UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Implementación hardware de un controlador de memoria cache de reconfiguraciones en VHDL

Proyecto de Sistemas Informáticos de Ingeniería en Informática

Abel Chocano Gómez y Carlos del Prado Mota

Profesor Director:
Juan Antonio Clemente Barreira

Madrid, 2014

UNIVERSIDAD COMPLUTENSE DE MADRID
Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Implementación hardware de un controlador de memoria cache de reconfiguraciones en VHDL

Proyecto de Sistemas Informáticos de Ingeniería en Informática

Profesor Director: Juan Antonio Clemente Barreira

Madrid, 2014

Autorización

Los autores de este proyecto autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos no comerciales tanto la memoria, como el código, la documentación y el prototipo del controlador hardware desarrollado.

Abel Chocano Gómez

Carlos del Prado Mota

Agradecimientos

Aprovechamos la oportunidad que se nos brinda para agradecer a todas esas personas que nos han apoyado tanto durante este tiempo. En primer lugar, agradecer a Juan Antonio Clemente, nuestro director de proyecto y profesor en alguna asignatura a lo largo de la carrera, por ofrecernos este proyecto con el que culminar la carrera e iniciar la transición al mundo laboral. Gracias al cual hemos podido llevar a cabo dos artículos, uno en español y otro en inglés, los cuales han sido enviados al congreso nacional Jornadas de Computación Reconfigurable y Aplicaciones (JCRA), y a la prestigiosa revista científica *IEEE Transactions on Very Large Scale Integration Systems* (IEEE TVLSI), respectivamente. Esto sin su apoyo incondicional y su esfuerzo no hubiera sido posible.

En segundo lugar y como no podía ser de otra forma, agradecer a nuestros familiares y en particular a nuestros padres por todo su apoyo, tanto en los momentos buenos como en los más difíciles, donde nunca nos han fallado.

Por último, pero no por ello menos importante, a Virginia y Lara, nuestras novias, junto a todos aquellos amigos que siempre han confiado en nosotros, con su apoyo y amistad, fortaleciéndonos en los momentos más difíciles, así como a todos los profesores y tutores que nos han encaminado hacia este momento durante toda nuestra vida.

Gracias a todos.

Resumen

Este proyecto presenta una implementación hardware de un controlador que gestiona de manera eficiente las reconfiguraciones que se realizan en tiempo de ejecución en un sistema que aplica cacheo de reconfiguraciones. Esta técnica consiste en utilizar una memoria *on-chip* que sirve de cache entre la memoria de configuración del dispositivo reconfigurable y la memoria principal, donde se guardarán todas y cada una de las reconfiguraciones que se quieran cargar en el dispositivo. La eficiencia de la técnica se puede mejorar particionando las configuraciones en bloques, y mapeando las configuraciones en diferentes memorias cache, en vez de en una sola.

De este modo, dada una asignación de reconfiguraciones de tareas en diferentes memorias *on-chip*, el controlador hardware presentado gestiona la reconfiguración de las tareas de manera adecuada y eficiente. Los resultados experimentales que se presentan muestran que nuestro controlador realiza las operaciones necesarias en unos pocos cientos ciclos de reloj, mientras que su coste de implementación en términos de recursos hardware es muy asequible.

Abstract

This project presents a hardware implementation of a controller that manages efficiently the run-time reconfigurations in a system that applies reconfiguration caching. This technique consists on using an on-chip memory that acts as a cache and that is placed between the reconfiguration memory of the device and the main memory where all the configurations are stored. The efficiency of this technique can be further improved by dividing the configurations into several blocks, each one of which can be mapped to a different on-chip memory.

Thus, given a mapping of the configuration blocks on the different existing on-chip memories, the hardware controller proposed manages the reconfiguration of these tasks in a proper and efficient way. The presented experimental results demonstrate that our module carries out the operations needed in just a few hundreds of clock cycles, while its implementation cost in terms of hardware resources is very affordable.

Lista de palabras clave

- Hardware reconfigurable.
- Cacheo de configuraciones.
- Jerarquía de memoria de configuraciones.
- Mapeo de configuraciones.
- Controlador de configuraciones.
- FPGA.
- Virtex-5.
- Xilinx ISE.
- ICAP.

Índice

Lista de figuras

Lista de tablas

1. Introducción

Una de las principales limitaciones de los dispositivos reconfigurables en general, y de las FPGAs en particular, es que el proceso de reconfiguración de las tareas en el hardware reconfigurable involucra penalizaciones importantes, en términos de tiempo y de consumo energético que no existen en ASICs (*Application Specific Integrated Circuits*) [1, 2].

Normalmente, este proceso de reconfiguración consiste en copiar los datos correspondientes a la configuración de las tareas desde una memoria *off-chip* a la memoria de configuración del dispositivo, a través de un circuito de configuración dedicado. Sin embargo, una desventaja importante de este esquema es que cargar las configuraciones directamente de una memoria *off-chip* implica grandes penalizaciones no sólo en términos de rendimiento (normalmente, del orden de cientos de milisegundos [3]), pero también en términos de consumo de energía [4]. Esta tendencia se ha observado especialmente durante la última década [5]. Así, durante los últimos años, el *gap* existente entre el ancho de banda y las prestaciones del sistema se ha ido agrandando significativamente. Y al mismo tiempo, los accesos a las memorias *on-chip* de tecnologías actuales consumen de media en torno a 250 veces menos energía por bit que a memorias *off-chip*.

Para solventar este problema, se han propuesto una serie de técnicas basadas en cacheo de reconfiguraciones [6-9]. Estas técnicas consisten en añadir una o varias memorias *on-chip* intermedias que actúan como caches, y que están situadas entre la memoria *off-chip* donde se guardan las reconfiguraciones y el hardware reconfigurable. Su objetivo es acelerar las reconfiguraciones de las tareas que se ejecutan, así como ahorrar energía en el proceso de reconfiguración. Asimismo, estas memorias pueden implementar diversos compromisos entre prestaciones y consumo de energía [10, 11].

Sin embargo, una limitación típica de estas técnicas es la capacidad reducida de las memorias RAM *on-chip* disponibles en los dispositivos reconfigurables modernos. Por ejemplo, la FPGA de tipo Virtex-5 disponible en la placa de desarrollo de Xilinx XUPV5-LX110T contiene sólo 6.448 Kb de memoria *on-chip*, en comparación con los 256 MB de los bancos de memoria DDR2 que se le pueden agregar. De este modo, en la mayoría de los casos sólo unas pocas reconfiguraciones se pueden guardar simultáneamente en las memorias *on-chip*.

Este proyecto propone extender las estrategias de cacheo de reconfiguraciones clásicas para dividir las configuraciones en bloques, reduciendo así su granularidad. Esta idea permite mapear cada uno de los diferentes bloques en los cuales las configuraciones están divididas en diferentes memorias *on-chip*. Por tanto, este esquema permite mapear en las memorias *on-chip* sólo los bloques de configuraciones necesarios para conseguir la relación deseada entre prestaciones y consumo de energía, mejorando así la flexibilidad del sistema.

Sin embargo, este esquema tiene la desventaja de que son necesarios una serie de servicios de gestión de reconfiguraciones que se tienen que realizar en tiempo de ejecución (es decir, a medida que se realizan las reconfiguraciones). Además, estos servicios se tienen que proporcionar de manera transparente al sistema operativo o *middleware* que gestiona el funcionamiento general del sistema. Este proyecto también se centra en este problema, y propone una implementación hardware de un controlador que gestiona de manera transparente y muy eficiente estas operaciones. Este controlador se ha implementado utilizando una placa de prototipado Xilinx XUPV5-LX110T, la cual contiene una FPGA Virtex-5 XC5VLX110T.

Los resultados experimentales demuestran que el controlador propuesto tiene un consumo de recursos muy asequible. Asimismo, la penalización introducida por este controlador es del orden de cientos de ciclos de reloj, lo cual es despreciable.

1.1. Hardware reconfigurable

En mayo de 1960, Gerald Estrin, matemático y físico muy respetado, considerado hoy en día como el padre de la computación reconfigurable moderna, publicó un artículo en la *Western Joint Computer Conference* titulado “Organization of Computer Systems – The Fixed Plus Variable Structure Computer”. En dicho artículo, planteó la preocupación de uno de sus compañeros de la Universidad de California en Los Ángeles (UCLA), John Pasta, acerca de los problemas computacionales de vital importancia del momento y cuyas soluciones estaban fuera del alcance utilizando los computadores del momento. En su opinión, los fabricantes de computadores habían perdido el interés en invertir en arquitecturas arriesgadas e innovadoras.

El Dr. Pasta lo desafió a encontrar nuevas formas de estructurar los sistemas informáticos de tal forma que supusiese la revolución en las computadoras del momento. A raíz del desafío, el Dr. Estrin propuso un "equipo de estructura fija más variable". Dicho proyecto consistía en utilizar hardware "reconfigurable", controlado por un computador de propósito general fijo, con la intención de aprovechar tanto el alto rendimiento del hardware como la flexibilidad del software. A pesar de que el Dr. Estrin implementó un prototipo, esta idea no tuvo éxito debido, principalmente, a la falta de tecnología en ese momento para poder llevarla a cabo.

Durante muchos años, las nuevas aplicaciones y algoritmos se han aplicado en procesadores de propósito general, von Neuman (vN). Las aplicaciones se describían como una secuencia de instrucciones, las cuales indicaban que operaciones debían ser realizadas paso a paso. Sin embargo, a medida que las aplicaciones comenzaron a requerir mayor cálculo computacional, debido a su complejidad, los equipos basados en vN comenzaron a mostrar carencias en cuanto al rendimiento requerido. Por lo que se necesitaba un apoyo adicional. Una forma sencilla de conseguirlo era añadir aceleradores hardware, que se habían utilizado normalmente como ASICs.

Este tipo de circuitos se personalizan para un uso en particular, más que para usos de propósito general. Por lo tanto, permiten obtener el rendimiento deseado de cualquier aplicación, así como bajo consumo de energía y ocupan un porcentaje pequeño del chip. Sin embargo estas características tienen altas consecuencias en el precio, ya que debido a su diseño personalizado, estos componentes no pueden ser casi nunca reutilizados. Esto despertó el interés en encontrar una nueva solución al problema.

Esto motivó el hecho de que, en la década de 1980, el hardware reconfigurable experimentara un renacimiento, que aún continúa a día de hoy. Tanto bajo fines de investigación como comerciales, este tipo innovador de dispositivo ha demostrado ser una alternativa eficaz a las basadas puramente en software o en hardware. Desde su nacimiento hasta nuestros días, los dispositivos reconfigurables han experimentado evoluciones en términos de rendimiento y cantidad de recursos reconfigurables, entre otros. Sin embargo, el concepto sigue siendo el mismo: combinar en un solo dispositivo tanto la flexibilidad de un programa de software como la velocidad de un circuito de hardware, tal y como se muestra en la Figura 1.

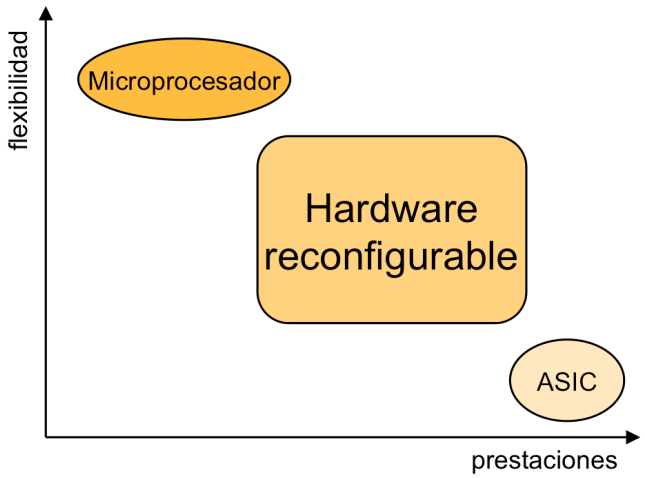

Figura 1. Clasificación de las diferentes tecnologías en términos de flexibilidad y prestaciones.

La característica principal del hardware reconfigurable es la capacidad de alterar la funcionalidad de un diseño hardware mediante la carga de un nuevo circuito en la estructura reconfigurable del dispositivo. Además, si el dispositivo es capaz de cambiar dinámicamente su funcionalidad en tiempo de ejecución (es decir, sin tener que reiniciar el sistema), estaríamos hablando de hardware dinámicamente reconfigurable (HWDR). Hoy en día, la mayoría de los dispositivos reconfigurables existentes son en realidad dinámicamente reconfigurables. Por lo tanto, en lo que sigue nos referiremos a HWDR y no a hardware reconfigurable.

Dependiendo de la forma en la que se lleve a cabo el proceso de reconfiguración, se pueden identificar varios modelos. En concreto, hay tres criterios principales que permiten clasificar HWDR. Por su granularidad (reconfiguración de grano fino y de grano grueso), por el número de reconfiguraciones que cada uno almacena (reconfiguración de un único contexto o múltiples contextos) o por el estilo

de reconfiguración (total o parcial). El primer criterio es el que ha predominado mayoritariamente en el mercado en los últimos años y es el que siguen los fabricantes a la hora de elaborar sus dispositivos. Es por ello que la siguiente subsección lo describe en mayor detalle.

1.1.1. Dispositivos de grano fino y grano grueso

Podemos clasificar el HWDR en función del nivel de detalle del sistema, es decir, en función del tamaño del elemento reconfigurable más pequeño en el dispositivo. Esto puede dar lugar a dos clasificaciones: HDWR de grano fino y grano grueso.

1.1.1.1 Grano fino

Permite describir el hardware a nivel de bit, es decir, se pueden asignar las diferentes operaciones a cada bit único del sistema. En este modelo, la configuración de los elementos básicos se compone de *Look-Up Tables* (LUTs) y biestables (*flip-flops*), los cuales, a su vez, se suelen organizar en unidades lógicas de mayor tamaño. En el caso de dispositivos Xilinx Virtex, éstos son conocidos como *Configurable Logic Blocks* (CLBs).

Este tipo de granularidad es muy adecuada para cálculos a nivel de bit, ya que los elementos lógicos también están conectados entre sí a nivel de bit. Sin embargo, no sería eficaz para cálculos con datos con ancho de palabra. Por lo tanto, podemos decir que este modelo de configuración permite obtener la máxima flexibilidad, ya que para cada algoritmo podremos optimizar al máximo los recursos disponibles utilizados para su implementación. En su defecto, aumentaría el coste en términos de área, energía, longitud, número de interconexiones existentes e incluso, tendría repercusiones negativas en el rendimiento.

Las arquitecturas de grano fino más populares son las FPGAs (*Field Programmable Gate Arrays*). Éstas se configuran a través de un mapa de bits almacenado en la memoria RAM de configuración del dispositivo. A pesar de ser arquitecturas reconfigurables de grano fino, muchas de ellas también incluyen elementos de grano grueso, como son los multiprocesadores o procesadores integrados (en el caso de dispositivos Xilinx, *Digital Signal Processors* o DSPs), los cuales ofrecen un buen rendimiento y simplifican el proceso de diseño. Una gran ventaja de este enfoque combinado es que permiten el uso de los procesadores, los requisitos de funcionamiento son bajos y además se puede utilizar el resto de los recursos configurables disponibles para definir las aplicaciones más exigentes. Por lo tanto este tipo de plataformas pueden proporcionar un consumo moderado de energía, comparable con el de un sistema integrado de bajo consumo y a la vez proporcionar el máximo rendimiento.

1.1.1.2 Grano grueso

A diferencia de los dispositivos de grano fino, en los dispositivos de grano grueso, la mayor parte de las operaciones lógicas se realizan a nivel de palabra, entre 4 y 128 bits. Los elementos lógicos que nos encontramos normalmente son unidades aritmético-lógicas (ALUs), así como elementos de interconexión y recursos de almacenamiento. Las ALUs pueden realizar diferentes operaciones, tales como operaciones aritméticas entre números enteros, desplazamiento de bits y operaciones lógicas bit a bit, dependiendo del dispositivo de grano grueso en particular.

Este tipo de granularidad proporciona mejor rendimiento y menor consumo de energía que el de grano fino, siempre y cuando la tarea pueda ser adaptada convenientemente al ancho de palabra requerido por la plataforma de grano grueso en particular. Siendo, además, menor el número de bits requeridos para configurar la tarea. Sin embargo, el aumento en el rendimiento conlleva una pérdida de flexibilidad del sistema, gran inconveniente en muchos contextos dinámicos. Es este el motivo por lo que este modelo de configuración existe principalmente en el mundo académico, ya que sistemas industriales, como son Morphosys y MATRIX, no han tenido un gran éxito entre los usuarios.

1.1.2. FPGAs Xilinx Virtex-5

Acabamos de explicar en detalle los dispositivos de grano fino en donde podemos encontrar las FPGAs, entre otros. A continuación vamos a explicarlas con más detalle, adentrándonos en el modelo que hemos utilizado para la realización del proyecto, la FPGA Xilinx Virtex-5.

Las FPGAs son circuitos electrónicos prefabricados reconfigurables que contienen bloques de lógica cuya funcionalidad e interconexión se puede programar, lo que permite crear circuitos que se comporten a voluntad del diseñador. Son típicamente volátiles. Aparecieron como una evolución de los CPLDs (*Complex Programmable Logic Devices*). Las ventajas que tienen frente a éstos son: su mayor densidad de puertas, la flexibilidad de su arquitectura y que habitualmente contienen módulos que implementan funciones de alto nivel, como sumadores o multiplicadores, y bloques de memoria empotrados en la matriz de interconexionado.

Una de las grandes ventajas de las FPGAs es que al diseñar un sistema no es necesaria su fabricación, sino que se puede realizar programando correctamente el dispositivo, por lo que los costes se reducen.

Como hemos comentado anteriormente, hoy en día las FPGAs no sólo incluyen CLBs, bloques de entrada/salida o enrutado, si no que también incluyen una mezcla equilibrada de componentes como memorias *on-chip*, multiplicadores o unidades aritmético-lógicas y en algunos casos procesadores. Un ejemplo de estas FPGAs es la familia Xilinx Virtex-5 (mostrada en la Figura 2), con la que hemos realizado las pruebas. A continuación la explicaremos en mayor detalle.

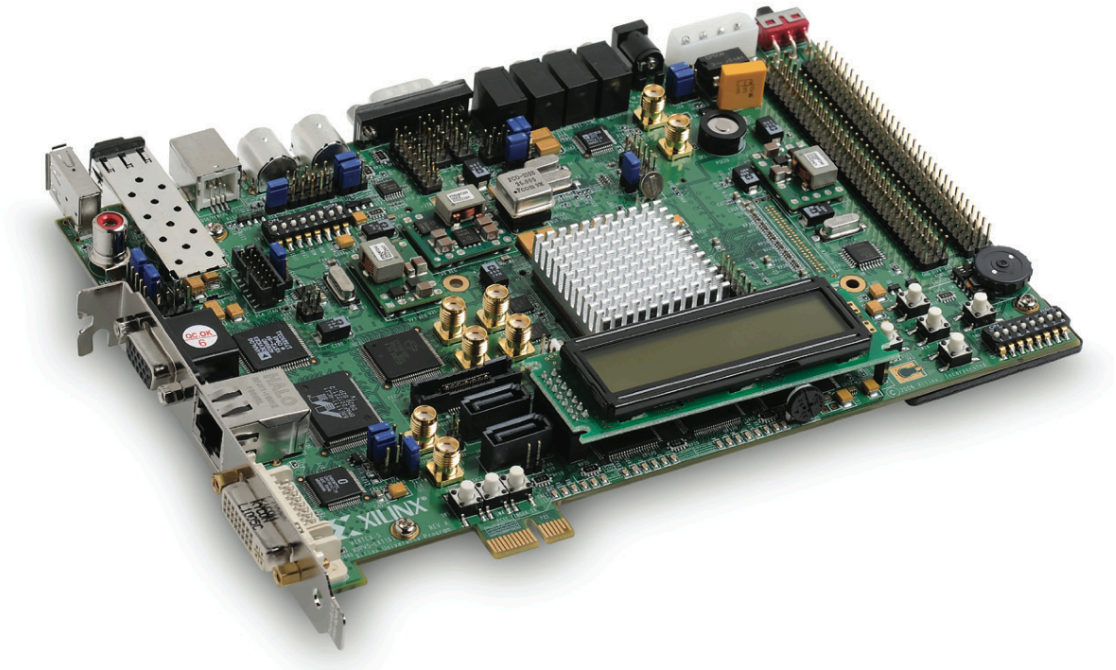

Figura 2. FPGA Virtex-5 XC5VLX110T

La arquitectura básica de una FPGA Virtex-5 consiste en una matriz de interconexionado que comunica los módulos CLBs, IOBs (*Input/Ouput Blocks*), CMTs (*Clock Manager Tiles*), BRAM (*Block RAMs*) y DSPs. A continuación se explican cada uno de estos módulos en mayor detalle.

- **CLBs**: se componen de dos elementos denominados *slices* y cada uno cuenta con cuatro LUTs de seis salidas cada uno. Estos *slices* se comunican con los demás bloques a partir de una matriz de interconexionado, tal y como se muestra en la Figura 3 y en la Figura 4. En la tabla siguiente se puede visualizar la composición de un CLB y posteriormente, se explicarán en detalle algunos de sus componentes.

Slices	LUTs	Flip-Flops	Clocks, Clock-Enables, Reset	RAM distribuida
2	8	8	2 de cada	256 bits

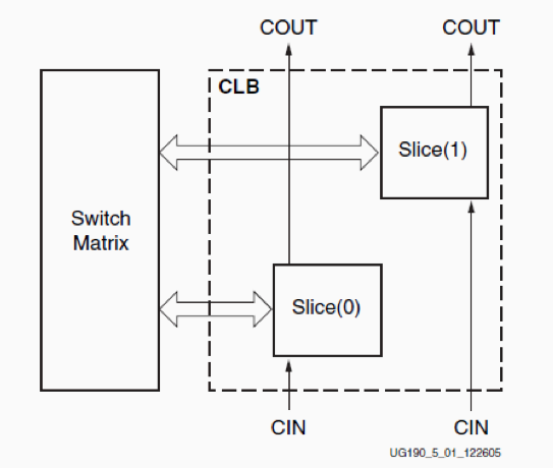

Figura 3. Disposición de los *slices* en el CLB

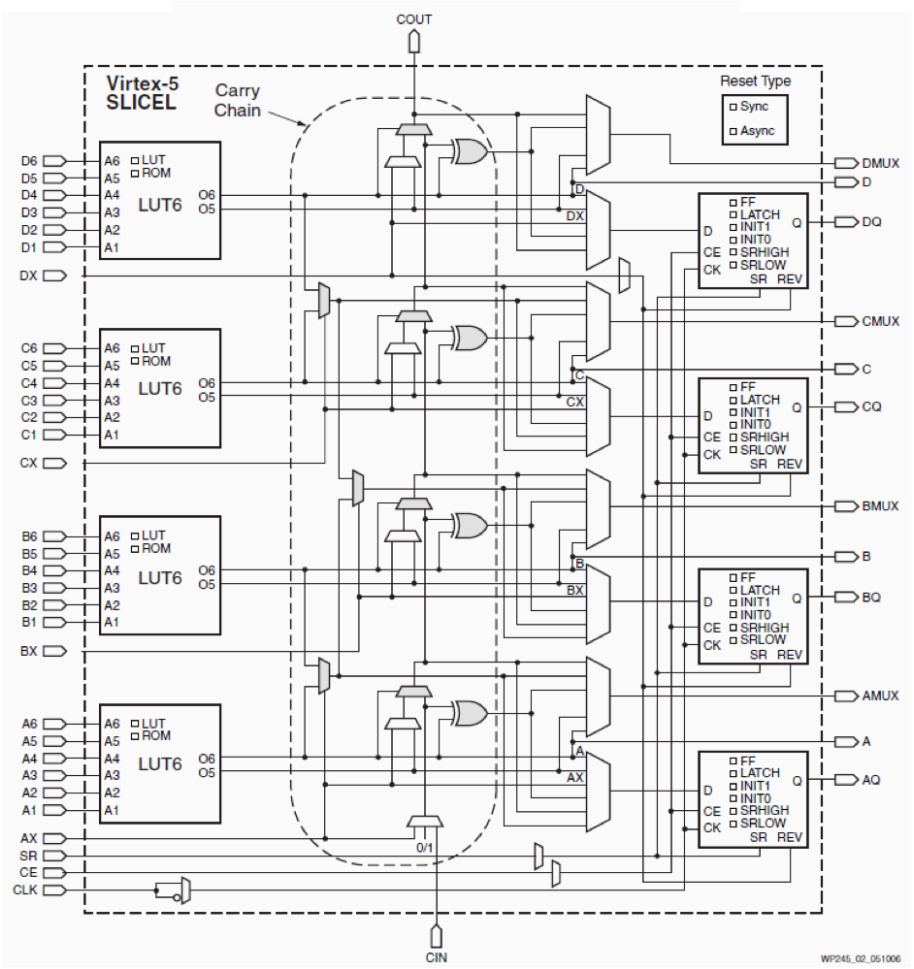

Figura 4. Estructura interna de un slice.

En la familia de las Virtex-5, las LUTs han sido extendidas de 4 entradas (las cuales requerían implementar muchos niveles de puertas lógicas, debido a sus limitaciones, para implementar un circuito complejo), a 6 entradas aumentando así su capacidad. Éstas pueden ser utilizadas como funciones booleanas de 6 entradas y 1 salida o como 2 funciones booleanas de 5 entradas y 1 salida, tal y como se muestra en la Figura 5.

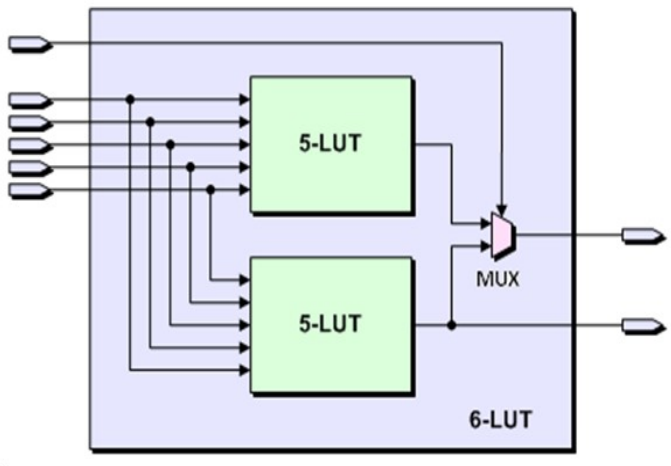

Figura 5. Estructura de una LUT de 6 entradas.

- **IOBs**: son módulos que comunican la FPGA con el exterior y suelen rodear a la matriz de bloques lógicos. Estos módulos, como los CLBs o DSPs, se pueden programar para que se comporten como puertos de salida, entrada o entrada-salida.

- **CMTs**: en la familia de Virtex-5 los módulos DCMs (*Digital Clock Managers*) han sufrido una evolución a los llamados CMTs, los cuales están compuestos por 2 DCMs, módulos que manipulan la señal de reloj, pudiendo multiplicar o dividir su frecuencia; y un PLL, que es un módulo analógico realimentado cuyo objetivo es perfeccionar la señal de reloj. Esto se muestra en detalle en la Figura 6.

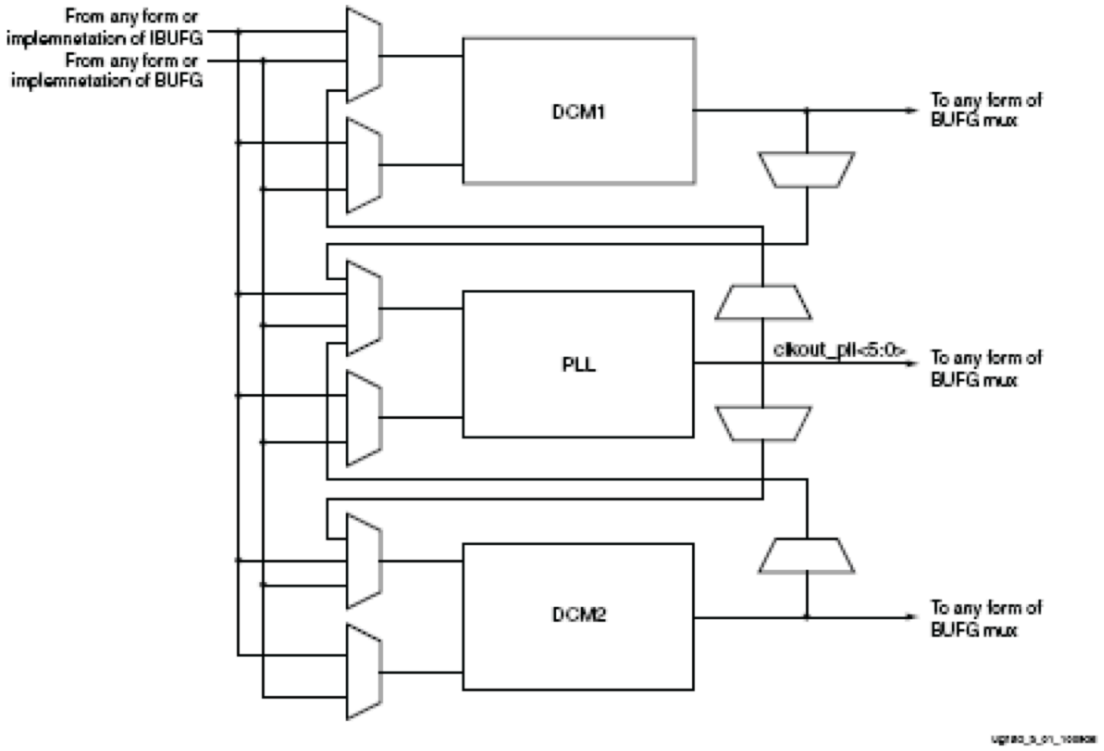

Figura 6. Estructura de un CTM.

- **BRAMs**: bloques de memoria RAM empotrados, de 18 o 36Kb cada uno, situados en columnas a lo largo de la FPGA.

- **DSPs**: en la familia Xilinx Virtex-5, desaparecen los multiplicadores y éstos son integrados en los módulos DSP (llamados “DSP48E” en este caso). Se utilizan para realizar operaciones aritmético lógicas y están compuestos por un multiplicador de 25x18 bits, un sumador de 48 bits y un acumulador, tal y como se muestra en la Figura 7.

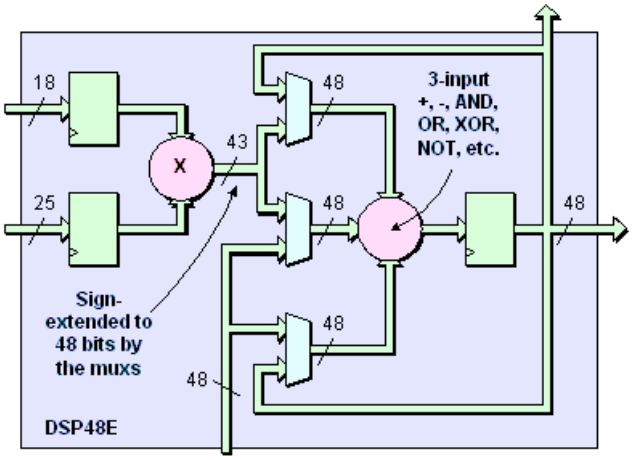

Figura 7. Estructura de un DSP.

1.2. Trabajo relacionado

Muchos grupos de investigación han desarrollado técnicas que buscan reducir las penalizaciones temporales y el consumo energético asociado a las cargas de las tareas en el hardware dinámicamente reconfigurable [12]. Entre éstas se pueden subrayar el desarrollo de nuevas arquitecturas [13], compresión de configuraciones [14] y planificación de tareas [15-17]. Esta última tiene en cuenta el hecho de que, en sistemas empotrados, las aplicaciones se suelen representar como grafos dirigidos acíclicos (o DAGs, por sus siglas en inglés). Este tipo de estrategias han sido estudiadas con mucho detalle en la literatura, y de hecho se pueden encontrar técnicas estáticas [15], dinámicas [16] o híbridas [17]. Estas últimas son una combinación entre estáticas y dinámicas y presentan las ventajas de ambas.

Dos artículos de investigación recientes [18, 19] han analizado el impacto de cargar las reconfiguraciones en tiempo de ejecución desde una memoria externa. Los resultados experimentales publicados en [18] permiten concluir que, si se utiliza la memoria *flash* para almacenar las configuraciones, la velocidad de reconfiguración alcanzada es tres órdenes de magnitud más lenta que la máxima velocidad de reconfiguración posible. En [19], se analiza el impacto de las reconfiguraciones en las prestaciones del computador reconfigurable de altas prestaciones Cray XD1, el cual incluye una o varias FPGAs y un sistema multiprocesador convencional. Según sus mediciones, cargar las configuraciones desde una memoria externa ralentiza en hasta tres órdenes de magnitud la velocidad pico teórica. En estos dos artículos, la razón que explica los retardos de reconfiguración adicionales es el acceso a las memorias *off-chip*. Por tanto, incluir un nivel intermedio de memorias *on-chip* parece una buena idea para atacar este problema.

De hecho, esta técnica, comúnmente conocida como “cacheo de reconfiguraciones” en la literatura, ha sido estudiada en la literatura desde hace ya algún tiempo [6, 7]. El trabajo publicado en [6] está entre los primeros en explorar esta idea. Propone un conjunto de algoritmos adaptados para dispositivos reconfigurables de un sólo contexto, multi-contexto y parcialmente reconfigurables. Sus resultados muestran que se puede obtener un factor de reducción de penalizaciones temporales

de hasta 5 con respecto a metodologías pensadas únicamente para sistemas con memorias *off-chip*. El trabajo presentado en [7] extiende estas ideas para tareas de tamaño variable, y presenta dos técnicas llamadas *Penalty-based* y *History-based*.

Otros algoritmos interesantes se pueden encontrar en [8, 9, 20]. Por un lado, [8, 9] proponen técnicas híbridas que combinan cacheo de reconfiguraciones y de datos para tareas en FPGAs. Por otro lado, [20] presenta un sistema reconfigurable heterogéneo que incluye varios procesadores reconfigurables. También propone incluir una memoria cache de configuraciones para cada procesador, así como una técnica de prebúsqueda de configuraciones.

Aparte del tiempo de acceso a memoria, muchos investigadores han recalcado que, para sistemas empotrados, el consumo de energía debido a la jerarquía de memoria supone un porcentaje muy importante (alrededor del 30%) del consumo de energía global del sistema [21, 22]. Y esto es cierto tanto para arquitecturas de grano fino, como de grano grueso. En este sentido, los trabajos presentados en [10, 11] han propuesto también extender las técnicas clásicas de cacheo de reconfiguraciones para tener en cuenta este punto, además de las penalizaciones temporales.

De este modo, [10] presenta una jerarquía de memoria compuesta por dos bloques de memoria *on-chip* con diferentes compromisos entre prestaciones y consumo de energía, tal y como se muestra en la Figura 8. Este esquema permite acelerar las reconfiguraciones que sean más críticas para las prestaciones del sistema, y al mismo tiempo, permite reducir el consumo de energía generado por las reconfiguraciones. El trabajo presentado en [11] desarrolla estas ideas en profundidad, y presenta dos algoritmos de mapeo de tareas que se adaptan muy bien a diferentes contextos de ejecución de aplicaciones.

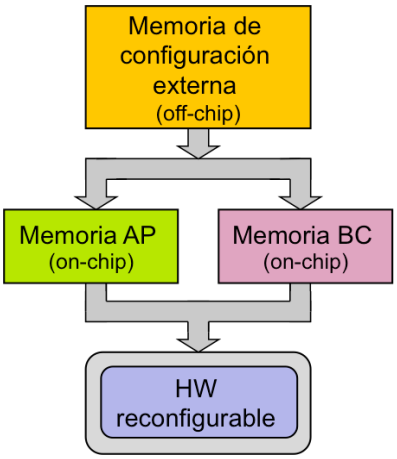

Figura 8. Jerarquía de memoria presentada en [10] y compuesta por dos bloques de memoria *on-chip*, uno de altas prestaciones (AP) y otro de bajo consumo (BC).

Sin embargo, todas estas metodologías de cacheo de reconfiguraciones tienen una limitación común: **asumen que las configuraciones de las tareas son indivisibles**. Esto limita su aplicabilidad en sistemas empotrados modernos, donde las memorias *on-chip* normalmente tienen unos tamaños muy reducidos. Además, no

proporcionan ningún soporte hardware para gestionar el cacheo de las reconfiguraciones de manera eficiente. El módulo hardware desarrollado en este proyecto asume que las configuraciones están divididas en uno o varios bloques, lo cual aumenta la flexibilidad del sistema y permite mejorar sus prestaciones y reducir su consumo de energía debido a las reconfiguraciones. Esto está motivado debidamente en el siguiente apartado.

1.3. Ejemplo de motivación

La Figura 9 ilustra los beneficios potenciales de la utilización de la jerarquía de memoria descrita en la Figura 8, en combinación con la idea de dividir configuraciones en bloques introducida en la sección anterior. En este caso, se asume que el sistema tiene 3 unidades reconfigurables (URs), cada una de las cuales puede ejecutar hasta una tarea simultáneamente en el hardware reconfigurable. También se supone que el tamaño y la forma de estas unidades, así como la planificación de las tareas en este sistema ya han sido determinadas de antemano. Por ejemplo, esta planificación no reemplaza la tarea 1 a la hora de cargar las tareas 2, 3, 4 o 5. De esta forma, cuando la aplicación se ejecuta varias veces consecutivas, la tarea 1 será cargada sólo la primera vez, y será reutilizada en las sucesivas ejecuciones. Por tanto, la reconfiguración R1 sólo generará penalizaciones en tiempo y en energía la primera vez que esta aplicación se ejecute. Sin embargo, cómo todas estas decisiones se toman están fuera del alcance de este proyecto.

En este ejemplo, se asume que el sistema utiliza una jerarquía de memoria de configuraciones que está compuesta de una memoria *off-chip* (llamada “EXT” en el resto del presente documento) y dos memorias *on-chip*: una con altas prestaciones (AP) y la otra con bajo consumo (BC), que tiene menor consumo energético que AP, pero mayor tiempo de acceso. Asumamos que tanto AP como BC tienen cada una capacidad para guardar hasta 2 configuraciones completas.

Las características concretas de estas memorias también están incluidas en la tabla de la Figura 9. Estos datos son realistas, y han sido obtenidos con la herramienta CACTI 4.0 [24]. No hemos querido utilizar los resultados de este simulador como valores absolutos, ya que las características reales de cada memoria varían considerablemente dependiendo de la tecnología que se utiliza. En lugar de eso, hemos utilizado esta información para obtener la relación en el tiempo y la energía consumida por acceso a memoria para cada una de las diferentes memorias. Por este motivo, hemos utilizado valores normalizados para el consumo energético. En este caso, hemos seleccionado una memoria externa de 1 MB, y memorias AP y BC de 64 kB, ya que, según nuestras mediciones, estos tamaños eran los que ofrecían el mejor compromiso entre capacidad de memoria, consumo de energía y tiempo de acceso a memoria para cada configuración. En este caso, la memoria AP es un 50% más rápida, pero consume un 30% más de energía por acceso que la BC. Además, AP es 3 veces más rápida y 4 veces más eficiente energéticamente que la memoria externa.

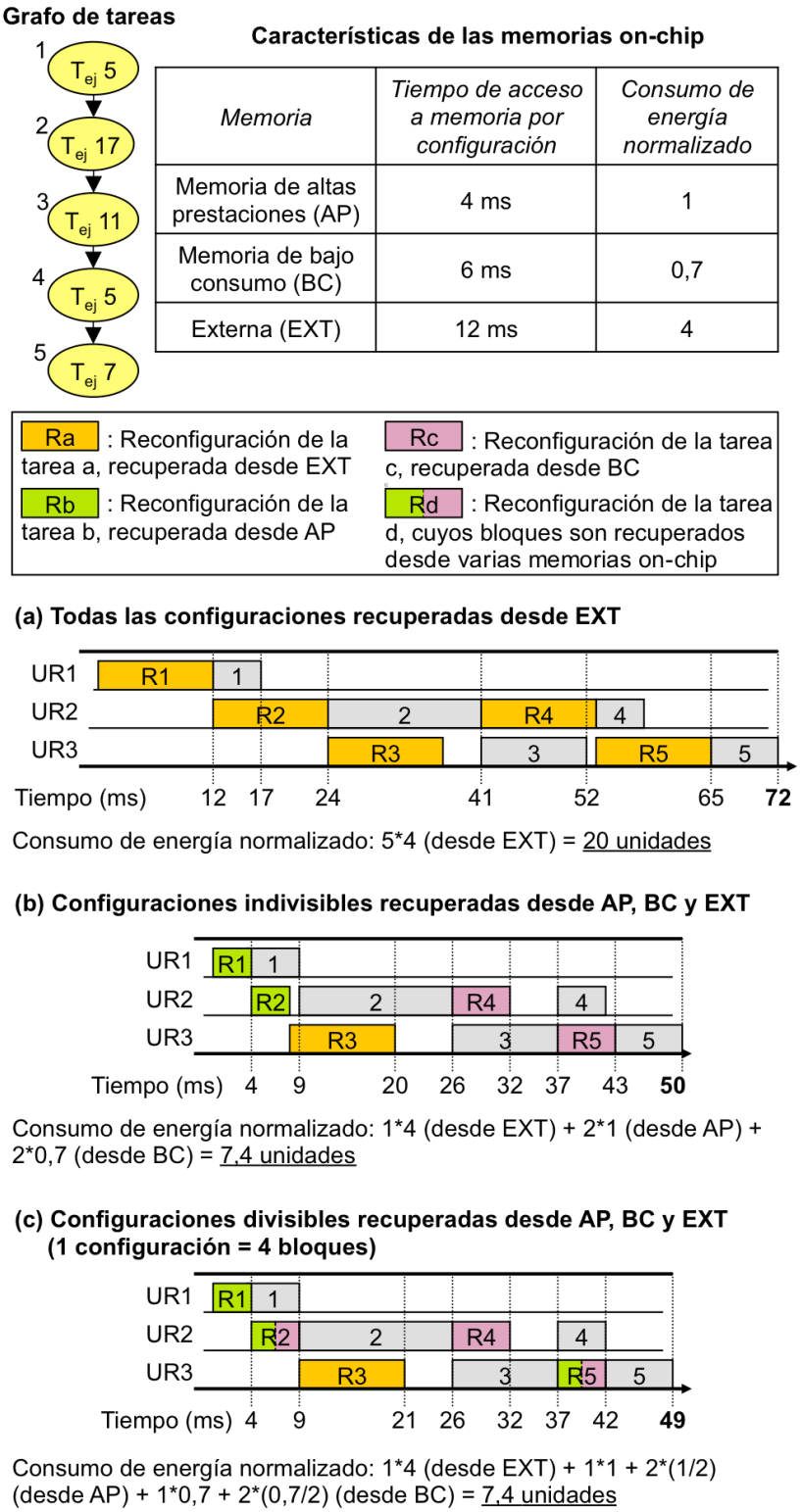

Figura 9. Ejemplo de motivación que muestra los beneficios de un almacenamiento de configuraciones en cache basada en bloques.

En primer lugar, en la Figura 9.a todas las configuraciones se asignan (y en consecuencia, son accedidas desde) la memoria externa. Esto implica un tiempo de ejecución de 72 milisegundos y un consumo de energía normalizado de 20 unidades energéticas (5 lecturas de EXT). Tanto estas prestaciones como este consumo

energético pueden mejorarse en gran medida utilizando la jerarquía de configuraciones descrita anteriormente, ya que R1 y R2 pueden asignarse a AP; R4 y R5, a BC, y R3, a EXT (Figura 9.b). Esta asignación permite reducir el tiempo de ejecución total a 50 milisegundos, y el consumo energético a 7.4 unidades.

Sin embargo, si las configuraciones se particionan en varios bloques, las prestaciones del sistema se pueden mejorar aún más. En el ejemplo de la Figura 9.c las configuraciones se dividen en 4 bloques. Esto permite flexibilizar la asignación de R2, asignando 2 bloques a AP y 2 bloques a BC. Esto no sólo reduce el consumo energético debido a R2 (de 1 a 0.75 unidades), sino que también deja espacio disponible en la memoria AP para repetir la misma operación con R5. De este modo, el tiempo de ejecución total se reduce a 49 milisegundos, manteniendo además el mismo consumo energético debido a las reconfiguraciones.

Por último, es importante destacar que, por simplicidad, en este ejemplo se asume que todas las reconfiguraciones que se solicitan ya existían en las memorias *on-chip*. Por tanto, este ejemplo no muestra una ejecución anterior en la cual todas las tareas se cargan primeramente desde EXT y son escritas en las memorias *on-chip* en donde son asignadas.

1.4. Paquetes software utilizados

Para el desarrollo de la implementación en VHDL del módulo hardware propuesto y su posterior testeo se han empleado los siguientes programas: Xilinx ISE 14.6 e ISim 14.6 respectivamente. Ambas son herramientas muy utilizadas en el mundo del diseño hardware y que facilitan mucho las tareas de crear, verificar, simular y sintetizar diseños basados en FPGAs o CPLDs.

1.4.1. Xilinx ISE 14.6

El entorno de desarrollo ISE de Xilinx posee un aspecto similar al de los entornos de programación utilizados en la actualidad, es decir, posee diversas ventanas para la visualización de tareas específicas sobre cada una de ellas. En concreto, en dicho software nos encontramos con cuatro ventanas, las cuales las podemos apreciar en la Figura 10.

1. **Ventana de ficheros fuente** → Muestra 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 Proceso** → 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. **Ventana 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** → Muestra mensajes de error, avisos o información emitidos por la ejecución de los programas de compilación, implementación, etc. Es muy importante tener presente dicha ventana a la hora del diseño hardware, ya puede avisar de que el hardware que estamos diseñando no es sintetizable.

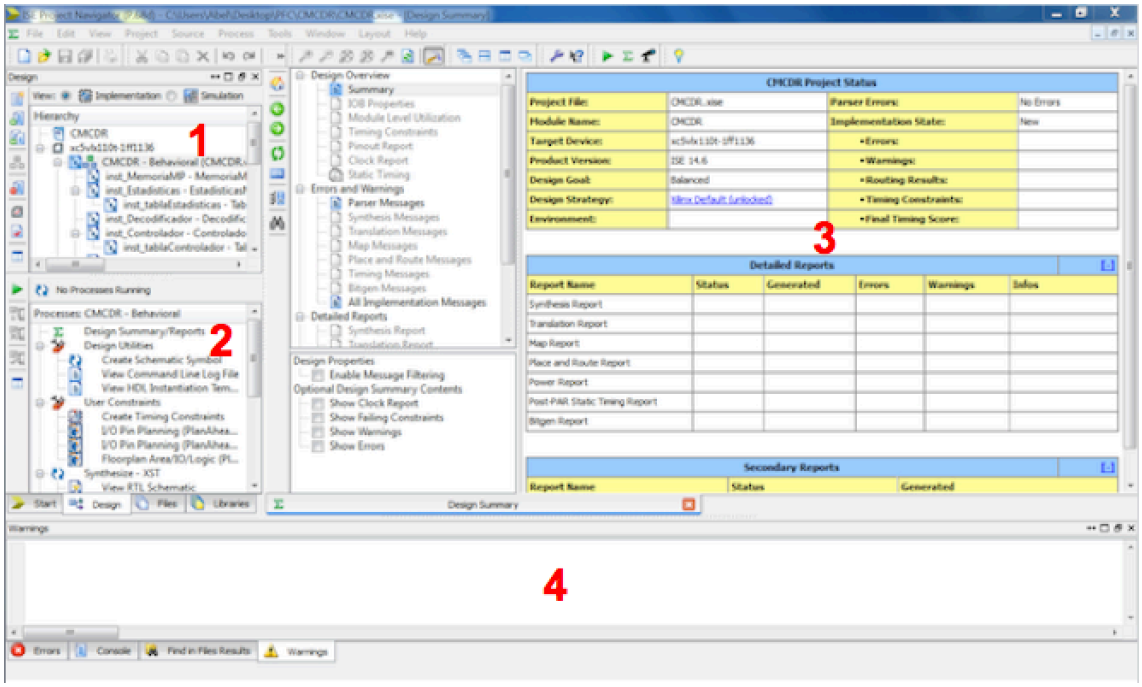

Figura 10. Ventana principal de la herramienta Xilinx ISE, en concreto la versión 14.6.

Una vez presentada la herramienta, vamos a describir los pasos que realiza ésta en el diseño e implementación. Veremos cómo, a partir del código VHDL o bien de un diseño esquemático creados por el usuario, a través de una serie de pasos, genera un fichero de extensión .bit con toda la información necesaria del hardware diseñado para que la FPGA pueda ser programada. Todo esto que acabamos de explicar se puede observar de manera mucho más sencilla a partir de diagrama mostrado en la Figura 11.

El primer paso que realiza la herramienta a la hora de interpretar el diseño hardware realizado por el usuario es la función de síntesis. Para ello, transforma el código en una *netlist* descrita en un lenguaje intermedio para un hardware concreto. En este fichero se definen restricciones temporales, asignación de pines y optimizaciones de cada uno de los componentes a partir de la descripción de comportamiento del sistema. Para realizar este proceso se debe seleccionar la opción *Synthesize-XST* (Figura 12). Una vez finalizado, ISE muestra un informe con toda la información de la síntesis.

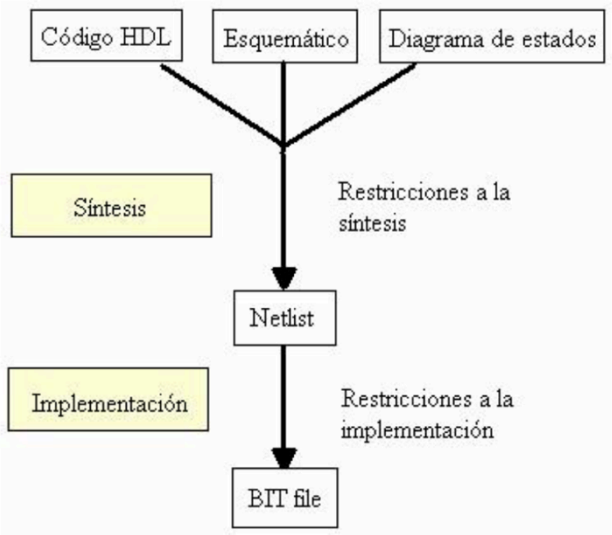

Figura 11. Diagrama de flujo de la herramienta Xilinx ISE.

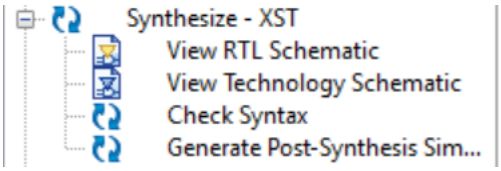

Figura 12. Ventana proceso *síntesis* de la herramienta Xilinx ISE 14.6.

Una vez finalizado el proceso de síntesis, pasaremos al segundo paso. Este proceso se denomina implementación y está dividido en tres fases: *translate*, *map* y *place&route*. Lo que se hace en este proceso es enlazar los módulos con las condiciones indicadas en el fichero de restricciones (.ucf) y reducirla a primitivas de Xilinx. A continuación, se adapta al dispositivo en el cual se va a volcar el diseño y se colocan y se rutan los componentes físicos finales. Para realizar este proceso se selecciona *Implement Design*. Cada una de las fases genera un informe que es importante tener presente por si la herramienta no está diseñando el hardware con las especificaciones deseadas.

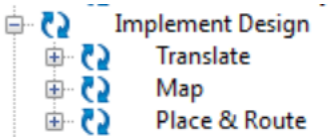

Figura 13. Ventana proceso *implementación* de la herramienta Xilinx ISE 14.6.

En el fichero de restricciones se especifican los pines a los que se debe conectar cada señal de entrada y salida y su tecnología. Se pueden añadir restricciones temporales, como retardo máximo de una señal o frecuencia del reloj, y colocación de módulos en la FPGA.

El último paso es la generación del *bitstream* (fichero .bit) que será lo que se cargue en la FPGA. Este fichero contiene toda la información de localización de los dispositivos y los elementos de rutado. Para este proceso, se selecciona la opción *Generate Programming File*. Una vez finalizado este paso, ya tenemos nuestro diseño hardware listo para probarlo en la FPGA.

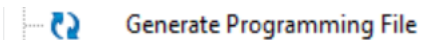

Figura 14. Ventana generación archivo .bit de la herramienta Xilinx ISE 14.6.

En el proyecto, este software se ha utilizado para la implementación en VHDL del hardware propuesto y para el testeo de los módulos que componen dicho hardware, con el propio simulador que él mismo incluye.

1.4.2. ISim 14.6

Los bancos de prueba (o *testbenches*) son una parte esencial en el proceso de diseño hardware, ya que permiten comprobar su correcto funcionamiento y ayudan en la automatización del proceso de verificación del diseño. Es importante ir realizando el testeo de los módulos por separado y una vez se esté seguro que el módulo funciona correctamente continuar con el proceso de diseño. Por otro lado, recoger estadísticas de la cobertura del código durante la simulación ayuda a asegurar la calidad y la minuciosidad de las pruebas.

El software ISim 14.6, que incorpora la herramienta de diseño comentada anteriormente, proporciona un entorno integrado de depuración (*Integrated Debugging Environment*) que facilita el depurado eficiente de diseños basados en FPGAs, programados en cualquiera de los tres lenguajes siguientes: VHDL, Verilog y SystemC.

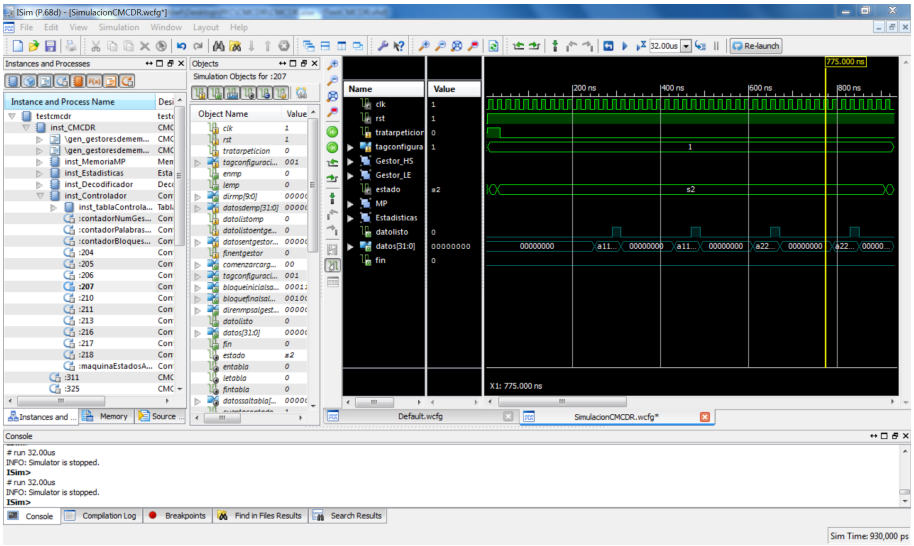

Figura 15. Ventana principal del simulador ISim 14.6.

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

En la Figura 15 se visualiza un ejemplo concreto de simulación, se trata del controlador diseñado. En concreto, se observan las dos instancias de los gestores (el de altas prestaciones y el de bajo consumo), la memoria externa, el módulo de reemplazo, el de estadísticas, etc. Para cada uno de ellos están agrupadas las señales más importantes que durante el desarrollo del proyecto han servido de indicadores para comprobar si los módulos se comportaban correctamente.

Por último comentar que, aunque esa es la simulación del hardware desarrollado completo, como se ha adelantado antes, es importante testear los módulos de forma independiente para así solucionar los posibles errores de manera mucho más sencilla. Esta es la metodología de depuración que hemos seguido a lo largo de todo el desarrollo de este proyecto.

1.5. Contribuciones del proyecto

La principal contribución del proyecto ha sido el desarrollo de un módulo hardware que realiza de forma transparente y eficiente la gestión de las reconfiguraciones en un sistema que aplica la técnica de cacheo de reconfiguraciones [6-9].

Existen técnicas que buscan reducir las penalizaciones temporales y el consumo energético asociado a las cargas de las tareas en el hardware dinámicamente reconfigurable, pero todas estas asumen que las reconfiguraciones son indivisibles. El módulo hardware propuesto soporta la división de las configuraciones en bloques, mejorando así la flexibilidad del sistema. La principal ventaja de este esquema de particionamiento de configuraciones en bloques consiste en poder obtener mapeos que ofrecen mejores prestaciones y un menor consumo energético, tal y como hemos podido constatar con el ejemplo de motivación.

El controlador propuesto tiene un consumo de recursos muy asequible y asimismo, introduce unas penalizaciones en tiempo de ejecución insignificantes respecto al tiempo ideal de ejecución de las tareas. Esto se comentará en detalle en el capítulo 4.

Por último, comentar que el trabajo de este proyecto de Sistemas Informáticos ha dado lugar a dos artículos, uno en español y otro en inglés, los cuales han sido enviados al congreso nacional Jornadas de Computación Reconfigurable y Aplicaciones (JCRA), y a la prestigiosa revista científica *IEEE Transactions on Very Large Scale Integration Systems* (IEEE TVLSI), respectivamente. En el momento de redactar esta memoria, ambos artículos se encuentran en proceso de revisión.

2. Modelo de arquitectura supuesto

El controlador hardware que hemos desarrollado ha sido diseñado para trabajar con el modelo de arquitectura objetivo descrito en la Figura 16, el cual es una extensión del modelo de jerarquía de memoria comentado en la Figura 8 del capítulo anterior. Según este modelo, existen un conjunto de memorias *on-chip* situadas entre la memoria de configuración (*off-chip*) y el hardware reconfigurable. Estas memorias son conocidas habitualmente como caches, pero normalmente no son realmente caches, sino memorias de tipo SRAM controladas por software (es decir, *scratchpads*). Si se utilizan adecuadamente, pueden mejorar drásticamente las prestaciones de la jerarquía de memoria, así como su consumo de energía. El motivo es que evitan que el sistema acceda a buses externos al hardware reconfigurable, los cuales tienen una gran capacitancia [23]. Estas memorias *on-chip* se caracterizan por tener unos consumos de energía y tiempos de acceso mucho menores que las memorias *off-chip*.

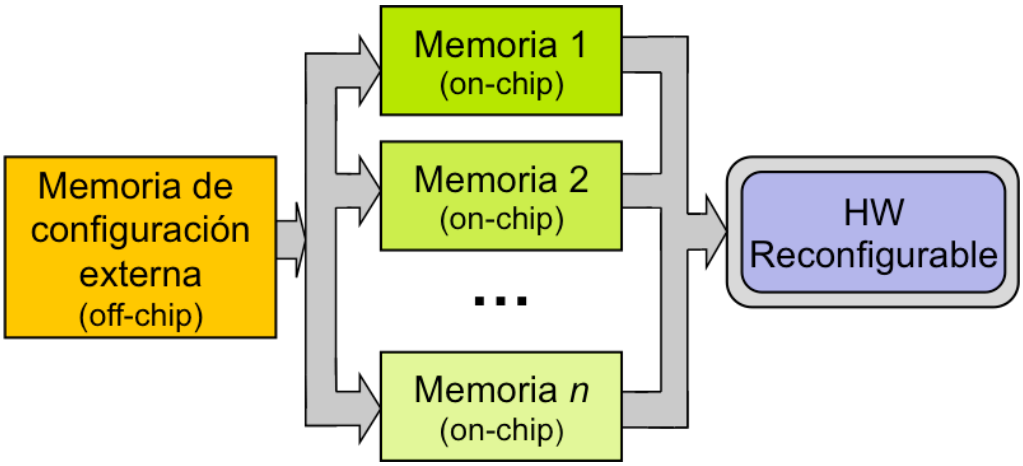

Figura 16. La jerarquía de la memoria de configuración supuesta. El HW reconfigurable también está conectado a la memoria *off-chip* por medio de una conexión dedicada, que no se muestra en la figura por simplicidad

Además, cada una de ellas puede tener diferentes propiedades. Por ejemplo, en el caso de dispositivos reconfigurables, estas memorias pueden estar construidas componiendo una o varias BRAMs. Si no hay suficientes BRAMs disponibles, otra posibilidad es utilizar la RAM que se encuentra distribuida entre todos los *slices* de la FPGA. En este caso, una combinación de ambos tipos de memoria constituye una jerarquía de memoria heterogénea, cuyos diferentes tipos de memorias trabajan a diferentes velocidades y tienen diferentes consumos de energía.

En esta arquitectura, se asume que los diferentes bloques en los que las configuraciones están divididas se asignan a las diferentes memorias existentes en el sistema. Estas asignaciones (llamadas “mapeos” en el resto de la memoria) se asignan dependiendo de las restricciones de las tareas. Lo más habitual es que se asignen en tiempo de diseño; es decir, antes de la ejecución de las aplicaciones. Sin embargo, también pueden actualizarse en tiempo de ejecución, en función de la carga de trabajo del sistema y de la presión que se ejerce sobre las memorias *on-chip*. Sea cual sea la manera en la que estos mapeos se realice, el controlador que hemos desarrollado es compatible con cualquiera de estas metodologías. De hecho, como acabamos de comentar, hemos incorporado la posibilidad de realizar el mapeo en tiempo de ejecución, por lo que nos pareció muy interesante incorporar también un módulo de estadísticas. De esta manera, podemos consultar en tiempo de ejecución el número de aciertos y fallos que ha sufrido una determinada tarea, pudiendo recalcular el mapeo sobre dicha tarea en el caso de que el número de fallos fuese muy elevado o si se considerase oportuno. Esta medida ofrece la posibilidad de mejorar la eficiencia en tiempo de ejecución.

Cuando el sistema arranca por primera vez, se asume que, inicialmente, todos los bloques de las reconfiguraciones están almacenados en la memoria *off-chip*. Sin embargo, cuando el sistema demanda la reconfiguración de una tarea en el hardware reconfigurable, el controlador que hemos desarrollado busca esta reconfiguración en la memoria *off-chip*, guardando una copia en la memoria *on-chip* correspondiente, de acuerdo a lo que el mapeo de esa tarea especifique. Si la memoria *on-chip* en cuestión está llena, el controlador también debe decidir cuál de los bloques que ya están escritos en esa memoria debe ser reemplazado para dejar sitio al nuevo. Esto se explica en mayor detalle en los capítulos siguientes.

3. El controlador de memoria caché de reconfiguraciones desarrollado

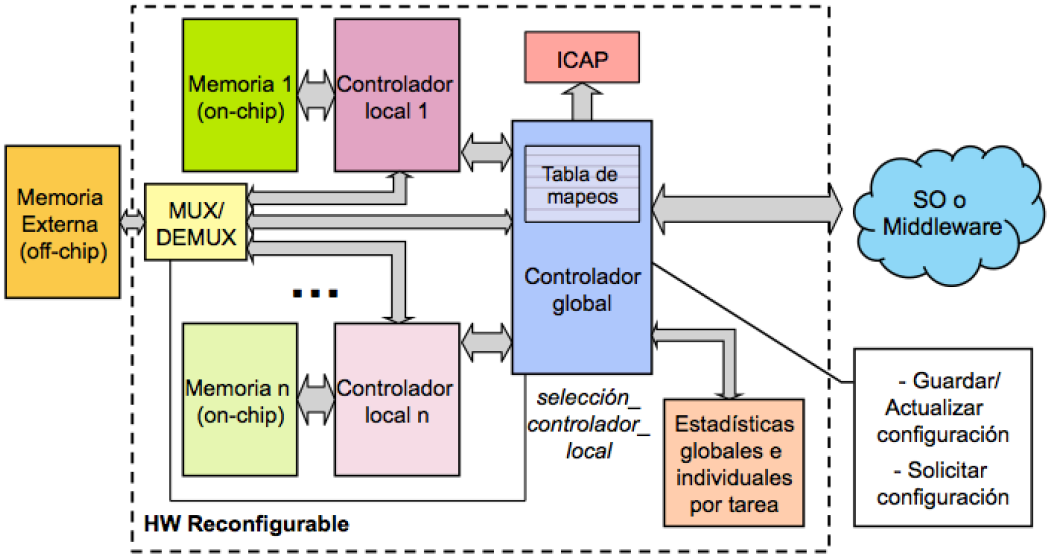

Figura 17. Esquema general del controlador hardware propuesto.

El esquema general del controlador hardware propuesto está descrito en la Figura 17. Tal y como se puede apreciar, contiene varias memorias *on-chip*, cada una de ellas gestionadas directamente y de manera exclusiva por una instancia de un *Controlador Local*. El *Controlador global* gestiona el funcionamiento de los controladores locales y también se comunica con el sistema operativo o con algún *middleware*. Todos estos módulos tienen acceso a la memoria *off-chip*, que, en este caso, es una memoria de tipo DDR2 de 256 MB de tamaño que está disponible en la placa de prototipado XUPV5-LX110T utilizada para implementar el diseño. Por ello, sus interfaces de comunicación están conectadas al módulo *MUX/DEMUX*, el cual tiene un conjunto de multiplexores (para las señales de entrada a la memoria) y demultiplexores (para las señales de salida de la memoria). Tanto los multiplexores como los demultiplexores son controlados a través de la señal *seleccion_controlador_local*. Finalmente, el módulo *Estadísticas* se encarga de almacenar el número de aciertos y fallos globales en una ejecución y también el número de aciertos y fallos que ha sufrido cada una de las configuraciones solicitadas por el *controlador global*. Estas estadísticas se pueden consultar en cualquier momento de la ejecución, ya que el módulo ha sido diseñado para que atienda peticiones aún cuando uno de los controladores locales está trabajando.

A continuación (apartados 3.1, 3.2 y 3.3) se explicarán con mayor detalle el controlador local, el controlador global y el módulo de estadísticas respectivamente, incidiendo especialmente en su arquitectura hardware y su funcionalidad.

3.1. Detalles a bajo nivel del controlador local

El controlador local es el encargado de atender las peticiones solicitadas por el controlador global. Éste siempre solicitará un determinado número de bloques de una tarea en cuestión. El controlador local deberá recuperar y devolver la información relativa a esos bloques y avisar al controlador global cuándo este proceso ha finalizado. Para ello, necesita de los siguientes módulos que vamos a comentar a continuación.

3.1.1. Tabla asociativa

La Figura 18 describe en detalle cómo están construidos los controladores locales que hemos comentado anteriormente. En primer lugar, y tal y como se puede observar, contienen una *Tabla asociativa* que guarda la información referente a aquéllos bloques que hay guardados en cada momento en la memoria *on-chip* asociada a este controlador.

Cada entrada de esta tabla contiene información acerca de los bloques de cada una de las tareas que se encuentran en la memoria asociada a dicho controlador local. Esta información se actualiza en tiempo de ejecución a medida que entran y salen bloques de dichas memorias.

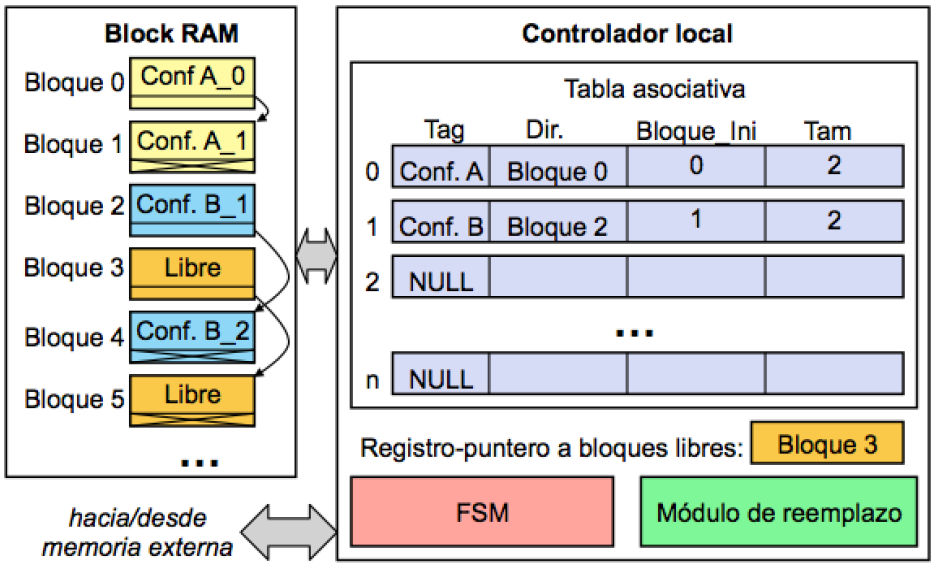

Figura 18. Esquema general del módulo controlador local.

La información que almacena la tabla se puede dividir en 4 campos:

- *Tag*: Indica de manera unívoca el nombre de la configuración en cuestión.
- *Dir*: Contiene la dirección inicial de la memoria *on-chip* donde está almacenado el primer bloque de dicha configuración. A partir de ella nos podemos posicionar en cualquiera de los bloques que esa configuración tenga alojados en memoria *on-chip* rápidamente gracias al valor del campo *Tam*.
- *Tam*: Indica el número de bloques almacenados de dicha configuración en memoria *on-chip*.
- *Bloque_Ini*: Indica el bloque inicial que tenemos almacenado de dicha configuración.

Como podemos observar, además de la dirección del primer bloque de la configuración, sólo almacenamos el bloque inicial de la configuración y el número de bloques almacenados, sin dejar constancia del resto de bloques almacenados a partir del inicial. Esta decisión se ha tomado bajo la suposición de que los bloques de una configuración que se van a solicitar siempre van a ser contiguos. Todo el sistema trabaja en base a esto para mantener la coherencia.

Las memorias *on-chip* se pueden implementar utilizando RAM distribuida en la FPGA, o instanciando BRAMs. En este caso, se han utilizado este último tipo de bloques de memoria, a través de la primitiva *RAMB16_S36* [24]. Esta primitiva permite instanciar bloques de memoria de 512 entradas de 32 bits, es decir, 2 KB de capacidad. Este tamaño ha sido suficiente para ejecutar los experimentos explicados en el capítulo 4 del proyecto. Sin embargo, si se necesitasen bloques de memoria más grandes, este esquema se puede extender instanciando varias primitivas BRAM y añadiendo un descodificador y un multiplexor para acceder a la BRAM correcta en función de la dirección de entrada.

También es importante destacar que el controlador asocia a cada memoria *on-chip* su tiempo de acceso a memoria (en ciclos de reloj). Por ejemplo, un valor de "1" indica que se puede leer una nueva palabra cada ciclo de reloj, de esta forma podemos simular memorias de altas prestaciones y de bajo consumo, variando simplemente un parámetro genérico que se corresponde al número de ciclos con los que trabaja la memoria.

En la memoria *on-chip*, los bloques que forman parte de la misma configuración están enlazados con punteros. El puntero asociado al último bloque en la secuencia tiene valor *null* (por ejemplo, los punteros de los bloques 1, 4 y 5 en la Figura 18). Esta metodología permite al controlador local encontrar fácilmente la información que está guardada en la memoria, ya que en la tabla tenemos guardada la dirección del bloque inicial de la tarea y para posicionarnos en otro bloque sólo se tiene que seguir los punteros "siguiente" tantas veces como sea necesario hasta posicionarse en el bloque deseado. El tamaño de la tabla asociativa es configurable a través de un parámetro genérico en su fichero fuente VHDL.

Por último, vamos a explicar cómo se comportan ambos módulos conjuntamente, la tabla asociativa y la memoria *on-chip*, cuando llega una solicitud al controlador local. Como hemos comentado anteriormente, el objetivo de éste consiste en devolver los datos referentes a la tarea solicitada por el controlador global. La petición se puede dividir en dos partes, por una lado se indica la tarea y por otro lado se especifica número de bloques que se solicitan, indicando el bloque inicial y final que se requiere de dicha tarea. Con esta información, el controlador local devuelve la información referente a cada uno de los bloques al controlador global.

Una vez hemos comprendido cuál es el funcionamiento a grandes rasgos del controlador local, vamos a explicar más en detalle el comportamiento del mismo desde el momento que recibe una petición del controlador principal hasta que realiza el procesado de los bloques. La Figura 19 muestra a través de un diagrama de flujo el comportamiento del controlador local.

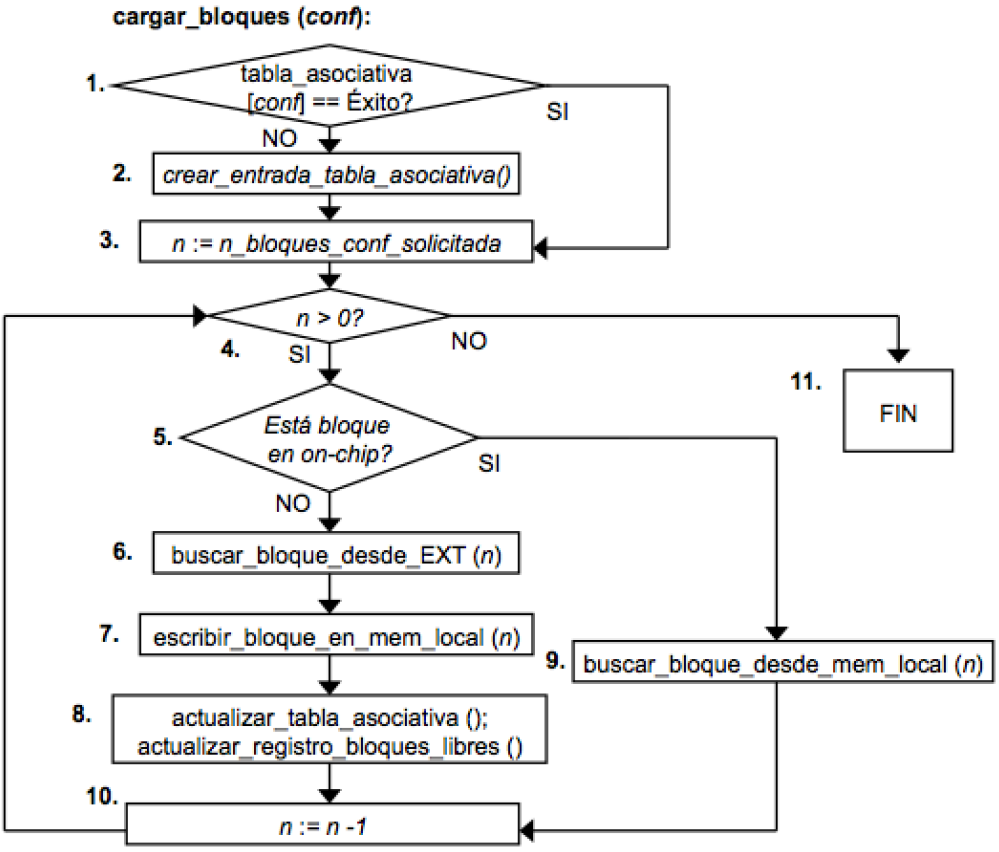

Figura 19. Diagrama de flujo que describe el funcionamiento del controlador local

Lo primero que se comprueba es si la configuración seleccionada existe en la memoria *on-chip* asociada al controlador local. Para ello, se comprueba si hay una entrada relativa a esa tarea en la tabla asociativa (paso 1). En caso de no existir, la máquina de estados del módulo transitará por distintos estados para crear una nueva entrada en la tabla del controlador correspondiente a la tarea que estamos procesando para dejar constancia de que esa configuración va a ser almacenada (paso 2). De esta forma, se consigue que, si esa misma tarea se solicita tarea de nuevo en el futuro, la tabla del controlador tenga constancia de ello.

Una vez realizada la comprobación, se extrae de la tabla asociativa la información de cuántos bloques existen de esa configuración en la memoria *on-chip* (paso 3). Si esa configuración no existía en la tabla asociativa, este número (*n*) se inicializa a 0. Llegados a este punto, se procesan todos y cada uno de los bloques correspondientes a la tarea solicitada para este controlador en concreto (paso 4). Pudiendo suceder lo siguiente:

1. **Que el bloque a procesar se encuentre en la memoria *on-chip* asociada a dicho controlador (acierto, pasos 5 y 9)**. Se leerán todas y cada una de las palabras que constituyen el bloque y se le devolverán al controlador global a medida que se van leyendo. En este caso es importante comentar que la tabla no sufre variaciones ya que estamos leyendo parte de una configuración ya existente en la memoria *on-chip*, por lo que no se ha modificado la memoria *on-chip* y en consecuencia no se debe modificar la tabla.

2. **Que el bloque a procesar no se encuentre en la memoria *on-chip* asociada a dicho controlador (fallo, pasos 6, 7, 8 y 10)**. En este caso se debe ir a memoria externa (paso 6), donde reside el bloque a procesar, y éste se ha de copiar en la memoria *on-chip* asociada a dicho controlador (paso 7). A la vez que se va realizando la lectura desde la memoria externa, se realiza la copia en la memoria *on-chip* y además se devuelve al controlador global la información referente al bloque que se está procesando. Como se puede observar, en este caso se está trayendo de memoria externa un bloque que antes no estaba almacenado en la memoria *on-chip*, por lo que se debe dejar constancia en la tabla (paso 8) para que así, si se vuelve a solicitar el mismo bloque al controlador local, éste sepa que tiene ese bloque y que, por tanto, no vaya a memoria externa a buscarlo. Es importante comentar lo que puede suceder al traer un nuevo bloque a la memoria *on-chip* (algo que se obvia en el diagrama por cuestión de simplicidad en el paso 7):

   2.1. Que la memoria *on-chip* no se encuentre llena. En este caso se escribe el bloque allí donde apunte el puntero de bloques libres (explicado en el apartado 3.1.3), y una vez escrito éste, procederemos a actualizar dicho puntero. Más adelante explicaremos con más detalle la organización de la memoria en referencia a la gestión de los bloques libres.

   2.2. Que la memoria *on-chip* se encuentre llena. En este caso no es trivial ni directo decidir dónde escribir el bloque, porque un bloque ya existente en la memoria *on-chip* debe ser sustituido por el bloque entrante. Esta decisión la toma el módulo de reemplazo, el cual se explicará en detalle más adelante. Este módulo es el encargado de decidir la dirección donde se debe escribir el bloque entrante. Como podemos suponer, antes de escribirlo, la dirección devuelta por el módulo de reemplazo era la dirección de comienzo de un bloque ya almacenado en memoria *on-chip*. Por tanto, al salir éste de la memoria *on-chip*, es necesario actualizar la entrada de la tabla correspondiente a la configuración del bloque saliente, para mantener así la

coherencia. Pueden suceder dos cosas a la hora de eliminar un bloque de una configuración almacenada en la memoria *on-chip*:

2.2.1. Se trate del bloque inicial o final de una configuración. En este caso se debe actualizar la tabla con el nuevo número de bloques almacenados y, en el caso de que el bloque saliente sea el inicial, se debe actualizar también el campo *Dir* de la tabla asociativa (paso 8). En este caso, se asigna a este campo la dirección donde se encuentra el siguiente bloque existente en memoria *on-chip*. Esto se realiza muy fácilmente consultando el valor del puntero "siguiente" del bloque saliente. En el caso de que no hubiese más bloques de dicha configuración y que éste fuese el único, se elimina la entrada de la tabla por razones de eficiencia.

2.2.2. No se trate del bloque inicial ni del bloque final de una configuración. En este caso, para mantener la coherencia del sistema, nos vemos obligados a eliminar todos aquéllos bloques comprendidos entre el bloque inicial almacenado y el bloque que vamos a eliminar. Esto es necesario porque *en nuestros controladores locales sólo se almacenan bloques contiguos de una configuración*. Por tanto, en la tabla asociativa debe quedar constancia de que ahora, los bloques almacenados están comprendidos desde el siguiente bloque al eliminado hasta el bloque final que ya existía.

3.1.2. Módulo de reemplazo

Es el encargado de facilitar al controlador local la dirección inicial del bloque a reemplazar, en el caso de que la memoria *on-chip* no disponga de la capacidad suficiente para almacenar un nuevo bloque. Se ha implementado como un módulo independiente del gestor para así ofrecer una mayor flexibilidad en el diseño. De esta forma, si se desease cambiar la política de reemplazamiento con la que funciona el controlador, bastaría con crear un módulo similar al existente, con las mismas entradas y salidas que el actual, y cambiar la lógica interna para que éste implemente la política deseada.

Es importante comentar que se trata de un módulo combinacional, lo que hace muy eficiente el reemplazo. En todo momento, devuelve la dirección del bloque a reemplazar. Esta dirección siempre se corresponde con la dirección inicial de algún bloque de los almacenados en memoria *on-chip*.

En nuestro caso hemos implementado una política de reemplazo LRU (*Least Recently Used*). Es decir, el bloque reemplazado es siempre aquél cuyo último uso fue el menos reciente de todos los posibles bloques candidatos. Se puede observar su esquema general en la Figura 20. Dicho módulo consta de una serie de registros que guardan los *tags* de las tareas correspondientes a los bloques existentes en la memoria *on-chip*, así como las direcciones donde éstos están guardados. Esta información es necesaria para realizar los cambios pertinentes en la tabla asociativa del controlador local. Por tanto, esta implementación hace posible que éste obtenga

esta información al instante, haciendo así que la actualización de la tabla sea extremadamente eficiente.

También existen tantos contadores como bloques pueden estar alojados en la memoria *on-chip*. Están inicializados de forma ascendente y que saturan cuando alcanzan su valor máximo, el cual se corresponde con el número de contadores existentes. Estos valores son una medida de cuánto tiempo llevan esos bloques en la memoria *on-chip* (en lo sucesivo, "edad"; a mayor edad, mayor tiempo llevan esos bloques en la memoria). En este caso, hay 4 contadores. El módulo de reemplazo siempre selecciona el bloque cuyo contador tenga el valor más alto (de acuerdo a la política LRU), lo cual se consigue a través de los comparadores, indicados también en la Figura 20. En este caso, el bloque cuyo contador tiene el valor más alto es el bloque *B_2*. El valor de todos estos contadores se compara con el máximo valor posible (4), por lo que los resultados de las comparaciones es "0001" ("1" indicando igualdad y "0" indicando desigualdad). El codificador servirá para que los dos multiplexores devuelvan correctamente el *Tag* y la dirección del bloque seleccionado para su reemplazo.

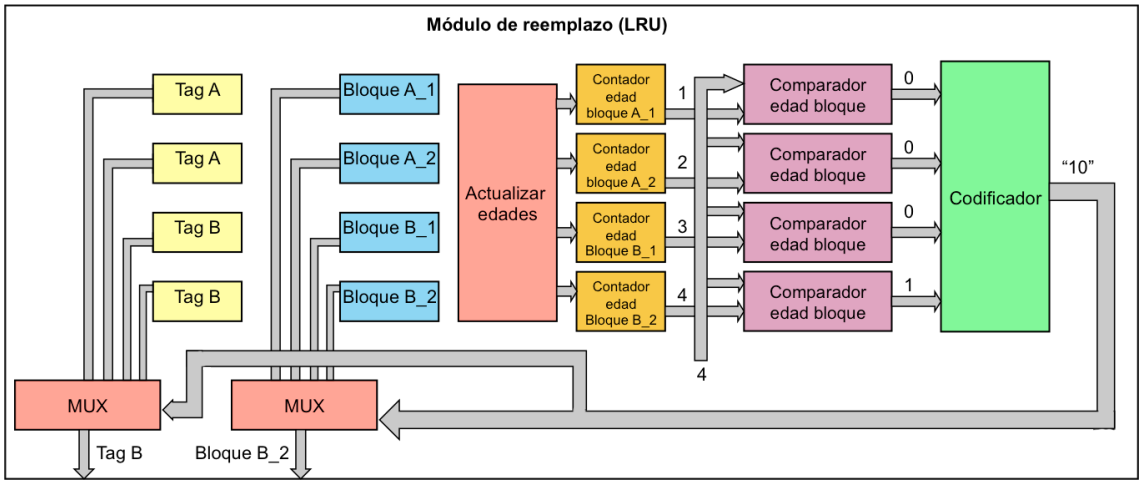

Figura 20. Esquema general del módulo de la política de reemplazamiento (el módulo desarrollado se comporta como la política LRU).

Por otro lado, a medida que se suceden los accesos/reemplazos de bloques, las "edades" de estos bloques se tienen que actualizar en consecuencia. Esto se explica en detalle más adelante en este apartado.

Por último, vamos a comentar detalladamente cómo se comporta el módulo de reemplazo desarrollado. Como hemos adelantado antes, los contadores de cada uno de los bloques en los que está dividida la memoria *on-chip* están inicializados de forma ascendente al iniciar la ejecución del controlador global. Una vez estén inicializados, ya están preparados para funcionar. A medida que el controlador local procesa los bloques solicitados, puede ocurrir lo siguiente:

- **Que la memoria *on-chip* asociada al controlador no se encuentre llena**. En este caso este módulo no realiza ninguna acción, ya que al tener la memoria capacidad necesaria, no es necesario reemplazar ningún bloque.

- **Que la memoria *on-chip* asociada al controlador se encuentre llena**. En este el módulo debe realizar su funcionalidad, pudiendo suceder dos cosas:

    - Que el bloque que se está procesando se encuentre en memoria *on-chip*. En este caso, se debe actualizar la edad de este bloque a "1", y se debe aumentar en 1 la edad del resto de los bloques. Esto lo realiza el módulo *actualizar edades* en la Figura 20.

    - Que el bloque que se está procesando no se encuentre en memoria *on-chip*. En este caso se tiene que eliminar un bloque de una determinada tarea que había almacenada en esta memoria. Esto puede ocasionar que el bloque a eliminar no sea el bloque inicial o el bloque final de su tarea, sino un bloque intermedio. Como ya se ha explicado en el apartado 3.1.1, si esto ocurre, la tabla asociativa debe eliminar los bloques necesarios para que queden en memoria *on-chip* bloques contiguos de dicha tarea, manteniendo así la coherencia. Si ocurre esto, a los bloques implicados se les asigna la edad máxima. Esto consigue que el módulo los posicione como máximos candidatos para ser los reemplazos próximamente. Además, se asigna edad "1" al bloque entrante para así indicar que acaba de ser utilizado.

3.1.3. Gestión de bloques libres y máquina de estados

Tal y como se ha comentado ya en apartados anteriores, el controlador local también incluye un registro que guarda la dirección del primer bloque que se encuentra libre en la memoria *on-chip*. Al igual que los bloques ocupados, estos bloques libres se encuentran enlazados entre ellos a través de punteros. Esto es muy útil para ir llenando la memoria *on-chip* hasta que ya no pueda almacenar un bloque completo más. Una vez la memoria *on-chip* está completa, este registro deja de utilizarse.

Finalmente, el funcionamiento de todos estos módulos está gobernado por una máquina de estados finita síncrona (FSM en la Figura 18). Básicamente, se encarga de que el controlador local se comporte de acuerdo a lo ya especificado en el diagrama de flujo de la Figura 19.

3.2. Detalles a bajo nivel del controlador global

El controlador global es el módulo encargado de recibir peticiones de carga de configuraciones y hacerle llegar la información de estas configuraciones al puerto de configuración ICAP (*Internal Configuration Access Port*). A su vez, este puerto es el encargado de volcar esta información en la FPGA. Su arquitectura está detallada en la Figura 21. Las peticiones que puede recibir el controlador global pueden ser de dos tipos:

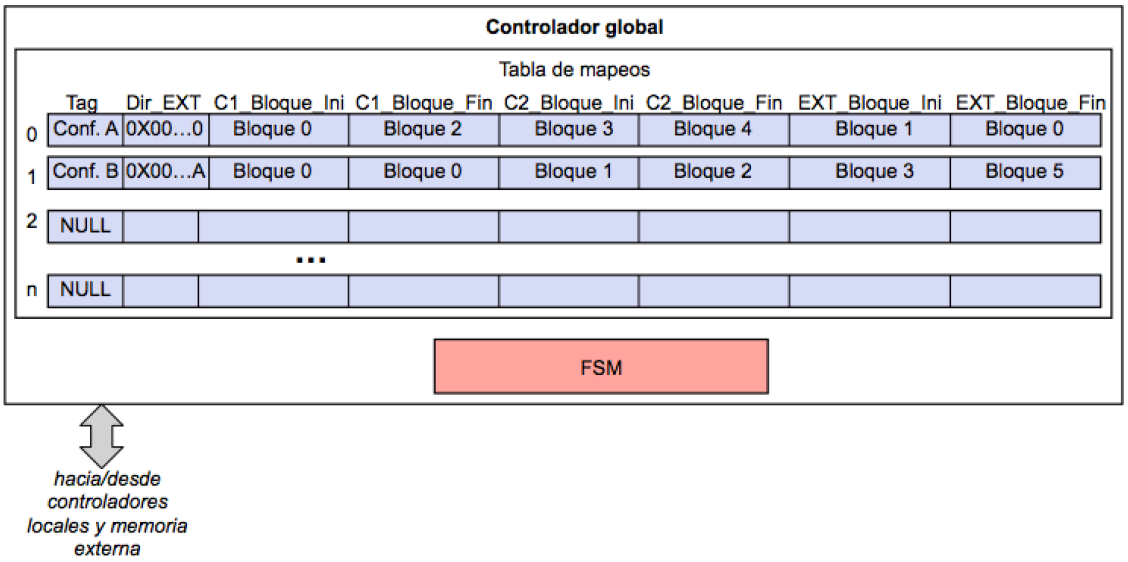

Figura 21. Esquema general del controlador global, incluyendo detalles de la tabla de mapeos del mismo.

- **Guardar/Actualizar mapeo de configuraciones:** Se especifica un mapeo para una reconfiguración dada. El controlador global guarda/actualiza esta información en una tabla (*Tabla de Mapeos* en la Figura 21) en consecuencia. Cada entrada en la tabla contiene la información acerca del nombre de la configuración en cuestión, la dirección donde está alojada el primer bloque de dicha configuración en memoria externa y cómo se distribuyen sus bloques entre las diferentes memorias *on-chip*. Esta tabla se puede implementar utilizando BRAMs de la FPGA, aunque en nuestro caso, sólo tuvimos necesidad de utilizar una.

  Es importante comentar que los bloques de las tareas no tienen por qué estar mapeados en todos los controladores locales ni en memoria externa. Es por ello por lo que se decidió en la fase de diseño que, cuando una tarea no tiene bloques en un determinado controlador local o en memoria externa, esta situación se indica en los campos *C<XXX>_Bloque_Ini* y *C<XXX>_Bloque_Fin* a valores 1 y 0, respectivamente. Esa combinación reservada representa que, en la memoria *on-chip* gestionada por el controlador *C<XXX>,* no hay bloques asignados.

Un ejemplo muy claro de esto último que cavamos de comentar se puede apreciar en la Figura 21, en concreto la tarea cuyo *Tag* es *Conf. A* no tiene mapeados bloques en memoria externa, ya que los campos *EXT_Bloque_Ini* y *EXT_Bloque_Fin* están asignados a valores 1 y 0, respectivamente.

- **Solicitar configuración:** En tiempo de ejecución, se solicita la reconfiguración de una tarea en el hardware reconfigurable. En este caso, el controlador global lee el mapeo de esa tarea de su tabla de mapeos (consultar párrafo anterior) y solicita de manera secuencial los bloques asignados a cada una de las memorias *on-chip*, tal y como se indica en el diagrama de flujo de la Figura 22 (Pasos 1-4). Para ello, primero invoca al controlador local correspondiente (Paso 3). Este módulo devuelve los bloques de configuraciones que han sido asignados a la memoria asociada con el controlador local en cuestión, ciclo a ciclo. El controlador global recibe esta información y la redirige al puerto de configuración ICAP de la FPGA. Para maximizar la compatibilidad con este puerto, se utiliza una señal de 32 bits [3]. Una vez que esto finaliza, el contador se actualiza (Paso 4). Los pasos 3 y 4 se ejecutan tantas veces como memorias *on-chip* diferentes existan en el sistema. Finalmente, se accede a la memoria externa para traer el resto de los bloques que no han sido mapeados en ninguna de las memorias *on-chip* (Paso 5). El controlador global ejecuta todas estas operaciones a través de una máquina de estados finitos, la cual no aparece en detalle en la Figura 21 por simplicidad.

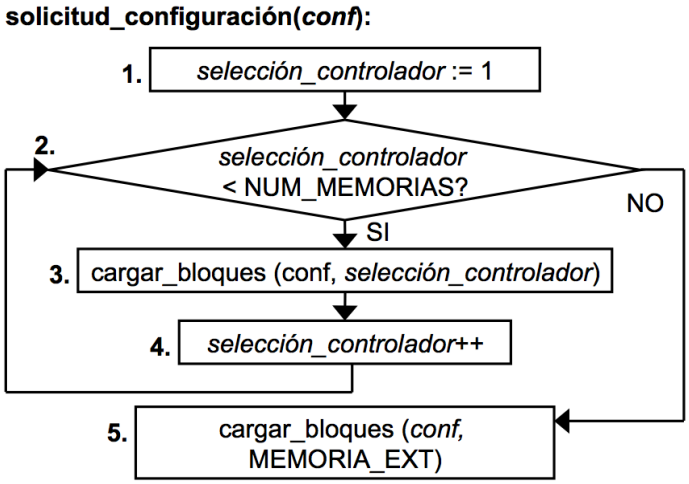

Figura 22. Diagrama de flujo que describe el funcionamiento del controlador global.

Por último, en la Figura 23 veremos a través de un ejemplo real el funcionamiento del sistema completo cuando recibe una petición de solicitud de configuración. En este caso se le solicita la tarea cuyo identificador es *Tag B*, cuyo mapeo en las memorias *on-chip* es el especificado en la Figura 21. Como podemos observar tiene un único bloque (bloque 0) asignado a la memoria del controlador 1, 2 bloques (1 y 2) asignados a la memoria del controlador 2, y 3 bloques (3, 4 y 5) mapeados en la memoria externa.

En este ejemplo, vamos a suponer que el bloque asignado a la memoria del controlador 1 (bloque 0) ya existe en esa memoria, mientras que los bloques 1 y 2 (asignados a la memoria del controlador 2) no se encuentran allí, por ejemplo, porque han sido reemplazados después de su último uso.

Por tanto, para acceder al bloque 0, éste se busca directamente de la memoria 1, tal y como se indica en la Figura 23. Sin embargo, para buscar los bloques 1-2, éstos han de buscarse en la memoria externa, ser traídos al controlador local 2, y éste realiza en paralelo 2 operaciones: 1) copiar esos bloques a la memoria 2 y 2) proporcionar esa información al controlador global. En este caso, es importante recordar que, cuando se busca información en la memoria externa, se utiliza la dirección de acceso especificada en la tabla de mapeos del controlador global, tal y coo se indicó en la Figura 21. Finalmente, los bloques 3-5 se buscan directamente de la memoria externa.

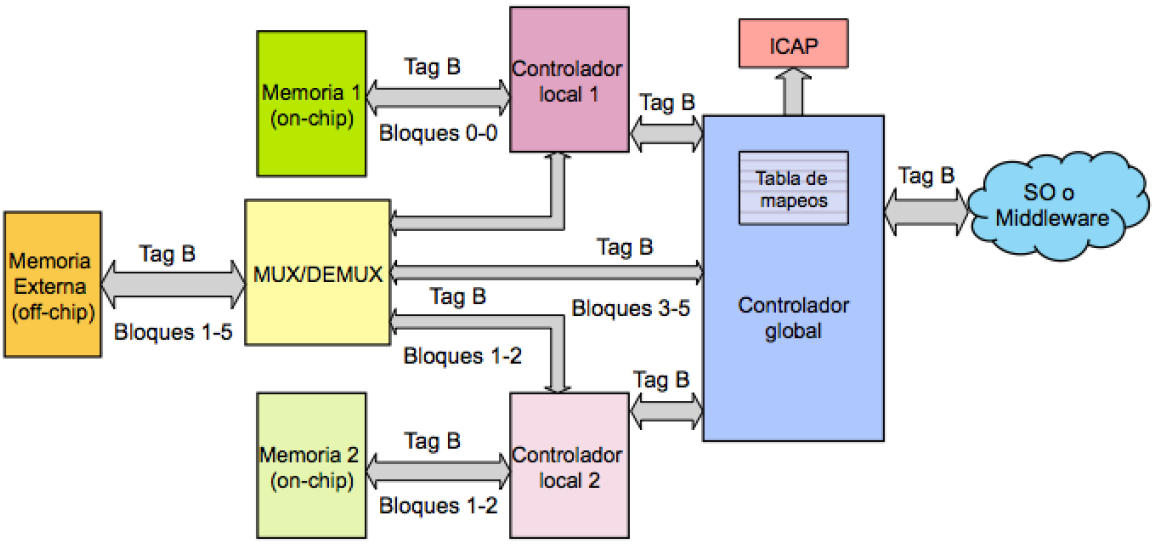

Figura 23. Ejemplo de ejecución. El mapeo de la configuración solicitada (*Tag B*) se puede apreciar en la Figura 21.

3.3. Detalles a bajo nivel del módulo de estadísticas

Este módulo da un valor añadido al controlador global diseñado, ya que ofrece la posibilidad de recalcular el mapeo de las reconfiguraciones en tiempo de ejecución, algo que hace más funcional si cabe el hardware diseñado. Nos pareció muy interesante observar cómo se comportaba el controlador en cualquier momento y así poder recalcular los mapeos de los bloques en las memorias *on-chip* en función de dicho comportamiento, con el único fin de mejorar el rendimiento a la hora de ejecutar las tareas.

Tiene varios objetivos: por una lado, llevar el recuento del número de aciertos y de fallos totales referentes a los bloques solicitados de todas las tareas ejecutadas. Por otro lado, almacena el número de aciertos y fallos de cada una de las tareas ejecutadas. Todo esto lo podemos observar en la Figura 24.

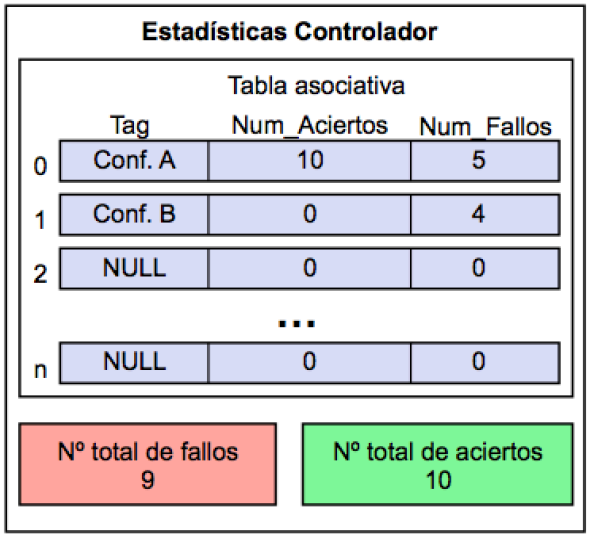

Figura 24. Esquema general del módulo de estadísticas.

Por tanto, este módulo ofrece al usuario la posibilidad de visualizar en cualquier momento de la ejecución el comportamiento del sistema en términos de aciertos y fallos, lo que permite, en el caso de que no esté funcionando de forma eficiente, volver a recalcular el mapeo de las reconfiguraciones con el único de fin de mejorar la eficiencia a la hora de ejecutar esas tareas.

4. Resultados experimentales

En este capítulo se evalúa el módulo hardware propuesto, en términos de consumo de recursos y de penalización temporal. Los resultados se explican en detalle en los apartados siguientes.

4.1. Consumo de recursos y escalabilidad

En primer lugar, la Tabla 1 presenta el consumo de recursos del controlador hardware descrito en el capítulo 3, cuando se instancian 2 controladores locales, conectados a 2 memorias *on-chip* respectivamente, con capacidad para albergar 4 bloques, siendo el tamaño de bloque de 128 Kb. Los resultados se muestran en términos de recursos reconfigurables de una FPGA Xilinx XUP Virtex-5 LX110T.

Tabla 1. Consumo de recursos del controlador hardware propuesto, con dos controladores locales, cada uno con su memoria asociada. Se asume que las memorias on-chip tienen una capacidad de 4 bloques cada una (para una FPGA Xilinx XUP Virtex-5 LX110T).

	Slice Registers	***Slice LUTs***	***Block RAMs***
Tabla asociativa	8 (0,01%)	42 (0,06%)	1 (0,68%)
FSM	48 (0,07%)	287 (0,42%)	0 (0%)
Módulo de reemplazo	14 (0,02%)	23 (0,03%)	0 (0%)
Controlador local	169 (0,24%)	458 (0,66%)	1 (0,68%)
Módulo de estadísticas	99 (0,14%)	106 (0,15%)	1 (0,68%)
Controlador global	18 (0,03%)	74 (0,11%)	1 (0,68%)
TOTAL	455 (0,66%)	1074 (1.55%)	4 (2,72%)

Para un controlador local (fila 4), los resultados se desglosan a su vez en sus componentes: la tabla asociativa (fila 1), la máquina de estados (fila 2) y el módulo de reemplazo (fila 3). También se presentan los resultados del controlador global (fila 6), teniendo en cuenta 2 instancias de controladores locales y su tabla asociativa, donde almacena la información de posicionamiento de los bloques tanto en los controladores locales como en memoria externa. Finalmente, se muestra el consumo de recursos del sistema completo (fila 7). Se puede observar que este último es muy asequible, ya que, en el peor de los casos, supone el 2.72% de la cantidad total de recursos disponibles en la FPGA. En este caso, la máxima frecuencia de reloj permitida a la que puede funcionar es 153.7 MHz, lo que permite sobradamente suministrar información de configuración al puerto ICAP a su máxima velocidad teórica (100 MHz).

También hemos estudiado la escalabilidad del diseño. En este caso, la complejidad del mismo aumenta con el número de gestores que se instancian, y con el número de bloques de configuración que se pueden almacenar en las memorias *on-chip*. Los resultados se muestran en la Figura 25 y en la Figura 26.

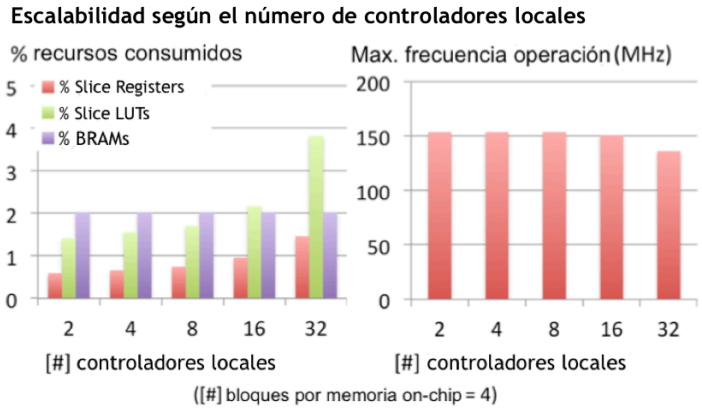

Figura 25. Escalabilidad del módulo hardware propuesto, tanto en términos de consumo de recursos y la frecuencia máxima de funcionamiento aumentando el número de instancias de controladores locales (para una FPGA Xilinx XUP Virtex-5 LX110T).

La Figura 25 muestra que el diseño propuesto escala muy bien con el aumento del número controladores locales (manteniendo el número de bloques por cada memoria *on-chip* constante a 4). De hecho, el consumo de recursos nunca supera el 4% de los recursos reconfigurables disponibles en la FPGA, y la frecuencia de funcionamiento máxima se mantiene siempre próxima a 150 MHz.

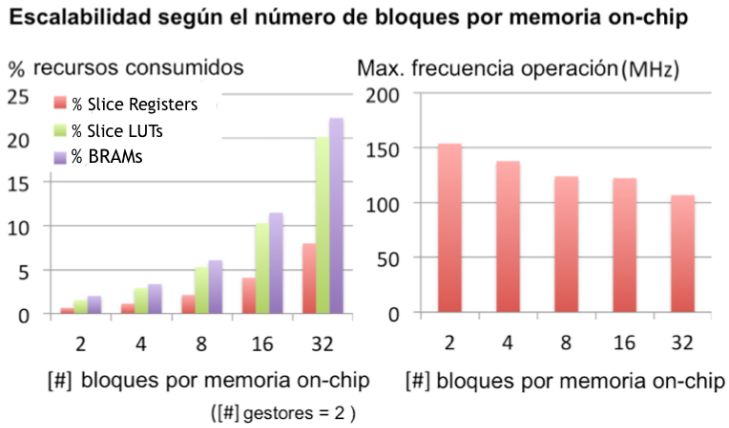

Figura 26. Escalabilidad del módulo hardware propuesto, tanto en términos de consumo de recursos y la frecuencia máxima de funcionamiento aumentando el número de bloques por memoria *on-chip* (para una FPGA Xilinx XUP Virtex-5 LX110T).

La Figura 26 muestra que el consumo de recursos se mantiene muy asequible (entorno al 10% o por debajo) cuando el número de bloques de configuración por memoria *on-chip* es 16 o menos. En este caso, se ha configurado el módulo para que instancie 2 controladores locales. Sin embargo, este consumo de recursos crece exponencialmente, y para 32 bloques por memoria *on-chip*, alcanza el 22% de los recursos disponibles. El motivo es que la complejidad del módulo de reemplazo de los controladores locales aumenta con el número de bloques diferentes que se pueden almacenar en las memorias *on-chip*. Por otro lado, la máxima frecuencia de funcionamiento de nuestro módulo se mantiene siempre por encima de 100 MHz, lo cual no supone ningún cuello de botella en las prestaciones del sistema, teniendo en cuenta la máxima velocidad del puerto de configuración ICAP (100 MHz).

4.2. Penalización por gestión transparente de reconfiguraciones

En esta sección se estudia la penalización temporal que el controlador desarrollado introduce en tiempo de ejecución (es decir, a medida que gestiona las peticiones de carga de configuraciones). Para ello, se han seleccionado un conjunto de *benchmarks* desarrollados por Texas Instruments [25]. Para ello, se ha utilizado una jerarquía de memoria similar a la explicada en la Figura 8 del apartado 1.2, en la que, por simplicidad, se han supuesto los tiempos de acceso a memoria correspondientes a la Tabla 2. Estos datos son consistentes con los mostrados en el ejemplo de motivación del apartado 1.3 de esta memoria, que a su vez fueron obtenidos a partir de la herramienta de simulación CACTI 4.0 [24].

Tabla 2. Tiempo de acceso ideal (en número de ciclos de reloj) a cada una de las memorias *on-chip* utilizadas en los experimentos de este capítulo.

Memoria	Retardo en número de ciclos
Externa	6
Altas prestaciones	3
Bajo consumo	2

Tabla 3. Número de configuraciones y tiempo ideal de ejecución para cada uno de los *benchmark*s utilizados.

Benchmark	Número de configuraciones	Tiempo ideal de ejecución
DSP_dot_prod	3	32 us
DSP_vec_sumq	2	32 us
DSP_q15_tofl	3	5645 us
DSP_neg_32	3	109 us
DSP_min_val	3	114 us
DSP_dotp_sqr	3	32 us
DSP_blkmove	3	109 us

En este apartado se mostrarán los resultados obtenidos al ejecutar estos *benchmarks*, cuyo tiempo ideal de ejecución y número de configuraciones viene dado en la Tabla 3, teniendo en cuenta que sus éstas (de tamaño constante) están compuestas de bloques de diferentes tamaños. Esto implica que, para cada uno de estos experimentos, las memorias *on-chip* tienen, capacidad para guardar diferentes números de bloques (siempre manteniendo su capacidad total constante a 4 configuraciones completas).

En concreto estas pruebas se han realizado para 4, 31 y 255 bloques por memoria *on-chip;* es decir, cuando las configuraciones son indivisibles, y constan de 8 y 64 bloques cada una, respectivamente. Nótese que, en los 2 últimos casos, las memorias *on-chip* no tienen capacidad para guardar el máximo número teórico de bloques (32 y 256, respectivamente). El motivo es que los punteros que enlazan los bloques consecutivos también ocupan espacio y esto reduce la capacidad efectiva de

las memorias en un bloque. Las tablas 4, 5 y 6 muestran los mapeos resultantes de las configuraciones en las memorias *on-chip* en cada uno de estos casos, cuando se utiliza el algoritmo de mapeo descrito en la referencia [11]. Aquí es interesante resaltar cómo, a medida que las configuraciones se particionan en más y más bloques, los mapeos de los bloques en las memorias *on-chip* se ajustan de manera cada vez más personalizada de acuerdo a las características particulares de los *benchmarks* utilizados. Así, por ejemplo, para *DSP_q15_tofl*, *DSP_neg_32* y *DSP_min_val*, la tabla 4 (configuraciones indivisibles) muestra los mismos mapeos en los tres casos, mientras que la tabla 5 ya muestra diferencias entre *DSP_min_val* y las otros dos *benchmarks*. Finalmente, la tabla 6 muestra un mapeo totalmente diferente para cada *benchmark*.

Tabla 4. Mapeo de las configuraciones de cada *benchmark* en las diferentes memorias *on-chip* cuando la capacidad máxima éstas es de 3 bloques y las configuraciones son indivisibles.

4 Bloques de capacidad por memoria on-chip			
Memoria / *Benchmark*	**Configuraciones indivisibles**		
	Externa	**Altas prestaciones**	**Bajo consumo**
DSP_dot_prod	0	1	2
DSP_vec_sumq	1	1	0
DSP_q15_tofl	1	1	1
DSP_neg_32	1	1	1
DSP_min_val	1	1	1
DSP_dotp_sqr	0	1	2
DSP_blkmove	0	2	0

Tabla 5. Mapeo de las configuraciones de cada *benchmark* en las diferentes memorias *on-chip* cuando la capacidad máxima éstas es de 31 bloques y las configuraciones están divididas en 8 bloques.

31 Bloques de capacidad por memoria *on-chip*			
Memoria / *Benchmark*	**Número de bloques por configuración: 8**		
	Externa	**Altas prestaciones**	**Bajo consumo**
DSP_dot_prod	10	8	6
DSP_vec_sumq	8	8	0
DSP_q15_tofl	14	8	2
DSP_neg_32	14	8	2
DSP_min_val	13	8	3
DSP_dotp_sqr	11	8	5
DSP_blkmove	0	16	0

Tabla 6. Mapeo de las configuraciones de cada *benchmark* en las diferentes memorias *on-chip* cuando la capacidad máxima éstas es de 255 bloques y las configuraciones están divididas en 64 bloques.

255 Bloques de capacidad por memoria *on-chip*			
Memoria / *Benchmark*	**Número de bloques por configuración: 64**		
	Externa	**Altas prestaciones**	**Bajo consumo**
DSP_dot_prod	92	64	36
DSP_vec_sumq	64	64	0
DSP_q15_tofl	118	64	10
DSP_neg_32	113	64	15
DSP_min_val	111	64	17
DSP_dotp_sqr	96	64	32
DSP_blkmove	0	128	0

A continuación, las tablas 7, 8 y 9 muestran las penalizaciones introducidas por el controlador propuesto, para los tamaños de bloques mencionados anteriormente. En estas tablas, las columnas 2 y 3 muestran los resultados cuando se ejecutan los *benchmarks* por primera vez en el sistema, mientras que las columnas 4 y 5 hacen lo propio con la segunda ejecución (y ejecuciones subsecuentes) en el mismo sistema. El objetivo de este experimento es observar cómo se comporta el controlador en dos situaciones:

- Cuando tiene que ir a buscar los bloques a la memoria externa y copiarlos a las memorias *on-chip* (primera ejecución).
- Cuando estas configuraciones ya se han cargado en las memorias *on-chip* correspondientes (segunda ejecución y sucesivas).

Finalmente, la fila 9 en todas las tablas muestra qué ocurre para la ejecución de los *benchmarks DSP_vec_sumq, DSP_q15_tofl, DSP_neg_32 y DSP_min_val* secuencialmente (*Benchmarks 2-5*). En este caso, tienen lugar reemplazos en las memorias *on-chip*, ya que en estas memorias sólo caben 4 configuraciones diferentes. Este último experimento no es aplicable a las columnas 4 y 5 de las tablas.

Tabla 7. Penalización introducida por el del controlador propuesto, cuando las configuraciones son indivisibles.

	Primera ejecución		**Segunda ejecución y sucesivas**	
Benchmark	**[#] Ciclos de reloj**	**% Penalización**	**[#] Ciclos de reloj**	**% Penalización**
DSP_dot_prod	426	1,33E-2	812	2,53E-2
DSP_vec_sumq	277	8,65E-3	405	1,26E-2
DSP_q15_tofl	419	7,42E-5	676	1,19E-4
DSP_neg_32	419	3,84E-3	676	6,21E-3
DSP_min_val	419	3,67E-3	676	5,92E-3
DSP_dotp_sqr	426	1,33E-2	812	2,53E-2
DSP_blkmove	284	2,61E-3	540	4,95E-3
Benchmarks 2-5	2042	3,46E-4		

Tabla 8. Penalización introducida por el del controlador propuesto, cuando el número de bloques por configuración es 8.

	Primera ejecución		Segunda ejecución y sucesivas	
Benchmark	**[#] Ciclos de reloj**	**% Penalización**	**[#] Ciclos de reloj**	**% Penalización**
DSP_dot_prod	358	1,11E-2	629	1,96E-2
DSP_vec_sumq	277	7,12E-3	405	1,17E-2
DSP_q15_tofl	332	5,88E-5	591	1,04E-4
DSP_neg_32	332	3,04E-3	591	5,42E-3
DSP_min_val	340	2,98E-3	654	5,73E-3
DSP_dotp_sqr	356	1,11E-2	780	2,43E-2
DSP_blkmove	284	2,61E-3	582	5,33E-3
Benchmarks 2-5	1252	2,12E-4		

Tabla 9. Penalización introducida por el del controlador propuesto, cuando el número de bloques por configuración es 64.

	Primera ejecución		Segunda ejecución y sucesivas	
Benchmark	**[#] Ciclos de reloj**	**% Penalización**	**[#] Ciclos de reloj**	**% Penalización**
DSP_dot_prod	321	1,01E-2	558	1,74E-2
DSP_vec_sumq	276	8,62E-3	608	1,91E-2
DSP_q15_tofl	557	9,86E-5	806	1,42E-4
DSP_neg_32	415	3,80E-3	844	7,74E-3
DSP_min_val	415	3,64E-3	849	7,44E-3
DSP_dotp_sqr	420	1,31E-2	973	3,01E-2
DSP_blkmove	284	2,61E-3	948	8,69E-3
Benchmarks 2-5	1683	2,85E-4		

En estas tres tablas se puede observar que las operaciones realizadas introducen retardos del orden de varios cientos de ciclos de reloj. Suponiendo que el controlador funciona a la máxima velocidad del puerto ICAP (100 MHz), estos retardos son insignificantes con respecto a los tiempos de ejecución ideales de los *benchmarks* (véanse las columnas 3 y 5 de las tablas). De hecho, en el peor de los casos, estos retardos pueden suponer; el 1,33E-2% (tabla 7), el 1,11E-2% (tabla 8) y el 1,31E-2% (tabla 9), del tiempo de ejecución ideal de la aplicación *DSP_dotp_sqr* (véase la columna 5).

Analizando estas tablas, se puede observar que nuestro diseño no introduce retardos significativos cuando se varía el número de bloques por configuración. De este modo, incluso si se utiliza un número muy elevado de bloques por configuración (tabla 9), nuestro controlador seguirá funcionando de forma correcta y eficiente. Por ejemplo, para la aplicación *DSP_blkmove* los mapeos de las configuraciones de la aplicación en todos los experimentos han sido realizados en una memoria de altas prestaciones y como podemos comprobar la penalización es la misma en todos los casos (2,61E-3%).

Analizando los resultados obtenidos en estas tres tablas observamos que, la penalización introducida tiene una fuerte dependencia con la distribución de los bloques en las diferentes memorias de la jerarquía.

Por ejemplo, para la aplicación *DSP_dot_prod* obtenemos un 1,33E-2%, un 1,11E-2% y un 1,01E-2% de penalización (tablas 7, 8 y 9 respectivamente). El resultado de la tabla 7 produce mayor penalización durante la primera ejecución, debido a que todas las configuraciones de esta aplicación son solicitadas a memorias *on-chip* para inmediatamente después ser buscados en la memoria externa. El motivo es que al ser una primera ejecución, todos sus bloques siempre producen fallo. En cambio, el resultado la penalización mostrada en la tabla 8 es menor, ya que en este caso hay un mayor porcentaje de bloques mapeados en memoria externa (41% de los bloques –consultar tabla 5-, frente al 0% de la tabla 4). Por último en la tabla 9, tendríamos un valor aún menor que en la tabla 8, lo cual es acorde a la distribución de los bloques en las diferentes memorias existentes. Por otro lado, si analizamos los resultados obtenidos para las segundas ejecuciones de las tareas, podemos constatar que los retardos introducidos también son directamente proporcionales a la cantidad de información que se guarda en las memorias on-chip (2,53E-2%, 1,96E-2% y 1,74E-2%, respectivamente).

Cabe destacar el caso del *benchmark DSP_q15_tofl*, esta aplicación tiene sus configuraciones mapeadas en gran medida en memoria externa para las tablas 8 y 9, y su penalización para una primera ejecución es de 5,88E-5 y 9,86E-5 respectivamente. En cambio, cuando esta aplicación se ejecuta dos veces consecutivas, la penalización asciende a valores de 1,04E-4% y 1,42E-4%. Esto se debe a que el número de configuraciones mapeadas en memorias *on-chip* es inferior al de configuraciones mapeadas en memoria externa.

Por último, es interesante comentar el caso del *benchmark DSP_blkmove*, el cual para una primera ejecución, obtiene la misma penalización independientemente del número de bloques en el que hayan sido particionadas sus configuraciones. En cambio, no sucede lo mismo en sucesivas ejecuciones, cuyas penalizaciones son 4,95E-3%, 5,33E-3% y 8,69E-3% (tablas 7, 8 y 9 respectivamente). Es interesante notar que, en este caso, la penalización aumenta a la vez que aumenta el número de bloques por configuración. Esto se debe a que el controlador tiene que invertir un mayor número ciclos en consultar a la tabla asociativa la existencia de estos bloques a medida que se van solicitando (aunque el primer bloque de la secuencia exista en la memoria *on-chip*, cada vez que es necesario buscar un nuevo bloque, se debe consultar a la tabla asociativa por si éste hubiese sido reemplazado). Sin embargo, esto no ocurre durante la primera ejecución, porque en este caso, al acceder al primer bloque, el controlador descubre que ninguno de los bloques de la tarea en cuestión está en la memoria *on-chip*. Por tanto, va a buscar todos los bloques a memoria externa sin volver a consultar la tabla asociativa sucesivas veces.

5. Conclusiones

Este proyecto ha presentado una implementación hardware de un controlador que gestiona de manera eficiente las operaciones de gestión necesarias que hay que realizar en tiempo de ejecución en un sistema reconfigurable que aplica cacheo de reconfiguraciones y que particiona las configuraciones en varios bloques. Se ha utilizado una FPGA Xilinx Virtex-5 para la implementación de este módulo, pero se podría portar a cualquier otra FPGA sin tener que realizar ninguna modificación. Los resultados experimentales presentados han demostrado que el módulo propuesto apenas introduce retardos insignificantes en la ejecución de los *benchmarks* utilizados. Además, su coste de implementación en términos de recursos hardware es muy asequible. Finalmente, su frecuencia de funcionamiento máxima permite explotar el puerto de configuración interno de las Virtex-5 (ICAP) a su máxima velocidad de funcionamiento posible, por tanto nuestro módulo no constituye ningún cuello de botella en las prestaciones del sistema.

Por último, comentar que el trabajo de este proyecto de Sistemas Informáticos ha dado lugar a dos artículos, uno en español y otro en inglés, los cuales han sido enviados al congreso nacional Jornadas de Computación Reconfigurable y Aplicaciones (JCRA), y a la prestigiosa revista científica *IEEE Transactions on Very Large Scale Integration Systems* (IEEE TVLSI), respectivamente. En el momento de redactar esta memoria, ambos artículos se encuentran en proceso de revisión.

6. Trabajo futuro

La realización de este trabajo deja las siguientes líneas de trabajo futuro abiertas:

- Implementar la funcionalidad necesaria para que el módulo hardware desarrollado se comunique con puerto de configuración ICAP.

- Integrar este módulo dentro de un gestor de ejecución de tareas, para así poder testearlo con un elevado número de tareas de forma sencilla y sin tener que introducir dichas tareas a mano.

- Estudiar diferentes algoritmos de mapeo adaptativos, con el fin de obtener el mejor rendimiento posible del módulo. En este sentido, el módulo de estadísticas desarrollado (explicado en detalle en el apartado 3.3) será de gran utilidad para estudiar en tiempo de ejecución la calidad de los mapeos realizados.

Referencias

[1] T. Becker, W. Luk, and P. Y. K. Cheung, “Energy-aware optimisation for run-time reconfiguration,” in Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010, pp. 55–62.

[2] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 26, no. 2, pp. 203–215, 2007.

[3] Xilinx, “Virtex-5 FPGA configuration user guide, ug191(v3.10),” 2011.

[4] S. Liu, R. N. Pittman, A. Form, and J.-L. Gaudiot, “On energy efficiency of reconfigurable systems with run-time partial reconfiguration,” in Proceedings of the IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), 2010, pp. 265– 272.

[5] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, “GPUs and the future of parallel computing,” IEEE Micro, vol. 31, no. 5, pp. 7–17, 2011.

[6] Z. Li, K. Compton, and S. Hauck, “Configuration caching management techniques for reconfigurable computing,” in Proceedings of the anual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 22–36.

[7] S. Sudhir, S. Nath, and S. Goldstein, “Configuration caching and swapping,” in Field-Programmable Logic and Applications (FPL), ser. Lecture Notes in Computer Science, 2001, vol. 2147, pp. 192–202.

[8] D. Deshpande, A. K. Somani, and A. Tyagi, “Configuration caching vs data caching for striped fpgas,” in Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA), 1999, pp. 206–214.

[9] D. Deshpande, A. Somani, and A. Tyagi, “Hybrid data/configuration caching for striped fpgas,” in Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1999, pp. 294–295.

[10] E. Pérez, J. Resano, D. Mozos, and F. Catthoor, “Memory hierarchy for high-performance and energy-aware reconfigurable systems,” IET Computers & Digital Techniques, vol. 1, no. 5, pp. 565–571, 2007.

[11] J. Clemente, E. Ramo, J. Resano, D. Mozos, and F. Catthoor, “Configuration mapping algorithms to reduce energy and time reconfiguration overheads in reconfigurable systems,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, issue 6, pp. 1248–1261, 2014.

[12] E. Pérez, J. Resano, D. Mozos, and F. Catthoor, “Reducing the reconfiguration overhead: a survey of techniques,” in in Proceedings of the International Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA), 2007, pp. 191–194.

[13] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in Proceedings of the conference on Design, automation and test in Europe (DATE), 2001, pp. 642–649.

[14] B. Sellers, J. Heiner, M. Wirthlin, and J. Kalb, “Bitstream compression through frame removal and partial reconfiguration,” in International Conference on Field Programmable Logic and Applications (FPL), 2009, pp. 476–480.

[15] R. Cordone, F. Redaelli, M. Redaelli, M. Santambrogio, and D. Sciuto, “Partitioning and scheduling of task graphs on partially dynamically reconfigurable fpgas,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 28, no. 5, pp. 662–675, 2009.

[16] K. Ganeshpure and S. Kundu, “Game theoretic approach for run-time task scheduling on an multi-processor system on chip,” IET Circuits, Devices & Systems, vol. 7, no. 5, pp. 243–252, 2013.

[17] J. Clemente, J. Resano, C. Gonzalez, and D. Mozos, “A hardware implementation of a run-time scheduler for reconfigurable systems,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 7, pp. 1263 1276, 2011.

[18] K. Papadimitriou, A. Anyfantis, and A. Dollas, “An effective framework to evaluate dynamic partial reconfiguration in FPGA systems,” IEEE Transactions on Instrumentation and Measurement (TIM), vol. 59, no. 6, pp. 1642–1651, 2010.

[19] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, “Exploiting partial runtime reconfiguration for high-performance reconfigurable computing,” ACM Transactions on Reconfigurable Technology Systems (TRETS), vol. 1, no. 4, pp. 21:1–21:23, 2009.

[20] S. Chevobbe and S. Guyetant, “Reducing reconfiguration overheads in heterogeneous multicore RSoCs with predictive configuration management,” International Journal of Reconfigurable Computing, vol. 2009, pp. 8:4–8:4, 2009.

[21] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, “A power modeling and estimation framework for VLIWbased embedded systems,” in ST Journal of System Research, 2001, pp. 26–28.

[22] M. Jayapala, F. Barat, T. Vander, F. Catthoor, H. Corporaal, and G. Deconinck, “Clustered loop buffer organization for low energy VLIW embedded processors,” IEEE Transactions on Computers, vol. 54, no. 6, pp. 672–683, 2005.

[23] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, “The energy efficiency of iram architectures,” in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), 1997, pp. 327–337.

[24] H. Labs, “http://www.hpl.hp.com/research/cacti/,” 2012. Xilinx, “Virtex-5 libraries guide for schematic designs, ug622 (v 13.1),” 2011.

[25] Texas Instruments, http://focus.ti.com/general/docs/dsnsuprt.tsp, 2010.