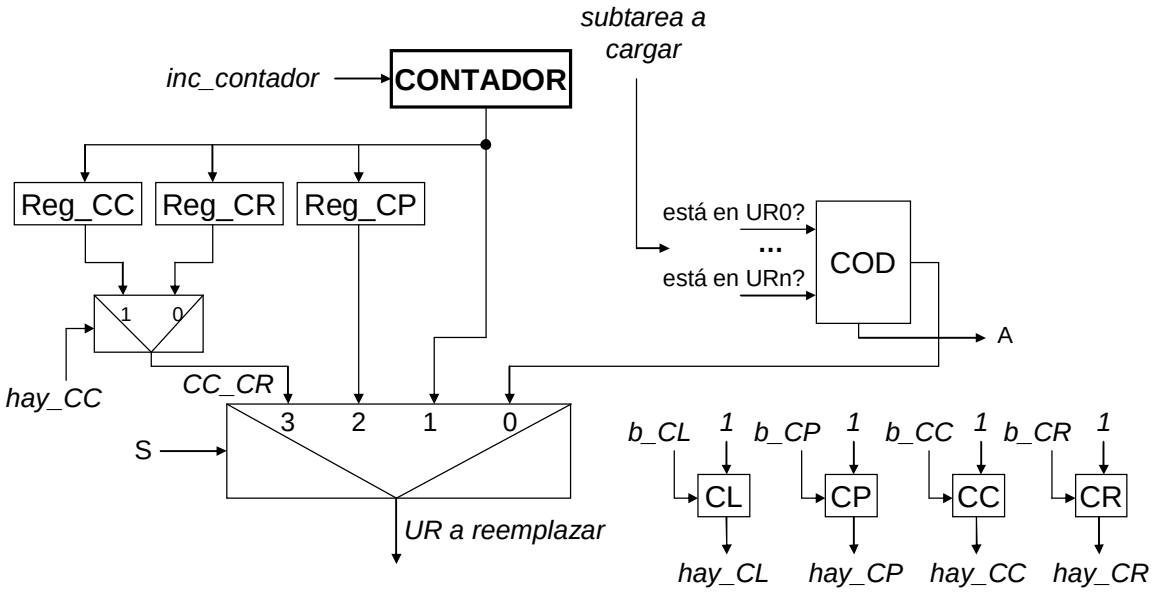


Figura 35. Estrategia LF+C. HW que selecciona los candidatos

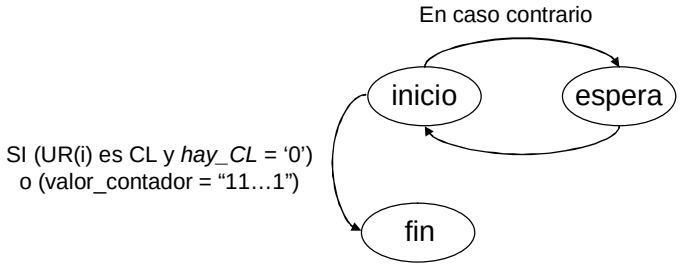
Por su parte, la figura 36 muestra el resto del módulo de reemplazo. Consiste en una máquina de estados y una ruta de datos que toman la decisión de en qué UR realizar una carga cada vez que se solicite.

En la ruta de datos utilizamos el mismo contador que la figura 35, cuya función es iterar entre 0 y *NUM_URs*-1 (en el peor de los casos) y seleccionar mediante el HW de dicha figura la información de las URs. Entonces en cada iteración, para cada UR, se identifica qué tipo de candidato es y se actualiza el registro y/o el biestable correspondiente: tenemos tres registros (para guardar el índice del primer CR, CC, o CP encontrado, respectivamente) y cuatro biestables (para guardar si se encontró un CC, CP, CL o CR, respectivamente). Por otro lado, utilizamos un codificador que indica si la tarea a cargar se puede reutilizar en la UR seleccionada (señal A = ‘1’).

RUTA DE DATOS



MÁQUINA DE ESTADOS



En estado *inicio:*

inc_contador = NOT *CL*

SI UR(i) es CP y *hay_CP* = '0' → actualiza *reg_CP*; *b_CP* = '1'

SI UR(i) es CC y *hay_CC* = '0' → actualiza *reg_CC*; *b_CC* = '1'

SI UR(i) es CR y *hay_CR* = '0' → actualiza *reg_CR*; *b_CR* = '1'

SI UR(i) es CL y *hay_CL* = '0' → ; *b_CL* = '1'

En estado *fin:*

comenzar carga = *A* OR *hay_CP* OR *hay_CL* OR *hay_CC* OR *hay_CR*

Figura 36. Estrategia LF+C. Resto del HW del módulo de reemplazo

También disponemos de dos multiplexores que realizan la selección del candidato, dando más prioridad a las reutilizaciones, CLs, CPs, CCs y CRs; en este orden. El multiplexor de dos entradas selecciona entre la salida del registro CR y la del registro CC, en función de si la salida del biestable CC es 1 o no. A continuación, se realiza la selección del multiplexor de 4 entradas, que elige entre *CC_CR*, *salida_CP*, la salida del contador y la salida del codificador. La salida del contador representa un CL en el caso de que se identifique como tal, ya que la máquina de estados (que comentaremos más adelante) da la búsqueda por finalizada en cuanto detecta que la UR actual es un CL. Por este motivo no hace falta un registro para tal tipo de candidato. Como se puede ver, la selección de este multiplexor es también sencilla, y sigue las siguientes reglas:

- Si *A* = '1', S = "00"; en caso contrario:
- Si *hay_CL* = '1', S = "01"; en caso contrario:
- Si *hay_CP* = '1', S = "10"; en caso contrario:
- S = "11" (selecciona entre un CC y un CR; dicha selección se hace en el otro MUX).

La salida del multiplexor es el índice de la UR que se va a reemplazar. Esta señal se conectará a un decodificador, cuya salida se conecta a las entradas *cargar* de todas las URs, como se hacía en la estrategia LRU (figura 33). En la figura 36 esto no aparece por simplicidad.

Por otro lado, disponemos de una máquina de 3 estados, que gobierna el HW detallado anteriormente. Su función es iterar el contador entre 0 y *NUM_URs*-1 (en el peor de los casos), actualizar adecuadamente los registros de los candidatos y terminar cuando se cumplan las condiciones para ello. Es una máquina de tipo MOORE y tiene tres estados:

"***inicio***": En este estado se ordena la actualización de los registros en función del tipo de candidato que sea la UR seleccionada, tal y como se puede ver en la figura. En este estado también se actualiza el contador, pero sólo si no se ha encontrado un CL, ya que en ese caso, ese valor será el índice de dicho candidato, y hará falta más adelante para ordenar la carga en la UR correspondiente. De este estado pasamos al estado "*espera*", a no ser que se encuentre un CL. En este caso, se da la búsqueda por finalizada.

"***espera***": Este estado sirve para realizar los accesos a la tabla asociativa (recordemos que para detectar si un candidato es CR tenemos que comprobar si está en la tabla). La tabla tarda un ciclo entre que se le da una orden de comprobación y se obtiene su respuesta.

"***fin***": Este estado sirve para ordenar la carga de la tarea en la UR seleccionada. Esta carga sólo se realizará si la tarea se puede reutilizar o si se ha encontrado un candidato. En caso contrario, no se hace nada. Por tanto, comenzar_carga = *A* OR *hay_CP* OR *hay_CL* OR *hay_CC* OR *hay_CR.*

En nuestra implementación, esta pequeña máquina de estados realmente está integrada en la unidad de control, que es una máquina de estados mucho más grande que realiza el control del sistema. Por otro lado, y a diferencia de las estrategias anteriores, el tiempo que se tarda en este caso en tomar una decisión sobre un reemplazo oscila entre 1 y 2x*NUM_URs* ciclos (en el peor de los casos).

EJEMPLO DE EJECUCIÓN DE UNA TAREA

En esta sección se describe paso a paso la ejecución de los mismos grafos de tareas de la figura 18, que sirvieron para ilustrar el funcionamiento de nuestra estrategia de reemplazo. En este apartado nos centraremos mayoritariamente en detallar el funcionamiento del módulo gestor, al margen de la estrategia de reemplazo; y cómo, mediante la generación y gestión de los eventos, la ejecución de los grafos llega a buen puerto. En la figura 37 se muestra dicha ejecución.

Inicialmente, en tiempo de diseño, se calculan los pesos de las tareas, la secuencia de reconfiguraciones y se identifica qué tareas son críticas. Seguidamente se envía esta información al planificador en tiempo de ejecución. Cuando la información esté correctamente almacenada en un buffer de entrada, el procesador generará el evento *nueva tarea*. En consecuencia, se inicia la primera reconfiguración de la secuencia (tarea 1) en la UR1; ya que está libre y es la primera que encuentra el módulo de reemplazo. Cuando la tarea 1 termine su reconfiguración, la UR1 generará un evento *fin de reconfiguración*.

Ahora, la unidad de control comprobará si la tarea 1 puede empezar su ejecución. En este caso, no hay dependencias no resueltas, por lo tanto comenzará justo después de que la reconfiguración termine. Además, comenzará la siguiente reconfiguración (tarea 2) ya que el

circuito de reconfiguración está libre y la UR2 es la primera que el módulo de reemplazo encuentra libre.

Cuando esa reconfiguración termine, generará un nuevo evento *fin de reconfiguración*, y la unidad de control comenzará la reconfiguración de la tarea 3. Además, se comprobará a su vez si la tarea 2 puede comenzar su ejecución, pero en este caso la tabla asociativa indicará que hay una dependencia aún no resuelta.

Cuando esta reconfiguración finalice, un nuevo evento será procesado, pero esta vez, no dará comienzo una nueva reconfiguración, ya que no hay más nodos en el grafo actualmente en ejecución (grafo 1). En este instante, tampoco se puede iniciar la ejecución de ninguna tarea, al no haber finalizado aún la ejecución de 1.

No obstante, cuando la tarea 1 finalice su ejecución se generará un evento *fin de ejecución*, que actualizará las dependencias e intentará comenzar una reconfiguración. Como no hay reconfiguraciones pendientes en el grafo 1, esto no es posible. También se comprueba si alguna tarea puede comenzar su ejecución; esto ocurre para la tarea 2. Por tanto, comienza su ejecución.

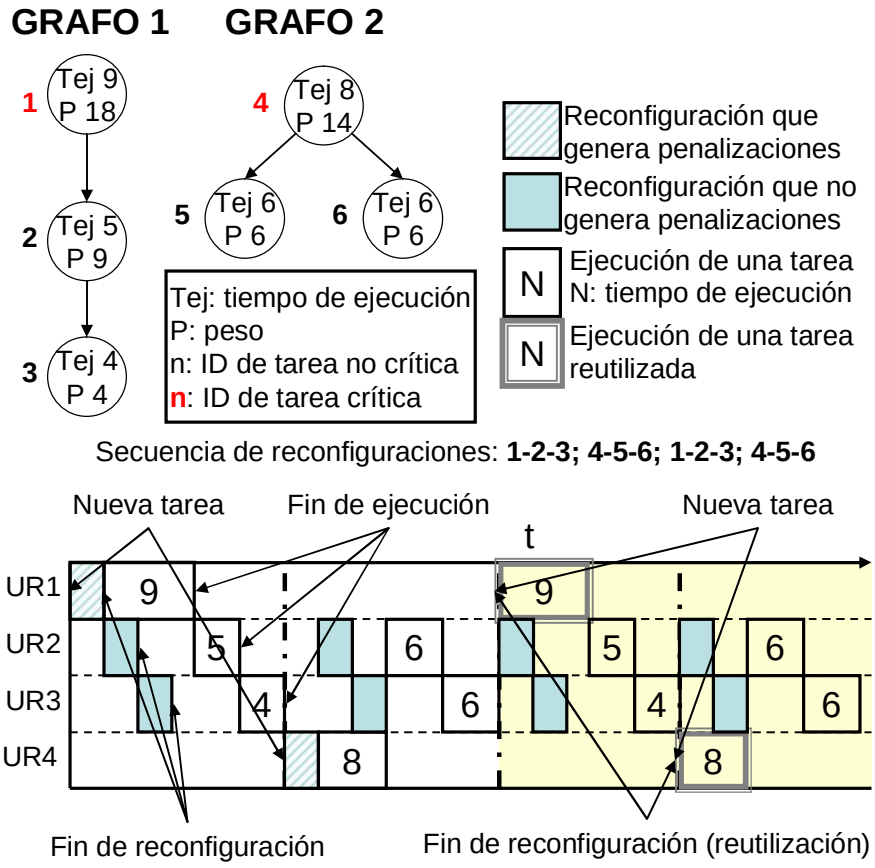


Figura 37. Ejemplo de ejecución de un grafo de tareas: gestión de eventos

Este proceso continúa hasta que finaliza la ejecución del grafo 1. En ese instante, se detecta que hay más grafos pendientes de ejecución y se genera nuevamente un evento de *nueva tarea*. Esto hace posible que comience la ejecución del grafo 2, cuya ejecución en el planificador es muy similar a la del grafo 1. (Recordemos que la estrategia de reemplazo no reemplaza las tareas 5 y 6 en la UR1, al detectar que allí hay cargada una tarea crítica (1)).

Cuando finaliza la ejecución del grafo 2, se vuelve a generar un evento *nueva tarea* y, en consecuencia, comienza la ejecución del grafo 3. Al intentar efectuar la reconfiguración de la tarea 1, el módulo de reemplazo detecta que ya está cargada en la UR1, por lo que decide su

reutilización y automáticamente genera un evento de *tarea reutilizada* (que, como ya se mencionó, es equivalente al evento *fin de reconfiguración*). El resto de la ejecución de este grafo es idéntica a la del grafo 1.

Por último, cuando se genera el evento *nueva tarea* cuando se llega al fin de la ejecución del grafo 3, vuelve a detectarse que la siguiente reconfiguración (tarea 4) puede reutilizarse, y se procede del mismo modo que con la anterior reutilización. Finalmente, cuando el resto de las tareas del último grafo terminen, el planificador generará una interrupción para informar al procesador que la ejecución de los grafos ha sido completada.

COSTE DE IMPLEMENTACIÓN

En esta sección, se evaluará el coste de la implementación HW del planificador, expresada en términos de área. Para este fin se ha implementado el circuito en una FPGA *VIRTEX-II PRO xc2rp30* utilizando *ISE 9.1i* y *EDK 9.1i* como plataformas de desarrollo.

En los diseños realizados se ha empleado en todos ellos genéricos para que sean ajustables; como por ejemplo, el tamaño de la tabla asociativa, el máximo número de sucesores por tarea y el número de URs en el sistema. Puesto que estos parámetros influirán en el coste del sistema, se evaluará un pequeño planificador con una tabla asociativa de tan sólo ocho entradas y cuatro URs, y más adelante se estudiará cómo evoluciona el coste cuando el sistema aumenta de tamaño.

La tabla 3 muestra el coste de cada uno de estos módulos y el coste total del planificador. Como se puede ver, se necesitan aproximadamente el 13% de los recursos de la FPGA; pero si no incluimos el controlador DMA, se necesitaría el 9%. En este caso el módulo más costoso es la tabla asociativa ya que ha sido diseñada para dar el mayor rendimiento posible. Sin embargo, el coste es todavía bastante reducido para este tamaño.

Módulo	Número de *slices*	*Slices (%)*	*Block RAMs*	*RAMs (%)*
Unidad de control	638	4.66%	0	0%
FIFO rec.	10	0.06%	1	0,74%
UR	44	0.32%	0	0%
Cola de eventos	10	0.06%	1	0,74%
Tabla asociativa	496	3.6%	3	2.2%
Controlador DMA	621	4%	0	0%
Planificador	**1198 (1819)**	**12.7% (8.7%)**	**5**	**3.68%**

Tabla 3. Coste de implementación para un planificador con una tabla asociativa de ocho entradas y cuatro URs

La tabla asociativa es además el módulo con mayor retardo, por lo que limita la frecuencia de funcionamiento del sistema. Por lo tanto, resulta interesante su implementación para distintos tamaños, con el fin de estudiar la evolución de su coste y su frecuencia. Estos datos se presentan en la tabla 4. Como se puede ver, el área necesaria crece aproximadamente de forma lineal con respecto al tamaño de la tabla. Por ejemplo una tabla con 8 entradas utiliza casi el 4% de los recursos de la FPGA y una tabla de 16 entradas utiliza el 8%. Para tamaños grandes, el consumo de recursos es más y más elevado; por lo que es conveniente utilizar tamaños de tabla moderados. Por otro lado, el periodo de reloj soportado es prácticamente constante, 100 MHz. Sin embargo, se debe remarcar que las FPGAs actuales incluyen soporte para el uso de múltiples relojes, por lo tanto, la frecuencia de reloj de 100 MHz puede que sólo reduzca el rendimiento de nuestro planificador, pero no el rendimiento de las tareas que son ejecutadas en las URs, ya que éstas pueden utilizar su propio reloj.

Tamaño (entradas)	8	16	32	64
Número de *slices*	496	1113	2068	3864
Slices (%)	3.6%	8.1%	15%	28.2%
Block RAMs(%)	2.2%	2.2%	2.2%	2.2%
Frecuencia de reloj (MHz)	100	100	99,2	98,7

Tabla 4. Coste y frecuencia de reloj para distintos tamaños de la tabla asociativa

EVALUACIÓN DEL RENDIMIENTO

En esta sección se realiza una evaluación del rendimiento de nuestro planificador. En primer lugar, evaluaremos la mejora en el rendimiento que se obtiene al aplicar las ya mencionadas técnicas de prebúsqueda y reemplazo. Seguidamente mostramos los resultados obtenidos al aplicar la estrategia LF+C; en concreto, evaluamos el porcentaje de reutilización y la mejora en el rendimiento que se producen para distintas estrategias de reemplazo en el módulo planificador. Finalmente, evaluamos qué impacto en el rendimiento que se obtiene al utilizar nuestro planificador. Para ello, hemos desarrollado una versión SW equivalente que servirá para hacer un estudio comparativo entre ambas versiones y mostrar las ventajas e inconvenientes de la utilización de una sobre otra.

Técnicas de prebúsqueda y reemplazo

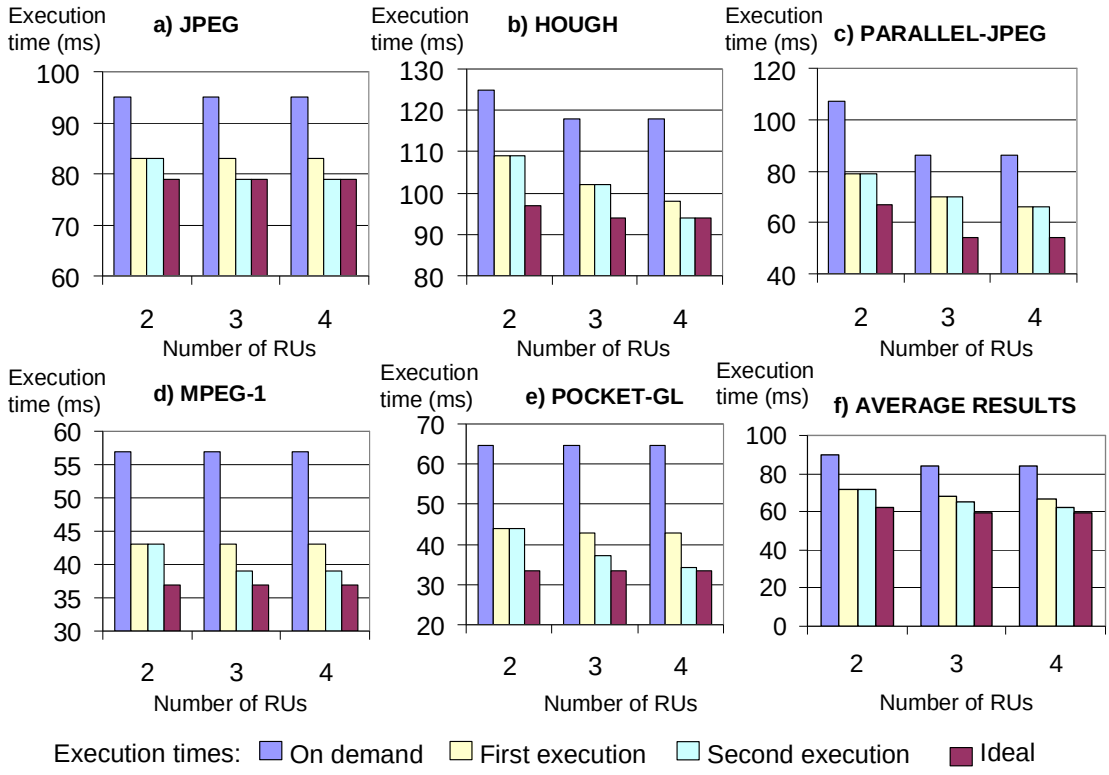


Figura 38. Impacto de la optimización de la prebúsqueda y reutilización en el funcionamiento del sistema

En la figura 38 se muestra la reducción de la penalización de la reconfiguración debido a las técnicas de optimización aplicadas al planificador. La figura incluye cuatro tiempos de ejecución:

On demand: Presenta una situación en la que no se realiza ninguna optimización. Las reconfiguraciones comienzan cuando una tarea puede ser ejecutada; por tanto, todas las reconfiguraciones producen retardos.

First execution: En este caso, se aplican las técnicas de prebúsqueda y reemplazo a la ejecución de un grafo una sola vez.

Second execution: En este caso, se aplican también las técnicas de prebúsqueda y reemplazo a la ejecución de un grafo dos veces consecutivas. El tiempo de ejecución mostrado es el de la segunda ejecución. Como es de esperar, en este caso algunas veces es posible reutilizar alguna tarea más debido a que después de la primera ejecución, algunas tareas quedan reconfiguradas en las URs. Por tanto, en este caso se pueden reducir aún más las penalizaciones por reconfiguración.

Ideal: Representa el tiempo de ejecución sin penalizaciones por reconfiguación; es decir, como si las reconfiguraciones no produjesen retardos.

En estos experimentos la latencia de reconfiguración se ha establecido en 4 ms, que es el tiempo necesario para reconfigurar la quinta parte de una FPGA XC2VP30, ya que todas las tareas consideradas en este experimento caben en esa área.

Examinando la figura 38, vemos que en la estrategia *on demand* es muy ineficiente y que las optimizaciones de prebúsqueda y reemplazo pueden pueden mejorar enormemente el funcionamiento del sistema: Por regla general las reconfiguraciones generan una penalización del 42% en tiempo de ejecución cuando se utiliza la aproximación *on demand*, sin embargo sólo se genera alrededor de un 13% de penalización cuando se utiliza la técnica de prebúsqueda. Además, esta penalización se reduce a tan sólo el 9% cuando el planificador puede reutilizar alguna de las tareas.

En este experimento se ha utlizado un pequeño número de URs para crear una situación en la que exista mucha competencia por utilizar las URs. Sin embargo, si las penalizaciones por reconfiguración son todavía elevadas, nuestro planificador puede reducirlas si se incluyen más URs. Por ejemplo, la figura 38 no muestra ninguna mejora en la columna *second execution* respecto a *first execution* para la aplicación Parallel-JPEG, ya que se trata de un grafo con muchos nodos. Pero a medida que se van incluyendo más URs, aumentan las probabilidades de reutilización en una segunda ejecución del grafo. En la tabla no aparece, pero si se incluye una UR adicional (es decir cinco URs en total) a la aplicación Parallel-JPEG, una de las tareas puede ser reutilizada y las latencias debidas a las reconfiguraciones se reducirían a 4 ms.

En cualquier caso, la tendencia es la reducción de las penalizaciones por reconfiguración a medida que:

- aumenta el número de ejecuciones consecutivas del mismo grafo. En este caso, se facilita la reutilización de tareas de una ejecución a la siguiente.

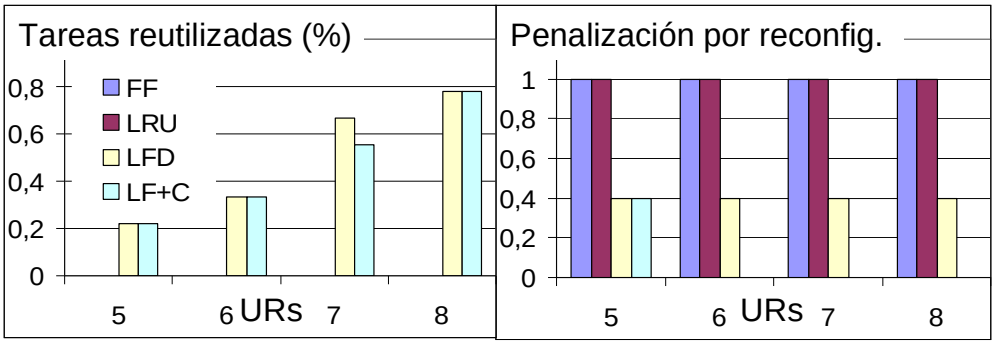
- aumenta el número de URs en el sistema. A mayor cantidad de URs, aumentan las probabilidades de que haya una UR libre cuando se quiera realizar una prebúsqueda. Asimismo, se producen menos reemplazos, con lo que se favorece también la reutilización.

Nuestra estrategia de planificación: LF+C

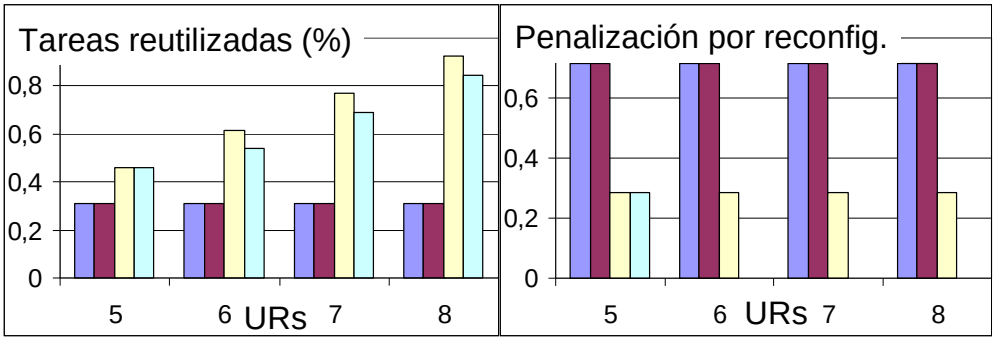
En este apartado se evalúa el funcionamiento de la estrategia de planificación LF+C, cuyo desarrollo e implementación ha ocupado gran parte del presente trabajo. Se presenta un estudio comparativo con las demás estrategias de planificación mencionadas en la sección que describía el módulo de reemplazo: *First Free* y *LRU*. También se incluye la estrategia LFD [Bela66].

Para ello, hemos ejecutado en el planificador dos grafos de tareas multimedia actuales: JPEG (4 nodos) y MPEG (5 nodos). La figura 39 muestra los resultados obtenidos. Como se puede ver, se muestran tres experimentos, que corresponden a diferentes patrones de ejecución para estas dos tareas. En primer lugar, en la figura 39 a) se muestran los resultados para la ejecución del grafo JPEG y el MPEG una vez cada uno, y así sucesivamente. En el

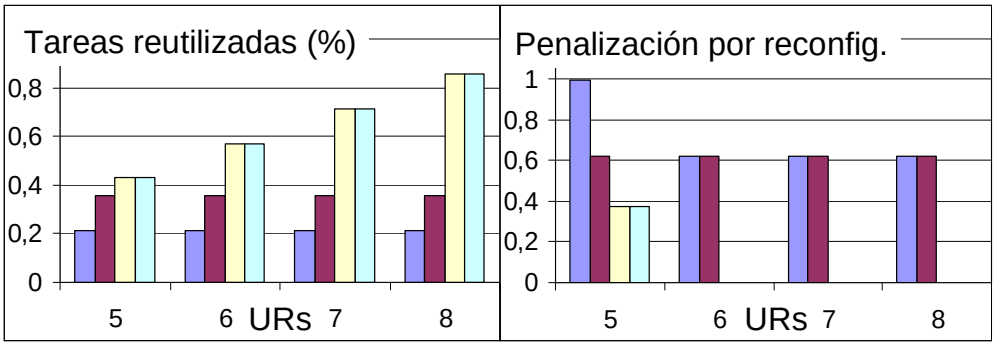
apartado b) aparecen los resultados para la ejecución de JPEG dos veces y una vez el MPEG. Finalmente, en el apartado c) se ejecuta el JPEG una vez y el MPEG dos veces. En los tres casos, se ejecutan las tareas siguiendo el patrón que corresponda y se muestran los resultados para la segunda ejecución, ya que la primera vez no es posible reutilizar nada. Y para cada caso, se muestra el porcentaje de reutilización de tareas que se produce (gráfica izquierda) y el porcentaje de las penalizaciones que no se consiguen ocultar (gráfica derecha).



(a) 1 JPEG-1MPEG



(b) 2 JPEG-1MPEG



(c) 1 JPEG-2MPEG

Figura 39. Tasas de reutilización y penalizaciones de reconfiguración cuando se ejecutan los grafos de tareas JPEG y MPEG para diferentes estrategias de reemplazo

Por un lado, examinando el porcentaje de reutilizaciones, se observa que tanto *First Free* como LRU producen resultados pobres (apenas se llega al 40%), mientras que *LFD* y *LF+C* aumentan considerablemente este porcentaje, llegando a tasas de cerca del 80% cuando se tienen 8 URs. También es significativo el hecho de que, mientras las dos primeras estrategias de reemplazo obtienen los mismos resultados independientemente del número de URs; en las dos últimas, este porcentaje va aumentando considerablemente; y puede llegar a ser hasta tres veces mayor. Esto según los resultados de la gráfica; ya que con suficiente número de URs (más que tareas), en la estrategia LRU no habría competencia entre tareas y no habría penalizaciones por reconfiguración. Vemos que nuestra estrategia *LF+C* obtiene en la mayoría de los casos los mismos resultados que *LFD*; mientras que en el resto de las ocasiones este

porcentaje de reutilización es muy similar, aunque algo menor. Esto es algo que esperábamos, ya que *LFD* tiene conocimiento de los eventos que se producirán en el futuro, y *LF+C* no utiliza esta información. No obstante, en el peor de los casos, ambos porcentajes no difieren en más del 11%, lo cual es un resultado excelente.

Por otro lado, más importante que el porcentaje de reutilizaciones es minimizar las penalizaciones debido a las reconfiguraciones existentes. Examinando este dato, se observa que, en todos los casos, las estrategias *First Free* y *LRU* ocultan muy pocas latencias: por ejemplo, en el caso a) no se consigue ocultar ninguna aunque haya 8 URs y en el caso b) estas penalizaciones se mantienen en torno al 75% de la penalización máxima. Sin embargo, las estrategias *LFD* y *LF+C* obtienen muy buenos resultados, manteniendo esta penalización en torno al 40% en el peor de los casos. Y lo que es más importante, *LF+C* mejora los resultados de *LFD*, tal y como se puede ver en la parte derecha de la figura. Esto es debido a que, mientras que *LFD* maximiza el número de reutilizaciones que se producen, nuestra heurística identifica qué tareas son críticas e intenta reutilizarlas prioritariamente sobre las demás. Por ello, aunque se consiga reutilizar menos tareas, se intentan reutilizar prioritariamente las críticas, que son aquéllas que producen las penalizaciones por reconfiguración.

Comparativa entre versiones HW/SW

En esta sección se evalúa el impacto en el rendimiento que se obtiene al utilizar ambas versiones del planificador, al margen de si aplican o no técnicas de optimización (prebúsqueda, reemplazo y estrategias de planificación). Es decir, ahora sólo nos fijaremos en los retardos que añade el planificador por el simple hecho de funcionar. Para ello, hemos desarrollado además una versión SW equivalente, con la que haremos un estudio comparativo de las prestaciones de nuestro planificador. Ambas versiones (HW y SW) se han implementado en una FPGA Virtex-II PRO xc2vp30 utilizando el entorno EDK, y se han evaluado utilizando un conjunto de grafos de tareas sacados de aplicaciones multimedia actuales, incluyéndose dos versiones de un decodificador JPEG (JPEG y Parallel-JPEG), un codificador MPEG-1, un reconocedor de patrones (HOUGH), y una aplicación de renderización 3D basada en la librería de código libre Pocket-GL [Shang02b] (Pocket GL). En el último caso la aplicación incluye 20 grafos diferentes, por eso en los resultados sólo se presenta la media de los resultados. Estos grafos han sido planificados para una plataforma con 4 URs utilizando el entorno de planificación presentado en [Res05a], sin embargo, se puede utlizar cualquier otra planificación. Finalmente, comprobaremos que ambas versiones ofrecen diferentes compromisos área/rendimiento. Por tanto, el diseñador puede elegir la que más convenga dependiendo de si se quiere minimizar el área utilizada (versión SW) o las penalizaciones temporales (versión HW).

En el siguiente apartado, se detallará la implementación SW alternativa. Finalmente, en la siguiente sección se realizará un estudio pormenorizado de las prestaciones de ambas versiones.

1) Versión SW equivalente

En esta sección explicaremos la implementación SW de nuestro planificador. La figura 40 muestra una visión general del mismo. Como puede observarse a primera vista, la principal diferencia es la desaparición del planificador HW (cuya tarea ahora será realizada por el procesador) y la comunicación directa entre el procesador y las URs a través de las líneas de interrupción (por lo que aumentarán las comunicaciones HW/SW). También hemos conectado contadores programables (*OPB_timers*), tantos como URs queremos que tenga nuestro sistema. Estos contadores simularán el comportamiento (las reconfiguraciones y ejecuciones) de las tareas que se ejecutan en esas unidades, ya que aún no contamos con el soporte

necesario para reconfigurar realmente grafos de tareas en la FPGA mediante reconfiguraciones parciales.

Cada *OPB_Timer* tiene dos contadores, que utilizamos para simular los tiempos de reconfiguración y ejecución respectivamente. De este modo, para simular una latencia de reconfiguración, utilizamos el contador 0 del *OPB_Timer* correspondiente. Para las latencias de ejecución utilizaríamos el contador 1. Igual que con la versión HW, también tenemos un *OPB_Timer* adicional para medir el tiempo (en ciclos) que ha tardado en ejecutarse el grafo.

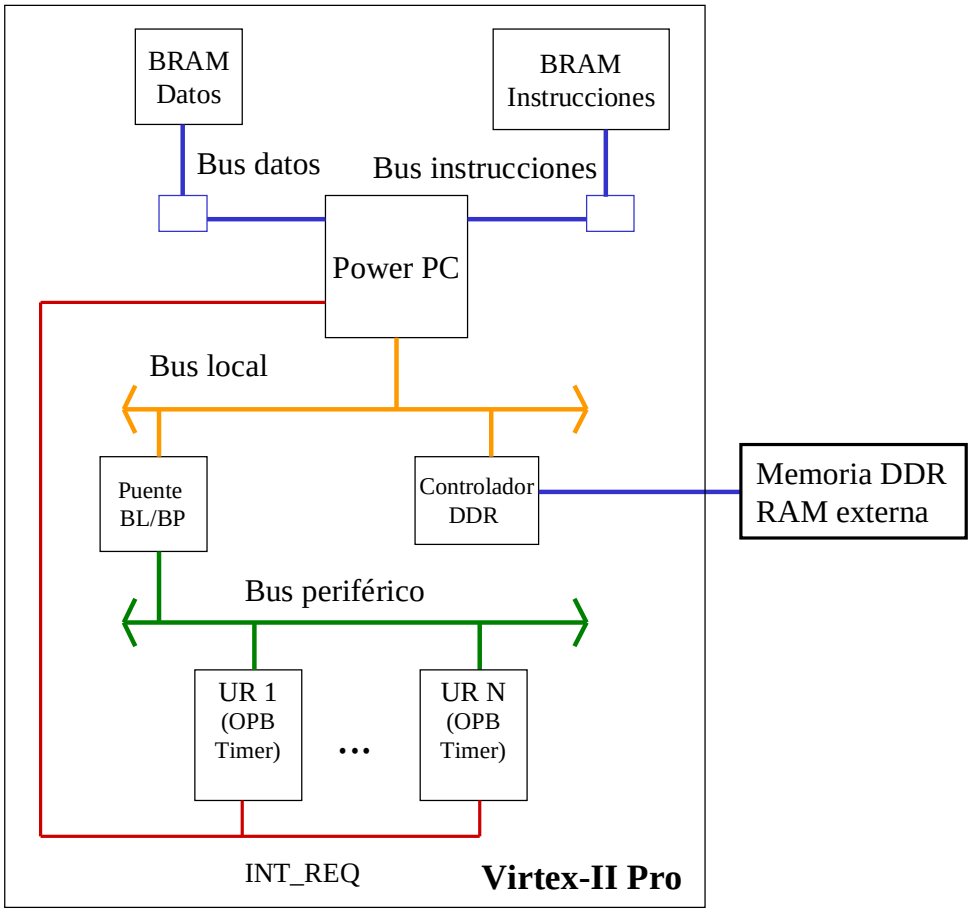


Figura 40. Implementación SW equivalente

Por tanto, esta versión SW del planificador consiste en un programa escrito en C que incluye un código de inicialización de la plataforma (inicialización de los componentes y configuración de las interrupciones) y una colección de rutinas de tratamiento de interrupción (RTIs), una para cada UR, que son las encargadas de gestionar la captura y tratamiento de eventos.

Las estructuras de datos que utilizamos son las que aparecen en la figura 41. Representamos un nodo con una estructura que contiene: un identificador único, el número de sucesores, el número de predecesores y un array con los identificadores de sus sucesores. Representamos un evento con una estructura que contiene: un identificador de evento único (distinto para cada tipo de evento) y un puntero al nodo que lo generó. Finalmente, representamos cada UR con una estructura que contiene: el estado de la unidad (*parada*, *reconfigurando*, *esperando ejecución* o *ejecutando*), un array con los identificadores de los nodos que se deben reconfigurar localmente en esa unidad y un puntero al *OPB_Timer* que simula su comportamiento. Nótese la complejidad de estas estructuras de datos, lo que hace que el código del planificador (detallado a continuación) sea complejo también.

```
TNodo: struct{                    //PARA REPRESENTAR UN NODO
       int id;                    //id único
       int num_sucesores;         //número de sucesores
       int num_predecesores;      //número de predecesores
       sucesores: array [1..NUM_MAX_SUCESORES];      //sucesores
}
TEvento: struct{                  //PARA REPRESENTAR UN EVENTO
       int id;
       TNodo* fuente;        //El nodo que ha generado el evento
}
TUR: struct{                      //PARA REPRESENTAR UNA UR
       int estado;
       int tarea_actual;
       fifoLocal: array [1..NUM_MAX_RECONF_LOCALES];
       XTmrCtr* contador;
}
UR URs[NUM_URS];                  //URs
NodoGrafo grafos [NUM_GRAFOS*NUM_MAX_NODOS_GRAFO]; //Los grafos
```

Figura 41. Estructuras de datos utilizadas en la versión SW de nuestro planificador

El pseudo-código de este planificador está descrito en la figura 42 (La totalidad del código desarrollado en C se adjunta al final de la memoria como un apéndice). Su funcionamiento es el siguiente: En primer lugar, inicializamos todos los componentes: los *OPB_Timers* y el controlador de interrupciones; y configuramos las interrupciones para asignar a cada contador una RTI diferente. A continuación, las estructuras de datos necesarias: el grafo, el array de eventos (inicialmente con un evento de tipo *nueva_tarea*, que será el primero que se procese), el de reconfiguraciones (inicialmente con todas las reconfiguraciones que se producirán) y el de las URs (inicialmente sin tareas en ejecución y con sus respectivas reconfiguraciones a la espera de que se produzcan). Entonces se ejecuta el bucle principal, del que no se saldrá en tanto en cuanto no se haya generado *NUM_NODOS* veces el evento fin de ejecución, siendo *NUM_NODOS* el número de nodos de nuestro grafo.

Cada vez que el planificador deba enviar una orden de reconfiguración o de ejecución a una UR, se envía una orden de que cuente un determinado número de ciclos (de acuerdo a sus tiempos de reconfiguración o de ejecución; y a la frecuencia de funcionamiento del bus) al contador del *OPB_Timer* correspondiente, simulando así la latencia deseada. Cuando el contador finaliza, genera una interrupción; con lo que el flujo del programa salta a la RTI correspondiente. De este modo, ya sabemos la UR que la ha generado. A continuación, averiguamos qué contador ha interrumpido; y generamos el evento consecuente: fin de reconfiguración para el contador 0 y fin de ejecución para el contador 1. Cuando esta gestión finaliza, el flujo de ejecución vuelve al bucle principal, que inmediatamente detectaría que se ha generado un nuevo evento, con lo que le daría el tratamiento adecuado (descrito en secciones anteriores), actualizando convenientemente las estructuras de datos. Esta gestión se realiza en las funciones *evento_nueva_tarea*, *evento_fin_reconf* y *evento_fin_ejecucion*. Cuando este bucle tiene constancia de que se ha tratado *NUM_NODOS* veces el evento fin de ejecución, termina. Lo último que se hace a continuación es medir cuántos ciclos han transcurrido en el *OPB_Timer* adicional para saber cuál ha sido el tiempo de ejecución del grafo en nuestro planificador SW.

```
//Declaración de las constantes NUM_NODOS, NUM_MAX_EVENTOS...
Grafo: array [1..NUM_NODOS] de TNodo;
Eventos: array [1..NUM_MAX_EVENTOS] de TEvento;
Reconfiguraciones: array [1..NUM_MAX_RECONFIG] de int;
URs: array [1..NUM_URs] de TUR;
void codigo_gestor(){
   inicializa_sistema ();
   mientras (not terminado){
      if (Eventos not vacia) {
         extrae_evento(&ev);
         switch(ev){
            case 0: evento_nueva_tarea();
            case 1: evento_fin_reconf(ev.fuente);
            case 2: evento_fin_ejecucion(ev.fuente);
         }
      }
   }
}
void RTI_0{ codigo_RTI(0); }     //RTIs de los contadores
…
void RTI _N{ codigo_RTI(n); }
void codigo_RTI (int n){
   int p_address = addr(n);      //Se obtiene la dirección del contador
   if (contador_activado(p_address) == 0){
      genera_evento_fin_reconfiguracion();
      stop_contador(p_address, 0);
   }else{
      genera_evento_fin_ejecucion();
      stop_contador(p_address, 1);
   }
}
int main(){                      //PROGRAMA PRINCIPAL
   inicializa_perifericos();
   inicializa_temporizador(&timer);
   int t1 = getValue_timer(&timer);
   codigo_gestor();
   int t2 = getValue_timer(&timer);
   printf(t2-t1);                //Se muestra el tiempo transcurrido
}
```

Figura 42. Pseudo-código de la versión SW de nuestro planificador

2) Comparación de prestaciones

La tabla 5 presenta los retardos que el planificador introduce en la ejecución de los grafos mencionados: JPEG, PARALLEL-JPEG, MPEG-1, HOUGH y POCKET GL. Las columnas 2 y 3 incluyen el número de tareas de cada grafo y el tiempo de ejecución inicial; es decir, asumiendo que la gestión de la ejecución no genera ninguna penalización.

Por su parte, las columnas de la 4 a la 7 proporcionan detalles en cuanto a la penalización generada por la versión SW, mientras que la última columna se refiere a la versión HW. Estos retardos se deben al tiempo de computación y a las comunicaciones HW/SW entre el procesador y las URs.

Para la implementación SW, la columna 4 muestra el tiempo total de ejecución del grafo correspondiente. Hemos desglosado ese tiempo en:

Tiempo de comunicaciones: Es únicamente el tiempo que se emplea en las comunicaciones entre el planificador y las URs, que se representan como contadores programables. Por ello, en este tiempo se han considerado las comunicaciones SW-HW

que se producen al dar órdenes a estos contadores utilizando la interfaz de comunicación que proporciona EDK.

Tiempo de cómputos: Es el tiempo total de penalización no dedicado a comunicaciones.

Estos tiempos se muestran en las columnas 5 y 6. Finalmente, la columna 7 muestra el impacto que estos retardos tienen sobre el funcionamiento general de las aplicaciones.

Por otro lado, la columna 8 muestra el retardo total producido por la versión HW. Para esta implementación, el tiempo de gestión incluye tanto el retardo introducido por la lógica del planificador como el tiempo de las comunicaciones. En este caso, el retardo introducido debido a la computación para la gestión son solamente unos pocos cientos de ciclos en un sistema funcionando a 100 MHz (es decir, unos pocos milisegundos). Respecto a las comunicaciones, estos retardos son de aproximadamente 2200 ciclos cuando se utiliza un DMA.

Tarea	Número de tareas	Tiempo inicial de ejecución (ms)	RENDIMIENTO DEL PLANIFICADOR				
			SW				HW
			Tiempo de ejecución (ms)	Tiempo de gestión (ms)		% penalización	Tiempo de ejecución (ms)
				Cómputos	Comunicaciones		
JPEG	4	83	83.87	0.466	0.405	1.04	83.022
PARALLEL-JPEG	8	66	67.40	0.588	0.809	2.18	71.023
MPEG-1	5	43	44.05	0.597	0.455	2.45	43.023
HOUGH	6	98	98.89	0.325	0.557	0.91	98.022
POCKET GL	5	44	45.06	0.551	0.510	2.41	44.022

Tabla 5. Evaluación del rendimiento del planificador de grafos de tareas

Como puede verse en la tabla, la implementación SW genera pequeños retardos que oscilan desde el 1% al 3% del tiempo inicial de ejecución. Si estos retardos no son aceptables, se pueden reducir enormemente los tiempos de computación y ejecución utilizando la implementación HW. Así, el tiempo de comunicación se reduce de un promedio de 0,54 ms a tan sólo 0,022 ms, y el tiempo de gestión se reduce drásticamente (al menos en tres órdenes de magnitud).

Como hemos mostrado en la tabla 5, el hecho de que el planificador aplique en tiempo de ejecución las técnicas de prebúsqueda, apenas genera retardos. Esto es especialmente cierto en el caso de la versión HW del planificador. Por tanto, estamos ante una mejora importante cuando comparamos con otras técnicas de prebúsqueda presentadas anteriormente, porque, como se ha explicado en el apartado de trabajo relacionado, la mayoría de ellas no pueden aplicarse en tiempo de ejecución, mientras otras proponen cómputos complejos en tiempo de ejecución, pero no evalúan la penalización en tiempo de ejecución generada por esos cómputos.

Para terminar este apartado, haremos un breve comentario acerca del tamaño de los códigos SW de ambas versiones del planificador. En sistemas empotrados, el tamaño del código es también una medida importante ya que normalmente el espacio en memoria es muy reducido. (Por ejemplo, en la FPGA Virtex-II PRO xc2vp30 el tamaño de la RAM interna es de 306KB). El tamaño del código de la implementación SW del planificador es de 197KB mientras que para la versión HW es de 82KB. Estos tamaños no sólo incluyen el planificador, sino también todos los drivers para los periféricos incluidos en el sistema. Por tanto, que a pesar de que la implementación HW del planificador tenga un coste de área HW significativo, reduce enormente los requerimientos de memoria del sistema. Esto puede útil ser sobre todo para reducir los accesos a memorias externas.

CONCLUSIONES

En este trabajo hemos desarrollado y testeado un planificador de grafos de tareas en una plataforma HW multi-tarea. Con ello, pretendemos reducir las penalizaciones por reconfiguración que se producen en tales sistemas sin castigar excesivamente el rendimiento general del mismo. Para lograr este objetivo, dicho planificador aplica técnicas de prebúsqueda y reutilización, además de una técnica de reemplazo (*LF+C*) que nos hemos encargado de desarrollar y testear. Todas estas técnicas colaboran entre sí para mejorar sus resultados individuales.

Nuestro planificador ha demostrado una gran eficiencia a la hora de reducir penalizaciones por reconfiguraciones, anticipando y reutilizando inteligentemente las tareas. Nuestra estrategia de reemplazo consigue tasas de reutilización de hasta el 80% y reducir así, de media, las penalizaciones por reconfiguración del 42% a aproximadamente el 9%. Se debe remarcar que hemos comparado *LF+C* con *LFD*, una estrategia que es óptima en cuanto a número de reutilizaciones conseguidas. Aunque no conseguimos alcanzar tasas de reutilización óptimas, sí reducimos más penalizaciones por reconfiguración que *LFD*. Esto es debido a que en nuestro caso nos centramos en reutilizar las tareas críticas, que son las que producen dichas penalizaciones, y en consecuencia, es más interesante reutilizar.

Por otro lado, hemos comparado el coste y el rendimiento entre la versión HW del planificador y una versión SW equivalente para un conjunto de aplicaciones multimedia actuales. Ambas implementaciones representan diferentes *trade-off*s entre rendimiento/coste en área/tamaño de los códigos. La versión HW tiene un coste en área de aproximadamente el 13% de los recursos HW de una FPGA Virtex-II PRO xc2vp30, no introduce prácticamente penalizaciones de rendimiento en el sistema (unos pocos cientos de ciclos de reloj) y el tamaño del código a ejecutar en el POWER-PC es reducido (82 KB). Por otro lado, la versión SW no tiene coste en área, pero introduce penalizaciones que oscilan entre el 1% y el 3% del tiempo total de ejecución de los grafos de tareas, de acuerdo a nuestros experimentos. Asimismo, el tamaño del código ejecutable que se genera es mayor (197 KB). Por tanto, el usuario puede elegir la alternativa de implementación más adecuada a los requisitos de área/memoria/prestaciones que desee.

TRABAJO FUTURO

Tras mucho esfuerzo y dedicación, podemos estar orgullosos al afirmar que hemos conseguido cumplir prácticamente todos los objetivos que nos propusimos al inicio del proyecto: hemos diseñado, implementado y testeado un planificador para la ejecución de tareas HW, así como una comparativa con una versión SW equivalente del sistema.

No obstante, creemos que aún queda mucho trabajo por hacer si queremos obtener una implementación eficiente de un planificador de tareas que permita la ejecución real de grafos de tareas en una plataforma dinámicamente reconfigurable.

En primer lugar, creemos que podemos mejorar la eficiencia de la implementación HW de nuestro planificador: en concreto, queremos diseñar una tabla asociativa más escalable, y también reducir su retardo mediante la aplicación de técnicas de anticipación de operandos en la red iterativa y/o añadiendo algún ciclo extra. Esto haría que la frecuencia mínima de reloj a la que el sistema funcionase correctamente se redujera, con la consecuente ganancia en velocidad. No obstante, si con estas ampliaciones no consiguiéramos la mejora en velocidad deseada, pensamos explorar una implementación alternativa del planificador SW, totalmente diferente a la que actualmente tenemos: se trataría de un programa escrito en código ensamblador (*assembly code*) de la arquitectura POWER-PC o MICROBLAZE. Pensamos que tal implementación puede reducir los tiempos de gestión (no así los tiempos de comunicación, que en principio, deberían ser similares), al tratarse de una implementación más optimizada que un código originalmente escrito en C y compilado para una arquitectura en concreto. Esto implicaría tratar a muy bajo nivel con interrupciones, comunicación procesador-periféricos o incluso manejo de protocolos para realizar reconfiguración parcial.

Cabe mencionar que nuestro planificador no se limita a utilizar una única estrategia de reemplazo. Como ya hemos mencionado en secciones anteriores, hemos integrado en él varias estrategias para compararlas entre sí (*First Free*, *LRU* y *LF+C*). Naturalmente, podríamos integrar cualquier otra que nos pareciera interesante y con la que se consiga reducir significativamente la penalización por reconfiguraciones a cambio de no castigar excesivamente el rendimiento del sistema. También nos parece interesante explotar el paralelismo a nivel de grafo: ejecutar varios grafos simultáneamente, en vez de secuencialmente, como se hace en este trabajo.

Finalmente, y como trabajo a medio-largo plazo, pretendemos implementar un sistema HW multi-tarea basado en reconfiguración parcial, por lo que será necesario investigar cómo poder llevarla acabo de manera eficiente. En este sentido, creemos que podría ser interesante la colaboración con otros grupos de investigación que trabajen en ello. De esta forma, nuestro planificador dejaría de ser simplemente una plataforma de simulación para convertirse en un sistema completo que reciba tareas reales de entrada, planifique y gestione su ejecución y devuelva los resultados.

Explorar todas estas mejoras y alternativas de implementación ayudaría, sin duda, a proporcionar soporte a la investigación del HW reconfigurable en general y al desarrollo de una tecnología con posibles usos comerciales en el futuro; algo que nos parece especialmente interesante.

PUBLICACIONES GENERADAS

Se presentan a continuación las publicaciones mediante las que se ha divulgado el trabajo de investigación realizado en esta memoria. Están ordenadas por fecha, de más antigua a más reciente:

- **"HW implementation of an execution manager for reconfigurable systems",** Javier Resano, Juan Antonio Clemente, Carlos Gonzalez, Jose Luis Garcia and Daniel Mozos. Engineering of Reconfigurable Systems and Algorithms (ERSA), June 25-28 2007.

- **"Un sistema para la gestión eficiente del HW reconfigurable",** Carlos González, Juan Antonio Clemente, José Luis García, Javier Resano y Daniel Mozos. Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA), 11-14 septiembre de 2007.

- **"Efficiently scheduling run-time reconfigurations",** Javier Resano, Juan Antonio Clemente, Carlos González, Daniel Mozos and Francky Catthoor. Transactions on Design Automation of Electronic Systems, 2008.

- **"Task-graph management for reconfigurable multi-tasking systems",** Juan Antonio Clemente, Carlos González, Javier Resano and Daniel Mozos. Proceedings of ReCoSoc'2008. Barcelona, July 9-11 2008.

- **"Task-execution and reconfiguration management for reconfigurable multitasking systems",** Juan Antonio Clemente and Javier Resano. ACACES 2008. Poster Abstracts, Advanced Computer Architecture and Compilation for Embedded Systems, L'Aquila, July 13-19 2008.

- **"Implementaciones HW y SW de un gestor de ejecución de grafos de tareas en un sistema multitarea reconfigurable",** Juan Antonio Clemente, Carlos González, Javier Resano y Daniel Mozos. Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA), 18-19 septiembre de 2008.

GLOSARIO DE TÉRMINOS

Asíncrono:
Hace referencia al suceso que no tiene lugar en total correspondencia temporal con otro suceso. Si se refiere a una comunicación asíncrona, ésta es en la que se realiza el envío de datos sin la sincronización de un reloj externo.

Bit:
Dígito en el sistema binario.

Bus:
Conjunto de conductores eléctricos en forma de pistas metálicas impresas sobre la placa base del computador, por donde circulan las señales que corresponden a los datos binarios del lenguaje máquina. Si el bus es de datos, este conectará dos dispositivos hardware.

Byte:
8 bits.

Chip:
Circuito integrado o pastilla en la que se encuentran todos o casi todos los componentes necesarios para que un ordenador pueda realizar alguna función.

Ciclo:
Un ciclo es la distancia temporal entre el principio y el final de una onda completa.

Circuito impreso:
Medio para sostener mecánicamente y conectar eléctricamente componentes electrónicos, a través de rutas o pistas de material conductor, grabados desde hojas de cobre laminadas sobre un sustrato no conductor.

Depuración:
Proceso de mejora de un sistema hasta que realiza su función correctamente.

Empotrado:
Ubicado dentro del chip.

Esquemático:
Es un diagrama, dibujo o boceto que detalla los elementos de un sistema.

Evento:
Acontecimiento ocurrido en el sistema.

FIFO:
First In, First Out. Protocolo que implementa una cola: lo que viene primero, se maneja primero, lo que viene segundo espera hasta que lo primero haya sido manejado, etc.

Flanco:
Transición del nivel bajo al alto (flanco de subida) o del nivel alto al bajo (flanco de bajada) de una señal.

FPGA:
Dispositivo donde se pueden programar infinidad de diseños digitales distintos.

Gestor:
Elemento que se encarga de organizar y dar órdenes a otros elementos de menor nivel en la jerarquía de diseño.

Grafo de tareas dirigido acíclico:
Conjunto de nodos relacionados mediante aristas de simple o doble sentido. No contiene ciclos.

Hardware:
Es un término general usado para describir artefactos físicos de una tecnología.

LRU:
Least Recently Used. Algoritmo de reemplazo de páginas, en el que un reemplazo se produce en la posición cuyo último uso fue más lejano en el tiempo respecto a la actualidad.

Planificación:
Modelo de ejecución para una tarea.

Precarga:
Técnia que prevé y lleva a cabo una reconfiguración parcial con anterioridad a su utilización.

Predecesor:
Que se encuentra con anterioridad en la jerarquía.

Prototipo:
Ejemplar original o primer molde en que se fabrica un diseño.

Reconfiguración:
Proceso en el que parte de la FPGA (reconfiguración parcial) o toda (reconfiguración total) se reprograma.

Síncrono:
Hace referencia al suceso que tiene lugar en correspondencia temporal con otro suceso. Si se refiere a una comunicación síncrona, ésta es en la que se realiza el envío de datos bajo la sincronización de un reloj externo.

Speed-up:
Cociente que nos dice la mejora de rendimiento obtenida.

Sucesor:
Que se encuentra con posterioridad en la jerarquía.

Tarea:
Cada una de las partes de un grafo de tareas con significado propio.

Unidad reconfigurable:
Parte de la FPGA que puede ser reconfigurada de modo independiente.

VHDL:
Very High Speed Integrate Circuit Hardware Description Language, es un lenguaje de descripción y modelado, diseñado para describir la funcionalidad y la organización de sistemas *hardware* digitales, placas de circuitos, y componentes.

BIBLIOGRAFÍA Y REFERENCIAS

[Actel97] Actel Corporation: *Accelerator Series FPGAs: ACT3 Family*. http://www.actel.com, 1997.

[Actel01] Actel Corporation: *SX Family of High Performance FPGAs*. http://www.actel.com, 2001.

[Altera08] www.altera.com/products/devices/stratix-fpgas/about/stx-about.html. May, 2008.

[Ahr90] M. Ahrens, A. El Gamal, D. Galbraith, J. Greene y S. Kaptanoglu: *An FPGA family optimized for high densities and reduced routing delay*. En *Proceedings of the IEEE Custom Integrated Circuits Conference*, páginas 31.5.1-31.5.4, 1990.

[And99] T. Anderson: *System-on-Chip Design with Virtual Components*. Circuit Cellar Online, http://www.circuitcellar.com, Agosto 1999.

[Bela66] L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," IBM Systems J., vol. 5, no. 2, 1966, pp. 78-101.

[Bon02] K. Bondalapati y V.K. Prasanna: *Reconfigurable Computing Systems*. Proceedings of the IEEE, 90(7):1201-1217, Julio 2002.

[Burns97] J. Burns, A. Donlin, J. Hogg, S. Singh y M. De Wit: *A Dynamic Reconfiguration Run-Time System*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 66-76, Abril 1997.

[Chou00] Y. Chou, P. Pillai, H. Schmit y J.P. Shen: *PipeRench Implementation of the Instruction Path Coprocessor*. En *Proceedigns of the 33th Annual International Symposium on Microarchitecture (MICRO)*, páginas 147-158, 2000.

[Com02a] K. Compton y S. Hauck: *Reconfigurable Computing: A Survey of Systems and Software*. ACM Computing Surveys, 34(2):171-210, Junio 2002.

[Com02b] K. Compton, Z. Li, J. Cooey, S. Knol y S. Hauck: *Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing*. IEEE Transactions on VLSI Systems, 10(3):209-220, Junio 2002.

[CS00] Inc. Chameleon Systems: *CS2000 Advance Product Specification*. http://www.isis.vanderbilt.edu/projects.asp, 2000.

[Dan03] K. Danne, C. Bobda y H. Kalte: *Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration*. Lecture Notes in Computer Science, 2778:272-281, 2003.

[DeHon96] A. DeHon: *Reconfigurable Architectures for General Purpose Computing*. A.I. Technical report No. 1586. Artificial Intelligence Laboratory. Massachusetts Institute of Technology, Septiembre 1996.

[EDK08] www.xilinx.com/ise/embedded/edk_docs.htm. May, 2008.

[Fu05] W. Fu, K. Compton. An execution environment for reconfigurable computing. FCCM'05, p.149-158. 2005.

[Gar06] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu. "An Overview of Reconfigurable Hardware in Embedded Systems". EURASIP Journal on Embedded Systems, vol. 2006.

[Gau02] J. Gause, P.Y.K. Cheung y W. Luk: *Reconfigurable Shape-Adaptive Template Matching Architectures*. En *Proceedigns of the IEEE FPGA Custom Computing Machine Conference 2002*, páginas 98-110, 2002.

[Hau97] J. Hauser y J. Wawrzynek: *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 12-21, Abril 1997.

[Hauck98a] S. Hauck: *Configuration Prefetch for Single Context Reconfigurable Coprocessors*. En *Proceedigns of the ACM International Symposium in Field Programmable Gate Arrays (FPGA'98)*, páginas 65-74, Febrero 1998.

[Hauck98b] S. Hauck, Z. Li y E. Schewabe: *Configuration Compression for the Xilinx XC6200 FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, páginas 138-147, Abril 1998.

[Hauck02] Z. Li, S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation", FPGA'02, pp. 187-195. 2002

[Hauck04] S. Hauck, T.W. Fry, M.M. Hosler y J.P. Kao: *The Chimaera Reconfigurable Functional Unit*. IEEE Transactions on VLSI Systems, 12(2):206-217, Febrero 2004.

[Hsieh90] H. Hsieh, W. Carter, J. Y. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin y L. Tinkey: *Third-generation Architecture Boosts Speed and Density of Field-programmable Gate arrays*. En *Proceedings of the IEEE Custom Integrated Circuits Conference*, páginas 31.2.1-31.2.7, 1990.

[Huang00] W.J. Huang and N. Saxena y E.J. McCluskey: *A reliable LZ data compressor on reconfigurable coprocesors*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 249-258, Abril 2000.

[Laz04] J. Lázaro, A. Astarloa, U. Bidarte, J. Arias y C. Cuadrado: *High Throughput Serpent Encryption Implementation*. Lecture Notes in Computer Science, 3203:996-1000, 2004.

[Leung00] K.H. Leung, K.W. Wong y P.H.W. Leong: *FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 68-76, Abril 2000.

[Li00] Z. Li, K. Compton y S. Hauck: *Configuration Caching Techniques for FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 22-38, Abril 2000.

[Li02] Z. Li, S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation", FPGA'02, pp. 187-195. 2002.

[Lys06] P. Lysaght et al.," Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs", FPL'06, p. 1-6, 2006.

[Mares02] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", FPL,p. 795-805, 2002.

[Nogue04a] J. Noguera, M. Badia., "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling". ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, p. 385-406. 2004

[Nogue04b] J. Noguera, M. Badia., "Power-Performance Trade-Offs for Reconfigurable Computing". CODES+ISSS.pp. 116-121. 2004

[Nollet04a] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde. "Operating System controlled Networkon-Chip". DAC'04, p. 256-259, 2004.

[Nollet04b] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde. “Operating System controlled Networkon-Chip”. Proceedings of the Design Automation Conference (DAC), pag. 256-259, 2004.

[Qu06] Y. Qu, J. Soininen, J..Nurmi, "A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead”, DATE’06, pp. 965- 969. 2006.

[Res05a] J. Resano, D. Mozos, F Catthoor, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Runtime the Reconfiguration Overhead of Dynamically Reconfigurable HW" DATE05, pp.106-111. 2005

[Res05b] J. Resano, D. Mozos, D Verkest, F Catthoor, "*A Reconfiguration Manager for Dynamically Reconfigurable Hardware*", IEEE Design&Test, Vol. 22, Issue 5, pp. 452-460. 2005.

[Rupp98] C.R. Rupp, M. Landguth, T. Garverick, E. Comersall, H. Holt, J.M. Arnold y M. Gokhale: *The NAPA adaptive processing architecture*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, páginas 28-37, Abril 1998.

[Sch97] H. Schmit: *Incremental Reconfiguration for Pipelined Applications*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 16-18, Abril 1997.

[Shang02a] L. Shang, N.K. Jha, “Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs”, ASP-DAC’02, pp. 345-360, 2002.

[Shang02b] L. Shang, N.K. Jha, “Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs”, ASP-DAC’02, pp. 345-360, 2002.

[Sid99a] R. Sidhu, A. Mei y V.K. Prassana: *Genetic Programming using Self-Reconfigurable FPGAs*. Lecture Notes in Computer Science, 1673:301-312, 1999.

[Sid99b] R. Sidhu y A. Mei V.K. Prasanna: *String Matching on Multicontext FPGAs using Self-Reconfiguration*. En *Proceedigns of the ACM International Symposium in Field Programmable Gate Arrays (FPGA'99)*, páginas 217-226, Febrero 1999.

[Sid01] R. Sidhu y V.K. Prasanna: *Fast Regular Expression Matching using FPGAs*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, páginas 698-709, Abril 2001.

[So06] H. Kwok-Hay So, R. W. Brodersen, "Improving Usability of FPGA-Based Reconfigurable Computers through Operating System Support", FPL '06, p. 1-6, 2006.

[SOPC08] www.altera.com/products/software/products/sopc/sop-index.html. May, 2008.

[Tes01] R. Tessier y W. Burleson: *Reconfigurable Computing for Digital Signal Processing: a survey*. Journal of VLSI Signal Processing, (28):7-27, Mayo 2001.

[Tri97] S. Trimberger, D. Carberry, A. Johnson y J. Wong: *A time-multiplexed FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 22-28, Abril 1997.

[Tuley97] J. Tuley: *Soft Computing Reconfigures Designer Options*. Embedded Systems, página 76, Abril 1997.

[Vik06a] K. N. Vikram, V. Vasudevan. “Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures”. IEEE Trans. on VLSI Systems, Volume 14, Issue 9, Sept. 2006, p.1010-1023.

[Vik06b] K. N. Vikram, V. Vasudevan. “Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 14, Issue 9, Sept. 2006, pp.1010-1023.

[Wal03] H. Walder, M. Platzner, “Online Scheduling for Block-partitioned Reconfigurable Devices”. DATE’03 p. 10290-10295. 2003.

[Xil00] Xilinx Corp.: *Using Block SelectRAM+ Memory in Spartan II FPGAs*. Xilinx Application Notes, http://www.xilinx.com, Deciembre 2000.

[Xil02a] Xilinx Corp.: *XC6200 Field Programmable Gate Arrays*. Xilinx Documentation, http://www.xilinx.com, 2002.

[Xil02b] Xilinx Corp.: *Virtex FPGA Series Configuration and Readback*. Xilinx Application Notes, http://www.xilinx.com, Julio 2002.

[Xil02c] Xilinx Corp.: *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. Xilinx Application Notes, http://www.xilinx.com, Mayo 2002.

[Xil03a] Xilinx Corp.: *Virtex 2.5V Field Programmable Gate Array*. Xilinx Documentation, http://www.xilinx.com, Deciembre 2002.

[Xil05] Xilinx Inc., Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet, Xilinx, San Jose, Calif, USA, 2005.

[Xil08] www.xilinx.com/products/silicon_solutions/index.htm. May, 2008.

[Yang02] P. Yang et al, "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia systems", ISSS'02, pp. 112-119, 2002.

[ZLi02] Zhiyuan Li, “Configuration management techniques for reconfigurable computing”, Ph.D. thesis, ISBN:0-493-65106-3. 2002.

APÉNDICE

CÓDIGO C DE LA VERSIÓN SW

```
#include "xintc.h"
#include "xexception_l.h"
#include "xparameters.h"
#include "stdio.h"
#include "xbasic_types.h"
#include "xtmrctr.h"
#include "stdlib.h"

#define UARTLITE_DEVICE_ID        XPAR_RS232_UART_DEVICE_ID
#define INTC_DEVICE_ID            XPAR_OPB_INTC_0_DEVICE_ID
#define INT_IRQ_UR3               XPAR_OPB_INTC_0_CONTADOR_UR3_INTERRUPT_INTR
#define INT_IRQ_UR2               XPAR_OPB_INTC_0_CONTADOR_UR2_INTERRUPT_INTR
#define INT_IRQ_UR1               XPAR_OPB_INTC_0_CONTADOR_UR1_INTERRUPT_INTR
#define INT_IRQ_UR0               XPAR_OPB_INTC_0_CONTADOR_UR0_INTERRUPT_INTR
#define NUM_MAX_SUCESORES         8
#define NUM_URS                   4
#define NUM_MAX_RECONFIGURACIONES 16
#define NUM_MAX_RECONFIGURACIONES_LOCALES 8
#define NUM_MAX_EVENTOS           16
#define NUM_MAX_NODOS_GRAFO       3
#define NUM_GRAFOS                4
#define FIRST_FREE                0
#define LRU                       1
#define LF_C                      2

struct NodoGrafo{
      //primer identificador para las reconfiguraciones/ejecuciones 1->2->1
      int id;
      //segundo identificador para comprobar las dependencias 1->2->3
      int idGrafo;
      Xuint32 tReconfiguracion;
      Xuint32 tEjecucion;
      int nPredecesores;
      int nSucesores;
      int sucesores[NUM_MAX_SUCESORES];
      int critico; //dice si el nodo es crítico (1) o no (0)
};

/*los índices de las reconfiguraciones y ejecuciones son SIEMPRE en términos
de id's de grafos. Sólo se comprueba su id estándar. Cuando se quiere
comprobar si, a la hora de cargar una subtarea, debe reutilizarse o no.*/

struct NodoEvento{
      struct NodoGrafo*   fuente;      //la subtarea que lo produce
      //El evento que se genera: 0->Nueva Tarea; 1->FR; 2->FE
      int                 evento;
};

struct UR{
      int estado;  //el estado de la UR: 0->Finalizado;
                   //1->Cargando; 2->Esperando ejecución; 3->Ejecutando
      int subtarea_actual;      //la subtarea actual
      XTmrCtr contador;         //el contador para los tiempos de
                                //reconfiguración (el 0) y ejecución (el 1)
};

struct FifoReconfiguraciones{
      int    reconfiguraciones[NUM_MAX_RECONFIGURACIONES];
      int    indice;
      int    num;
```

```
};

struct FifoEventos{
	struct NodoEvento   eventos[NUM_MAX_EVENTOS];
	int                             indice;
	int                             num;
};

typedef struct NodoGrafo NodoGrafo;
typedef struct NodoEvento NodoEvento;
typedef struct UR UR;
typedef struct FifoReconfiguraciones FifoReconfiguraciones;
typedef struct FifoEventos FifoEventos;

void RTI(void * direccion_base);
void codigo_RTI(int UR);
void RTI_UR0(void);
void RTI_UR1(void);
void RTI_UR2(void);
void RTI_UR3(void);
void inicializaGrafos ();
void inicializaURs ();
void inicializaFIFOs(void);
void eventoNuevaTarea(void);
void eventoFinReconfiguracion(NodoGrafo* fuente);
void eventoFinEjecucion(NodoGrafo* fuente);
void elegirSubtareaACargar(void);
int inicializaPerifericos(void);
void carga (int i, int siguienteSubtarea);
void inicializaEstrategia(void);

FifoEventos fifoE;
FifoReconfiguraciones fifoR;
int circuitoReconfiguracionLibre;       //1 -> SI; 0 -> NO
XTmrCtr counter;                        //el contador de ciclos
XIntc ControladorInterrupciones;        //El controlador de interrupciones
UR URs[NUM_URS];                        //URs
NodoGrafo grafos[NUM_GRAFOS*NUM_MAX_NODOS_GRAFO];

/*Los grafos de tareas, todos los nodos seguidos. En la posición i del grafo
está la subtarea con id i+1*/
int numNodosGrafos[NUM_GRAFOS];         //Los números de nodos de los grafos
int numGrafosActuales;                  //El número de grafos que quedan

//El número de reconfiguraciones que quedan por hacerse en el grafo activo
int numReconfiguracionesActuales;
int numNodosActuales;      //Número de nodos que quedan en el grafo activo
int terminadoGrafo;        //Para ver si el grafo actual ha terminado
int terminado;             //Para ver si he terminado

//*****VARIABLES PARA LAS ESTRATEGIAS DE REEMPLAZO
/*La estrategia de reemplazo que se usará: FIRST_FREE, LRU o LF_C*/
int estrategiaReemplazo;
int listaLRU[NUM_URS];     //La lista de valores de las URs para LRU

//LAS LISTAS DE CANDIDATOS (ESTRATEGIA LF+C)
int ECs[NUM_URS];   //Empty Candidates --> candidatos en los que no haya nada
int PCs[NUM_URS];
int CCs[NUM_URS];
int RCs[NUM_URS];
int numPCs = 0;
int numCCs = 0;
int numRCs = 0;
int numECs = 0;


//*************************************************************************
```

```
void elegirSubtareaACargar(){

	int i;
	if (fifoR.num > 0){
		int siguienteSubtarea = fifoR.reconfiguraciones[fifoR.indice];

		//Estrategia de reemplazo FIRST FREE
		if (estrategiaReemplazo == FIRST_FREE){
			for (i=0;i<NUM_URS;i++){
				// Si el estado es finalizado
				if(URs[i].estado == 0){
					carga (i, siguienteSubtarea);
					break;
				}
			}

		//Estrategia de reemplazo LRU
		} else if (estrategiaReemplazo == LRU) {
			int i;
			for (i=0; i<NUM_URS; i++){
				if (listaLRU[i] == NUM_URS-1)
					break;
			}

			if(URs[i].estado == 0){

				carga (i, siguienteSubtarea);
				int j;

				//Actualizo la lista de valores LRU
				for (j=0; j<NUM_URS; j++){
					(listaLRU[j])++;
				}
				listaLRU[i] = 0;
			}

		//Estrategia de reemplazo LF_C
		} else if (estrategiaReemplazo == LF_C) {
	//Para cada UR, decido el tipo de candidato que es: EC, PC, CC o RC
			numECs=0; numPCs=0; numCCs=0; numRCs=0;

			int indiceCandidato;	//El índice de la UR candidata
			int reutilizar = 0;	//Si reutilizo o no

			int i;
			for (i=0; i<NUM_URS; i++){
				int j;
				int esta = 0;

				NodoGrafo* nodoCandidato =
					&grafos[URs[i].subtarea_actual-1];

				for (j=fifoR.indice;
				j<fifoR.indice+numReconfiguracionesActuales; j++){
					if (URs[i].subtarea_actual != 0 &&
		nodoCandidato->id == grafos[fifoR.reconfiguraciones[j]-1].id){
						esta = 1;
						break;
					}
				}

				if (URs[i].estado == 0 &&
						URs[i].subtarea_actual == 0){
					ECs[numECs] = i;
					numECs++;
				}else if (URs[i].estado == 0 &&
					esta == 0 && nodoCandidato->critico == 0){
					PCs[numPCs] = i;
```

```
                                numPCs++;
                        }else if (URs[i].estado == 0 &&
                                esta == 0 && nodoCandidato->critico == 1){
                                CCs[numCCs] = i;
                                numCCs++;
                        }else if (URs[i].estado == 0 && esta == 1){
/*Si aparte de estar en la FIFO de reconfiguraciones, la puedo reutilizar
ahora mismo*/
                                if (nodoCandidato->id ==
                                        grafos[siguienteSubtarea-1].id){
                                        reutilizar = 1;
                                        indiceCandidato = i;
                                }else{
                                        RCs[numRCs] = i;
                                        numRCs++;
                                }
                        }
                }

                //Selecciono el candidato
                if (reutilizar == 1){
                        carga(indiceCandidato, siguienteSubtarea);

                }else if (numECs > 0){
                        indiceCandidato = ECs[0];
                        carga(indiceCandidato, siguienteSubtarea);
                }else if (numPCs > 0){
                        indiceCandidato = PCs[0];
                        carga(indiceCandidato, siguienteSubtarea);
                }else if (numCCs > 0){
                        indiceCandidato = CCs[0];
                        carga(indiceCandidato, siguienteSubtarea);
                }else if (numRCs > 0){
                        indiceCandidato = RCs[0];
                        carga(indiceCandidato, siguienteSubtarea);
                }

        }
    }
}

//*************************************************************************

void carga (i, siguienteSubtarea){

    // Sólo la cargo si es distinta (si fuera igual se reutiliza)
    Xuint32 id2Verdad = 0;
    Xuint32 id2Grafo = 0;
    if (URs[i].subtarea_actual != 0){
        //la id de verdad de la subtarea que está ahora en la UR
        id2Verdad = grafos[URs[i].subtarea_actual-1].id;
            id2Grafo = grafos[URs[i].subtarea_actual-1].idGrafo;
    }

    //la id de verdad de la subtarea que quiero cargar
    Xuint32 id1Verdad = grafos[siguienteSubtarea-1].id;
    Xuint32 id1Grafo = grafos[siguienteSubtarea-1].idGrafo;

    if (id1Verdad != id2Verdad){
        circuitoReconfiguracionLibre = 0;
        XTmrCtr c = URs[i].contador;
        Xuint32 tRec = grafos[siguienteSubtarea-1].tReconfiguracion;
        XTmrCtr_SetResetValue(&c, 0, tRec);
        XTmrCtr_Reset(&c, 0);
        XTmrCtr_mEnableIntr(c.BaseAddress, 0);
        XTmrCtr_Start(&c, 0);
        // El estado es ahora "reconfigurando"
        URs[i].estado = 1;
```

```
	}else{
		//Generar el evento de SUBTAREA REUTILIZADA
		NodoEvento n;
		// La fuente es la subtarea actualmente cargada en la UR0
		n.fuente = &(grafos[siguienteSubtarea-1]);
		n.evento = 1;
		fifoE.num++;
		fifoE.eventos[(fifoE.indice + fifoE.num - 1)%NUM_MAX_EVENTOS] = n;
		// El estado es ahora "esperando ejecución"
		URs[i].estado = 2;
	}

	// Actualizo la subtarea actual
	// Actualizo la fifo global de reconfiguraciones
	URs[i].subtarea_actual = siguienteSubtarea;   fifoR.indice =
				(fifoR.indice + 1)%NUM_MAX_RECONFIGURACIONES;
	fifoR.num--;

}

//*************************************************************************

void eventoNuevaTarea(){
	if (circuitoReconfiguracionLibre == 1){
		elegirSubtareaACargar();
	}
}

//*************************************************************************

void eventoFinReconfiguracion(NodoGrafo* fuente){

	numReconfiguracionesActuales--;
	circuitoReconfiguracionLibre = 1;

	// Buscamos la unidad en la que esté
	int k,m;
	for (m=0; m<NUM_URS ; m++){
		if (URs[m].subtarea_actual > 0 && URs[m].subtarea_actual ==
		 fuente->idGrafo){
			k = m;
			break;
		}
	}

	// Comenzar ejecución
	URs[k].estado = 2;

	if (fuente->nPredecesores == 0){

		// Uso el contador 1, que es el de las ejecuciones.
		XTmrCtr c = URs[k].contador;
		Xuint32 tEje = fuente->tEjecucion;
		XTmrCtr_SetResetValue(&c, 1, tEje);
		XTmrCtr_Reset(&c, 1);
		XTmrCtr_mEnableIntr(c.BaseAddress, 1);
		XTmrCtr_Start(&c, 1);
		// Actualizar el estado de la unidad
		URs[k].estado = 3;
	}

	if (circuitoReconfiguracionLibre == 1 && numReconfiguracionesActuales !=
0){
		elegirSubtareaACargar();
	}
}

//*************************************************************************
```

```
void eventoFinEjecucion(NodoGrafo* fuente){

	// Buscamos la unidad en la que esté
	int k,m,j,i;

	for (m=0; m<NUM_URS ; m++){
		if (URs[m].subtarea_actual > 0 && URs[m].subtarea_actual ==
			fuente->idGrafo){
			k = m;
			break;
		}
	}


	URs[k].estado = 0;

	// Actualizar el estado de la unidad -> "finalizado" y dependencias
	for (j=0; j< fuente->nSucesores; j++){
		grafos[  grafos[fuente->idGrafo-1].sucesores[j]-1   ]
                                                    .nPredecesores--;
	}

	int siguienteSubtarea;
	for (i=0; i<NUM_URS ;i++){

		siguienteSubtarea = ( URs[i].subtarea_actual );
		if (URs[i].estado == 2 && siguienteSubtarea > 0 &&
			grafos[siguienteSubtarea-1].nPredecesores == 0){

			// ENVIAR SEÑAL PARA QUE COMIENCE LA EJECUCION
			XTmrCtr c = URs[i].contador;
			Xuint32 tEje = grafos[siguienteSubtarea-1].tEjecucion;
			XTmrCtr_SetResetValue(&c, 1, tEje);
			XTmrCtr_Reset(&c, 1);
			XTmrCtr_mEnableIntr(c.BaseAddress, 1);
			XTmrCtr_Start(&c, 1);

			// Actualizar el estado de la unidad -> "ejecutando"
			URs[i].estado = 3;
		}
	}

	if (circuitoReconfiguracionLibre == 1 && numReconfiguracionesActuales !=
0){
		elegirSubtareaACargar();
	}

	numNodosActuales--;
	if (numNodosActuales == 0){
		terminadoGrafo = 1;
	}
}

//*************************************************************************

void codigoGestor(void){

	inicializaGrafos();
	inicializaFIFOs();
	inicializaURs();
	inicializaEstrategia();

	// espera activa grafos totales
	while(terminado == 0) {

		// inicializo FIFO de eventos con un "nueva tarea"
		NodoEvento n;
```

```
		n.fuente = NULL;
		n.evento = 0;

		fifoE.eventos[0] = n;
		fifoE.indice = 0;
		fifoE.num = 1;

		// espera activa grafo actual
		while(terminadoGrafo == 0){
			if (fifoE.num>0){
				NodoEvento n = fifoE.eventos[fifoE.indice];
				switch (n.evento){
					//Nueva tarea
					case 0: eventoNuevaTarea();
						break;
					//Fin de reconfiguración
					case 1 : eventoFinReconfiguracion(n.fuente);
						break;
					//Fin de ejecución
					case 2: eventoFinEjecucion(n.fuente);
						break;
					default:
						printf("-- Evento desconocido --\r\n");
					break;
				}

				//avanzo una posicion en la fifo
				fifoE.indice = (fifoE.indice + 1)%NUM_MAX_EVENTOS;
				fifoE.num--;
			}
		}

		terminadoGrafo = 0;
		numGrafosActuales--;
		numNodosActuales = numNodosGrafos[NUM_GRAFOS-numGrafosActuales];
		numReconfiguracionesActuales =
				numNodosGrafos[NUM_GRAFOS-numGrafosActuales];
		if (numGrafosActuales == 0){
			terminado = 1;
		}
	}

}

//*****************************************************************************
//AQUÍ CONFIGURO LA RTI DE CADA CONTADOR!!!!
//*****************************************************************************

int inicializaPerifericos(void){
    XStatus Estado;

    /*
     * Initialize the interrupt controller driver so that it is ready to use
     */
    Estado = XIntc_Initialize(&ControladorInterrupciones, INTC_DEVICE_ID);
    if (Estado != XST_SUCCESS){
        print("--> ERROR Inicializando el OPB Interrupt Controller...\r\n");
            return XST_FAILURE;
    }


    /*
     * Connect a device driver handler that will be called when an interrupt
     * for the device occurs, the device driver handler performs the specific
     * interrupt processing for the device
     */
    Estado = XIntc_Connect(&ControladorInterrupciones,
      XPAR_OPB_INTC_0_CONTADOR_UR3_INTERRUPT_INTR, (XInterruptHandler)RTI_UR3,
```

```
                                        (void *)XPAR_CONTADOR_UR3_BASEADDR);
    if (Estado != XST_SUCCESS){
        print("--> ERROR conectando el contador 3 al IH...\r\n");
               return XST_FAILURE;
    }


    Estado = XIntc_Connect(&ControladorInterrupciones,
       XPAR_OPB_INTC_0_CONTADOR_UR2_INTERRUPT_INTR, (XInterruptHandler)RTI_UR2,
                                        (void *)XPAR_CONTADOR_UR2_BASEADDR);
    if (Estado != XST_SUCCESS){
        print("--> ERROR conectando el contador 2 al IH...\r\n");
               return XST_FAILURE;
    }

    Estado = XIntc_Connect(&ControladorInterrupciones,
       XPAR_OPB_INTC_0_CONTADOR_UR1_INTERRUPT_INTR, (XInterruptHandler)RTI_UR1,
                                        (void *)XPAR_CONTADOR_UR1_BASEADDR);
    if (Estado != XST_SUCCESS){
        print("--> ERROR conectando el contador 1 al IH...\r\n");
               return XST_FAILURE;
    }

    Estado = XIntc_Connect(&ControladorInterrupciones,
       XPAR_OPB_INTC_0_CONTADOR_UR0_INTERRUPT_INTR, (XInterruptHandler)RTI_UR0,
                                        (void *)XPAR_CONTADOR_UR0_BASEADDR);
    if (Estado != XST_SUCCESS){
        print("--> ERROR conectando el contador 0 al IH...\r\n");
               return XST_FAILURE;
    }

    /*
     * Start the interrupt controller such that interrupts are enabled for
     * all devices that cause interrupts, specific real mode so that
     * the UR can cause interrupts thru the interrupt controller.
     */
    Estado = XIntc_Start(&ControladorInterrupciones, XIN_REAL_MODE);
    if (Estado != XST_SUCCESS){
        print("--> ERROR iniciando el OPB Interrupt Controller...\r\n");
               return XST_FAILURE;
    }


    /*
     * Enable the interrupt for the URs
     */
    XIntc_Enable(&ControladorInterrupciones, INT_IRQ_UR3);
    XIntc_Enable(&ControladorInterrupciones, INT_IRQ_UR2);
    XIntc_Enable(&ControladorInterrupciones, INT_IRQ_UR1);
    XIntc_Enable(&ControladorInterrupciones, INT_IRQ_UR0);

    /*
     * Initialize the PPC405 exception table
     */
    XExc_Init();

    /*
     * Register the interrupt controller handler with the exception table
     */
    XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
                         (XExceptionHandler)XIntc_InterruptHandler,
                         &ControladorInterrupciones);

    /*
     * Enable non-critical exceptions
     */
    XExc_mEnableExceptions(XEXC_NON_CRITICAL);
```

```
	return 0;
}

//*****************************************************************************

int main (void) {

	Xuint32 t1,t2;
	print("\n\r**************************\r\n");
	print("\n\r-- Entrando en el main() --\r\n");
	print("\n\r**************************\r\n");
	terminadoGrafo = 0;
	terminado = 0;

	XStatus e = inicializaPerifericos();

	print("-- Inicio del codigo del gestor...\r\n");

	//Configuración del contador de ciclos
	XTmrCtr_Initialize(&counter, XPAR_OPB_TIMER_0_DEVICE_ID);
	XTmrCtr_Reset(&counter, 0);
	XTmrCtr_Start(&counter, 0);

	t1=XTmrCtr_GetValue(&counter, 0);
	codigoGestor();

	t2=XTmrCtr_GetValue(&counter, 0);
	xil_printf("Tiempo transcurrido = %d\r\n", t2-t1);

   print("-- Saliendo del main() --\r\n");
   return 0;
}

//*****************************************************************************

void RTI(void * direccion_base_p){
	print("-- Interrupcion!! --\r\n");
}

//*****************************************************************************

void codigo_RTI(int UR){
	Xuint32 csr0, csr1, direccion;

	switch (UR){
		case 0: direccion = XPAR_CONTADOR_UR0_BASEADDR;
			break;
		case 1: direccion = XPAR_CONTADOR_UR1_BASEADDR;
			break;
		case 2: direccion = XPAR_CONTADOR_UR2_BASEADDR;
			break;
		case 3: direccion = XPAR_CONTADOR_UR3_BASEADDR;
			break;
	}

	csr0 = XTmrCtr_mGetControlStatusReg(direccion, 0);
	csr1 = XTmrCtr_mGetControlStatusReg(direccion, 1);

	// Evento "fin de reconfiguración"
	if (csr0 & XTC_CSR_INT_OCCURED_MASK) {
		XTmrCtr_Stop(&(URs[UR].contador), 0);
		NodoEvento n;

		// La fuente es la subtarea actualmente cargada en la UR0
		n.fuente = &(grafos[URs[UR].subtarea_actual-1]);
		n.evento = 1;
		fifoE.num++;
		fifoE.eventos[(fifoE.indice + fifoE.num - 1)%NUM_MAX_EVENTOS] = n;
```

```
			XTmrCtr_mDisableIntr(direccion, 0);

		// Evento "fin de ejecución"
		}else if (csr1 & XTC_CSR_INT_OCCURED_MASK){
			XTmrCtr_Stop(&(URs[UR].contador), 1);
			NodoEvento n;

			// La fuente es la subtarea actualmente cargada en la UR0
			n.fuente = &(grafos[URs[UR].subtarea_actual-1]);
			n.evento = 2;
			fifoE.num++;
			fifoE.eventos[(fifoE.indice + fifoE.num - 1)%NUM_MAX_EVENTOS] = n;
			XTmrCtr_mDisableIntr(direccion, 1);
		}
}

//*****************************************************************************

void RTI_UR0(void){
		codigo_RTI(0);
}

void RTI_UR1(void){
		codigo_RTI(1);
}

void RTI_UR2(void){
		codigo_RTI(2);
}

void RTI_UR3(void){
		codigo_RTI(3);
}

//*****************************************************************************

void inicializaGrafos(){

		NodoGrafo n1;
		n1.id = 1;
		n1.idGrafo = 1;
		n1.critico = 1;
		n1.tReconfiguracion = 400000;
		n1.tEjecucion = 900000;
		n1.nPredecesores = 0;
		n1.nSucesores = 1;
		n1.sucesores[0] = 2;
		grafos[0] = n1;

		NodoGrafo n2;
		n2.id = 2;
		n2.idGrafo = 2;
		n2.critico = 0;
		n2.tReconfiguracion = 400000;
		n2.tEjecucion = 500000;
		n2.nPredecesores = 1;
		n2.nSucesores = 1;
		n2.sucesores[0] = 3;
		grafos[1] = n2;

		NodoGrafo n3;
		n3.id = 3;
		n3.idGrafo = 3;
		n3.critico = 0;
		n3.tReconfiguracion = 400000;
		n3.tEjecucion = 400000;
		n3.nPredecesores = 1;
		n3.nSucesores = 0;
```

```
grafos[2] = n3;

numNodosGrafos[0] = 3;

NodoGrafo n4;
n4.id = 4;
n4.idGrafo = 4;
n4.critico = 1;
n4.tReconfiguracion = 400000;
n4.tEjecucion = 800000;
n4.nPredecesores = 0;
n4.nSucesores = 2;
n4.sucesores[0] = 5;
n4.sucesores[1] = 6;
grafos[3] = n4;

NodoGrafo n5;
n5.id = 5;
n5.idGrafo = 5;
n5.critico = 0;
n5.tReconfiguracion = 400000;
n5.tEjecucion = 600000;
n5.nPredecesores = 1;
n5.nSucesores = 0;
grafos[4] = n5;

NodoGrafo n6;
n6.id = 6;
n6.idGrafo = 6;
n6.critico = 0;
n6.tReconfiguracion = 400000;
n6.tEjecucion = 600000;
n6.nPredecesores = 1;
n6.nSucesores = 0;
grafos[5] = n6;

numNodosGrafos[1] = 3;

NodoGrafo n7;
n7.id = 1;
n7.idGrafo = 7;
n7.critico = 1;
n7.tReconfiguracion = 400000;
n7.tEjecucion = 900000;
n7.nPredecesores = 0;
n7.nSucesores = 1;
n7.sucesores[0] = 8;
grafos[6] = n7;

NodoGrafo n8;
n8.id = 2;
n8.idGrafo = 8;
n8.critico = 0;
n8.tReconfiguracion = 400000;
n8.tEjecucion = 500000;
n8.nPredecesores = 1;
n8.nSucesores = 1;
n8.sucesores[0] = 9;
grafos[7] = n8;

NodoGrafo n9;
n9.id = 3;
n9.idGrafo = 9;
n9.critico = 0;
n9.tReconfiguracion = 400000;
n9.tEjecucion = 400000;
n9.nPredecesores = 1;
n9.nSucesores = 0;
```

```
	grafos[8] = n9;

	numNodosGrafos[2] = 3;

	NodoGrafo n10;
	n10.id = 4;
	n10.idGrafo = 10;
	n10.critico = 1;
	n10.tReconfiguracion = 400000;
	n10.tEjecucion = 800000;
	n10.nPredecesores = 0;
	n10.nSucesores = 2;
	n10.sucesores[0] = 11;
	n10.sucesores[1] = 12;
	grafos[9] = n10;

	NodoGrafo n11;
	n11.id = 5;
	n11.idGrafo = 11;
	n11.critico = 0;
	n11.tReconfiguracion = 400000;
	n11.tEjecucion = 600000;
	n11.nPredecesores = 1;
	n11.nSucesores = 0;
	grafos[10] = n11;

	NodoGrafo n12;
	n12.id = 6;
	n12.idGrafo = 12;
	n12.critico = 0;
	n12.tReconfiguracion = 400000;
	n12.tEjecucion = 600000;
	n12.nPredecesores = 1;
	n12.nSucesores = 0;
	grafos[11] = n12;

	numNodosGrafos[3] = 3;

	numGrafosActuales = 4;
	numNodosActuales = numNodosGrafos[0];
	numReconfiguracionesActuales = numNodosGrafos[0];
}

//*****************************************************************************

void inicializaURs (){
	UR UR0,UR1,UR2,UR3;

	UR0.estado = 0;
	UR0.subtarea_actual = 0;
	XTmrCtr contUR0;
	XTmrCtr_Initialize(&contUR0, XPAR_CONTADOR_UR0_DEVICE_ID);
	XTmrCtr_SetOptions(&contUR0, 0, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetOptions(&contUR0, 1, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetHandler(&contUR0, (XTmrCtr_Handler)RTI_UR0, NULL);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR0_BASEADDR,0);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR0_BASEADDR,1);
	UR0.contador = contUR0;
	URs[0]=UR0;

	UR1.estado = 0;
	UR1.subtarea_actual = 0;
	XTmrCtr contUR1;
	XTmrCtr_Initialize(&contUR1, XPAR_CONTADOR_UR1_DEVICE_ID);
	XTmrCtr_SetOptions(&contUR1, 0, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
```

```
	XTmrCtr_SetOptions(&contUR1, 1, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetHandler(&contUR1, (XTmrCtr_Handler)RTI_UR1, NULL);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR1_BASEADDR,0);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR1_BASEADDR,1);
	UR1.contador = contUR1;
	URs[1]=UR1;

	UR2.estado = 0;
	UR2.subtarea_actual = 0;
	XTmrCtr contUR2;
	XTmrCtr_Initialize(&contUR2, XPAR_CONTADOR_UR2_DEVICE_ID);
	XTmrCtr_SetOptions(&contUR2, 0, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetOptions(&contUR2, 1, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetHandler(&contUR2, (XTmrCtr_Handler)RTI_UR2, NULL);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR2_BASEADDR,0);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR2_BASEADDR,1);
	UR2.contador = contUR2;
	URs[2]=UR2;

	UR3.estado = 0;
	UR3.subtarea_actual = 0;
	XTmrCtr contUR3;
	XTmrCtr_Initialize(&contUR3, XPAR_CONTADOR_UR3_DEVICE_ID);
	XTmrCtr_SetOptions(&contUR3, 0, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetOptions(&contUR3, 1, XTC_DOWN_COUNT_OPTION |
				XTC_INT_MODE_OPTION); //Modo predeterminado
	XTmrCtr_SetHandler(&contUR3, (XTmrCtr_Handler)RTI_UR3, NULL);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR3_BASEADDR,0);
	XTmrCtr_mEnableIntr(XPAR_CONTADOR_UR3_BASEADDR,1);
	UR3.contador = contUR3;
	URs[3]=UR3;
}

//*************************************************************************

void inicializaFIFOs(void){
	fifoR.indice = 0;
	fifoR.num = 12;
	fifoR.reconfiguraciones[0] = 1;
	fifoR.reconfiguraciones[1] = 2;
	fifoR.reconfiguraciones[2] = 3;
	fifoR.reconfiguraciones[3] = 4;
	fifoR.reconfiguraciones[4] = 5;
	fifoR.reconfiguraciones[5] = 6;
	fifoR.reconfiguraciones[6] = 7;
	fifoR.reconfiguraciones[7] = 8;
	fifoR.reconfiguraciones[8] = 9;
	fifoR.reconfiguraciones[9] = 10;
	fifoR.reconfiguraciones[10] = 11;
	fifoR.reconfiguraciones[11] = 12;

	circuitoReconfiguracionLibre = 1;
}

//*************************************************************************

void inicializaEstrategia(void){
	//estrategiaReemplazo = FIRST_FREE;
	//estrategiaReemplazo = LRU;
	estrategiaReemplazo = LF_C;

	int i;
	for (i=0; i< NUM_URS; i++){
		listaLRU[i] = i;
```

```
        }
}
```