UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

TESIS DOCTORAL

Entorno para multitarea hardware en dispositivos reconfigurables con gestión dinámica de particiones y complejidad constante

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Sara Román Navarro

Bajo la dirección de los doctores:

Hortensia Mecha López,
Daniel Mozos Muñoz y Julio Septién del Castillo

Madrid, 2010

ISBN: 978-84-693-3227-6

Entorno para multitarea hardware en dispositivos reconfigurables con gestión dinámica de particiones y complejidad constante

Universidad Complutense de Madrid

Departamento de Arquitectura de Computadores y Automática

Sara Román Navarro

2009

Entorno para Multitarea Hardware en Dispositivos Reconfigurables con Gestión Dinámica de Particiones

Memoria presentada por Sara Román Navarro para optar al grado de Doctora por la Universidad Complutense de Madrid. Trabajo realizado en el Departamento de Arquitectura de Computadores y Automática de la Facultad de Informática de la Universidad Complutense de Madrid bajo la dirección de los doctores Hortensia Mecha López, Daniel Mozos Muñoz y Julio Septién del Castillo.

Madrid Mayo 2009

A Román y Carmen

A todos los Budas

Esta tesis ha sido posible gracias a la financiación de la Comisión Interministerial de Ciencia y Tecnología, a través de los proyectos TIC2002-00160, TEC2005-04752 y TIN2006-03274

Todas las olas del mar deben la belleza de su perfil

a las que precedieron y se retiraron

André Gide

Las cosas importantes no son cosas

Anónimo

No hay un camino que lleve a la felicidad: la felicidad ES el camino

Buda Sakyamuni

Agradecimientos

Ninguno estaríamos hoy donde estamos si no fuera por los que nos han precedido, como expresa André Gide de una forma tan bella. Es por esto que la lista de personas a las que agradecer el que se haya podido realizar este trabajo es, en realidad, infinita ...

Los más cercanos y evidentes, sin cuya valiosa existencia y ayuda no tendríamos hoy estas páginas en las manos, son mis directores de tesis : Horten, Daniel y Julio, que además fueron compañera y profesores en la carrera, respectivamente.

Y otros que, no siendo tan evidentes, son sin embargo importantes, porque han ayudado con muchos pequeños detalles, engorrosos pero imprescindibles: a Marcos por su insuperable capacidad crítica y a Juan Carlos por su maestría con el LaTeX y su generosidad con el tiempo.

A todos mis compañeros de pasillo por su paciencia y cariño, especialmente a Guadalupe, compañera de despacho y de sincronías.

Y a todos los de los pasillos de más allá, por orden geográfico: Josele, Inma, Victoria, David, Javi, Jose Luis, Ángel, Miguel. A los compañeros de otros edificios: Silvia, Dani, Nacho, Manel. A los que ya no vemos por estos pasillos: Jose, Javi, Fredy. A todos ellos por los buenos ratos, los consejos y el

apoyo.

También a mi padre y mi hermana María, por compartir conmigo su valiosa experiencia como doctores.

Y a mi madre, por ser mi madre.

A la hermana Jotika por ayudarme a ampliar la visión y poner las cosas en contexto en esos momentos en que la mente se cierne sobre un solo objeto de concentración y parece que no existiera nada más.

A Ross Freeman, visionario inventor y con un precioso apellido.

Y a la infinita lista de todos lo seres, sin los cuales nos estaríamos ninguno aquí ahora.

GRACIAS

Índice general

Capítulo 1

Introducción

Puede decirse del mundo desarrollado actual, esta parte del mundo donde nosotros vivimos ahora, que está caracterizado por dos aspectos fundamentales:

- **Es una cultura audio-visual:** lo que no puede verse ya no vende. Los libros están siendo desplazados por toda clase de medios audiovisuales: televisión, videojuegos, internet (videos, fotografías ... todo en imágenes). Las mentes de las nuevas generaciones apenas pueden manejarse ya con conceptos abstractos no apoyados sobre imágenes, preferiblemente acompañadas por sonido (música, voz, pero algo que suene; el silencio también ha caído en desuso).

- **Es una cultura de prisas:** Vivimos en un tiempo donde todo va muy rápido, la tecnología avanza muy rápido, los coches circulan a gran velocidad, las noticias vuelan, la gente corre de un lado a otro... y las personas ya no estamos acostumbradas a esperar.

Es un círculo que se realimenta a sí mismo: la tecnología ofrece mayores y mejores productos y los usuarios nos acostumbramos y demandamos cada vez más. Si fue antes el huevo o la gallina: la pregunta del millón. Nosotros nos limitaremos a asumir la realidad y a proponer una solución para una pequeña parte del vasto problema, solución que "a pesar de" su sencillez, y como probaremos a lo largo de estas páginas, funciona.

1.1. Demandas actuales en computación

Podemos constatar la gran inversión que la industria tecnológica ha realizado en los últimos diez años en el desarrollo de sistemas portátiles y audiovisuales cada vez más complejos y espectaculares, que atrapan nuestra atención.

Y todo ello siempre bajo la presión de un mercado muy competitivo que se mueve también a gran velocidad. Los fabricantes de tecnología necesitan sacar al mercado productos cada vez más pequeños, más ligeros, que consuman poco y que sean baratos.

Esta marcada tendencia se traduce en que los sistemas deben ofrecer cada vez mayores velocidades de proceso de la información (por ejemplo para las aplicaciones gráficas), menores consumos (para los sistemas portátiles) y sin aumentar, o preferiblemente disminuir, su tamaño, peso y precio.

Esta situación plantea un serio reto para los diseñadores de sistemas, que necesitan soluciones buenas, baratas y que salgan rápido al mercado.

Por esta razón han surgido numerosas líneas de investigación en bajo consumo, arquitecturas especializadas, nuevas tecnologías ...

Y es dentro de este contexto donde encontramos también numerosas líneas

de investigación relacionadas con el uso de **Dispositivos de Hardware Reconfigurable** (en adelante DHWR), ya que ofrecen un seguro "camino medio" entre la rigidez de los dispositivos especializados (ASICs) y el limitado rendimiento de los dispositivos de propósito general (microprocesadores), a un coste cada vez menor y ofreciendo cada vez mejores prestaciones en cuanto a consumo, disponibilidad de elementos básicos programables y memoria *on-chip*, gracias al aumento en la densidad de integración conseguida en los últimos años. Esto se muestra en al figura 1.1.

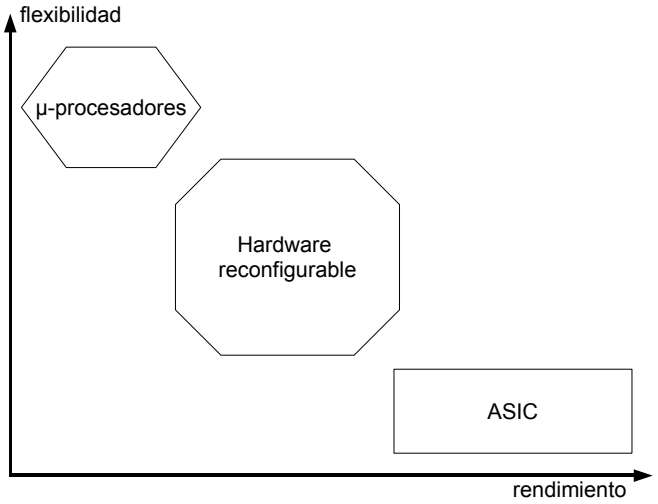

FIGURA 1.1: DHWR: el camino medio

Esta apuesta por los DHWR está fundamentada en el análisis de la tendencia de los sistemas computacionales actuales, como ha reflejado [Poz06] en su trabajo, en el que expone su convicción de que las **FPGAs**, cuyas siglas significan Matriz de Puertas Programables (de *Field Programmable Gate Array*), terminarán siendo las grandes competidoras de las CPUs. En esta línea, [Und04] hace un análisis del rendimiento pico de FPGAs y CPUs y sostiene

que el de las FPGAs puede llegar a ser superior.

Otros autores destacan la importancia del uso de FPGAs como piezas fundamentales integradas en sistemas tradicionales, como por ejemplo [MTAB07], [SLM00], para poder así aprovechar las ventajas computacionales de estos dispositivos sin perder las ya de sobra conocidas de las CPUs.

1.2. Dispositivos HW reconfigurables

Los DHWR se caracterizan por disponer de una serie de **elementos básicos configurables**, generalmente dispuestos en filas y columnas, que se pueden programar y combinar para realizar funciones específicas con más eficacia que un dispositivo de propósito general.

Estos elementos se configuran a través de algún tipo de mecanismo físico cuya disposición determina la función que realizan. Dicha función puede cambiarse de manera sencilla y rápida, tantas veces como sea necesario.

La **reconfiguración** del dispositivo puede ser **total** o **parcial**, afectando a todos los elementos básicos configurables o solamente a una parte de ellos. La reconfiguración parcial dinámica supone la posibilidad de reconfigurar una parte del dispositivo sin afectar al funcionamiento del resto.

Los elementos reconfigurables se conectan entre sí y a los pines de entrada y salida del dispositivo a través de una **red de conexiones**, también reconfigurables.

Los DHWR se dividen en dos grandes grupos: **grano fino** y **grano grueso**. Lo que distingue a unos de otros es la complejidad de los elementos básicos programables y la anchura de las rutas de datos disponibles. Si se trata de

elementos configurables muy básicos (por ejemplo memorias de 16 palabras de 1 bit), hablamos de DHWR de grano fino. Si por el contrario los elementos básicos programables son relativamente complejos (como por ejemplo ALUs), hablamos de DHWR de grano grueso.

Las arquitecturas de grano fino más populares son las FPGAs, que trabajan a nivel de bit, ya que es la unidad de información que procesan las celdas lógicas programables (CLBs) y por tanto que se transmite de unas celdas a otras por medio de la red de conexiones del dispositivo. Las FPGAs se configuran a través de un mapa de bits almacenado en la memoria RAM de configuración.

Los DHWR han demostrado su eficacia en muchos campos de aplicación, entre los que destacan el procesamiento de imágenes [KCMO06], [OPF06], [JTR+05], etc., los dispositivos portátiles [MVVL02], [GP02] y el encriptamiento de información [SM06], [CG05].

1.2.1. Grano fino vs. grano grueso

Dentro de las líneas de investigación que trabajan con DHWR hay algunos autores que proponen utilizar arquitecturas de grano grueso [Har01], [RSÉHB08], [Mei03]. Otros, en realidad la mayoría de ellos, optan por arquitecturas de grano fino, fundamentalmente FPGAs: [BKS00], [HV04], [DP05], [DEM+00], [WP02], [ABF+07]. Ambas opciones tienen ventajas e inconvenientes, que comentamos a continuación.

Las arquitecturas de grano grueso trabajan con una anchura de palabra que puede variar entre los 8 bits de MATRIX, [MD96], a los 32 bits de Chameleon, [SC01]. Esta anchura de palabra permite diseñar rutas de datos con

una estructura regular y hacer un uso de área más eficiente que las FPGAs, al necesitar una memoria de configuración de menor tamaño.

La otra ventaja de las arquitecturas de grano grueso es la menor cantidad de información necesaria para configurar las unidades básicas programables, lo que se traduce en un menor tiempo de reconfiguración que el necesario para reconfigurar FPGAs.

Las desventajas de las arquitecturas de grano grueso son su menor flexibilidad a la hora de diseñar circuitos con anchos de palabra diferente y el hecho de que las arquitecturas existentes no son comerciales y por tanto son menos accesibles para el investigador que las FPGAs. Otro grave inconveniente, el mayor de todos posiblemente, es la ausencia de herramientas de diseño automático para este tipo de sistemas. Debido a que las arquitecturas de grano grueso se desarrollan en entornos académicos de reducido impacto, no se invierten recursos en generar compiladores ni herramientas de síntesis, por lo que la compilación de aplicaciones a HW es larga y costosa y no es portable a otras arquitecturas.

Por su parte, las FPGAs proporcionan una gran cantidad de elementos muy básicos programables junto a otros elementos más complejos (multiplicadores e incluso en algunos casos, procesadores en el chip) que las dotan de una gran versatilidad para el diseño de circuitos. Existen diferentes fabricantes que las comercializan a precios asequibles y ofrecen una variedad de tamaños de dispositivo que permite seleccionar la adecuada para cada tipo de aplicación. Asimismo, en los últimos años los fabricantes de FPGAs han desarrollado herramientas de diseño y entornos de trabajo que permiten hacer un uso eficiente de sus funcionalidades y que parten de una descripción HW de alto nivel

estándar, portable a otras arquitecturas.

Estas circunstancias, junto al hecho de que en un principio se utilizaron extensamente en la industria para prototipado rápido de sistemas, las han convertido en los DHWR más populares tanto en el ámbito académico como en el sector industrial, a pesar de los inconvenientes antes mencionados: dificultad de rutado de señales en el dispositivo y tiempos de reconfiguración todavía altos.

1.2.2. Evolución de las FPGAs

El concepto de hardware reconfigurable se remonta a los años 60 [EBTB63]. Estrin fue pionero en la propuesta de acelerar las computaciones por medio de sistemas que pudieran adaptarse a diferentes tipos de cálculos y él y su grupo publicaron numerosos trabajos en esta línea. Reddi y Freustel le siguieron con una propuesta de arquitectura reconfigurable publicada en [RF78].

Pero el avance real en el campo de la computación reconfigurable no llegó hasta mediados de los años 80, cuando aparecieron las primeras FPGAs. Son numerosos los autores que han escrito acerca de la evolución de los DHWR, como el propio Estrin [Est02] y autores como Compton [CH02], Hauck [Hau98], Kuon [KTR08] y Leong [Leo08] y es denominador común el hacer un mayor hincapié en las FPGAs, por su importancia en este ámbito.

Las FPGAs fueron inventadas por Ross Freeman (uno de los fundadores de Xilinx), en el año 1984, en el Silicon Valley (CA, EEUU) [Xil09]. Se trataba de una tecnología novedosa que integraba la idea de programabilidad de los **PLDs**, Dispositivos Lógicos Programables (de *Programmable Logic Devices*),

y la eficiencia en el rendimiento de los ASICs.

Las FPGAS utilizaban una gran cantidad de transistores y en aquella época muchos pensaron que su invento no tendría éxito. Sin embargo, Ross Freeman confiaba en que el precio de los transistores disminuyera rápidamente, conforme a lo postulado por la Ley de Moore. Su acertada intuición convirtió a Xilinx en una de las empresas más prósperas y populares en la fabricación de FPGAs y en un referente para otras empresas que también las fabrican, como son Altera y Atmel.

Las primeras FPGAs eran similares en capacidad a los PLDs más grandes y fueron utilizados en un principio como alternativa a los costosos ASICs en partes de los diseños que correspondían a módulos no existentes en el mercado. La primera familia de FPGAs, la XC2064, salió al mercado en 1985. A modo de curiosidad podemos comentar que formaba parte de los componentes reconfigurables del computador personal Commodore-Amiga.

Las capacidades de las FPGAs siguieron aumentando y empezaron a utilizarse en prototipado rápido y de bajo coste de circuitos a medida, incluído el prototipado de procesadores. Este popular uso de las FPGAs se ha mantenido hasta nuestros días.

En 1991 salió al mercado la XC4000, que suele citarse como la arquitectura o familia típica de FPGAs de gama media. Están compuestas por una matriz de bloques básicos configurables (CLBs), cada uno de los cuales contiene una sección (*slice*) con tres **LUTs**, Tablas de Verdad (de *Look-up Tables*), que son tablas de 16 x 1 bit que permiten implementar funciones lógicas sencillas) y dos *flips-flops*. Además disponen de lógica esencial para propagación de acarreo. Las capacidades de estos dispositivos varían desde la más pequeña con una

matriz de 8 x 8 CLBs hasta la mayor de ellas, con 56 x 56.

La interconexión de CLBs se realiza por medio de líneas de conexiones organizadas jerárquicamente y de una matriz de **PSMs** o Matrices de Conmutación Programables (de *Programmable Switching Matrix*) intercaladas a intervalos regulares en la matriz de CLBs, como muestra la figura 1.2. Además existen líneas de interconexión específicas para la distribución de la señal de reloj y para la propagación de acarreo (local, de alta velocidad).

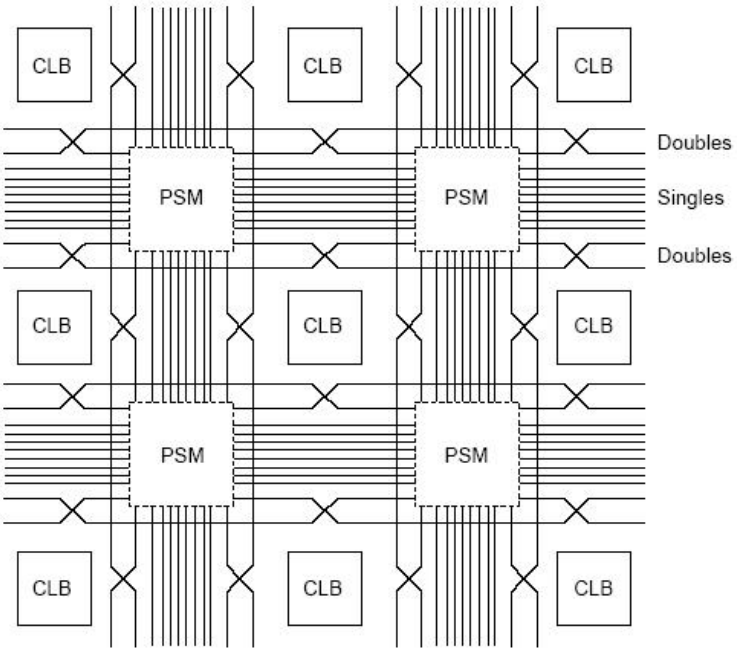

FIGURA 1.2: Interconexiones jerárquicas ($Xilinx^{TM}$)

Los bloques de entrada y salida (IOB) se encuentran situados en los bordes del dispositivo, cada uno de ellos conectado a un pin del chip FPGA. Estos bloques son también configurables (por medio de celdas RAM) y puede elegirse el sentido del flujo de información a través del pin así como características eléctricas y de temporización, lo que permite al dispositivo integrarse en una gran variedad de sistemas.

Este dispositivo se puede reconfigurar infinitas veces y la configuración es total, lo que significa que no se pueden configurar partes del dispositivo de forma independiente. La configuración se realiza por medio de la escritura de un conjunto de bits, llamado *mapa de bits de configuración*, que determinan el estado de una celda de memoria estática. Los valores de estas celdas controlan la función realizada por una LUT, la entrada de control de un multiplexor o un transistor de una celda PSM.

Esta arquitectura inicial de la familia XC4000 ha servido de base para las sucesivas familias de Xilinx, que pasamos a comentar.

En el año 1998 empieza la saga Virtex, que citando a Xilinx supone "un gran paso en la arquitectura de las FPGAs" y cuyas características resumimos en la tabla 1.1 y comentaremos más adelante.

La figura 1.3 muestra las arquitecturas básicas de las dos primeras familias Virtex, que fueron lanzadas al mercado en años sucesivos. Su estructura es similar y muy básica: bloques de memoria RAM intercalados con la matriz de CLBs, un elevado número de CLBs disponibles en relación a la XC4000 y las PLDs existentes hasta el momento y reconfigurables tantas veces como se quiera pero siempre la totalidad del dispositivo.

El siguiente año sale un nuevo miembro de la familia Virtex al mercado: la Virtex-2, que además de mejorar a sus hermanos mayores en cuanto a capacidad de procesamiento, cantidad de RAM disponible en el chip y aumento de la frecuencia de trabajo, dispone de módulos multiplicadores de 18 x 18 bits y capacidad de reconfiguración parcial dinámica.

Como ya se mencionó anteriormente, esta característica de la familia Virtex permite reconfigurar partes del chip sin afectar al funcionamiento de otras,

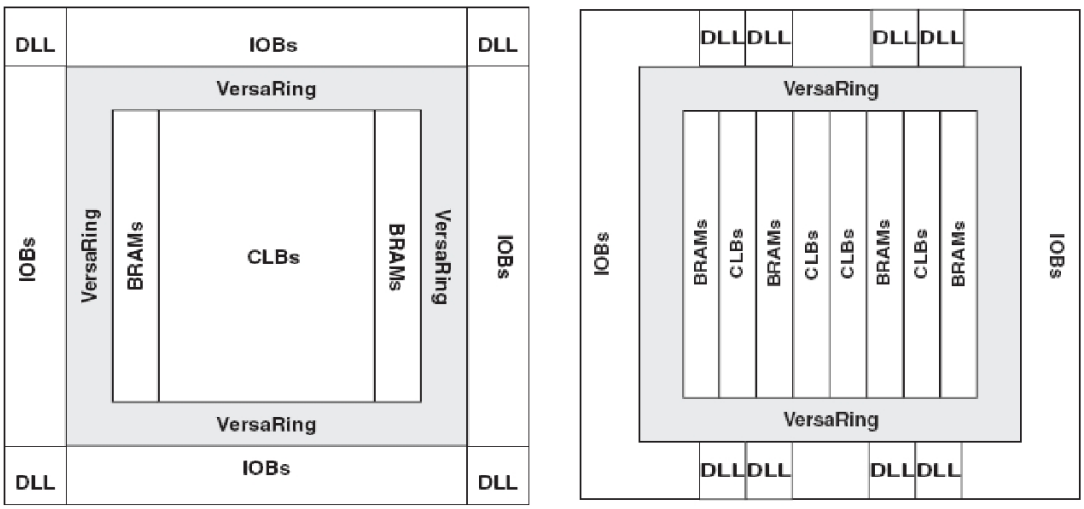

FIGURA 1.3: Arquitectura de la Virtex (izda) y la Virtex-E (dcha) ($Xilinx^{TM}$)

lo cual dota a la FPGA de una versatilidad muy alta y la capacita para realizar multitarea hardware. Esta característica de las FPGAs ha hecho crecer su popularidad notablemente en estos últimos ocho años. La reconfiguración parcial se realiza por columnas: la mínima parte de la FPGA que se puede reconfigurar de forma independiente al resto es un *frame*, cuya altura es igual a la del dispositivo y con anchura de 1 bit.

Esta novedosa característica supone una dificultad mayor a la hora de diseñar con estos dispositivos y las herramientas, documentación y entorno de diseño empiezan a aumentar en volumen y dificultad.

La figura 1.4 muestra un esquema de la arquitectura general de esta familia, bastante similar a la de sus predecesoras.

Dos años después Xilinx lanza al mercado la Virtex-2 Pro, una FPGA que supera a todas las anteriores en cuanto a capacidad de procesamiento ya que incluye dos procesadores Power PC. Se pueden configurar hasta dos de estos procesadores, incrustados en la matriz de CLBs, según indica la figura 1.5.

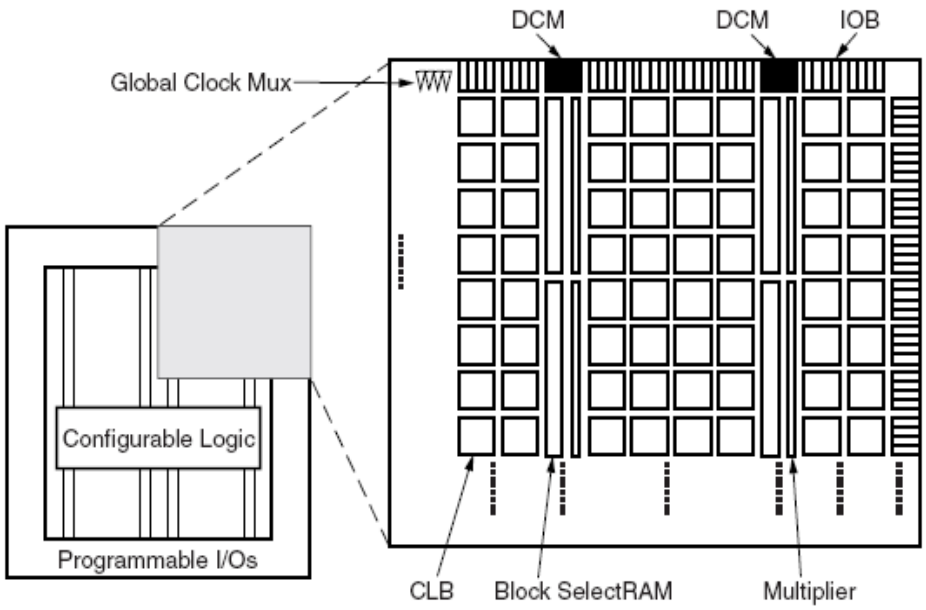

FIGURA 1.4: Arquitectura de la Virtex-2 ($Xilinx^{TM}$)

Con otros dos años de diferencia, en 2004, nace la Virtex-4, una gran FPGA que incluye todos los elementos antes citados, supera a sus antecesoras en velocidad, memoria en el chip, módulos aritméticos y que es reconfigurable en dos dimensiones.

Para esta FPGA ya no es necesario reconfigurar una columna completa de CLBs, sino que un *frame* abarca una altura de 16 CLBs. Esto significa que el mínimo bloque reconfigurable es de 1x16 CLBs y permite realizar reconfiguración parcial dinámica en 2D.

Con características similares, aparece la Virtex-5 en el mercado otros dos años después, con una cantidad mucho mayor, tanto de elementos configurables

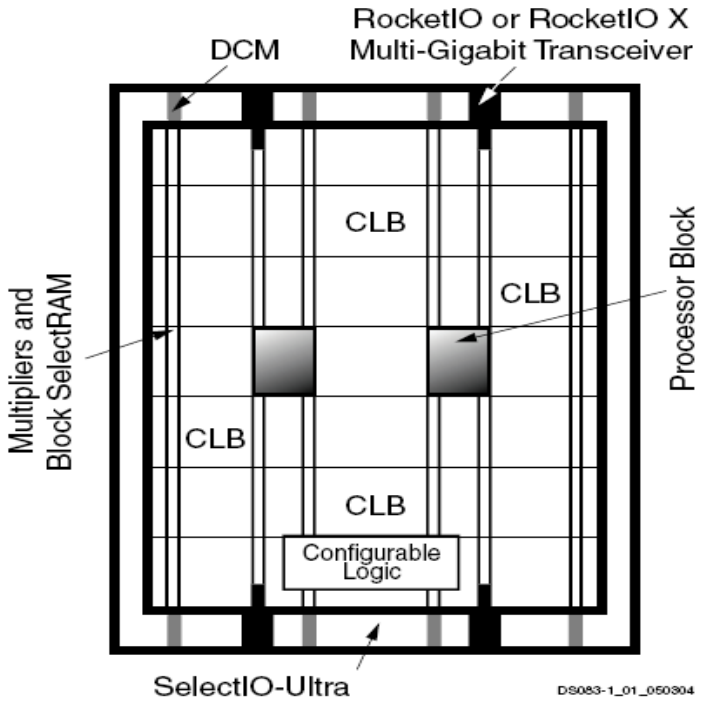

FIGURA 1.5: Arquitectura de la Virtex-2 Pro ($Xilinx^{TM}$)

como de memoria RAM interna.

Conforme evoluciona la complejidad de la familia Virtex, Xilinx desarrolla herramientas de diseño cada vez más completas y complejas, requiriendo del diseñador una preparación mayor y el manejo de volúmenes de documentación muy considerables.

La tabla 1.1 muestra que los tamaños de los mapas de bits de configuración aumentan notablemente con la capacidad y complejidad del circuito FPGA. Esto redunda en mayores tiempos de reconfiguración y la necesidad de utilizar más cantidad de memoria para almacenarlas.

El proceso de reconfiguración se hace más complejo y las necesidades de

TABLA 1.1: Resumen de la familia Virtex

Familia	**Año**	**Rec.**	**CLBs**	**CLK (MHz)**	**Mem.**	**Mapa de bits**
Virtex	1988	1D	64 x 96	200	122 Kb	6.127.744 bits
Virtex-E	1989	1D	104 x 156	300	832 Kb	16.283.712 bits
Virtex-2	2000	1D	112 x 104	420	3 Mb	26.194.208 bits
Virtex-2 Pro	2002	1D	120 x 94	450	8 Mb	34.292.768 bits
Virtex-4	2004	2D	192 x 116	500	10 Mb	51.325.440 bits
Virtex-5	2006	2D	240 x 108	550	16 Mb	70.848.000 bits

entrada y salida del dispositivo aumentan. Aunque no viene reflejado en la tabla, el número de pines de entrada y salida de las FPGAs ha ido creciendo, así como la velocidad de transmisión de información a través de ellos, con el fin de proporcionar suficientes pines para las tareas ejecutándose en el dispositivo y para paliar el aumento de tiempo de reconfiguración debido al aumento de tamaño de los mapas de bits.

Intercalada con la familia Virtex, en el año 2003, aparece la familia Spartan de Xilinx, de bajo coste (hoy pueden comprarse por 1$) y menores prestaciones y consumo de potencia que las FPGAs de la familia Virtex, pensada para su utilización en productos de fabricación a gran escala, y cuya existencia es una prueba de la creciente popularidad y uso comercial de las FPGAs.

Merece la pena mencionar también una familia de FPGAs de la que Xilinx no hace mención en su web, la XC6200, y que sin embargo ha sido muy popular entre los investigadores que han trabajado con DHWR en los años 90. La razón por la que se ha utilizado tan extensamente esta familia en el campo de la investigación ha sido la posibilidad (exclusiva de esta FPGA en ese momento) de reconfigurarla dinámicamente en 2D. Se trata de FPGAs de pequeña

capacidad (matrices de CLBs muy sencillos de dimensiones entre 48 x 48 y 128 x 128) que se puede reconfigurar a nivel de celda, a través de un interfaz paralelo llamado *FastMap* y que dispone de varios mecanismos para comprimir mapas de bits de configuraciones y agilizar el proceso de reconfiguración.

En resumen, nos encontramos ante una tecnología novedosa en pleno desarrollo y rápida evolución (solamente ocho años desde la aparición de la primera Virtex en el mercado), con un brillante futuro por delante y todavía mucho camino que recorrer en cuanto a desarrollo de herramientas de diseño, disminución de los tiempos de reconfiguración y mejora en las comunicaciones entre los elementos configurables.

Y siguiendo la intuición de Ross Freeman, que falleció en el año 1989, tan sólo cuatro años después de la comercialización de la primera FPGA de Xilinx, nosotros apostamos también por esta "tecnología del camino medio".

1.2.3. Diseño con FPGAs

El proceso de diseño con diferentes tecnologías HW es conocido como **compilación hardware**. El diseño con FPGAs se enmarca dentro de este contexto. Un algoritmo o tarea debe ser descrito en un lenguaje de alto nivel que esté directamente relacionado con elementos HW para su fácil traducción a un circuito.

En los últimos 15 años se ha popularizado y extendido el uso de los llamados lenguajes de descripción HW (HDL, *Hardware Description Language*), entre los cuales destacan Verilog, [ver08] y VHDL, [vhd06].

El primer paso para compilar una tarea a HW es describir el algoritmo en

alguno de los HDLs disponibles, típicamente en VHDL.

El siguiente paso es determinar qué tipo de bloques HW son necesarios y cómo están conectados entre sí y después, el tercer paso, que consiste en asignar bloques básicos configurables concretos de la FPGA (CLBs) y rutado de señales entre ellos.

En el cuarto paso del proceso se obtiene el mapa de bits de configuración necesario para que los CLBs utilizados en el diseño del circuito realicen cada uno la funcionalidad necesaria.

Por último, es necesario escribir dicho mapa de bits en la memoria de configuración del dispositivo.

La figura 1.6 refleja el proceso explicado.

En la actualidad existen herramientas de compilación HW que traducen de forma automática (y con bastante eficiencia) un algoritmo descrito en VHDL a la arquitectura de FPGA que se esté utilizando, lo que da como resultado un bajo tiempo de compilación HW junto a un aceptable uso de recursos del dispositivo. Además, estas herramientas permiten al diseñador imponer ciertas restricciones al programa a la hora de asignar CLBs, rutado etc.

La parte más delicada del proceso de compilación HW, y la más difícil de realizar de forma automática es, es el rutado de señales dentro de los dispositivos. Por este motivo, los diseños con grandes restricciones temporales generados de forma automática con herramientas de diseño HW suelen ser revisados por el diseñador antes de la generación del mapa de bits de configuración, y el rutado generado de forma automática es frecuentemente retocado de forma manual.

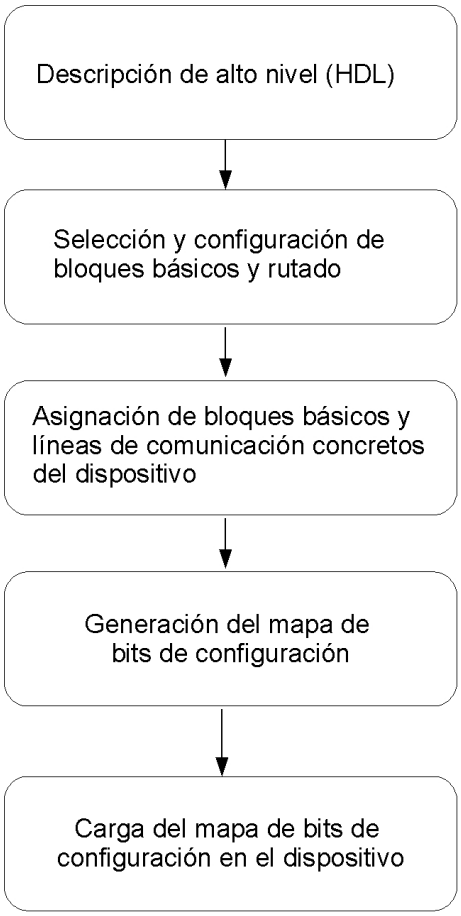

FIGURA 1.6: Etapas de diseño con FPGAs

1.3. Ventajas e inconvenientes de las FPGAs

1.3.1. Ventajas

A pesar de que no carecen de inconvenientes, las características de estos dispositivos los han convertido en la actualidad en una equilibrada solución para las demandas actuales de los sistemas de computación.

A continuación pasamos a enumerar la lista de los beneficios de utilizar FPGAs como parte de un sistema de computación de propósito general.

- **Aumento de la velocidad de procesamiento:** ya que con las FPGAs podemos obtener mejores tiempos de ejecución gracias a dos factores:

    1. **Procesamiento hardware:** la velocidad de ejecución de un circuito hardware específicamente diseñado para realizar una tarea es siempre mayor que la de un código software de alto nivel ejecutado en un procesador de propósito general.
    2. **Explotación de paralelismo:** la posibilidad de descomponer una aplicación en tareas o subtareas y de explotar el paralelismo es una realidad en los dispositivos FPGAs, ya que diferentes tareas pueden ser ejecutadas simultáneamente en el dispositivo, si se dispone de suficientes elementos básicos para configurar los circuitos simultáneamente. La multitarea HW también es factible en los dispositivos actuales, gracias a las técnicas de diseño modular.

- **Reducción de Consumo:** ya que se pueden aplicar técnicas que evitan el consumo de energía innecesario, combinadas con otros factores:

    1. **Avances en la tecnología:** el consumo de las FPGAs ha ido disminuyendo desde su aparición en el mercado, alcanzando unos niveles muy aceptables en el presente. Asimismo existen ya propuestas de novedosas tecnologías que podrían fabricarse en un futuro cercano y que según los resultados presentados por [BTAS07] consumen hasta 25 veces menos que un circuito reconfigurable fabricado con tecnología CMOS.
    2. **Eficiencia de diseño:** debido a la gestión multitarea, solamente

aquellas partes de una aplicación que se necesiten en cada momento estarán siendo ejecutadas en el dispositivo, y por tanto se reduce el consumo al mínimo necesario para la ejecución de una aplicación en cada momento.

3. **Técnicas:** existen numerosas líneas de investigación dedicadas a desarrollar técnicas de trabajo con las FPGAs que permiten ahorrar energía [NB04], [PRMC07].

- **Flexibilidad:** la posibilidad de reconfigurar los elementos básicos, por partes o en su totalidad, dota a los diseños realizados sobre FPGAs de la flexibilidad necesaria para adaptarse a bajo coste a un mercado que cambia con rapidez.

- **Coste:** el aumento en la popularidad de las FPGAs y los avances en la tecnología de fabricación han permitido que los precios de estos dispositivos bajen también a una velocidad que empieza a ser competitiva con respecto a otros dispositivos tradicionales.

A todas estas ventajas podemos sumarle la aparición de herramientas y entornos de desarrollo cada vez más completos y documentados para trabajar con estos dispositivos.

1.3.2. Inconvenientes

Aunque los puntos a favor de utilizar FPGAs son muchos y están sólidamente probados por la literatura, existen sin embargo dos aspectos importantes a tener en cuenta a la hora de su utilización:

1. **Rutado de señales:** como se comentó en la sección 1.2.3, el rutado obtenido a partir de una compilación automática no es el óptimo y suele ser necesaria una revisión por parte del diseñador y la realización de algunos retoques manuales. No obstante, los fabricantes de FPGAs conocen esta dificultad y están trabajando para ofrecer al mercado arquitecturas de FPGAs donde se mejoren las posibilidades de conexión entre celdas lógicas. Algunos autores han propuesto ya diseños en 3D [DWM+05], que ofrecen grandes posibilidades en este aspecto.

2. **Tiempo de reconfiguración:** una de las críticas más usuales al uso de FPGAs dinámicamente reconfigurables para ejecución multitarea HW es el elevado tiempo de reconfiguración del dispositivo o partes de él. Este tiempo está en la actualidad en el orden de los *ms* y es proporcional al área de dispositivo reconfigurada (tamaño del mapa de bits de configuración). Puesto que representa el cuello de botella real de los sistemas multitarea HW, encontramos pruebas de los numerosos esfuerzos dirigidos a mejorar este aspecto, que pasamos a comentar:

   a) **Investigación de nuevas tecnologías:** tal es el caso de [WK07], que han implementado una matriz de puertas dinámicamente reconfigurable con tecnología óptica y tiempos de reconfiguración del orden de los nanosegundos.

   b) **Propuestas de nuevas arquitecturas para FPGAs:** las propuestas de arquitecturas multicontexto, que permiten cargar una nueva configuración en el chip mientras todavía se está ejecutando una tarea en él, permitirían mejorar la velocidad de reconfiguración,

directamente ligada a la lectura del mapa de bits de memoria y su escritura en la FPGA. En trabajos como [MM06] se ha investigado en esta línea.

c) **Desarrollo de técnicas de reconfiguración:** especialmente diseñadas para minimizar el tiempo de reconfiguración. En [LH01] se presentan técnicas de compresión del mapa de bits de configuración y en [LCH00] se ha propuesto utilizar caches para guardarlos y con ello acelerar el proceso de escritura del mapa de bits en la FPGA, así como en [RMC05], donde se han desarrollado técnicas de pre-búsqueda de mapas de configuraciones basadas en una correcta planificación de los módulos a reconfigurar.

1.4. Extensión del Sistema Operativo

Como se mencionó anteriormente, muchos autores proponen la inclusión de dispositivos dinámicamente reconfigurables en sistemas tradicionales con objeto de mejorar su rendimiento. Esto supone un nuevo reto para el diseño de Sistemas Operativos ya que deben ser ampliados con nuevas funcionalidades relacionadas con la gestión del DHWR. Algunos autores han estudiado ya los requisitos de estos nuevos Sistemas Operativos, como es el caso de [DW99] y [WK02].

De hecho, el desarrollo de Sistemas Operativos para DHWR es un área de investigación en auge. Los primeros trabajos teóricos son de finales de los años 90 [DW99]. La dificultad de la tarea reside, entre otros factores, en el rápido avance en la tecnología de FPGAs que, como se revisó en la sección

1.2.2, ofrece cada poco tiempo nuevos y más completos modelos de FPGAs con funcionalidades y características en claro y rápido aumento, lo que por otro lado hace aumentar la complejidad y nivel de experiencia requerido para trabajar con ellas.

En estos casos, el Sistema Operativo (SO) debe decidir en cada momento qué tareas se ejecutarán en la CPU y cuáles en la FPGA, dependiendo de la disponibilidad de recursos (uso de la CPU, uso de la FPGA, tiempo límite para ejecutar una tarea, dependencias entre tareas, disponibilidad del mapa de bits, prioridad etc.). Además el SO dispondrá de un gestor específico para HW, parte del cual puede estar implementado en el propio DHWR.

Dicho gestor hardware es la parte novedosa de este tipo de sistemas y por tanto se hace necesario analizar en detalle cuáles serán las funcionalidades que debe desempeñar.

Se puede imaginar un recurso HW con reconfiguración parcial dinámica como una gran superficie de procesamiento que puede contener un conjunto de tareas, cada una de las cuales ha sido compilada a un mapa de bits reubicable con las herramientas de compilación disponibles, y puede ser cargada en una zona libre de la FPGA cuando se precise su ejecución. Cada tarea HW puede cargarse o abandonar la FPGA sin afectar a las otras tareas que se están ejecutando, de la misma manera que ocurriría en un sistema SW multitarea. Si la tarea necesita parámetros de entrada o produce resultados entonces hay que proporcionarle un mecanismo que la conecte con los pines de entrada y salida del circuito.

Veamos a continuación algunas de las funciones que consideramos sería preciso realizar para permitir que el SO gestione unos recursos HW dinámicamente

reconfigurables de gran capacidad:

- **Compilación de las tareas a código HW reubicable:** Las tareas que puedan ser susceptibles de ejecutarse en HW deberán haber sido compiladas para el HW objetivo, dando como resultado un código HW reubicable que pueda ser luego cargado donde se disponga de espacio libre.

- **Mantenimiento de la información sobre los recursos HW disponibles en cada momento:** El SO debe disponer en todo momento de la información necesaria acerca del estado de los recursos HW, para poder tomar decisiones relativas a su gestión.

- **Control de admisión/planificación de tareas HW:** En el contexto de un SO multitarea, la decisión de admisión o planificación de las tareas HW deberá basarse en criterios como secuenciamiento, disponibilidad de recursos, penalización en el rendimiento total en caso de no ejecutarse, tiempo máximo de ejecución de tareas, etc. Las consideraciones de prioridad podrían hacer incluso que algunas tareas menos prioritarias en ejecución salgan de HW temporalmente para dejar paso a otras más urgentes.

- **Asignación de los recursos HW concretos en los que se va a ubicar cada tarea:** Partimos, como ya hemos dicho antes, de que el código HW de la tarea es de naturaleza reubicable, por lo que podemos elegir la ubicación que más nos convenga. Esta función puede conllevar una **defragmentación** previa, si se comprueba la existencia de recursos

suficientes, pero distribuidos por el dispositivo.

- **Carga del código HW reubicable:** La carga del código HW, que conlleva la transferencia desde la memoria en la que se almacena hasta el HW reconfigurable (incluyendo la traducción de código reubicable a código absoluto, a partir de las decisiones de asignación de HW), y la consiguiente reconfiguración dinámica de parte del mismo, es la tarea que supone un mayor retardo.

- **Liberación del HW ocupado por tareas que han finalizado:** Tendría lugar cuando se detecta la finalización de una tarea, y conllevaría la actualización de la información disponible por el SO. Este dinamismo frecuentemente da lugar a la fragmentación del espacio libre, haciendo necesario en algunos casos llevar a cabo un proceso de defragmentación. Es posible también tener en cuenta otro tipo de consideraciones, como por ejemplo, no liberar el HW ocupado por una tarea si se prevé que puede ser vuelta a cargar y no es preciso reutilizar el espacio de HW reconfigurable que ocupa. En esta aproximación, la información del sistema sobre la ocupación del HW reconfigurable debería distinguir entre recursos libres, usados, y ocupados pero no usados. Esta distinción, que puede permitir eliminar la necesidad de la reconfiguración en algunos casos, complica notablemente la tarea de defragmentación, que se comenta a continuación.

- **Defragmentación de HW:** mediante reubicación de las tareas HW ya en ejecución para permitir el reagrupamiento del HW fragmentado a fin de que pueda ser aprovechado de manera más eficiente.

Como ya se ha mencionado al principio de esta sección, buena parte de las funciones y estrategias que se han ido proponiendo podrían ser implementadas en el propio DHWR, posiblemente incorporadas a su arquitectura interna.

1.5. Objetivos de este trabajo

Como se ha explicado en la sección 1.1, estamos trabajando en una línea de investigación con DHWR como solución a las crecientes necesidades en los sistemas de computación de propósito general. Y entre los DHWR existentes hemos optado por las FPGAs, por sus muchas ventajas frente al resto de dispositivos reconfigurables.

Para poder gestionar una FPGA como parte de un sistema de computación general, es necesario extender las funcionalidades tradicionales del SO. Una de las tareas que debe realizar un SO que gestione DHWR es la de mantener información actualizada del espacio libre en el dispositivo y gestionar dicho espacio, es decir, asignar recursos a las nuevas tareas que se van a ejecutar en él.

El presente trabajo se ha centrado en algunas de las funciones que el gestor de HW necesita realizar, en concreto: el mantenimiento de información sobre el dispositivo hardware, el control de admisión / planificación de ejecución de tareas y la ubicación de las tareas en la FPGA. La funcionalidad de defragmentación de hardware deja de ser necesaria en un modelo de gestión de área de FPGA como la que proponemos aquí.

Dentro de esta línea de investigación existen numerosos trabajos que tratan de resolver este problema y que explicamos en detalle en el capítulo 2. Existen

dos tendencias dentro del grupo de investigadores de este campo: una cuyo objetivo es aprovechar al máximo el área del dispositivo, lo cual obliga a utilizar estructuras de datos complejas y algoritmos de búsqueda de espacio libre lentos y complejos, y otra, dentro de la cual se encuentra el trabajo presente, que busca sencillez y rapidez de respuesta ante la llegada de una nueva tarea sin por ello perder eficiencia en el uso del dispositivo HW.

Los algoritmos de ubicación complejos tienen además el inconveniente de que asumen que una tarea se puede ubicar en cualquier posición dentro de la FPGA y esto en la actualidad resulta del todo alejado de las posibilidades reales de reconfiguración parcial dinámica de los dispositivos existentes. Presentan también el inconveniente de que dificultan la entrada / salida de datos. Además las nuevas FPGAs que están saliendo al mercado y las herramientas de diseño disponibles no solamente no lo permiten sino que no dan lugar a pensar que en corto o medio plazo esto pueda ser así.

Nuestro trabajo se ha centrado en desarrollar un planificador que asigne una ubicación para que una tarea pueda ejecutarse en una FPGA en la que hay otras tareas ejecutando, sin perturbar su funcionamiento (multitarea hardware) y nos hemos propuesto que cumpla los siguientes objetivos:

1. **Sencillez:** estructuras de datos sencillas para mantener la información actualizada respecto al espacio libre en la FPGA, fáciles de implementar en un sistema real.

2. **Rapidez de respuesta:** el algoritmo de asignación de espacio para una nueva tarea en la FPGA debe ser rápido y fácil de implementar.

3. **Realismo:** el algoritmo propuesto debe estar basado en las posibilidades

reales de la tecnología existente y poder ser implementado en las FPGAs actuales.

4. **Eficiencia:** el algoritmo no puede sacrificar los costosos recursos HW por su sencillez de planteamientos.

5. **Flexibilidad:** el algoritmo debe ser capaz de auto-ajustarse cuando las circunstancias así lo requieran, para aprovechar al máximo los recursos disponibles.

El resto del texto que presenta nuestro trabajo se ha estructurado de la siguiente manera: en el siguiente capítulo presentaremos el trabajo previo relacionado con nuestro tema de investigación. En el capítulo 3 describiremos el modelo de sistema utilizado junto con el entorno planificador básico. En el capítulo 4 expondremos el estudio realizado para implementar la adaptación dinámica del entorno a diferentes circunstancias y a continuación, en el capítulo 5 mostraremos en detalle cómo se ha implementado dicha funcionalidad.

A continuación, en el capítulo 6, presentamos los resultados experimentales que demuestran la eficiencia del algoritmo con y sin la funcionalidad de adaptación dinámica, y en el último capítulo las conclusiones obtenidas a partir de la realización de esta investigación y algunas líneas de trabajo futuro que de ella se podrían derivar.

Capítulo 2

Estado del arte

En este capítulo revisaremos el trabajo previo relacionado con nuestro tema de investigación. Presentaremos las principales aportaciones en el campo de los algoritmos de admisión, planificación y ubicación de tareas HW en un DHWR (que en adelante restringiremos a FPGAs).

Las primeras propuestas para este tipo de algoritmos son de los años 90 y muchos de ellos están basados en estudios matemáticos relacionados con el problema de empaquetamiento o *bin-packing*, aunque es preciso observar que en ninguno de los casos que mencionaremos se conoce *a priori* el conjunto de tareas que se van a ejecutar en la FPGA ni sus características. Se trata de resolver el problema de empaquetamiento "sobre la marcha" (técnicamente, *on-line*) y podríamos hacer un símil con el juego de tetris, en que figuras de formas diversas van cayendo y es preciso buscarles el mejor lugar posible en el espacio dejado por el montón de figuras que cayeron anteriormente y cuyo perfil va cambiando a lo largo del juego.

A diferencia de los algoritmos para *bin-packing*, en el caso de nuestros

algoritmos únicamente se dispone de información acerca de la tarea que ha llegado a la FPGA para su ejecución y se puede calcular el espacio disponible en la FPGA en ese momento y el que quedará libre cuando terminen de ejecutarse las tareas presentes en la FPGA. Cualquier decisión que se tome respecto al lugar donde dicha tarea se ubicará se hace sin conocimiento alguno de cuándo llegará la siguiente tarea ni cuál será su tamaño ni su tiempo de ejecución.

En la mayoría de los casos los autores trabajan con un modelo de tarea rectangular y los únicos cambios de forma permitidos se limitan a la rotación. En muy pocos casos los autores consideran tareas no rectangulares y cambios de forma.

Las metodologías utilizadas para resolver el problema son diversas pero pueden clasificarse en dos grupos claramente diferenciables: los algoritmos complejos y los algoritmos sencillos.

Llamamos algoritmos complejos (y añadimos la coletilla "avariciosos en uso de área") a aquellos algoritmos que dan prioridad al uso intensivo de área del dispositivo y proponen soluciones que tratan de optimizar el uso de área a costa de utilizar estructuras de datos de gran complejidad para representar el espacio libre en la FPGA y que precisan de algoritmos de complejidad alta para localizar un espacio libre que pueda albergar la tarea. El mantenimiento de la información del espacio libre en la FPGA (actualización de información cuando una tarea termina su ejecución) es también muy costoso en cálculos.

Las propuestas de algoritmos sencillos se centran en la rapidez de respuesta, una eficacia de uso de área competitiva y el pragmatismo (implementación sencilla y realista).

Pasamos ahora a revisar los trabajos publicados en la línea de algoritmos

complejos.

2.1. Algoritmos complejos, avariciosos en uso de área

2.1.1. Trabajo de O. Diessel

El pionero en este tipo de estudios es O. Diessel, de la Universidad de South New Wales (Australia), que escribió su tesis doctoral en el año 1998 [Die98]. Es un trabajo muy completo donde se presentan tres técnicas diferentes: dos de ellas para representar el espacio libre en la FPGA (una basada en una estructura en árbol y la otra en un grafo de visibilidad) y una tercera para 1D donde se propone una modificación arquitectónica que permite determinar el número de columnas libres contiguas.

Aunque este trabajo se centra en resolver el problema de la correcta representación del espacio libre disponible en la FPGA y la forma de asignar espacio para las nuevas tareas, propone también métodos de defragmentación asociados con las heurísticas de planificación que permiten agrupar el espacio libre de la FPGA.

Tanto en su tesis como en su artículo [DE01], el autor presenta dos heurísticas de representación del espacio disponible en la FPGA cuyo objetivo es representar con fidelidad la situación actual del dispositivo y permitir la aplicación de métodos eficaces que determinen si existe la posibilidad de compactar espacio y de aplicar heurísticas que determinen la mejor manera de hacerlo (mover el mínimo de tareas y causar el mínimo retraso en su ejecución).

El autor parte de un modelo de FPGA en el que se asume la posibilidad de insertar una tarea para ejecución en cualquier posición del dispositivo.

El primero de los algoritmos propuestos por Diessel utiliza una estructura en árbol para almacenar la información de área de FPGA disponible y debe recorrerlo para encontrar un hueco adecuado donde situar cada nueva tarea HW que llega al DHWR.

Dicha estructura se muestra en la figura 2.1: en cada nivel del árbol (siendo el raíz todo el área de la FPGA) cada rectángulo se divide en cuatro rectángulos menores, cada uno de los cuales es un nodo que se marca como libre, ocupado o parcialmente libre. Dicho árbol jerárquico se recorre con rapidez, pero esta estructura puede dar lugar a situaciones en las que exista un hueco libre suficientemente grande y que no se identifique debido a que esté repartido entre nodos/ramas diferentes del árbol.

El orden de complejidad del algoritmo que construye el árbol es de $O(m \cdot n)$ (donde n es el número de tareas y m la dimensión de la FPGA) y hay que repetirlo cada vez que una tarea termina su ejecución para mantener actualizada la información acerca del uso de área de la FPGA. Esta heurística, como hemos explicado anteriormente, debe inevitablemente actuar conjuntamente con otra que permita defragmentar la FPGA, y para ello propone una técnica de reubicación con una complejidad de $O(n^3)$, donde n es el número de tareas ejecutándose en la FPGA.

El segundo de los algoritmos utiliza un grafo de visibilidad, figura 2.2, en el que se dispone de la información de la posición de cada tarea en la FPGA y las posibles posiciones que podría desplazarse hacia su derecha. Recorriendo el grafo en orden inverso puede determinarse la máxima compactación que se

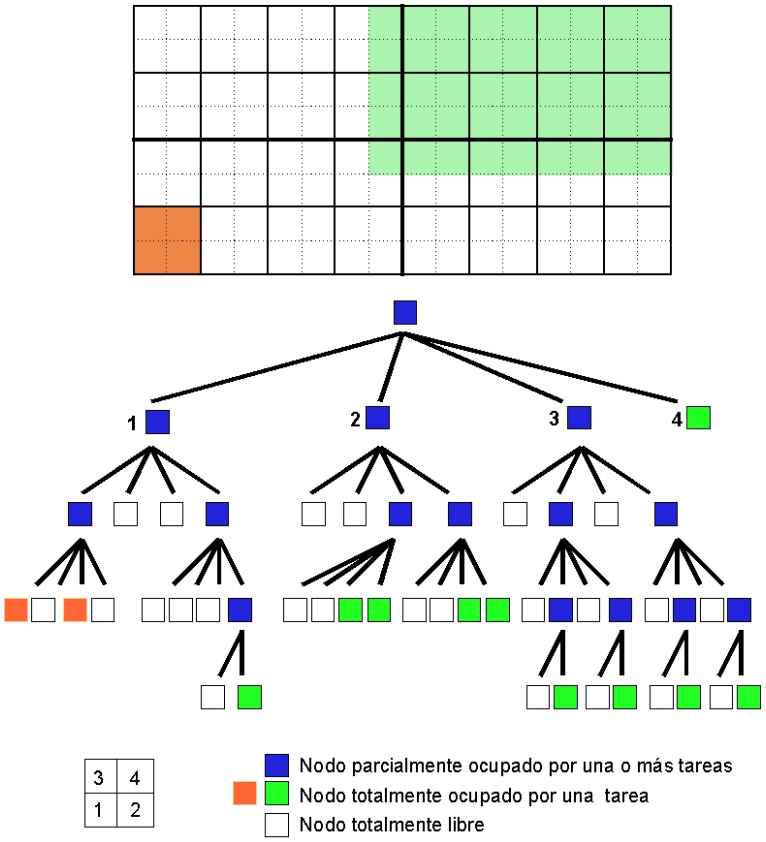

FIGURA 2.1: Estructura de árbol propuesta por Diessel

puede conseguir en la FPGA y a partir de ahí determinar si quedaría suficiente espacio libre en la FPGA tras la defragmentación.

En otro artículo [BD01] Diessel y sus colaboradores presentan una propuesta de implementación en 1D del último algoritmo mencionado, con HW específico para determinar el número de columnas contiguas libres en una FPGA y la posibilidad de aplicar el método de compactación (ligado a la heurística del grafo de tareas).

Destacamos el hecho de que en sus publicaciones más recientes, el grupo de Diessel está trabajando en temas más cercanos a la implementación y poniendo mucho énfasis en solucionar los problemas relacionados con las co-

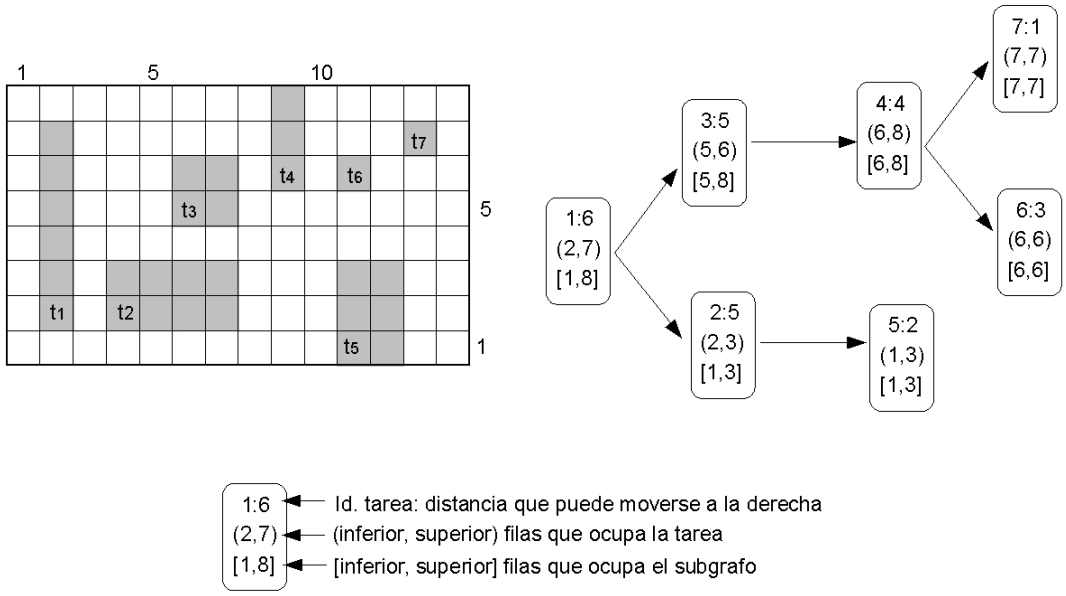

FIGURA 2.2: Grafo de visibilidad propuesto por Diessel

municaciones entre tareas y de las tareas con los pines de entrada/ salida. En su artículo [KD06] estudian las posibilidades reales de implementación de multitarea hardware en una Virtex-4 y proponen una gestión del espacio de la FPGA en 2D.

2.1.2. Trabajo de K.Bazargan

Otros autores, como K.Bazargan, de la Universidad de Minnesota (EEUU), utilizan heurísticas de *bin-packing* y aplican algunos de los algoritmos clásicos *on-line* y *off-line* (donde se conoce de antemano el conjunto de tareas a ejecutar) que existen para dicho problema teórico, pero todos tienen un orden de complejidad muy alto y se basan también en la hipótesis de que las tareas se pueden insertar en posiciones arbitrarias de la FPGA. Centraremos el estudio de este autor en las heurísticas *on-line* que ha presentado.

La figura 2.3 muestra un ejemplo de la técnica utilizada por Kia Bazargan en [BKS00] y [BS99] para representar el área libre de la FPGA, basada en un conjunto de Rectángulos Máximos Disponibles, los MER (*Maximum Empty Rectangles*), que en la figura aparecen nombrados por letras. Estos rectángulos representan máximas áreas libres en la FPGA y pueden estar solapadas entre sí.

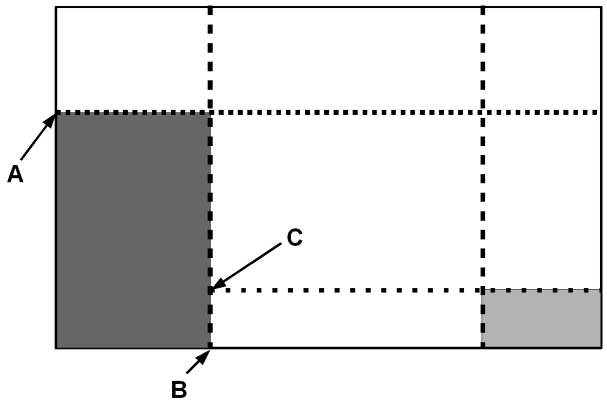

FIGURA 2.3: Organización en MER propuesta por Bazargan

El sistema dispone de una lista de MER. Cuando una nueva tarea llega al sistema, se recorre la lista en busca de un MER que sea suficientemente grande como para albergarla. A diferencia de la propuesta de O. Diessel, si no existe hueco en la FPGA, la tarea es rechazada, no se aplica ninguna técnica de defragmentación para hacerle hueco.

Si existen varios posibles MER en los que pueda ubicarse la tarea, se aplica una heurística de selección que puede ser desde la opción más sencilla, el primero encontrado, hasta una opción compleja que analiza cuál de ellos es mejor. Una vez seleccionado el rectángulo donde se ubicará la tarea, ésta se sitúa en la esquina inferior izquierda del rectángulo y se procede a la actualización de la

lista de MER. Esta lista también debe ser actualizada cada vez que una tarea termine su ejecución.

Como podemos observar, se trata de nuevo de un algoritmo de elevada complejidad. Solamente el mantenimiento de la lista actualizada de MER tiene complejidad $O(n^2)$, donde n es el número de tareas que se están ejecutando en la FPGA.

Para obtener un algoritmo de menor complejidad el autor propone una heurística en la que no se permite que los rectángulos vacíos se solapen. El resultado es un algoritmo de complejidad $O(n)$ y que resulta más rápido en su ejecución pero que pierde eficiencia en uso de área, ya que a veces existe espacio libre no identificado por los rectángulos marcados.

2.1.3. Trabajo de H. Walder (I)

Un autor que también ha trabajado en este problema es H. Walder, del Instituto Federal de Tecnología (Suiza). En un principio, es el único que considera tareas que pueden ser de forma no rectangular y tiene en cuenta posibles cambios de forma [WP02]. Este autor supone que las tareas están formadas por sub-tareas rectangulares (figura 2.4) y estudia los cambios de forma en función de las diferentes combinaciones posibles de subtareas, aunque abandonaron esta interesante idea por su complejidad.

En la publicación [WSP03] presenta una heurística basada en la de Bazargan en la que retrasa la decisión de dividir el área libre en rectángulos máximos no solapados y utiliza una tabla que permite un rápido acceso a la lista de rectángulos. La figura 2.5 nos muestra un ejemplo de la gestión del área libre

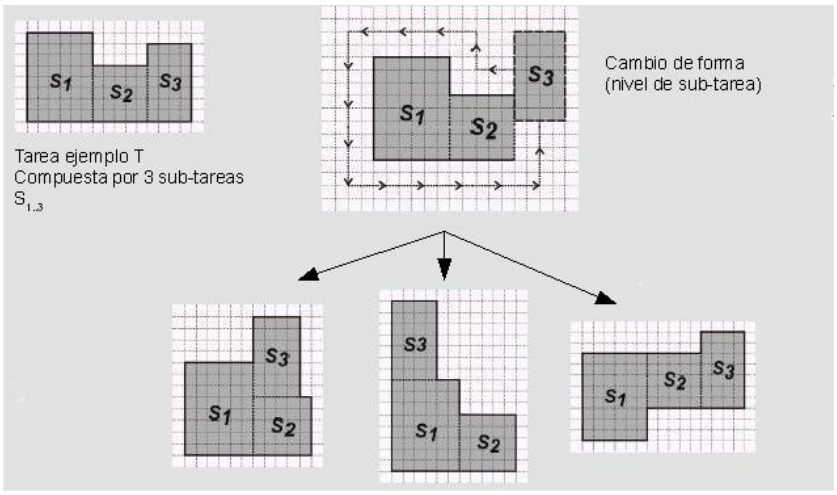

FIGURA 2.4: Cambios de forma, Walder

propuesta por este autor para 2D. Partiendo de una ocupación de la FPGA como la representada en 2.5a compara la técnica de Bazargan, que daría lugar a una división del área libre como la mostrada en 2.5b tras la llegada de una nueva tarea, con su propuesta, mostrada en 2.5c, que permite retardar la decisión de la división en MER hasta el momento de insertar la tarea.

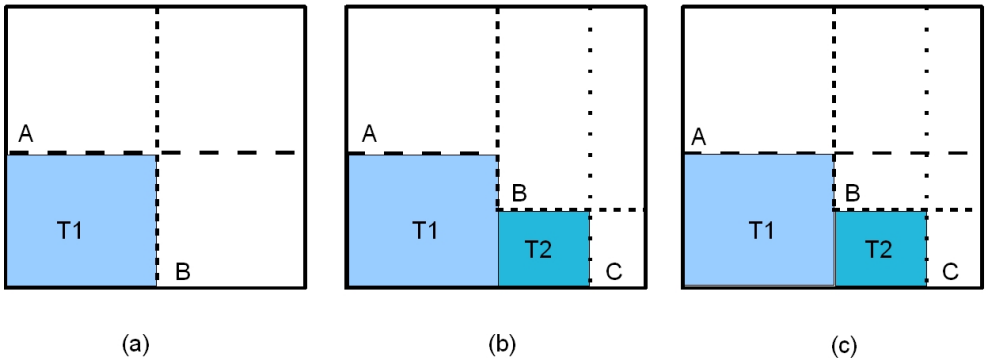

FIGURA 2.5: División del área libre en rectángulos: Walder

Esta propuesta está ampliamente desarrollada en otro trabajo, presentado en [SWP04] en el que presenta dos técnicas de planificación, cada una de ellas aplicada a 1D y 2D.

La primera técnica, llamada horizonte (del inglés *horizon*) utiliza tres listas de tareas: una de ellas contiene la información de las tareas actualmente en ejecución, otra de tareas en espera de ser ejecutadas y la tercera contiene un "horizonte de planificación", que contiene los intervalos (para el caso 1D) o los rectángulos máximos (para el caso 2D) que van a ser liberados en el futuro, ordenados por tiempo creciente de liberación del espacio. De esta forma se mantiene una información completa del espacio libre en la FPGA, que es consultada en el momento de la llegada de cada nueva tarea para comprobar si se puede ejecutar o no dentro de sus restricciones temporales.

La segunda técnica, llamada empaquetamiento (del inglés *stuffing*) es una técnica más compleja que la anterior, que utiliza también tres listas: la de tareas en ejecución, la de tareas en espera y la lista de espacio libre. En esta última se guarda la información de intervalos (para 1D) o rectángulos (para 2D) que están libres, ordenados ahora por su localización en la FPGA (de menor a mayor posición en el eje horizontal). Para dividir el área libre en rectángulos se basan en el trabajo presentado en [WSP03] y ya comentado.

2.1.4. Trabajo de A. Ahmadinia

Otro grupo que también ha realizado contribuciones muy interesantes en este tema es el de A. Ahmadinia, actualmente en la Universidad de Edimburgo (Reino Unido). Sus primeros trabajos publicados sobre algoritmos de ubicación de tareas son de 2004.

En uno de ellos trabaja sobre ideas de Bazargan [ABK+04], mientras que en el otro propone algunas ideas diferentes y originales [ABT04].

En [ABK+04] Ahmadinia describe la situación de la FPGA a partir de los rectángulos ocupados en lugar de los libres. Se basa en que el número de rectángulos libres crece con mayor rapidez que el de rectángulos ocupados y con ello consigue reducir la complejidad del algoritmo a O(n), donde n es el número de tareas ejecutándose en la FPGA.

La primera parte del algoritmo consiste en definir los IPR (*Impossible Placement Region*), zona rayada de la figura 2.6, que están formados por las áreas que ocupan las tareas y el área alrededor de ellas que produciría solapamiento con la nueva tarea a ubicar. Los IPR por tanto son dependientes de cada nueva tarea que llega y deben ser calculados cada vez que se quiere ubicar una nueva tarea en la FPGA. Se dispone de una lista de IPRs que se debe recorrer para comprobar si existe suficiente espacio libre en la FPGA, figura 2.6.

La segunda parte del algoritmo busca el mejor punto de ubicación en la FPGA dentro de los posibles, en función de la posición de los pines de E/S asignados a la tarea y de las posibilidades y necesidades de comunicación con otros módulos, a través de la evaluación de una función de coste de rutado.

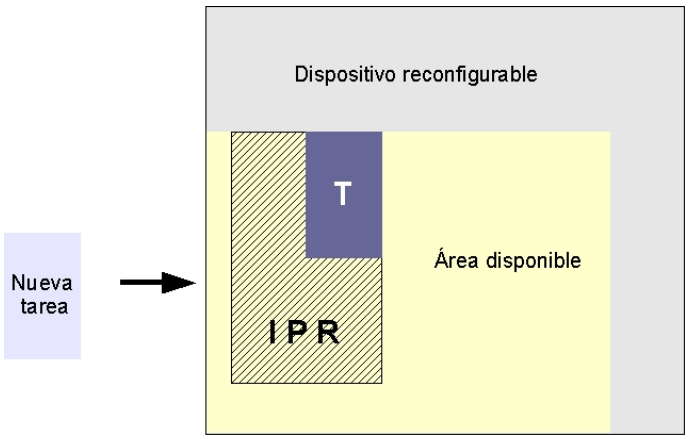

FIGURA 2.6: Ahmadinia: IPRs

Como vemos, este algoritmo también requiere de complicados cálculos para mantenimiento de la información del estado de la FPGA y la asignación de espacio a una tarea, y además también presupone que una tarea puede ubicarse en cualquier posición de la FPGA.

En su otro artículo del 2004, [ABT04] Ahmadinia propone agrupar las tareas en función del tiempo en que terminarán su ejecución (para evitar la fragmentación) y propone dividir la FPGA en franjas temporales que alberguen tareas que terminen en tiempos parecidos, dejando así todo el espacio libre a la vez, como muestra la figura 2.7. La FPGA queda entonces dividida en *slots* temporales, ya que cada nueva tarea que llega es ubicada en función de la proximidad de su tiempo de fin de ejecución con las que se están ejecutando en la FPGA.

Estos *slots* se van rellenando con tareas que terminarán más o menos a la vez y dejarán espacio libre contiguo en la FPGA cuando terminen su ejecución. Así la FPGA se irá ocupando y desocupando por franjas y se reduce el problema de la fragmentación del espacio disponible.

En el ejemplo mostrado en la figura 2.7 vemos que dos tareas con tiempo de fin parecido a $tfin_1$, que define el primer *slot*, son ubicadas en la parte izquierda de la FPGA. A continuación se han ubicado tres tareas con tiempos de fin parecidos entre sí, mayores a $tfin_1$ y menores o iguales a $tfin_2$, que define el fin del segundo *slot*. De forma análoga se ubican tareas en el tercer *slot*.

Cuando las tareas que estaban ejecutándose en el primer *slot* terminan su ejecución, el espacio de la izquierda de la FPGA queda libre para ejecutar nuevas tareas que tengan un tiempo de fin de ejecución parecido y que estará

definido por un nuevo tiempo $tfin_4 > tfin_3 > tfin_1$.

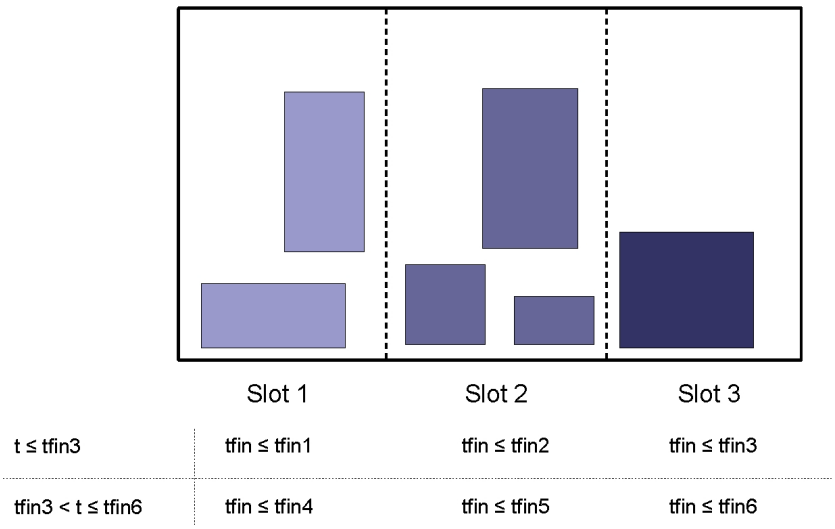

FIGURA 2.7: Ahmadinia: franjas temporales

En otro trabajo más reciente [ABF+07] el autor propone almacenar información acerca del contorno del espacio libre en la FPGA, aunque el resultado es también un algoritmo de complejidad alta, O($n \cdot logn$).

Por último mencionamos un trabajo en el que el autor y sus colaboradores han desarrollado un computador basado en una FPGA dinámicamente reconfigurable, la Virtex-2 [MTAB07]. Nos interesa destacar que de nuevo los autores utilizan un modelo de gestión del área similar al propuesto por nosotros cuando se trata de implementar un sistema real. En este caso, al tratarse de una FPGA reconfigurable en 1D, dividen el área que van a dedicar a ejecución de tareas en tres particiones de igual tamaño.

2.1.5. Trabajo de J. Tabero

Queremos también mencionar el trabajo del grupo de J. Tabero, de la Universidad Complutense de Madrid (España), publicado en [TSMM08] y [TSMM06], que utiliza una lista de vértices para representar el contorno del espacio libre en la FPGA, que combinada con diferentes heurísticas se utiliza para seleccionar la mejor ubicación dentro de las posibles para cada nueva tarea.

Las heurísticas utilizadas son dos:

1. **Basada en la fragmentación** producida por una determinada ubicación de la tarea en la FPGA, según la fórmula

$$F = 1 - \prod_i [\left(\frac{4}{V_i}\right)^n \cdot \frac{A_i}{A_{F_{FPGA}}}]$$

   donde V_i es el número de vértices, A_i es el área de la tarea, $A_{F_{FPGA}}$ es el área libre de la FPGA y i es el número del hueco.

   En esta fórmula, el término entre paréntesis representa la adecuación de la forma del hueco i para ubicar tareas rectangulares y el otro término representa el área del hueco normalizado.

2. **Basada en el estudio de adyacencia** de la tarea con respecto a las tareas que se están ejecutando en la FPGA según la expresión

$$Ady2D = \sum_h (VL_{lim} \bigcap lim\, T_{N_h})$$

   donde VL_{lim} es la lista de vértices que limitan con la tarea y $limT_{N_h}$ es

cada una de las aristas de la tarea. Esta expresión formaliza la idea de que el vértice seleccionado es el que permite mejor área de contacto con los límites de las tareas que se están ejecutando y con el área libre en la FPGA.

Estas heurísticas se amplían también al estudio 3D, en donde se tiene en cuenta el volumen de las tareas en la FPGA, siendo el tiempo la tercera dimensión considerada.

La figura 2.8 muestra un esquema del entorno utilizado por este autor. Cuando llega una nueva tarea al sistema, el planificador de tareas ejecuta el módulo selector de vértices para averiguar si existe espacio disponible en la FPGA para ejecutar la tarea de inmediato. Si no hay espacio, la tarea queda almacenada en una cola para su posterior ejecución. Las tareas en la cola de espera, Q_w, están ordenadas por su tiempo máximo de espera permitido.

Si hay espacio disponible, se utiliza una de las heurísticas mencionadas para seleccionar la mejor ubicación posible para la tarea. El módulo encargado de cargar el mapa de bits en la FPGA es el cargador / extractor.

Otro módulo, el analizador de la lista de vértices, monitoriza la fragmentación existente en la FPGA y en los casos en que detecte que se necesita realizar una defragmentación, ejecuta el módulo defragmentador, que reubica las tareas en la FPGA con el objetivo de que el espacio libre esté contiguo y pueda ejecutarse alguna de las tareas que están esperando en la cola o la ejecución inmediata de una nueva tarea que llegue al sistema.

Como vemos, se trata de una idea muy original pero que también da lugar a un algoritmo muy complejo.

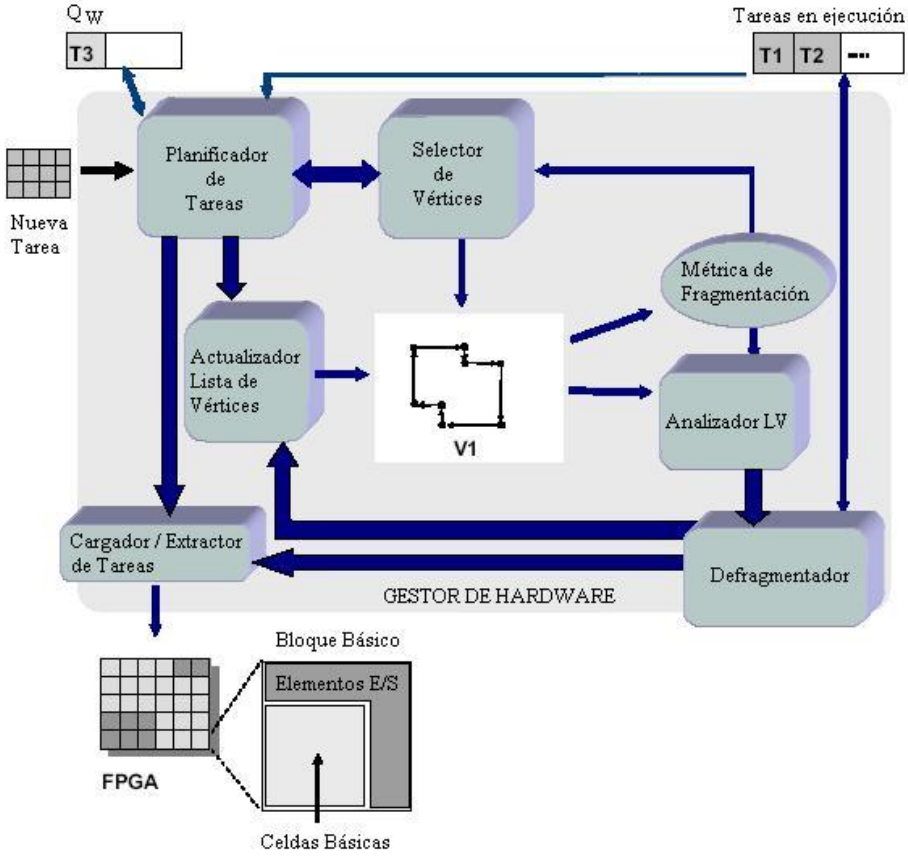

FIGURA 2.8: Esquema del entorno propuesto por Tabero

2.1.6. Trabajo de H. Kalte

H. Kalte, de la West University (Australia), ha realizado también algunos trabajos de investigación sobre algoritmos que representan el espacio libre en la FPGA y deciden la ubicación de las nuevas tareas. Se han centrado en el problema en 1D y los algoritmos propuestos son de complejidad alta.

Este grupo ha insistido fundamentalmente en el problema de la defragmentación del espacio de la FPGA y la reubicación de tareas que se están ejecutando.

En su artículo [KKK+04] proponen una estrategia que tiene en cuenta la flexibilidad a la hora de utilizar el espacio de la FPGA y el problema de las comunicaciones. Proponen una infraestructura de comunicaciones basada en la utilización de líneas horizontales que atraviesan la FPGA y que por tanto permiten el intercambio de datos y señales de control entre las tareas ubicadas en 1D, es decir, que ocupan la altura total de la FPGA y varían en anchura. Miden el rendimiento de su propuesta en el uso efectivo de CLBs de la FPGA y la comparan con las heurísticas 2D, consiguiendo mejoras con respecto a estas y sin necesidad de recurrir a la defragmentación.

En otro artículo, [KPK05], proponen dos heurísticas diferentes para 1D, aunque son también aplicables a 2D.

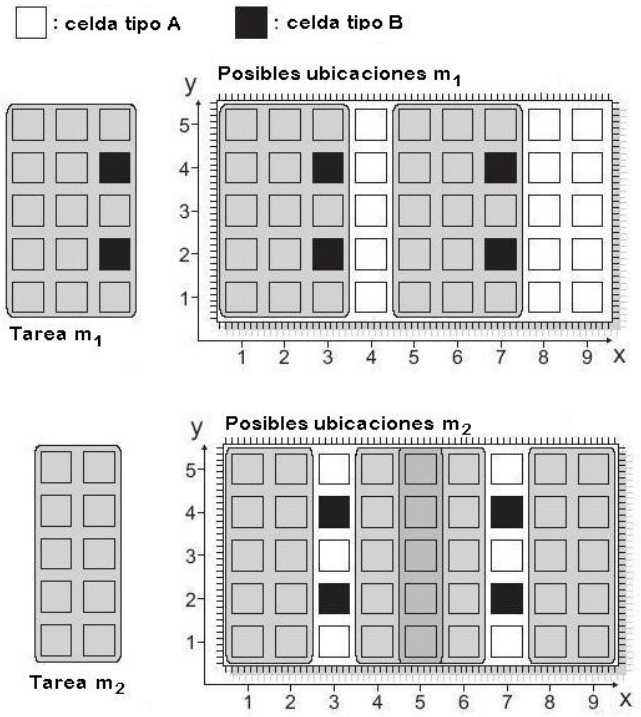

FIGURA 2.9: Gestión del espacio propuesta por Kalte

La primera de ellas examina todas las posibles ubicaciones de una tarea y calcula la probabilidad estática de utilización, determina los pesos de las diferentes posibles ubicaciones y las ordena de menor a mayor peso. En tiempo de ejecución, se examinan las ubicaciones posibles ordenadas por pesos y se selecciona la primera que esté disponible, figura 2.9. La segunda heurística utiliza ideas similares a la de la primera y es una versión mejorada en la que los pesos se re-calculan cada vez que una tarea termina su ejecución o una nueva comienza a ejecutarse en la FPGA. Las tareas se sintetizan para ocupar unas determinadas columnas de la FPGA pero en tiempo de ejecución el algoritmo puede desplazarlas para ajustarlas al espacio disponible.

En su artículo más reciente, [KKP06], este autor y su grupo se centran en el problema de defragmentación y re-ubicación de tareas en tiempo de ejecución. Hacen hincapié en los aspectos HW a solucionar para poder llevar a cabo la interrupción y reanudación de la ejecución de tareas sobre DHWR.

2.1.7. Trabajo de M. Handa

M. Handa, de la Universidad de Cincinnati (EEUU), ha trabajado también sobre este mismo problema. En su publicación [HV04] presentan un algoritmo que utiliza una estructura en escalera para representar el espacio libre en la FPGA. Su trabajo también se basa en el realizado por Bazargan, y se trata de una lista de MER representada de forma que resulta más rápido y sencillo localizarlos.

Una escalera está formada por todos los rectángulos libres que tienen en común su esquina inferior derecha, que se denomina origen de la escalera, según

muestra la figura 2.10. En esta figura el espacio libre de la FPGA ha quedado representada por varias escaleras, una de ellas con origen en el punto P, otra con origen en el punto M y otra en O. Estas dos últimas solamente contienen un escalón, mientras que P tiene tres escalones definidos por los puntos A, B y C.

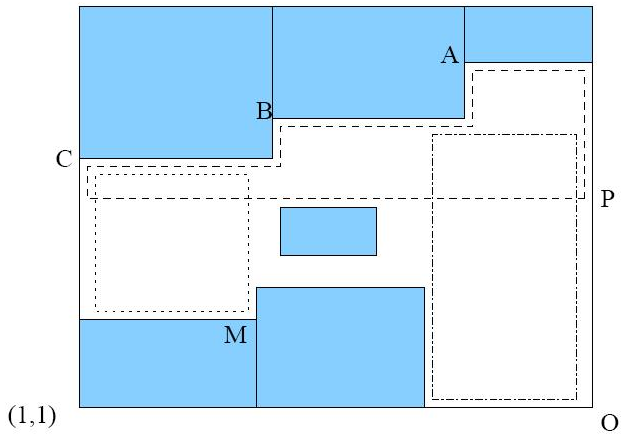

FIGURA 2.10: Estructura en escalera propuesta por Handa

Una vez determinadas todas las escaleras, es necesario detectar cuáles contienen MER (rectángulos máximos), es decir, las escaleras máximas, lo que reduce el proceso de búsqueda de posibles ubicaciones de las nuevas tareas, ya que solamente se consideran los MER.

La complejidad resultante de este algoritmo es también alta, $O(W * H)$, donde W es el número de filas de CLBs y H es el número de columnas.

No dan detalles acerca de cómo se selecciona uno de los MER para la ubicación de una tarea, y tampoco de la complejidad del algoritmo necesario para cargar una tarea en la FPGA, una vez seleccionada su ubicación.

2.1.8. Conclusiones: algoritmos complejos

Una vez presentados los principales autores que han trabajado en este tema, estamos en condiciones de hacer una serie de observaciones generales, aplicables a todos ellos:

- **Alta complejidad:** todos los autores mencionados han presentado ideas muy interesantes y eficientes pero de complejidad muy alta, y no siempre garantizando que el método empleado para localizar área libre en el dispositivo sea eficiente en el 100 % de los casos. Esta complejidad hace que los retardos asociados a la gestión de la ubicación de tareas penalice a las mismas, impidiendo algunas veces que cumplan sus restricciones temporales.

- **Ideas alejadas de las posibilidades actuales:** todos los autores presuponen que una tarea se puede ubicar en posiciones arbitrarias en una FPGA. Sin embargo esto, de momento, es una hipótesis alejada de las posibilidades reales de reconfiguración parcial dinámica en los dispositivos existentes, como prueban además los mismos autores, que en sus trabajos más recientes y dedicados al estudio de los problemas reales de implementación de sistemas computacionales basados en DHWR no recurren a los algoritmos propuestos en sus publicaciones anteriores para gestionar el área de los dispositivos sino a ideas más sencillas como las que se van a presentar a continuación.

2.2. Algoritmos sencillos

2.2.1. Trabajo de P. Merino

Dentro del grupo de algoritmos sencillos encontramos menos cantidad de trabajos. Destacamos la propuesta de P. Merino, de la Universidad de Málaga (España), [MLJ98a] y [MLJ98b], del año 1998 y cuya línea de investigación fue abandonada. Proponen una idea muy sencilla para la división del espacio disponible en la FPGA: cuatro particiones de igual área, según muestra la figura 2.11.

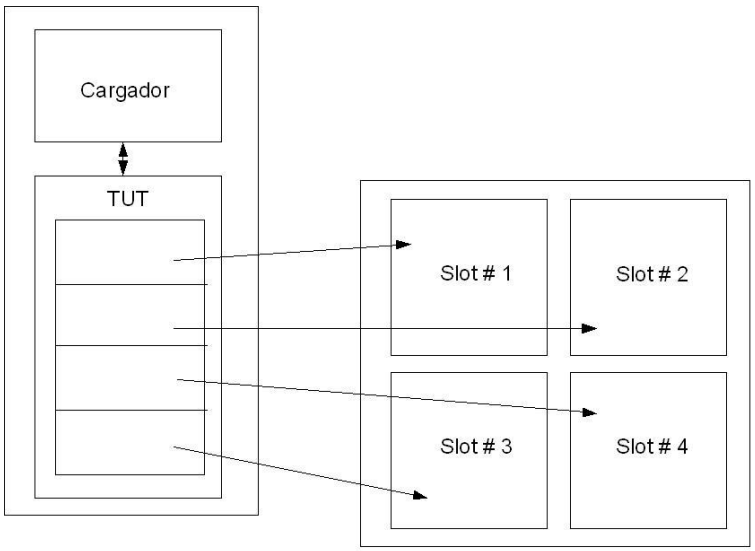

FIGURA 2.11: Propuesta de Merino

Tanto la estructura de datos como el algoritmo utilizados en este modelo son muy sencillos y fáciles de implementar. Las tareas que van a ser ejecutadas en la FPGA se almacenan en una tabla, la TUT (tabla para ubicación de tareas) y de ahí pasan a ejecutarse en una de las particiones de la FPGA.

Las ventajas y graves inconvenientes de este sistema resultan bastante ob-

vias: por un lado vemos que se trata de un algoritmo que efectúa la ubicación de tareas con mucha rapidez y sencillez, y por otro lado es evidente que el desperdicio de área del dispositivo resultará excesivo cuando la diferencia de tamaño entre la tarea y la partición sea grande. A este tipo de situación se le llama **fragmentación interna** y es común a todos los algoritmos que proponen trabajar con particiones pre-definidas en la FPGA.

Otro inconveniente es la limitación de ejecución de tareas a aquéllas de área igual o inferior al 25 % del tamaño de la FPGA.

Vemos entonces que para un conjunto de tareas de tamaños muy variados como los que se podrían esperar en un sistema de computación de propósito general, las tareas pequeñas desperdiciarían mucho área de la FPGA, y muchas tareas grandes no se podrían ejecutar en una partición y se tendrían que rechazar. En resumen, la ventaja en cuanto a rapidez y sencillez en esta propuesta no compensan la falta de eficiencia y rigidez de gestión que la acompañan. Se trata de una propuesta muy fácil de implementar que hubiera necesitado de un desarrollo posterior, que los autores no realizaron.

2.2.2. Trabajo de H. Walder (II)

En otro trabajo, H. Walder [WP03] ha publicado algunas ideas parecidas a las presentadas en nuestro trabajo, como la de dividir la FPGA en varios bloques con tamaños diferentes y fijos a lo largo de la ejecución de un lote de tareas, pero lo han aplicado exclusivamente a casos en 1D y no a FPGAs en 2D como hemos hecho nosotros. El esquema del funcionamiento de dicho algoritmo se muestra en la figura 2.12.

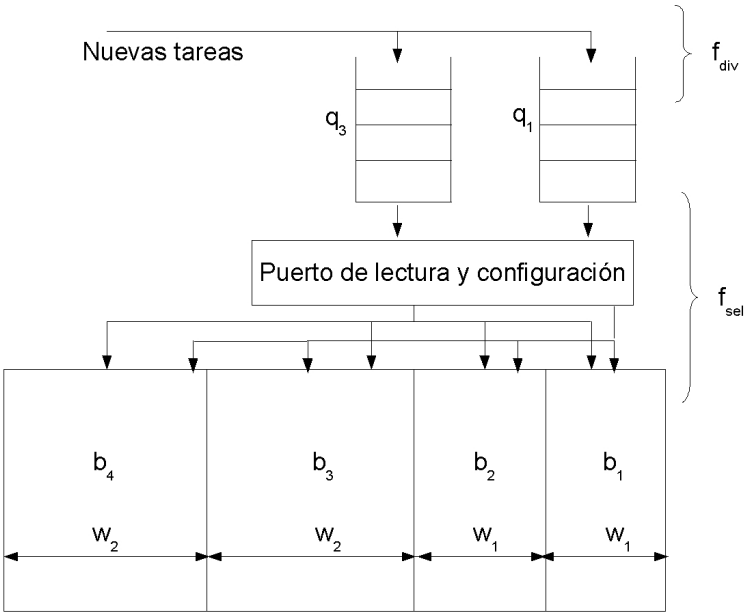

FIGURA 2.12: Propuesta de división en bloques de Walder

Los bloques son de tamaño fijo y diferente entre sí. Están ordenados de mayor a menor tamaño, estando los de mayor tamaño a la izquierda. La anchura de los bloques puede variar con cada nuevo lote de tareas a ejecutar, conforme a un estudio que se realiza partiendo de que se conoce de antemano el grupo de tareas a ejecutar.

El número de colas es también variable. Hay una cola por cada tamaño de bloque diferente. Una primera función, f_{div}, selecciona la cola que corresponda al menor tamaño de bloque capaz de ubicar a una tarea. Después inserta la tarea en la cola según un criterio que se ha seleccionado de antemano (por orden de llegada o por tiempo de ejecución).

Otra parte del algoritmo, f_{sel}, se ejecuta cada vez que una tarea termina su ejecución en la FPGA, determinando cuál de las tareas en la cabecera de cada cola se ejecutará la siguiente y en qué bloque de la FPGA se ubicará.

Esta función tiene dos modos de operación: el modo restrictivo y el modo preferente.

En el modo restrictivo, las tareas de una cola solamente pueden ejecutarse en los bloques asociados a dicha cola. En el modo preferente, las tareas de una cola pueden ejecutarse en cualquiera de los bloques que están libres en ese momento.

Conviene resaltar de este trabajo que además de ser un planteamiento para 1D, el algoritmo después de seleccionar una cola para la tarea entrante ordena las tareas en la cola y es posteriormente cuando selecciona la zona de la FPGA donde la tarea será ubicada. Esto quiere decir que la cola donde se ubica la tarea no determina la partición donde se ejecutará. También tiene el inconveniente de que no permite calcular el tiempo real que dicha tarea tendrá que esperar antes de ser ejecutada y que se retrasa la información de rechazo de tarea por no poder cumplir con sus requisitos de tiempo máximo hasta el momento en que éste haya expirado.

La decisión de cuál será la siguiente tarea a ejecutar y dónde se va a ubicar se toma cuando una tarea abandona la FPGA y deja espacio libre, por lo que este algoritmo tiene una complejidad alta, por un lado debido al algoritmo de ordenamiento de las tareas en las colas y por otro lado porque tiene que buscar en las particiones y contrastar la información con las tareas pendientes en las colas para decidir dónde ubicarla.

El número y tamaño de las particiones y el número de colas puede variar de una ejecución a otra, por lo cual su implementación no resulta sencilla, más aún si se realiza sobre el propio HW. Los autores hacen un estudio teórico de la eficiencia esperada para distintas distribuciones de particiones pero ad-

vierten de la dificultad de la implementación y de la necesidad de un estudio experimental posterior para llegar a conclusiones más determinantes.

2.2.3. Conclusiones: algoritmos sencillos

En resumen, en este grupo contamos con muchos menos trabajos publicados, cuyos objetivos son similares a los nuestros, es decir:

- **Complejidad simple**: Son algoritmos que manejan estructuras de datos sencillas para representar la situación de la ocupación de la FPGA.
- **Rapidez de respuesta**: Son algoritmos que encuentran rápidamente una ubicación para las nuevas tareas.

Tomándolos como punto de partida, podemos hacerles la siguientes críticas a los autores de estos grupos, que serán los aspectos que incorporaremos a nuestro trabajo de investigación, cuyos objetivos se han presentado en el capítulo anterior:

- **Eficiencia en planificación en 2D**: ya que no todos los algoritmos están orientados a la gestión del área de la FPGA en dos dimensiones.
- **Flexibilidad**: ya que algunas de las soluciones propuestas por estos autores carecen de la flexibilidad necesaria para la gestión del área de la FPGA.

Capítulo 3

Arquitectura del sistema y algoritmo básico

Este capítulo presenta la arquitectura global del sistema donde se realizará la implementación de un algoritmo que gestione los recursos HW disponibles, así como los modelos de tarea y FPGA que se han utilizado en esta tesis. Además, presentaremos el funcionamiento básico de nuestro planificador e indicaremos cómo se pueden ejecutar tareas de mayor tamaño que las particiones mediante la funcionalidad de fusión de particiones.

3.1. Arquitectura global del sistema

Este trabajo se basa en el uso de un recurso hardware dinámicamente reconfigurable como parte de un sistema de computación de propósito general. Esto quiere decir que el sistema dispone de una FPGA (además de la CPU tradicional) para ejecutar las aplicaciones, tanto las demandadas por el usuario

como las del propio SO.

La figura 3.1 muestra el esquema global de un sistema con estas características. Cualquier aplicación se descompone en tareas que pueden ser ejecutadas en la CPU (tareas tradicionales de software) o en la FPGA (tareas hardware).

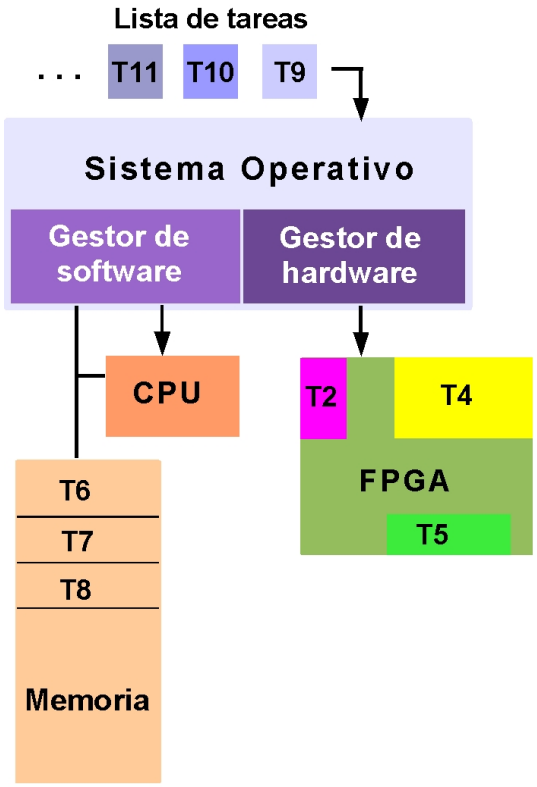

FIGURA 3.1: Arquitectura global del sistema

El SO decide en cada momento qué tareas se ejecutarán en la CPU y cuáles en la FPGA, dependiendo de la disponibilidad de recursos (uso de la CPU, uso de la FPGA, tiempo límite para ejecutar una tarea, dependencias entre tareas, disponibilidad del mapa de bits, prioridad etc.). Por ello es necesario extender las funcionalidades de un SO tradicional con las requeridas para gestionar el recurso hardware [DW99], [WK02]. Este aspecto fundamental viene reflejado

en la figura con la mención explícita de un gestor hardware y un gestor software.

3.2. Modelo de FPGA

Trabajamos con un modelo de FPGA en el que el dispositivo consiste en una matriz de W_T * H_T celdas básicas reconfigurables (cada una de ellas formada por un conjunto de CLBs correspondientes al tamaño mínimo de tarea), que son todas idénticas. W_T es el número total de columnas y H_T es el número total de filas, figura 3.2.

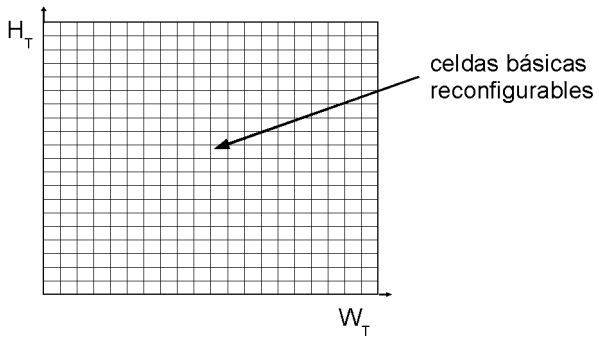

FIGURA 3.2: Modelo de FPGA

Según se muestra en la figura 3.3, en nuestro entorno hemos dividido el área de la FPGA de la siguiente manera:

- **Área para ejecución de tareas (AET):** Matriz de $W * H$ celdas básicas reconfigurables (CBRs), dividida en cuatro particiones de diferente tamaño, con $W < W_T$ y $H < H_T$.

  En nuestro trabajo gestionamos las celdas de la AET de la FPGA como si ésta estuviera dividida en cuatro particiones de diferente tamaño. Definimos las particiones como aquéllas que resultan de dividir la superficie

con dos líneas $x = w_p$ e $y = h_p$. Las particiones resultantes P_0, P_1, P_2 y P_3 tienen las siguientes características respectivamente:

- **Origen:** $(0,0)$, $(w_p,0)$, $(0,h_p)$ y (w_p,h_p)
- **Tamaños:** $w_p * h_p$, $(W-w_p)*h_p$, $w_p*(H-h_p)$ y $(W-w_p)*(H-h_p)$

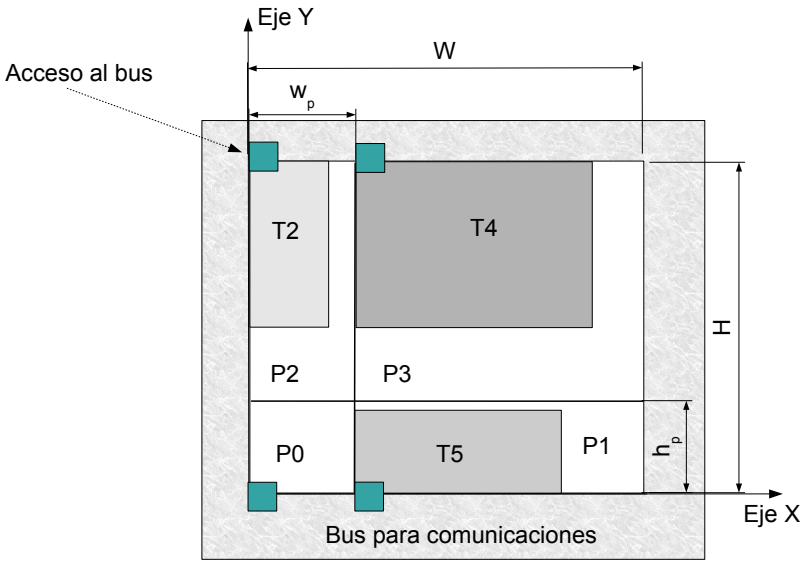

FIGURA 3.3: División de la FPGA

Los parámetros w_p y h_p determinan el tamaño de las particiones y pueden ser modificados a lo largo del funcionamiento del algoritmo para adaptarlas de la mejor manera posible a las necesidades del perfil de tareas en ejecución.

Se pueden rotar las tareas que se considere necesario para una mejor ubicación. Los cambios de forma de tarea aún no se están considerando, pero serían un aspecto a tener en cuenta en futuras versiones del algoritmo aunque supondrían una carga extra en la memoria y de computación

para el SO, puesto que sería necesario disponer de diferentes versiones de compilaciones de la misma tarea con diferentes geometrías.

- **Área para bus periférico:** área alrededor del AET, que garantiza el acceso de cada tarea a los pines de E/S de la FPGA y la comunicación entre tareas. El modelo de bus utilizado se explica en detalle en el apéndice A.

  El acceso al bus de comunicaciones se realiza a través de las conexiones fijas situadas en cada partición. Se ha elegido esta ubicación para evitar la reconfiguración del bus de comunicaciones en la mayor parte de los casos de funcionamiento del algoritmo.

- **Área para el Sistema Operativo:** el SO necesario para gestionar el DHWR su puede implementar en forma de módulos software, que se ejecutarán en una CPU tradicional (en versiones recientes de FPGA, en el propio procesador en el chip) y algunos módulos hardware que se configurarán en áreas de la FPGA reservadas para el SO.

3.3. Modelo de tarea

Las tareas HW están definidas por el mapa de bits que se cargará en la FPGA para su ejecución. Se representan como rectángulos en cuyas áreas están incluidos todos los recursos de procesamiento y rutado necesarios. El gestor de hardware incluye la posibilidad de establecer límites de tiempo para la ejecución de una tarea.

Cada tarea está definida por una tupla:

$$T_i = \{w_i,\ h_i,\ tej_i,\ tll_i,\ tmax_i\} \tag{3.1}$$

en la que w_i es el ancho de la tarea, h_i su altura, tej_i el tiempo de ejecución de la tarea (que incluye el tiempo de reconfiguración), tll_i es el momento en que llega la tarea al gestor, y $tmax_i$ el tiempo máximo para ejecución de la tarea, indicando el tiempo máximo desde la llegada de la tarea al gestor hasta el momento en que la tarea debe haber sido ejecutada, y limitando así el máximo tiempo de espera de cada tarea, $tespera_i$, según muestra la figura 3.4. Para cada tarea se debe cumplir:

$$tmax_i \geq tej_i + tespera_i \tag{3.2}$$

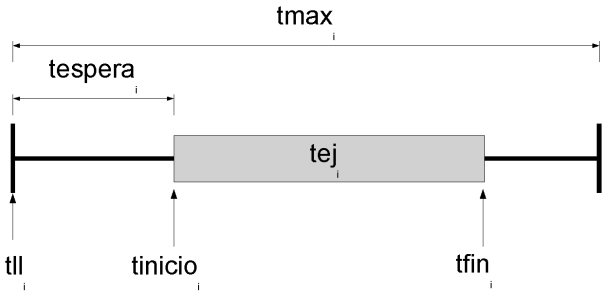

FIGURA 3.4: Máximo tiempo permitido para la terminación de la ejecución de una tarea

Las tareas se pueden rotar, para que el algoritmo disponga de mayor flexibilidad a la hora de asignarles una partición en que ejecutarse. Esto supone el coste adicional de disponer de dos compilaciones distintas para cada tarea en sus dos posibles versiones de $w_i * h_i$ o $h_i * w_i$, según muestra la figura 3.5, lo que supone un incremento de la memoria necesaria para almacenar los mapas

de bits de las tareas a ejecutar en el sistema.

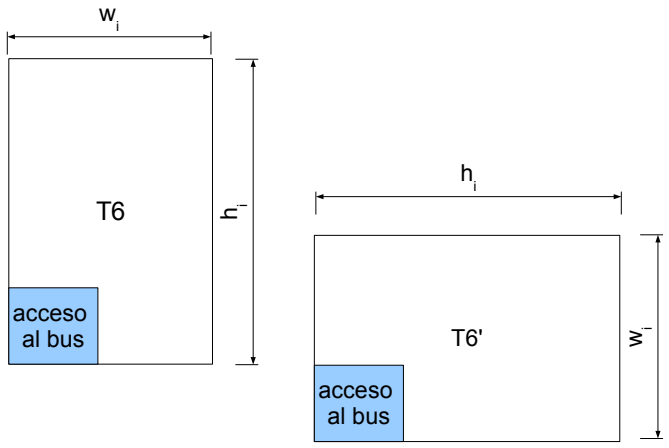

FIGURA 3.5: Rotación de la tarea

Los mapas de bits de las tareas se obtienen de manera que las tareas sean reubicables, es decir, que cuando se les asigna una partición en la que se ejecutarán, un sencillo cálculo permite generar el nuevo mapa de bits para la ejecución de la tarea, a partir del mapa de bits inicial.

Sin embargo, debido a la división de las FPGAs actuales en parte superior e inferior, en la actualidad, según se explica en detalle en el apéndice A, es necesario disponer de dos mapas de bits diferentes para cada tarea, según la mitad de la FPGA en que se vayan a ejecutar, de forma que se opte por la versión de compilación *inferior* para las particiones P0 y P1 y la versión *superior* para las particiones P2 y P3, según se muestra en la figura 3.6.

Por lo tanto, para la implementación de este modelo de gestión son necesarias cuatro compilaciones diferentes para cada tarea.

El modelo de reconfiguración parcial en que se basa el entorno propuesto en este trabajo está explicado en detalle en el apéndice A.

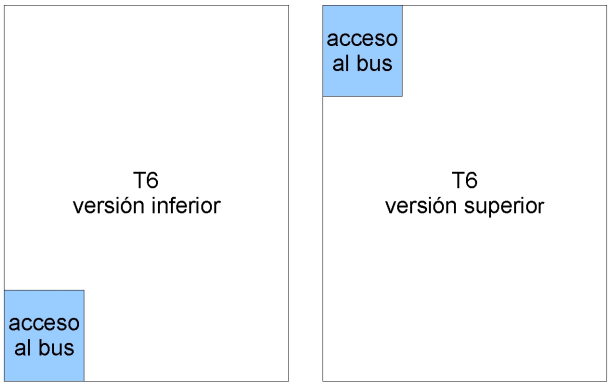

FIGURA 3.6: Dos compilaciones de la misma tarea

3.4. Esquema del planificador

La figura 3.7 muestra el esquema del planificador completo, compuesto por los siguientes elementos, que funcionan en paralelo:

- **Unidad de Planificación de Tareas (UPT):** Esta unidad examina las características de cada nueva tarea que llega al gestor HW para ser ejecutada en la FPGA y le asigna una cola, que determina la partición en la que la tarea se ejecutará. La selección de la partición se realiza mediante un algoritmo sencillo de complejidad constante que se describe en detalle en la siguiente sección de este capítulo.

- **Unidad de Lanzamiento de Tareas (ULT):** Esta unidad es la encargada de lanzar la ejecución de una nueva tarea a la FPGA cada vez que otra termina su ejecución y libera una partición. En ese momento la ULT lee la cola asociada y si hay alguna tarea esperando para ser ejecutada, pone en marcha el proceso de reconfiguración, encargándose de leer el mapa de bits de la tarea desde memoria y mandárselo al interfaz de reconfiguración. Asimismo, informa al SO de la terminación de la

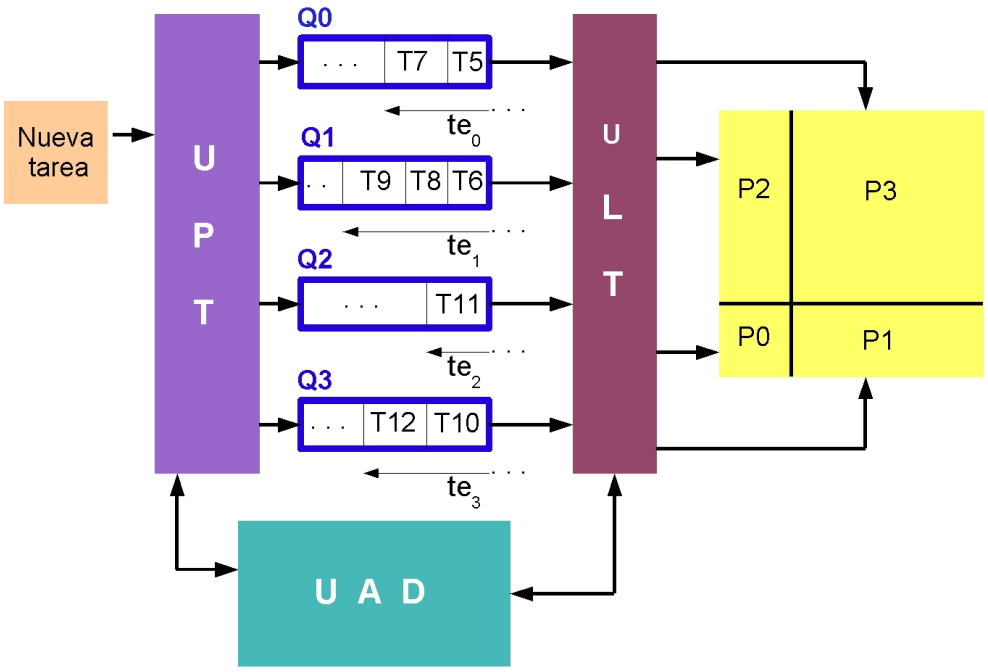

FIGURA 3.7: Esquema del planificador

ejecución de una tarea y el comienzo de la ejecución de una nueva.

- **Unidad de Adaptación Dinámica (UAD):** Esta unidad mantiene una estadística de las características de las tareas que están llegando al planificador HW y determina la necesidad de realizar cambios en las particiones (número y/o tamaño) y envía los comandos necesarios para ello a la ULT (reconfiguración del bus y de la FPGA si se cambia el número o tamaño de las particiones) y a la UPT (nuevos tamaños y número de particiones) para que se hagan efectivos. El funcionamiento detallado de esta unidad se describe en el capítulo 4.

- **Colas de tareas (Q_0, Q_1, Q_2, Q_3):** Hay una cola de tareas asociada a cada partición, de forma que todas las tareas esperando en la cola Q_i se ejecutarán en la partición P_i. El tiempo de espera de las tareas en

las colas puede aprovecharse para generar el mapa de bits absoluto, ya que una vez una tarea está escrita en una cola, la ubicación de la tarea en la FPGA queda determinada. Cada cola tiene un parámetro asociado llamado tiempo de espera, que indica el tiempo que una tarea tendrá que esperar a ser ejecutada si se asigna a dicha cola. Este parámetro es el que permite a la UPT calcular si una tarea asociada a una determinada partición cumplirá con el requisito de tiempo máximo de espera. El tiempo de espera se calcula de la siguiente forma:

$$te_k = \sum_{p \in Q_k} tej_p + tr_k \tag{3.3}$$

donde te_k es el tiempo de espera de la cola k, tej_p son los tiempos de ejecución de las tareas asignadas a la partición P_k y esperando en dicha cola, y tr_k es el tiempo restante de la tarea que se está ejecutando actualmente en la partición.

Se ha implementado este algoritmo con un código C++ donde para representar el estado de la FPGA basta con un sencillo array de 4x1 en el que cada posición está asociada con una partición, reflejando el estado de la partición por un entero que representa el instante de tiempo en el que la tarea que ocupa la partición termina su ejecución.

3.5. Algoritmo para asignación de espacio

La UPT examina las características de cada nueva tarea que llega y le asigna una cola/partición atendiendo a los siguientes criterios:

- **Mejor ajuste en área:** la tarea se ejecutará en la partición que más se ajuste a su tamaño, permitiendo así que particiones mayores queden libres para tareas de mayor tamaño y que el área del dispositivo se pueda aprovechar adecuadamente.

- **Tiempo de espera limitado:** la tarea se ejecutará dentro de la limitación de tiempo marcada por $tmax_i$.

El algoritmo que planifica la ubicación de la tarea en la FPGA es sencillo y de complejidad constante y consiste en un bucle en el cual se examina si la tarea cabe en las particiones, examinándolas en orden creciente de tamaño. La tarea se asignará a la primera partición que se encuentre donde quepa por tamaño y se cumpla el requisito de máximo tiempo de espera.

Si la UPT no encuentra ninguna partición en la que ejecutar la tarea debido a que los tiempos de espera son demasiado largos para las restricciones temporales de la tarea, dicha tarea se rechaza, es decir, no se escribe en ninguna cola y se informa inmediatamente al SO de que dicha tarea no será ejecutada en el plazo dado.

En nuestro modelo estamos utilizando una matriz de N x 4 variables X_{ij} que indican si la tarea T_i se ejecutará en la partición P_j o no. Las variables se definen de la siguiente manera:

$$X_{ij} = \begin{cases} 1 & \text{si } T_i \text{ se ejecuta en } P_j \\ 0 & \text{en caso contrario} \end{cases}$$

Cuando una tarea se ejecuta en la FPGA, una de las variables X_{ij} valdrá 1 y las restantes 0. Si todas las X_{ij} son 0 para una determinada tarea T_i significa

que no se pueden cumplir las restricciones de tiempo para dicha tarea y que se ha rechazado.

Como cada tarea únicamente se puede ejecutar en una partición, solamente una de las cuatro variables X_{ij} tomará el valor 1. Las otras tres restantes valdrán 0:

$$0 \leq \sum_{i=1}^{3} X_{ij} \leq 1 \tag{3.4}$$

El objetivo de nuestro algoritmo es ejecutar el máximo trabajo asignado para ejecución a la FPGA. Con el propósito de cuantificar la cantidad de trabajo ejecutada frente a la asignada hemos definido el **volumen de una tarea** de la siguiente manera:

$$vol_i = w_i \cdot h_i \cdot tej_i \tag{3.5}$$

que representa el volumen de FPGA que está ocupando, utilizando el tiempo como tercera dimensión.

Hemos definido el **volumen ejecutado** como la suma de los volúmenes de las tareas ejecutadas y el **volumen asignado** como la suma de los volúmenes de todas las tareas del lote. El resultado óptimo de nuestro algoritmo consiste en ejecutar el 100 % de la carga de trabajo asignada o volumen asignado. Por tanto se trata de maximizar la función volumen total ejecutado, que depende de la planificación que el algoritmo haga de las tareas que van llegando. La función se define así:

$$V_{ej} = \sum_{j=0}^{3} \sum_{i=1}^{N} X_{ij} \cdot vol_i \tag{3.6}$$

Esto nos permite evaluar el rendimiento del algoritmo a partir de la relación entre el volumen ejecutado V_{ej} y el volumen del lote de tareas o volumen asignado V_{as}, en términos percentiles. Cuanto mayor sea su valor mejor será el rendimiento del algoritmo:

$$R_{ej} = \frac{V_{ej}}{V_{as}} \cdot 100 \tag{3.7}$$

A continuación presentamos el algoritmo completo de la UPT.

Algoritmo 1 Algoritmo de selección de partición

```
for (i=1; i ≤ N; i++) do
  for (j=1; i ≤ 3; j++) do
    X_ij = 0;
  end for
end for
sel=false;
k=0;
while ((k < 4) and (sel == false)) do
  if (Cabe(T_i, P_k) == true) and (Cumplet(T_i, Q_k) == true) then
    Escribe(T_i,Q_k);
    X_ik=1;
    sel=true;
  end if
  k++;
end while
if (sel == false) then
  Rechaza(T_i);
end if
Salida();
```

Algoritmo 2 Subrutina Cabe(T_i,P_k)

if (($w_i \leq W_k$) and ($h_i \leq H_k$)) or (($w_i \leq H_k$) and ($h_i \leq W_k$)) **then**
  return true;
else
  return false;
end if

Algoritmo 3 Subrutina Cumplet(T_i,Q_k)

if ($tej_i + te_k \leq tmax_i$) **then**
  return true;
else
  return false;
end if

Algoritmo 4 Subrutina Salida()

V_{ej}=0;
V_{as}=0;
for (i=1; $i \leq N$; i++) **do**
  $V_{as} = +w_i \cdot h_i \cdot tej_i$;
  for (k=0; $k \leq 3$; k++) **do**
    $V_{ej} = +X_{ik} \cdot w_i \cdot h_i \cdot tej_i$;
  end for
end for
$R_{ej} = (V_{ej}/V_{as}) \cdot 100$;

3.6. Ejemplo de funcionamiento

En esta sección mostraremos un ejemplo detallado del funcionamiento del algoritmo y de su eficacia para gestionar el área de la FPGA. Para ello utilizaremos un lote de tareas pequeño y sencillo que pone de relieve las ventajas de nuestro algoritmo sencillo frente a otros complejos. Hemos utilizado una matriz de 20*20 CBRs para simular la FPGA con la distribución en 4 particiones que se muestra en la figura 3.8.

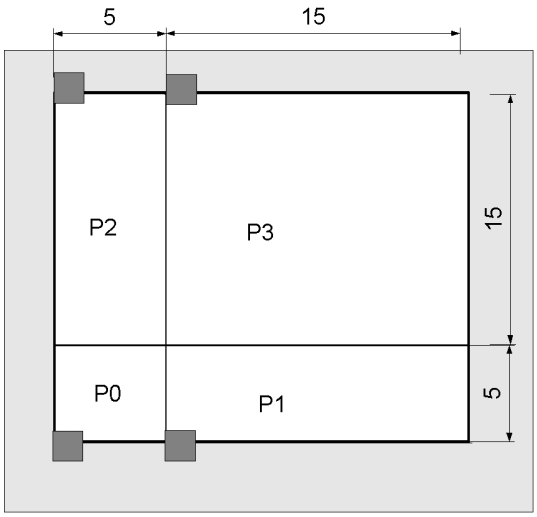

FIGURA 3.8: FPGA de 20 * 20 dividida en 4 particiones

El algoritmo que proponemos se basa en equilibrar la carga de trabajo en el dispositivo, clasificando las tareas por tamaños y reservando áreas específicas de la FPGA para tareas de determinados tamaños. Esto supone una gran ventaja cuando las tareas que pueden llegar al sistema para ser ejecutadas son de tamaños variados y especialmente en el caso en que pueden aparecer tareas de tamaño bastante grande con una frecuencia relativamente alta.

Para demostrar su eficacia, compararemos la gestión de nuestro algoritmo

basado en particiones (ABP) con un algoritmo comúnmente utilizado como referencia, el algoritmo de *First Fit* (FF), o Primer Ajuste.

FF es frecuentemente utilizado por los autores que trabajan en esta línea de investigación para comparar la eficiencia de sus propuestas. Este algoritmo puede ser *bottom left* (BL) o *top right* (TR), dependiendo de que se empiece la búsqueda de huecos para una nueva tarea por la esquina inferior izquierda de la FPGA (BL) o por la esquina superior derecha de la FPGA (TR). El FF BL examina las posibles posiciones de la FPGA para ubicar una nueva tarea entrante, de dimensiones w_i*h_i, empezando por la primera que encuentra libre en la esquina inferior izquierda y comprobando si la tarea entra en el hueco disponible. Para ello tiene que comprobar en cada posición libre si las w_i*h_i celdas contadas a partir de dicha celda están libres. FF ubica una tarea en el primer hueco libre que encuentra, según muestra la figura 3.9.

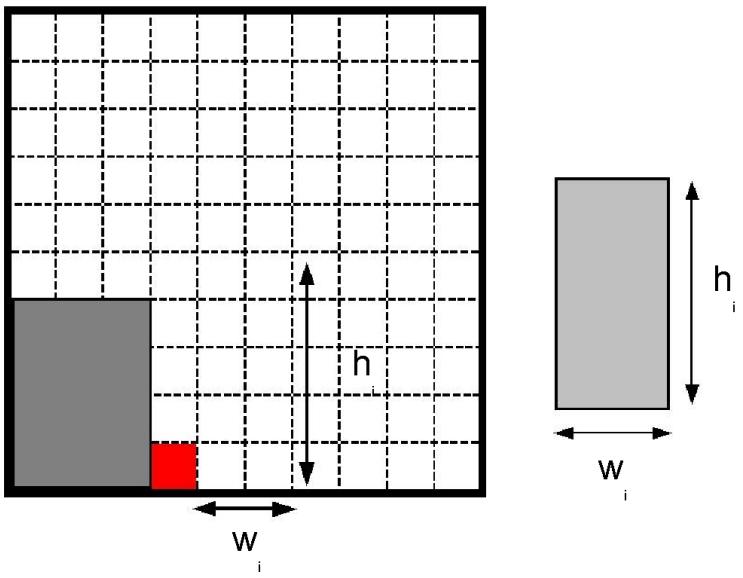

FIGURA 3.9: Algoritmo *First Fit BL*

Este algoritmo indudablemente puede llegar a hacer un uso razonable del

área de la FPGA en algunos momentos de su ejecución, pero también produce un nivel de fragmentación nada desdeñable debido a que en ocasiones una tarea puede estar ocupando una posición que impida la ejecución de otras, como podremos observar en nuestro ejemplo.

Además resulta de un orden de complejidad alto, $O(N^4)$ (donde $N*N$ son las dimensiones de una FPGA cuadrada) para reconfiguración parcial en 2D y resulta lento en su ejecución y por tanto en su respuesta, y aunque puede resultar eficiente en algunos momentos de su ejecución, en otros lo es muy poco. FF resulta bastante eficaz cuando los tamaños de las tareas son pequeños en relación al área de la FPGA, pero en la ubicación de tareas relativamente grandes resulta poco eficiente. Esto es debido a que si llegan tareas pequeñas o medianas intercaladas con las grandes, quedarán ubicadas en posiciones centrales de la FPGA e impedirán la ejecución de las tareas grandes ya que el espacio libre de la FPGA quedará muy fragmentado.

Para ilustrar lo comentado anteriormente, pasamos a explicar en detalle la ejecución del lote de tareas descrito en la tabla 3.1.

TABLA 3.1: Lote de tareas sencillo

Tarea	w_i	h_i	tll_i	tej_i	$tmax_i$
T1	5	5	1	6	13
T2	14	5	1	9	19
T3	15	13	1	8	17
T4	5	15	2	5	12
T5	4	4	8	12	32
T6	12	4	9	6	21
T7	13	13	11	5	21
T8	3	13	12	7	26

Para este ejemplo el área media de las tareas es de aproximadamente 80 CBRs, lo que supone un 20 % del tamaño de la FPGA. Observamos que se trata de un conjunto de 8 tareas de tamaños variados y que 2 de ellas son bastante grandes. Los tiempos máximos de las tareas son el doble de su tiempo de ejecución, lo que da poco margen para retrasarlas. Esto se ha hecho a propósito, ya que si no hubiera una limitación que fuera bastante restrictiva en cuanto al máximo tiempo que pueden esperar las tareas para su ejecución, el problema de planificación resultaría trivial y no tendría sentido la comparación de diferentes heurísticas.

Las figuras 3.10 a 3.15 nos muestran la planificación realizada por ABP, (a), columna de la izquierda, y la realizada por FF, (b), columna de la derecha. El número que aparece en cada tarea indica su tiempo de finalización.

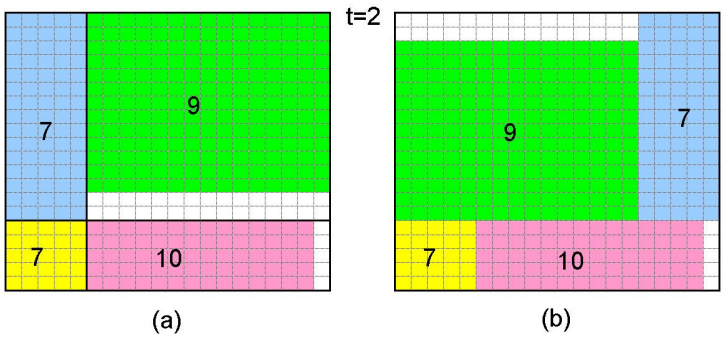

FIGURA 3.10: Ejemplo detallado t=2

En $t = 2$ han llegado cuatro tareas de diferentes tamaños (una de ellas es muy grande) que son ubicadas en la FPGA de diferentes maneras por ABP (a) y FF (b), según muestra la figura 3.10. Las tareas ocupan prácticamente todo el área de la FPGA y la fragmentación resultante de una y otra planificación son equivalentes.

En $t = 8$ llega una tarea de 4*4, que pasa a sustituir a la tarea de 5*5 que acaba de terminar su ejecución. Como el algoritmo FF busca el primer hueco libre, su posición coincidirá para los dos algoritmos, figura 3.11.

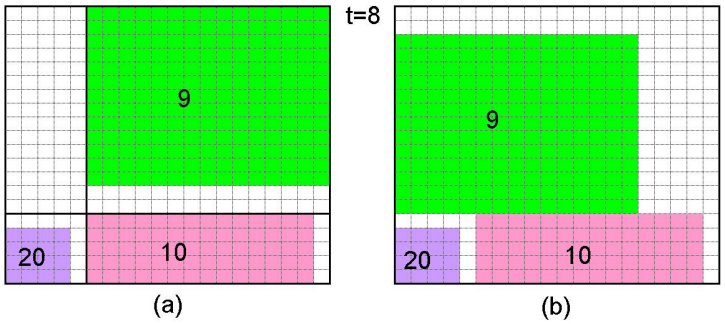

FIGURA 3.11: Ejemplo detallado t=8

En $t = 9$ llega una tarea de 12*4. Aquí empezamos a ver la diferencia en la política de uno y otro algoritmo. Como estamos simulando las versión más básica del algoritmo, suponemos que la tarea no se puede rotar y ejecutar en P_2. En el caso de FF, la ejecuta de inmediato situándola en el primer hueco libre que encuentra, justamente por encima de las otras dos que se están ejecutando. El algoritmo de particiones sin embargo decide esperar a que termine la tarea que se está ejecutando en P_1 (ya que la espera necesaria está dentro de los márgenes permitidos para esta tarea) y la escribe en Q_1. Aparentemente el algoritmo FF está aventajando al nuestro en este momento de la ejecución.

En $t = 10$ termina la tarea que estaba ocupando P_1 y pasa a ejecutarse la que espera en la cola. Nuestro algoritmo ha retrasado el fin de su ejecución en una unidad de tiempo debido a la espera.

Sin embargo podemos observar que el esperar para ejecutar la tarea de 12*4 permite al algoritmo de particiones mantener el espacio de la FPGA sin

fragmentar y dejar suficiente espacio libre para la ejecución tanto de tareas pequeñas o medianas que pudieran llegar como de tareas grandes y muy grandes, como muestra la figura 3.12a.

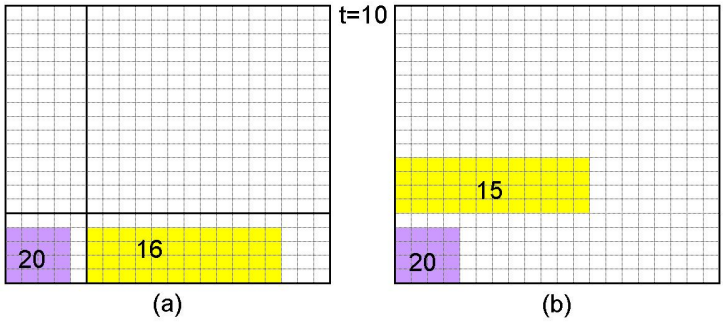

FIGURA 3.12: Ejemplo detallado t=10

En $t = 11$ llega la segunda tarea grande del lote y ABP la ejecuta de inmediato, figura 3.13a, mientras que FF debe ponerla en la cola de espera ya que el espacio libre disponible no es contiguo debido a la presencia de la tarea cuya ejecución adelantó con respecto al nuestro, figura 3.13b.

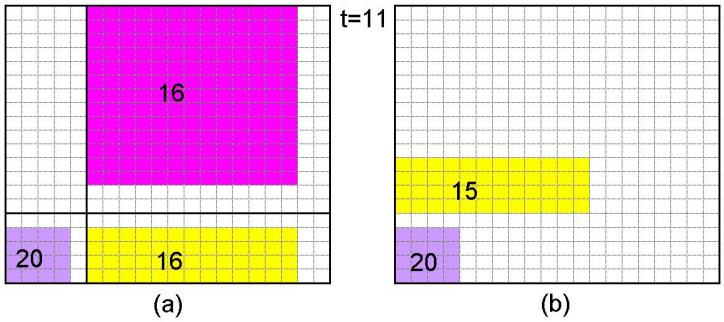

FIGURA 3.13: Ejemplo detallado t=11

La figura 3.14 muestra que la situación se agrava cuando en $t = 12$ llega una tarea de 3*13 que pasa a ser ejecutada en P_2 con ABP (a) y en el primer hueco disponible con FF (b), quedando ubicada en una posición central de la

FPGA y retrasando aún más el tiempo de espera de la tarea grande que está en la cola de FF.

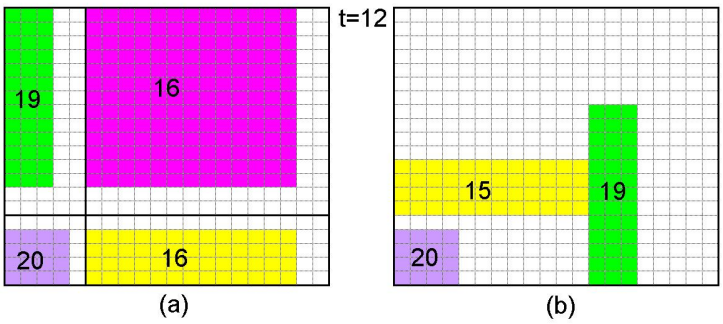

FIGURA 3.14: Ejemplo detallado t=12

Debido a la aparición de esta otra tarea de tamaño medio entre la tarea que espera en la cola y las tareas que se ejecutaban en la FPGA, en $t = 15$ ya no será posible para FF enviar a ejecutar la tarea que espera en la cola, teniendo esta tarea que esperar hasta $t = 19$ (y también debido a que ya no llegan más tareas para ejecutarse, si no la espera podría aumentar aún más), como muestra la figura 3.15b.

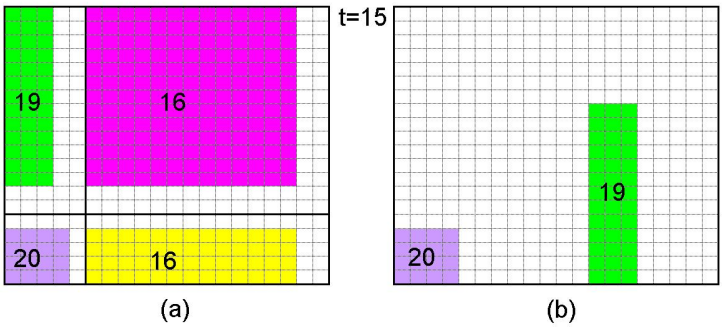

FIGURA 3.15: Ejemplo detallado t=15

En $t = 19$ la tarea se tiene que desechar puesto que se ha sobrepasado su tiempo máximo: $19 + 7 = 26 > 21$.

La primera fila de la tabla 3.2 muestra los resultados de la ejecución de este pequeño pero significativo lote de tareas para uno y otro algoritmo. Se muestra el rendimiento medido como la relación entre el volumen ejecutado y el volumen asignado (en porcentaje), el número de unidades tiempo (#u.t.) que se ha tardado en ejecutar el lote completo y el volumen de trabajo ejecutado. El volumen de trabajo de este lote es $V_{as} = 4{,}313$ CBRs*#u.t.

TABLA 3.2: Resultados del lote sencillo

	R (%)		$\mathbf{V_{ej}}$		**#u.t.**	
Lote	FF	Part	FF	Part	FF	Part
L1	80	100	3486	4313	20	20
L2	100	100	4313	4313	20	20
L3	80	80	3486	3468	20	20
L4	100	100	4313	4313	21	20
L5	64	100	2753	4313	18	19

Podemos observar que el rendimiento del algoritmo FF es más bajo que el de ABP precisamente debido a la tarea que ha tenido que desechar. Queremos resaltar la dificultad de este algoritmo para planificar la ejecución de tareas grandes, como se verá reflejado en los resultados experimentales presentados en el capítulo 6 y esta diferencia de rendimiento entre ABP y FF se mantiene o incluso aumenta a nuestro favor siempre que la presencia de tareas grandes en un lote de tareas sea significativa.

El resto de resultados de la tabla corresponden al mismo conjunto de tareas ordenado de diferentes formas.

En el lote L2 se ha retrasado la llegada de T7 a $t = 12$ y T8 se ha adelantado a $t = 11$. Vemos que en este caso el algoritmo FF sí puede planificar su ejecución

y el rendimiento se iguala para ambos.

En el lote L3 la llegada de T7 se ha adelantado a $t = 2$, es decir, a continuación de T3. El resto de las tareas mantiene el orden de llegada y se retrasan. La FPGA no es suficientemente grande como para ejecutar ambas simultáneamente y el tiempo entre las tareas con respecto al tiempo de ejecución de T3 es demasiado pequeño y la tarea no puede ejecutarse en ningún caso, ya que no depende de la planificación realizada. El rendimiento de ambos algoritmos también es igual para este caso.

En el lote L4 las tareas T2 y T3 llegan con suficiente separación en el tiempo, de ahí que cualquiera de los dos algoritmos ejecuta el total de tareas.

El lote L5 representa otra forma de ordenación de las tareas en el cual las de tamaño mediano y pequeño que llegan entre T2 y T3 interfieren, en el caso de FF, con la ejecución y dan lugar al rechazo de una de tamaño mediano además de una de las grandes.

Una clara ventaja de nuestro algoritmo basado en particiones frente a FF es que tarda mucho menos tiempo en planificar las tareas, dando además lugar a una planificación mejor.

3.7. Fusión de particiones

Es un hecho evidente que pueden llegar tareas al sistema que sean mayores que cualquiera de las particiones definidas. En estos casos, que asumimos como poco frecuentes, el algoritmo no puede permitirse rechazar una tarea debido a una excesiva rigidez en su definición.

Por esta razón se ha añadido una funcionalidad al algoritmo para el caso

en que una tarea tenga unas dimensiones tales que no quepa en ninguna de las particiones establecidas, pero sí en la FPGA: el algoritmo ofrece la posibilidad de unir dos o incluso las cuatro particiones para ubicar dicha tarea. Las posibles uniones de particiones son: P_0+P_1, P_0+P_2, P_1+P_3, P_2+P_3 y $P_0+P_1+P_2+P_3$, en orden creciente de tamaño.

Para incluir esta funcionalidad hemos añadido cinco particiones virtuales en el modelo: $P_4 = P_0 + P_1$, $P_5 = P_0 + P_2$, $P_6 = P_1 + P_3$, $P_7 = P_2 + P_3$ y $P_8 = P_0 + P_1 + P_2 + P_3$. De esta forma el algoritmo se mantiene en su forma original y solamente hay que aumentar el número de iteraciones del mismo, la función que calcula el tiempo de espera de las colas virtuales y la que escribe en las colas, que se muestran el sección 3.8.

La figura 3.16 muestra las particiones reales y las virtuales definidas en la FPGA.

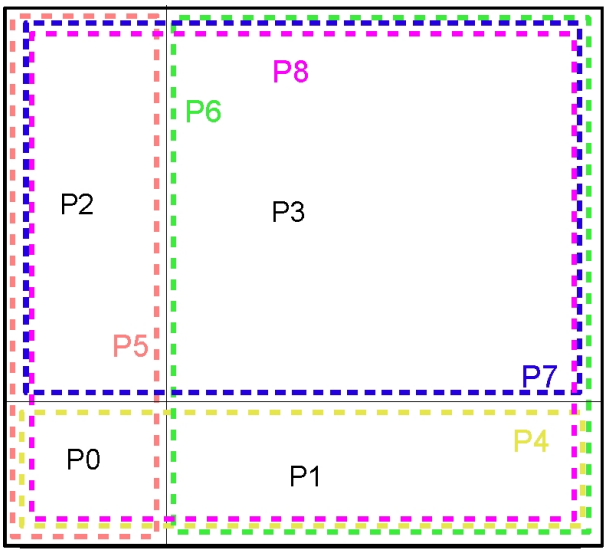

FIGURA 3.16: Particiones reales y virtuales en la FPGA

3.7.1. Descripción de la fusión de particiones

La UPT busca las dos particiones más pequeñas que pueden alojar a la tarea y comprueba que el tiempo que tendrá que esperar a que ambas queden libres simultáneamente no excede los límites del t_{max} de la tarea. Si los cumple, escribe la tarea en la cola con mayor tiempo de espera y reserva dicho intervalo de tiempo en la cola asociada a través de unos parámetros sencillos de calcular y de comprobar posteriormente. Si no los cumple, busca otra combinación de particiones que tengan menor tiempo de espera, con el coste de desperdiciar una mayor área de dispositivo. En el caso de que ninguna combinación de particiones cumpla los requisitos de tiempo máximo de la tarea, ésta será rechazada.

Los parámetros utilizados para reservar tiempo de ejecución en una partición en cuya cola no se ha escrito la tarea pero que se va a utilizar para la fusión, se denominan *tiempo de fusión* (tf_i) y *tiempo de reanudación* ($treanud_i$). El tiempo de fusión indica el instante de tiempo a partir de la cual una partición estará ocupada por una tarea no presente en la cola, y el tiempo de reanudación indica el momento en que dicha tarea abandonará la partición. Se muestran en la figura 3.17.

3.7.2. Ejemplo de fusión de particiones

La figura 3.17 muestra un ejemplo de tarea que necesita de la fusión de las particiones $P_0 + P_1$. La tarea en cuestión es la T12. El algoritmo examina el tiempo de espera de la cola 0 y de la cola 1. El tiempo de espera de la cola 1 es mayor y por tanto T12 se escribe en 0 *apuntando a 1*. El tiempo de espera de

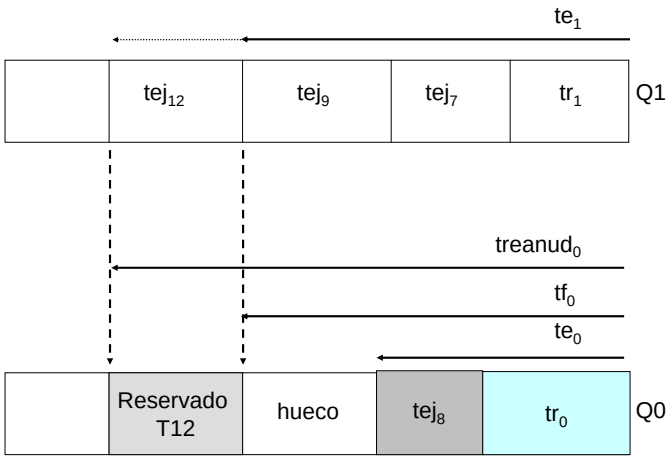

FIGURA 3.17: Parámetros utilizados para la fusión de particiones

la cola 1 se actualiza sumándole el tiempo de ejecución de T12 y en la cola 0 no se vuelve a copiar la tarea sino que se deja marcado el intervalo de tiempo que ocupará la parte de T12 que no cabe en la partición 1, a través de los parámetros tf_0 (tiempo de fusión en la cola 0) y $treanud_0$. Como observamos en la figura, ha quedado un hueco temporal de valor $tf_0 - te_0$ en la cola Q_0.

Este método permite mantener la complejidad constante del algoritmo y dejar el hueco correspondiente en la cola de menor tiempo de espera para un posible utilización posterior, en el caso de que llegara una tarea con un tiempo de ejecución menor al del hueco, como muestra la figura 3.18a. Si la siguiente tarea que llega a dicha cola no cabe en el hueco, se escribirá a continuación de la última tarea pero con un tiempo de lanzamiento igual al tiempo de reanudación, como muestra la figura 3.18b y manteniendo el hueco por si pudiera ser aprovechado posteriormente.

El coste de mantener la complejidad constante es el de dejar un solo hueco por cola, evitando así tener una lista de huecos que habría que recorrer cada

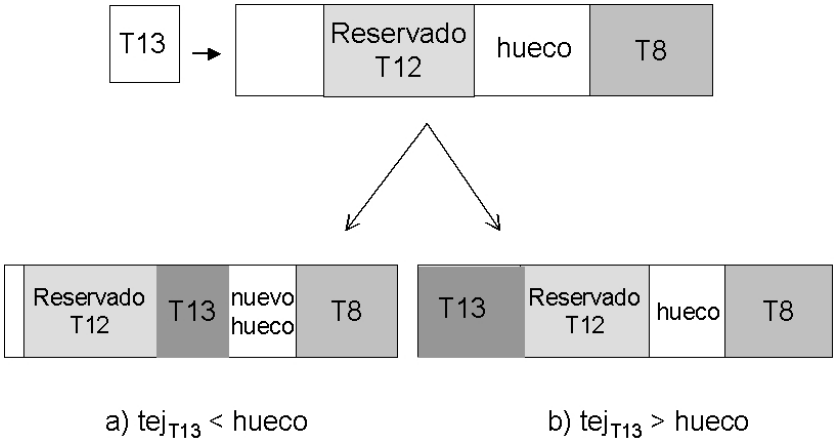

FIGURA 3.18: (a) aprovechamiento del hueco (b) imposibilidad de aprovechar el hueco

vez que llegara una tarea candidata a dicha cola (figura 3.19). Esto no supone un problema porque estamos asumiendo, de forma realista, que estos casos van a ser excepcionales y no habituales.

FIGURA 3.19: Mantenimiento de un solo hueco por cola

3.7.3. Implementación de la fusión de particiones

La fusión de particiones es una posibilidad real en arquitecturas actuales como la Virtex 4. El trabajo de Sedcole [SBB+06] muestra que con una adecuada técnica de reconfiguración es posible definir particiones contiguas en la FPGA. La unión de particiones se realiza a través de la escritura de un mapa de bits que abarca las particiones seleccionadas. La reconfiguración del bus periférico no es necesaria puesto que dichas tareas tienen garantizado el acceso

al bus exactamente de la misma manera que cualquier otra tarea, a través de la conexión fija en la esquina exterior de la partición resultante, como se muestra en la figura 3.20.

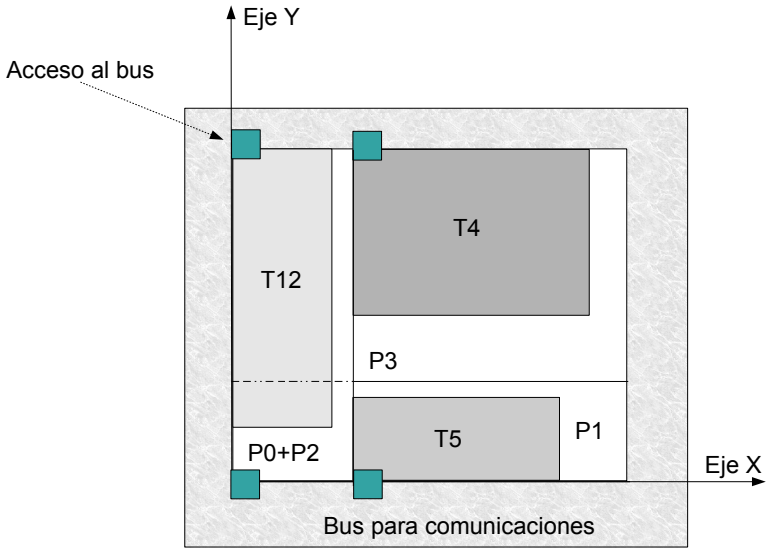

FIGURA 3.20: Acceso al bus en particiones unidas

3.8. Algoritmo con fusión de particiones

Como ya hemos comentado, se utilizan cinco particiones virtuales que representan la particiones unidas y así el algoritmo mantiene su forma original. Solamente hay que aumentar el número de iteraciones del mismo, la función que calcula el tiempo de espera de las colas virtuales y la función que escribe en las colas. La matriz M que pasa a ser una matriz de N*8.

En esta versión mejorada del algoritmo, la función que comprueba si cumple tiempos necesita calcular el tiempo de espera de la cola virtual k, que será el

máximo tiempo de espera de las colas asociadas con las particiones que se unen, l y m. La tarea se escribirá en la cola con mayor tiempo de espera y se reservará dicho hueco en la cola asociada, a través del parámetro *tiempo de fusión*.

Algoritmo 5 Algoritmo de selección de partición

```
for (i=1; i ≤ N; i++) do
  for (j=1; i ≤ P; j++) do
    X_ij = 0;
  end for
end for
sel=false;
k=0;
while ((k < 9) and (sel == false)) do
  if (Cabe(T_i, P_k) == true) and (Cumplet(T_i, Q_k) == true) then
    Escribe(T_i,Q_k);
    X_ik=1;
    sel=true;
  end if
  k++;
end while
if (sel == false) then
  Rechaza(T_i);
end if
Salida();
```

Algoritmo 6 Subrutina Cabe(T_i,P_k)

```
if ((w_i ≤ W_k) and (h_i ≤ H_k)) or ((w_i ≤ H_k) and (h_i ≤ W_k)) then
  return true;
else
  return false;
end if
```

Algoritmo 7 Subrutina Cumplet(T_i,Q_k)

if $((k \leq 3)$ and $(tej_i + te_k \leq tmax_i))$ **then**
  return true;
else
  if $((tej_i + max(te_l, te_m)) \leq tmax_i)$ **then**
    return true;
  else
    return false;
  end if
end if

Algoritmo 8 Subrutina Salida()

V_{ej}=0;
V_{as}=0;
for (i=1; $i \leq N$; i++) **do**
  $V_{as} = +w_i \cdot h_i \cdot tej_i$;
  for (k=0; $k \leq 8$; k++) **do**
    $V_{ej} = +X_{ik} \cdot w_i \cdot h_i \cdot tej_i$;
  end for
end for
$R_{ej} = (V_{ej}/V_{as}) \cdot 100$;

3.9. Comparación de resultados con y sin fusión

En la tabla 3.3 presentamos los resultados de comparar la ejecución de dos lotes de tareas con la versión básica del algoritmo y la versión que implementa la fusión de particiones. En ambos hay un pequeño número de tareas que no caben en ninguna de las particiones definidas y que por lo tanto necesitan la fusión de particiones para su ejecución.

TABLA 3.3: Comparación de la versión básica y con fusión de particiones

	R (%)		#**u.t.**		**A** (%)	
Lote	Básica	Fusión	Básica	Fusión	Básica	Fusión
L1	95	100	168	168	81,00	85,20
L2	77,79	100	138	154	76,40	88,00

Los resultados muestran que la versión con fusión de particiones obtiene un mejor rendimiento que la versión básica ya que no rechaza las tareas que no caben en ninguna de las particiones.

El algoritmo resultante de incluir esta funcionalidad no aumenta la complejidad del algoritmo ni supone una carga computacional excesiva, por lo que concluimos que la funcionalidad de fusión de particiones debe ser incorporada al planificador de tareas basado en particiones.

Capítulo 4

Gestión basada en particiones con adaptación dinámica

Este capítulo está dedicado a explicar el funcionamiento de la Unidad de Adaptación Dinámica (UAD). La UAD aparece como una mejora inevitable de nuestro algoritmo ya que proporciona la flexibilidad de la que carece la versión básica original. Como demostraremos a través de los experimentos realizados, nuestro algoritmo de complejidad constante ofrece un buen rendimiento frente a algoritmos complejos y resulta por tanto una propuesta que puede competir con las existentes debido a la excelente relación sencillez / eficacia que ofrece. Esto queda demostrado por una gama de experimentos, presentados en el capítulo 6, en los que podemos observar su buen comportamiento y eficiencia para distribuir las tareas entrantes entre las diferentes particiones de la FPGA.

Sin embargo es también evidente que se pueden dar situaciones en las que el algoritmo presente un rendimiento bajo y por ello se ha diseñado una unidad que funciona en paralelo con las otras y cuya labor consiste en observar la

naturaleza de las tareas entrantes (o lo que más adelante definiremos como **distribución de tareas**) y decidir si es necesario realizar cambios en la forma de gestionar el espacio disponible en la FPGA (es decir, en el tamaño/número de particiones). La hemos llamado Unidad de Adaptación Dinámica (UAD), y pasaremos a describir su funcionamiento después de realizar un análisis del problema de la planificación de una carga de trabajo en entornos *on-line* que nos permitirá sentar las bases para formalizar el problema y su solución.

La figura 4.1 nos recuerda la estructura de nuestro planificador y la relación de la UAD con el resto de unidades del sistema.

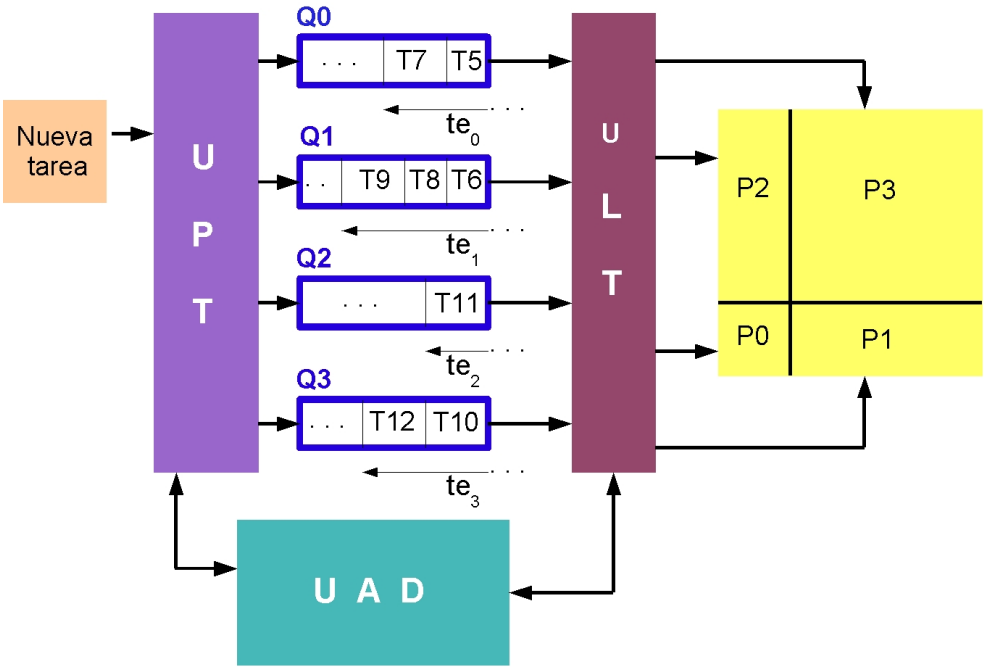

FIGURA 4.1: Esquema del planificador

4.1. Análisis del problema para una FPGA sin particiones

4.1.1. Carga de trabajo ideal

El punto de partida de nuestro análisis será la adecuación de la carga de trabajo asignada a la FPGA en función de su tamaño o capacidad.

La carga de trabajo se modela por medio de un conjunto o lote de tareas, L, compuesto por N tareas, según se ha explicado. Para estudiar la adecuación de una carga de trabajo a la capacidad de la FPGA tendremos que utilizar un valor t_L igual al intervalo de tiempo durante el cual llegan y se ejecutan las tareas del lote L.

Se definen las **condiciones ideales** de utilización de la FPGA como la situación en que es posible aprovechar su capacidad al 100 %: uso del 100 % del área de la FPGA el 100 % del tiempo en el que están llegando y ejecutándose las tareas que componen la carga.

Teniendo en cuenta que la carga de trabajo no se asigna de golpe sino que la llegada de tareas se reparte a lo largo de un intervalo de tiempo $tll_N - tll_1$, los volúmenes de las tareas entrantes tendrían que encajar de forma perfecta en el volumen libre de la FPGA sin dejar huecos (lo que no solamente depende del volumen de cada tarea sino también de sus tiempos de llegada y de ejecución). La primera llegada de tareas tendría que ser en tll_1 y estaría compuesta por un subconjunto de tareas cuyas áreas ocupasen el 100 % de la FPGA. Si todas terminan de ejecutarse a la vez, la siguiente llegada de tareas tendría que cumplir los mismos requisitos y llegar en un tiempo no superior al tiempo de

ejecución del primer subconjunto o sub-lote, L1.

Si los tiempos de ejecución de las tareas del primer sub-lote no tienen tiempos de ejecución iguales, las siguientes tareas tendrían que encajar de forma perfecta en los volúmenes que van quedando libres en el volumen de la FPGA y deben llegar en tiempos iguales o anteriores al tiempo de fin de ejecución de las tareas que están en la FPGA y cuyo hueco ocuparán. Además, las últimas tareas tendrían que terminar todas a la vez.

La figura 4.2 nos muestra un ejemplo sencillo de condiciones ideales. Los números que aparecen sobre cada tarea indican su tiempo de ejecución.

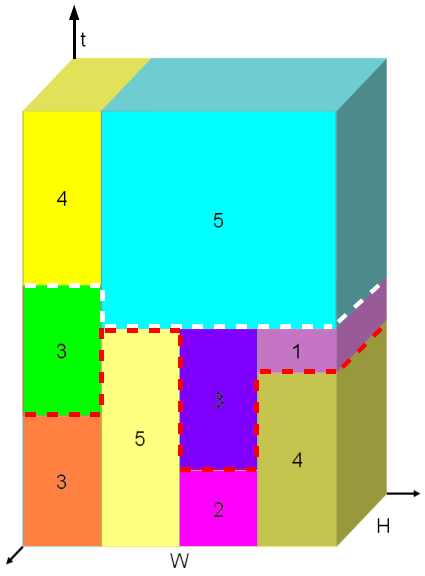

FIGURA 4.2: Condiciones ideales de ocupación de una FPGA

Este ejemplo corresponde a un grupo de 9 tareas, de las cuales las primeras 4 llegan a la vez, con diferentes tiempos de ejecución y todas las demás llegan repartidas en el tiempo, cumpliendo las condiciones ideales.

En la figura hemos resaltado con línea discontinua los tres grupos o sub-

lotes de tareas que se ejecutan en la FPGA: las primeras que llegan, las otras 3 que se ejecutan a continuación, y las últimas que llegan y se ejecutan, que terminan a la vez.

Podemos por tanto hacer una buena estimación del tiempo que tardará en ejecutarse un lote de tareas que cumple las condiciones ideales a partir del número de grupos de tareas que se ejecutarán en la FPGA (o capas de tareas que cubren por completo el área del dispositivo) y su tiempo medio de ejecución.

El número medio de tareas que se pueden ejecutar simultáneamente en la FPGA depende del tamaño medio de las tareas en relación al tamaño de la FPGA:

$$\overline{n} = \frac{A}{\overline{a}} = \frac{A \cdot n}{\sum_{i=1}^{n} a_i} \tag{4.1}$$

donde n es el número de tareas y a_i el área de la tarea t_i.

Si ejecutamos las tareas una a una, el tiempo que se tardará en ejecutarlas todas es

$$t_{tot} = \sum_{i=1}^{n} tej_i \tag{4.2}$$

Por lo tanto, si las ejecutamos de $\overline{n}$ en $\overline{n}$, el tiempo que se tardará será:

$$t_{ideal} = \frac{\sum_{i=1}^{n} tej_i}{\overline{n}} \tag{4.3}$$

Sustituyendo nos queda que:

$$t_{ideal} = \frac{\sum_{i=1}^{n} tej_i \cdot \sum_{i=1}^{n} a_i}{A \cdot n} \tag{4.4}$$

El número de capas de tareas en la FPGA en condiciones ideales es:

$$g = \frac{\sum_{i=1}^{n} a_i}{A} \tag{4.5}$$

El tiempo que se tarda en ejecutar el lote ideal de n tareas es:

$$t_{ideal} = g \cdot \overline{tej} \tag{4.6}$$

Resulta bastante obvio que las condiciones ideales son difíciles de cumplir y en la realidad se darán muchos tipos de situaciones que se alejarán de ellas.

El objetivo de esta parte del trabajo es analizar diferentes circunstancias que pueden darse a lo largo del funcionamiento de un sistema de computación de propósito general para poder adaptar el número y tamaño de las particiones de la forma más adecuada.

Hemos empezado planteando el caso ideal ya que nos permite analizar los factores relacionados con una adecuada gestión del dispositivo. A través de la ecuación obtenida para el tiempo de ejecución de un lote que cumple las condiciones ideales podemos observar que hay dos aspectos fundamentales que influyen de manera conjunta en las posibilidades reales que existen para ejecutar de forma óptima un conjunto de tareas en una FPGA:

1. Distribución de la carga de trabajo en el espacio: tamaños de las tareas

2. Distribución de la carga de trabajo en el tiempo: tiempos de ejecución de las tareas e intervalos de llegada de las tareas

A partir de la ecuación obtenida anteriormente se puede llegar a otra ecuación que nos permite realizar un análisis transversal de la distribución de la

carga en el espacio-tiempo (es decir, relacionar los tamaños de las tareas con los dos aspectos temporales: tiempo de ejecución e intervalos de llegada de las tareas) y que nos dará pautas para determinar la relación entre la carga de trabajo asignada a la FPGA y su capacidad, lo que ayudará a evaluar si el uso que se está haciendo del área de la FPGA es adecuado o no. Todo esto se explica en detalle en la siguiente sección.

4.1.2. Frecuencia ideal de llegada de tareas

El problema de definir qué es una adecuada distribución de la carga de trabajo en el tiempo y en el espacio es complejo ya que ambos aspectos están relacionados entre sí. Por ejemplo, si casi todas las tareas que llegan son muy grandes y ocupan casi todo el área de la FPGA, su llegada debe ser espaciada en el tiempo para que se pueda garantizar su ejecución y en intervalos de tiempo parecidos a sus tiempos de ejecución para que los tiempos de espera no se acumulen (esto daría como resultado el rechazo de tareas, como podremos comprobar también a partir de los resultados experimentales realizados). Sin embargo, si las tareas que llegan son en su mayoría pequeñas, entonces se puede garantizar su ejecución aunque lleguen muy cercanas en el tiempo, siempre que la relación entre los tiempos de ejecución de las tareas y la frecuencia con la que llegan esté en equilibrio.

Por lo tanto debemos encontrar una manera de relacionar estos tres parámetros: la frecuencia con la que llegan las tareas al sistema y sus características, es decir el área de FPGA que van a ocupar, $a_i = w_i * h_i$, y el tiempo que permanecerán ocupándola, tej_i.

La frecuencia de llegada de tareas al sistema es:

$$FLLT = \frac{n}{t_L} \tag{4.7}$$

donde t_L es el tiempo que transcurre desde que llega la primera tarea del lote L al sistema hasta que termina de ejecutarse la última.

Considerando que en condiciones ideales estas tareas tardarán un t_{ideal} en ejecutarse, podemos unir estas dos ecuaciones y definir la **frecuencia ideal de llegada de tareas** de la siguiente manera:

$$FLLT_{ideal} = \frac{n}{t_{ideal}} = \frac{n}{g \cdot \overline{tej}} = \frac{A}{\overline{a} \cdot \overline{tej}} \tag{4.8}$$

Esta relación entre los tiempos de llegada de las tareas, sus tiempos de ejecución y sus áreas indica las condiciones que nos permitirían utilizar el 100 % del dispositivo el 100 % del tiempo.

La ecuación obtenida es muy importante ya que incluye simultáneamente la condición de una adecuada distribución de la carga de trabajo en el espacio, a través del término $A/\overline{a}$ y la condición de una adecuada distribución de la carga de trabajo en el tiempo, a través de $1/\overline{tej}$.

Analizaremos en profundidad el significado de esta ecuación:

1. Si los tamaños medios de las tareas aumentan y sus tiempos de ejecución no lo hacen, lo que tendremos es un aumento de la carga de trabajo asignada. Para que el conjunto de tareas pueda ejecutarse completamente, la $FLLT_{real}$ deberá disminuir de forma proporcional. Si esto no sucede, el sistema habrá pasado a una situación de sobrecarga y es un problema no achacable a una incorrecta gestión del área de la FPGA.

2. Si los tiempos de ejecución de las tareas aumentan sin que lo hagan sus tamaños, estaremos en una situación parecida a la anterior.

3. Si ambos parámetros aumentan, entonces tendremos exactamente la misma situación que antes, aunque la disminución de la $FLLT_{real}$ tendrá que ser aún mayor.

4. Si lo que sucede es que las áreas o los tiempos de ejecución o ambos disminuyen, lo que tendría que ocurrir para no llegar a una situación en que el sistema esté siendo infrautilizado es que la $FLLT_{real}$ aumente. Si esto no sucede, la gestión del área de la FPGA se convierte en un problema trivial ya que sobra espacio y ningún algoritmo tendrá que rechazar tareas por no encontrar un lugar adecuado donde ejecutarlas.

4.1.3. Carga de trabajo real. Definición del parámetro α

Podemos por tanto decir que para analizar la adecuación de la gestión del área de la FPGA con respecto a las características de la carga de trabajo, es necesario que $FLLT_{real} \approx FLLT_{ideal}$. La relación entre la carga de trabajo asignada a la FPGA y su capacidad queda entonces definida por parámetros que se pueden medir en tiempo real.

Esta relación entre la frecuencia real de llegada de tareas al sistema y la que sería ideal para las caracterísitcas de la carga de trabajo en relación a la capacidad de la FPGA es un aspecto importante a tener en cuenta y lo hemos llamado **parámetro** α definido por la siguiente ecuación:

$$\alpha = \frac{FLLT_{real}}{FLLT_{ideal}} \tag{4.9}$$

Hemos definido la relación entre la carga de trabajo real y la capacidad del sistema en función de este parámetro del modo siguiente:

Sistema sobrecargado: Decimos que un sistema está sobrecargado cuando se está asignando más carga de trabajo al sistema (la FPGA) del que puede asumir, y que por tanto no existe ningún algoritmo capaz de garantizar la ejecución del 100 % de las tareas del lote:

$$\alpha > 1$$

Sistema infrautilizado: Decimos que un sistema está infrautilizado cuando la carga de trabajo asignada es muy pequeña con relación a la capacidad de la FPGA y por lo tanto no es necesaria una gestión complicada del espacio disponible:

$$\alpha \ll 1$$

Sistema equilibrado: Esto quiere decir que la carga de trabajo es proporcional a la capacidad del dispositivo:

$$\alpha \approx 1$$

Resumiendo, los valores de α indican la adecuación de la carga de trabajo a la capacidad de la FPGA. Hemos tomado una carga de trabajo que ocupa el 50 % del volumen ideal de la FPGA como el límite entre un sistema infrautilizado y uno equilibrado:

- **Sistema ideal:** $\alpha = 1$

- **Sistema sobrecargado:** $\alpha > 1$
- **Sistema infrautilizado:** $0 \leq \alpha < 0{,}5$
- **Sistema equilibrado:** $0{,}5 \leq \alpha < 1$

4.1.4. Ejemplo de variaciones de la carga de trabajo

Hemos realizado unas pruebas con cuatro lotes de 65 tareas cuyo volumen asignado es $V_{as} = 285,000\, CBRs * \#u.t.$ Se ha partido de un lote ideal, L1, sobre el que se han realizado variaciones en la frecuencia de llegada con el objetivo de conseguir simular cuatro tipos de situaciones: sistema ideal ($L1$), sistema sobrecargado ($L2$), sistema equilibrado ($L3$) y sistema infrautilizado ($L4$). Hemos simulado su ejecución en una FPGA de 50*50 CBRs. Para este caso $A = 2500$, $\overline{a} = 625$, $\overline{tej} = 7,02$ y $tmax_i = tll_i + 2 \cdot tej_i$ para todas las tareas. El área está dividida en 4 particiones según muestra la figura 4.3 y se ha preparado el lote de tareas para que sus tamaños sean exactamente iguales a los de las particiones definidas. Para este caso $FLLT_{ideal} = 0,57 (tareas/u.t)$.

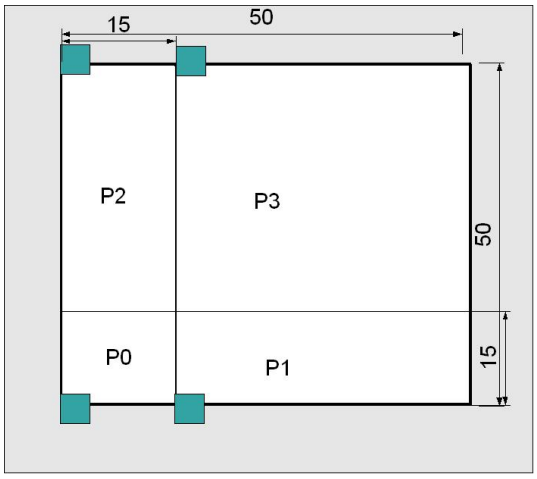

FIGURA 4.3: Distribución en 4 particiones para una FPGA de 50*50

La tabla 4.1 nos muestra las características de los lotes junto a los resultados obtenidos de la simulación. Hemos representado las unidades de tiempo (*u.t.*) que tarda la ejecución completa del lote, el porcentaje medio de área de FPGA utilizada en la ejecución completa del lote ($\%A$) y el rendimiento de la planificación, definido como la relación entre el volumen de la carga de trabajo asignada y el volumen de carga de trabajo ejecutado (R). Los 4 lotes tienen las mismas tareas.

TABLA 4.1: Resultados para comprobar la validez del parámetro α

Lote	$\boldsymbol{\alpha}$	**#u.t.**	**%A**	**R**
L1 (ideal)	1,00	114	100	**100**
L2 (sobrecargado)	1,07	114	96,75	**96,34**
L3 (equilibrado)	0,86	133	85,71	**100**
L4 (infrautilizado)	0,43	262	43,51	**100**

A partir de la tabla comprobamos que en condiciones ideales (L1) se hace un uso del 100 % de la FPGA durante el 100 % del tiempo, mientras que en situaciones de sobrecarga (L2) se tienen que desechar algunas tareas ($R <$ 100 % y uso del área de la FPGA cercano al 100 %) y en situaciones de equilibrio de la carga respecto a la capacidad de la FPGA (L3) o infrautilización (L4) se ejecuta el 100 % del volumen asignado aunque el uso del dispositivo es inferior al 100 %, tanto menor cuanto menor sea el valor de α).

4.1.5. Conclusiones

Podemos por lo tanto concluir que el parámetro α, que se obtiene a partir de medidas del sistema en tiempo real, nos dará pautas acerca de cuándo puede

ser conveniente la realización de cambios en el número y tamaño de particiones definidas en la FPGA.

Sin embargo, este parámetro por sí solo no es indicador de los cambios concretos que son necesarios en cada caso, y debemos por tanto definir algún parámetro más que nos indique el cambio en el número de particiones más adecuado en cada caso.

4.2. Análisis del problema para una FPGA con particiones

Pasamos ahora a ampliar el análisis a nuestro caso particular en que el área de la FPGA está dividida en P particiones, sin que P esté restringido al valor de 4 del algoritmo básico expuesto en el capítulo 3. Cuantas más particiones haya, más tareas se podrán ejecutar simultáneamente, pero más pequeño será el tamaño de las particiones. Por el contrario, disponer de particiones grandes significa reducir el número de tareas que se pueden ejecutar simultáneamente. Encontrar el equilibrio entre el número y tamaño de las particiones adecuado para las características de las tareas que llegan al sistema es el objetivo de esta parte del trabajo, que se concreta en el diseño del algoritmo para la Unidad de Adaptación Dinámica.

4.2.1. Distribución de particiones ideal. Parámetro D_p

Decimos que la distribución de particiones D_P es adecuada para el conjunto de tareas que están llegando a ejecutarse en la FPGA cuando se cumple la

siguiente condición:

$$P \approx \frac{A}{\overline{a}} \tag{4.10}$$

que quiere decir que el número y tamaño de particiones coincide con el número medio de tareas que se pueden ejecutar simultáneamente en la FPGA.

Si el sistema no está sobrecargado ni infrautilizado se cumplirá

$$FLLTreal \approx FLLTideal$$

y combinando la ecuación 4.8 con la 4.10, nos dará:

$$FLLTreal \approx \frac{P}{\overline{tej}} \tag{4.11}$$

que quiere decir que en cada intervalo de tiempo $\overline{tej}$ llegan P tareas al sistema, que se pueden ejecutar simultáneamente, y formula la condición de que la distribución de la carga de trabajo en el tiempo sea adecuada para la capacidad del sistema.

Un buen sistema de gestión del área debería conseguir que se cumpla la siguiente ecuación:

$$\sum_{T_i \in P_j} tej_i \approx \frac{\sum_{h=1}^{N} tej_h}{P} \quad \forall P_j \tag{4.12}$$

Esta ecuación refleja el hecho de que la carga de trabajo que llega la sistema será distribuida de forma equitativa entre las particiones de la FPGA y por lo tanto las tareas no se acumularán en las colas y será raro el caso de que se tenga que rechazar una tarea por no poder cumplir con su máximo tiempo de

espera.

Podemos entonces definir el **parámetro $\mathbf{D_p}$**, que mide la adecuación de la distribución en particiones como:

$$D_p = \frac{A}{\overline{a} \cdot P} \tag{4.13}$$

Para un sistema con un número de particiones adecuado, $D_p \approx 1$, que se deduce de la ecuación 4.10 y formaliza la idea de que el número y tamaño de las particiones debe ser proporcional a la relación entre el tamaño de las tareas y el tamaño de la FPGA.

Esta ecuación asimismo refleja el hecho de que cuanto más pequeñas son las tareas, más se pueden ejecutar simultáneamente y por tanto el número de particiones adecuado tiene que ser mayor que en el caso de que los tamaños de las tareas sea grande, en que necesitaremos particiones más grandes y por tanto también disminuirán en número, lo que es acorde con el hecho de que si las tareas son grandes no se podrán ejecutar muchas de ellas simultáneamente.

Para el caso particular de $P = 4$ la ecuación 4.11 refleja que, en promedio, no están llegando más de cuatro tareas por cada intervalo $\overline{tej}$ y cada tarea se puede ejecutar en una partición diferente.

Pasamos ahora a analizar los casos en que el algoritmo no esté realizando una gestión correcta del área de la FPGA, que serán los casos que la UAD debe detectar y corregir.

4.2.2. Distribución de particiones no adecuada

Podemos decir que si se cumple la condición

$$FLLTreal \approx \frac{A}{\overline{a} \cdot \overline{tej}} \tag{4.14}$$

o lo que es lo mismo

$$\alpha \approx 1$$

el sistema está recibiendo una carga de trabajo muy ajustada a su capacidad y si el algoritmo no está realizando una gestión adecuada del área de la FPGA se empezarán a rechazar tareas.

Tendremos dos posibilidades:

1. $D_p > 1$ Esto significa que los tamaños de las tareas son pequeños en relación al tamaño de la FPGA ($\overline{a} \cdot P < A$) y que el número de tareas que se están ejecutando simultáneamente, P, es insuficiente y por lo tanto se está desperdiciando mucha área de la FPGA

2. $D_p < 1$ Esto significa que los tamaños de las tareas son grandes en relación al tamaño de la FPGA ($\overline{a} \cdot P > A$) y que no se pueden ejecutar P tareas simultáneamente

4.3. Cambio dinámico del número de particiones

Disponemos de dos parámetros medibles en tiempo real que nos dan información acerca de la adecuación de la distribución de la carga de trabajo en el espacio y el tiempo con respecto a la capacidad del dispositivo y del uso que

se está haciendo de los recursos de la FPGA con un determinado número de particiones.

Si $\alpha \approx 1$ pero $D_p > 1$ resulta evidente que habrá que dividir el espacio de la FPGA en un número mayor de particiones de menor tamaño para que se pueda ejecutar un mayor número de tareas simultáneamente. Análogamente, si $\alpha \approx 1$ pero $D_p < 1$ es un claro indicativo de que el tamaño de las particiones es excesivamente pequeño para el tamaño de las tareas y una posible solución es reducir el número de particiones para que estas sean de tamaño mayor y no haya que recurrir a la unión de particiones constantemente, ya que esto podría perjudicar el rendimiento del algoritmo.

Sin embargo, la condición $\alpha \approx 1$ se refiere a unas condiciones de la carga de trabajo muy parecidas a las ideales y no es realista suponer que la carga de trabajo las cumplirá. Por lo tanto es preciso analizar el significado del resto de condiciones de la carga de trabajo y decidir qué se debe hacer con respecto al número/tamaño de las particiones en cada una de ellas.

Si α es muy bajo tendremos una carga de trabajo excesivamente pequeña con respecto a la capacidad del dispositivo, lo que significa que el dispositivo está siendo infrautilizado. Desde el punto de vista de su gestión, el problema de asignación de espacio resulta trivial ya que la frecuencia de llegada de las tareas es muy baja y encontrarán a su llegada numerosas opciones de ejecución. Pongamos por ejemplo el caso de una carga de trabajo donde las tareas son todas de un tamaño muy grande, por ejemplo del 50 % del tamaño de la FPGA, con tiempos de llegada entre las tareas iguales a sus tiempos de ejecución. Tendremos una ocupación teórica de la FPGA del 50 % (un $\alpha \approx 0,5$) y la versión básica de 4 particiones podrá ir ejecutando las tareas secuencialmente

sin problemas de no poder cumplir con su tiempo máximo (tendrá siempre la partición más grande disponible para cualquier tarea de tamaño 50 % que llegue al sistema). Lo mismo sucederá con tareas que sean de tamaño muy pequeño: el algoritmo les asignará particiones de mayor tamaño del que necesitan y se desperdiciará mucha área pero esto será debido a la baja afluencia de tareas al sistema y no a una mala gestión.

Por el contrario, si α es alto, tendremos una situación en la que se está demandando al dispositivo que ejecute una carga de trabajo que literalmente no cabe en él. En estas condiciones no puede achacarse el volumen de trabajo rechazado exclusivamente a la gestión que se está haciendo del área del dispositivo. Esto no quiere decir, sin embargo, que no se deba revisar el valor de D_p, ya que precisamente en este tipo de situaciones es muy importante hacer el mejor uso posible del área del dispositivo, con el objetivo de paliar la descompensación existente entre la carga de trabajo y la capacidad de la FPGA.

Si $\alpha > 1$, la adecuación del tamaño/número de particiones a las características de las tareas es vital para conseguir el mejor rendimiento posible en la situación de sobrecarga, aunque no se podrá garantizar en ningún caso la ejecución del 100 % de la carga de trabajo asignada.

En resumen, si α es muy bajo, no merece la pena hacer ningún cambio, y si α es equilibrado, ideal o excesivo, deberemos calcular D_p y realizar los cambios pertinentes para mejorar la eficiencia del algoritmo.

Estamos por lo tanto en condiciones de presentar el *esqueleto* de lo que será el algoritmo de la UAD, basándonos en el cálculo en tiempo real de los parámetros α y D_p, como se muestra en el algoritmo 9.

Algoritmo 9 Base para el algoritmo de la UAD

```
α = FLLT_real/FLLT_ideal;
D_p = A/ā · P;
if (0,5 ≤ α) then
  if (D_P > 1) then
    Aumentar-numero-de-particiones();
  end if
  if (D_p < 1) then
    Disminuir-numero-de-particiones();
  end if
end if
```

Como se ha mencionado, esto no es más que un esqueleto al que le faltan muchos detalles para estar completo, y que se listan a continuación:

1. Formalizar los diferentes tipos de situaciones que pueden darse y determinar el número y tamaño de particiones adecuado para cada una de ellas.

2. Determinar qué datos son necesarios para identificar correctamente estas situaciones, es decir, determinar los valores reales de D_p medidos en tiempo real y los valores concretos que representan las situaciones teóricas correspondientes a $D_p > 1$ y $D_p < 1$. También es necesario determinar el tamaño de la muestra que vamos a considerar durante la ejecución del algoritmo para obtener información realista de lo que está sucediendo en el sistema en tiempo real. Esta muestra debe permitir detectar las situaciones a tiempo y no tomar decisiones precipitadas.

3. Decidir la forma concreta en que se llevará a cabo el cambio en las particiones: qué parámetros es necesario cambiar, qué valores nuevos tomarán en cada situación y cómo se efectuará el cambio de particiones con el mí-

nimo coste posible.

Examinaremos cada uno de estos puntos en detalle en las secciones que siguen, presentando el estudio teórico y su comprobación experimental.

4.4. Estimación del número de particiones adecuado a una carga de trabajo

En esta sección plantearemos la solución al problema de decidir los nuevos tamaños y número de particiones adecuadas a cada carga de trabajo.

El buen rendimiento de nuestro algoritmo está basado en el reparto equitativo de la carga de trabajo entre las particiones. En la sección 4.2.1 hemos visto que si se cumplen las condiciones de carga de trabajo adecuada para el sistema, ecuación 4.10, y distribución adecuada de particiones, ecuación 4.11, obtendremos como consecuencia un reparto equilibrado de la carga de trabajo entre las particiones, ecuación 4.12.

Es necesario por tanto definir los parámetros que ayudarán a caracterizar la carga de trabajo y a identificar las distribución en particiones que es más conveniente en diferentes situaciones.

4.4.1. Caracterización de la carga de trabajo

El problema de representar la carga de trabajo es bastante complejo debido a que en la actualidad carecemos de datos experimentales de sistemas reales como el de nuestro modelo. Existe poca información acerca de los resultados de la síntesis de aplicaciones reales sobre hardware reconfigurable y los autores

que están trabajando en este campo realizan sus pruebas con lotes de tareas artificiales, generados aleatoriamente.

De los datos recopilados a partir de publicaciones donde aparecen datos de síntesis de aplicaciones reales, [KCMO06], [OPF06], [JTR+05], [NHCB02], [RMVC05], podemos concluir que las aplicaciones reales sintetizadas a hardware reconfigurable tienen tamaños muy variados y tiempos de ejecución también variados, y que hay muchas maneras de sintetizar una tarea para su ejecución en hardware reconfigurable y por lo tanto las aplicaciones se pueden dividir en sub-tareas de varias formas diferentes, dando lugar a conjuntos de tareas distintos con grafos de ejecución diferentes, como se muestra en [BdNSSL06].

4.4.1.1. Campana de Gauss

Tomando como ejemplo la realidad cotidiana, vemos que el éxito de la distribución en forma de campana de Gauss se debe precisamente a que refleja de una forma muy exacta lo que sucede en poblaciones estadísticas normales, es decir, donde no existen unas circunstancias específicas (que se considerarían como anómalas y de hecho la medicina occidental actual se basa en este principio) que hagan que predomine una cierta tendencia sobre otras. En estos casos lo que la distribución normalizada de Gauss está reflejando es que la mayoría de casos corresponden a una media (aunque esta media puede variar con el tiempo, la localización espacial, el factor racial etc., como son por ejemplo la estatura de las personas, la edad media en que las mujeres llegan a la menopausia, la prevalencia de determinadas enfermedades, etc.) y que solamente hay una minoría de casos que se alejan de esta media, tanto más minoría cuanto mayor es la desviación con respecto a lo normal.

Aplicando estos mismos principios a la carga de trabajo hipotética que podremos encontrar en este tipo de sistemas cuando empiecen a implementarse, podemos esperar un comportamiento bastante similar. Es decir, que podremos esperar una mayoría de tareas que no sean ni excesivamente pequeñas ni excesivamente grandes y que las tareas con tamaño muy pequeño o muy grande serán menos frecuentes que las de tamaño intermedio. Por supuesto que a lo largo de una ejecución de un sistema de propósito general puede haber períodos de tiempo en que el tipo de aplicaciones a ejecutar sea muy específico y dé lugar a conjuntos de tareas que tienen características muy parecidas entre sí y que por tanto o no corresponden a una distribución de tipo gaussiano, o corresponden a una campana desplazada.

Entendemos por tanto que una aproximación válida para plantear el análisis de las características de la carga de trabajo es la de tomar como caso base la situación en que las tareas de tamaños intermedios serían mayoría y las de tamaños grandes o pequeños serían minoría. La distribución de particiones propuesta y expuesta hasta ahora como punto de partida del planificador responde de forma intuitiva a esta hipótesis y sería la más acertada, al menos como media de la ejecución de un sistema de computación de propósito general. Esta forma de distribuir el área de la FPGA proporciona dos particiones para tareas de tamaño mediano (P_1 y P_2), una para tareas pequeñas (P_0) y otra para tareas grandes (P_3).

Y sin embargo, aunque este modelo de gestión del área de una FPGA funciona suficientemente bien en un amplio rango de casos, no podemos contentarnos con estos resultados ya que en muchos momentos de la ejecución la carga de trabajo de este hipotético sistema seguramente no se ajustará a las

condiciones descritas como normales.

Por ejemplo, para implementar el algoritmo JPEG, todas las tareas son de tamaño pequeño y no hay ninguna tarea de tamaños medianos o grandes (con respecto al área de la Virtex-2), lo que significa que si se ejecuta varias veces seguidas, y sin que haya otras aplicaciones ejecutándose simultáneamente en el sistema, la distribución de tamaños de tareas resultante no será la del caso base sino que corresponderá a otro tipo de distribución muy diferente. Al ser las tareas de tamaño pequeño, la frecuencia con la que podrían llegar al sistema puede ser excesivamente alta para una distribución en 4 particiones y el rendimiento del sistema puede llegar a ser bastante malo. Intuitivamente, sabemos que la solución consiste en aumentar el número de particiones, que serán de tamaños inferiores a las del caso base, pero suficientes para el tipo de tareas que están llegando al sistema.

La figura 4.4 nos muestra una representación del caso básico y dos posibles desviaciones de dicho caso. Empezaremos analizando este tipo de distribuciones y después pasaremos a comprobar la validez del estudio para otros tipos de distribuciones posibles.

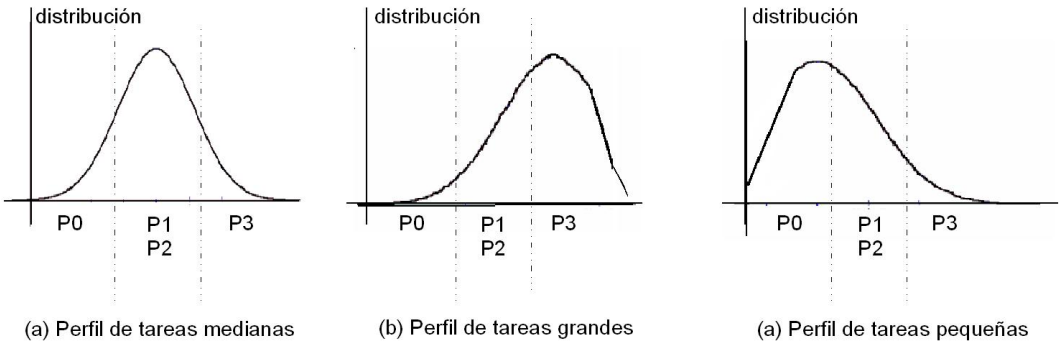

FIGURA 4.4: Distribuciones espaciales de la carga de trabajo

Aunque intuitivamente podamos describir el problema y sus soluciones, es imprescindible recurrir a la formalización, tanto para describir las posibles distribuciones de la carga en el espacio como la de las acciones que es adecuado tomar en cada caso y justificarlas. Es evidente que las posibles variaciones en la distribución de la carga de trabajo son prácticamente infinitas y que es imposible describirlas todas de manera exhaustiva. Nuestro objetivo por lo tanto es partir de una caso fácil de formalizar y analizar, y a partir de este caso base plantear posibles desviaciones y asegurarnos de que las conclusiones a las que llegamos y los cambios que proponemos implementar cubren un rango suficientemente amplio de posibles casuísticas como para poder garantizar la eficacia del algoritmo basado en particiones en prácticamente todas la situaciones.

4.4.1.2. Distribución de la carga de trabajo en el espacio: parámetros β, δ y γ

Hemos estado mencionando los conceptos de tarea pequeña, mediana y grande y manejándolos de una forma intuitiva. Para poder describir los aspectos de una carga de trabajo de forma objetiva necesitamos concretar estos conceptos.

La definición de tamaño de una tarea es relativa y el adjetivo pequeña debe ser utilizado siempre en comparación con otro tamaño. En nuestro caso la comparación del tamaño de las tareas se hace respecto al tamaño de la FPGA, A, medido en CBRs.

Teniendo en cuenta el considerable aumento en el número de CLBs de las FPGAs actuales, como se ha explicado en el capítulo 1, definiremos los tamaños de las tareas de la siguiente manera:

- **Tareas pequeñas:** son aquéllas que como máximo ocupan un 10 % del área de la FPGA. El área de las tareas pequeñas, a_p, cumple la condición $a_p \leq 0,1 \cdot A$

- **Tareas medianas:** son aquéllas que como máximo ocupan un 20 % del área de la FPGA: $0,1 \cdot A < a_m \leq 0,2 \cdot A$

- **Tareas grandes:** son aquéllas que como máximo ocupan un 50 % del área de la FPGA: $0,2 \cdot A < a_g \leq 0,5 \cdot A$

Una Virtex-5 tiene casi 26.000 CLBs, así que para esta FPGA una tarea pequeña sería cualquiera que ocupara menos de 2.600 CLBs, una tarea mediana ocupa entre 2.600 y 5.200 CLBs y una tarea que llegue a ocupar la mitad de la FPGA sería una tarea realmente grande, que ocuparía 13.000 CLBs.

Como el trabajo está desarrollado para gestionar la multi-tarea hardware, no tiene mucho sentido pensar que puede ser habitual que las tareas que se van a ejecutar ocupen más de la mitad del dispositivo. Si esto sucede, el algoritmo pondrá en marcha el mecanismo de unión de particiones y las podrá ejecutar igualmente, pero consideramos que estos casos son excepcionales y no representativos del tipo de sistema que estamos analizando.

Asimismo, el hecho de definir el tamaño de una tarea en función del tamaño de la FPGA dota a estos conceptos de la flexibilidad imprescindible en un contexto tecnológico altamente dinámico donde las FPGAs crecen casi de año en año y nos ha permitido obtener un método aplicable a cualquier FPGA.

Pasamos entonces a definir algunos tipos de distribuciones de la carga de trabajo, que expresaremos en función de los tiempos de ejecución de las tareas y no del número de tareas, ya que se trata de analizar la carga de trabajo por

particiones, y lo que nos interesa es saber el tiempo que una partición de un determinado tamaño estará ocupada. N es el número de tareas en el lote que representa la carga de trabajo del sistema.

Utilizaremos los parámetros β, δ y γ para caracterizar las cargas de trabajo en función de la proporción de tareas de un determinado tamaño:

1. **Distribución de tareas pequeñas:** es una carga de trabajo representada por un lote de tareas L_p donde la mayoría de tareas son de tamaño pequeño y el resto de tamaños se reparte en partes iguales. Cumple las condiciones:

   a) $\sum_{a_i \leq a_p} tej_i = \beta \cdot \sum_{h=1}^{N} tej_h$

   b) $\sum_{a_p < a_i < a_g} tej_i = \delta \cdot \sum_{h=1}^{N} tej_h$

   c) $\sum_{a_i \geq a_g} tej_i = \gamma \cdot \sum_{h=1}^{N} tej_h$

   d) $\beta + \delta + \gamma = 1$

   Los valores $\beta = 0,5$, $\delta = 0,25$ y $\gamma = 0,25$ caracterizan a este tipo de distribución y por lo tanto de ahora en adelante utilizaremos la notación $L_p = (0,5; 0,25; 0,25)$ para referirnos a ella ya que representa la idea expresada de forma intuitiva en la figura 4.4c.

2. **Distribución de tareas medianas:** es una carga de trabajo representada por un lote de tareas L_m donde la mayoría de tareas son de tamaño mediano y el resto de tamaños se reparte en partes iguales. Cumple las condiciones a), b), c) y d) con valores $\beta = 0,25$, $\delta = 0,5$ y $\gamma = 0,25$. De ahora en adelante nos referiremos a este tipo de distribución con la

notación $L_m = (0,25; 0,5; 0,25)$ ya que representa la idea expresada de forma intuitiva en la figura 4.4a.

3. **Distribución de tareas grandes:** es una carga de trabajo representada por un lote de tareas L_g donde la mayoría de tareas son de tamaño grande. Cumple las condiciones a), b), c) y d) con valores $\beta = 0,25$, $\delta = 0,25$ y $\gamma = 0,5$. De ahora en adelante nos referiremos a este tipo de distribución con la notación $L_g = (0,25; 0,25; 0,5)$ ya que representa la idea expresada de forma intuitiva en la figura 4.4b.

4. **Distribución de tareas casi total de pequeñas:** es una carga de trabajo representada por un lote de tareas L_{pp} donde prácticamente todas las tareas son de tamaño pequeño. Cumple las condiciones a), b), c) y d) con valores $\beta = 0,80$, $\delta = 0,10$ y $\gamma = 0,10$. De ahora en adelante nos referiremos a este tipo de distribución con la notación $L_{pp} = (0,80; 0,10; 0,10)$.

Vamos ahora a definir un método para relacionar las características de la distribución de tareas con el número de particiones adecuado para cada una de ellas.

4.4.1.3. Número de particiones ideal para cada tipo de distribución

Para realizar este cálculo debemos retomar la ecuación 4.13 que indicaba una correcta adecuación de la distribución de particiones para un determinado conjunto de tareas caracterizado por su área media $\overline{a}$:

$$D_p = \frac{A}{\overline{a} \cdot P}$$

Y recordando que la condición para que una distribución sea adecuada es:

$$D_p \approx 1$$

Definimos el **área media que caracteriza a un tipo de distribución** como la media ponderada de las áreas representativas de cada tipo de tareas. Una definición general válida para cualquier distribución será:

$$\overline{a}_{L_t} = \beta \cdot a_p + \delta \cdot a_m + \gamma \cdot a_g \tag{4.15}$$

donde a_p es el área que representa a las tareas de tipo pequeño, a_m es el área que representa a las tareas de tipo mediano y a_g es el área que representa a las tareas de tipo grande y los valores de β, δ y γ son los que caracterizan a la distribución L_t.

Si queremos saber cual es el número idóneo de particiones para distribuir el espacio de la FPGA conforme a la distribución de tamaños de la tareas, tendremos que tomar como valor representativo de cada tipo de tareas el máximo de cada tamaño, para asegurarnos de que todas las tareas de un tipo caben en las particiones asignadas a dicho tipo. De otra forma, la distribución de particiones obtenida no podrá asegurar que las tareas de mayor tamaño de un tipo quepan en las particiones diseñadas a tal fin.

Por lo tanto, hemos utilizado el máximo tamaño de tareas de cada tipo para calcular el valor de P adecuado para cada tipo de distribución y obtenemos los siguientes resultados:

1. $\overline{a}_{L_p} = 0,225 \cdot A \Rightarrow P = 4,44$

2. $\overline{a}_{L_m} = 0,250 \cdot A \Rightarrow P = 4$

3. $\overline{a}_{L_g} = 0,325 \cdot A \Rightarrow P = 3,07$

4. $\overline{a}_{L_{pp}} = 0,150 \cdot A \Rightarrow P = 6,6$

Para el caso que hemos utilizado como caso base, L_m, obtenemos un valor de $P = 4$, lo que coincide con la distribución básica presentada hasta este momento.

Para el caso de tareas grandes, obtenemos un valor de P cercano a lo que la intuición nos diría: si llegan menos tareas porque son muy grandes, es mejor tener menos particiones y que cada partición sea grande, para evitar tener que estar uniendo particiones con frecuencia y dado que con tamaños comprendidos entre el 20 % y el 50 % del área de la FPGA va a ser prácticamente imposible ejecutar más de 3 tareas simultáneamente.

El resultado que puede parecernos sorprendente, porque contradice lo que en un principio tenderíamos a pensar, que hay que aumentar el número de particiones para ejecutar más tareas de las pequeñas, es el del número de particiones obtenido para la distribución de tareas de tamaño pequeño.

Un resultado $P = 4,4$ significa que no está claro que en un caso así lo más conveniente sea aumentar el número de particiones. Y también significa que una distribución del área en 4 particiones puede resultar suficiente aunque la mayoría de tareas sean pequeñas.

En primer lugar y como ya se ha comentado anteriormente, el caso de que las tareas que llegan sean pequeñas no es el más crítico ya que se pueden ejecutar en cualquiera de las 4 particiones y por tanto es muy difícil que la partición más pequeña, P_0, llegue a saturarse.

En segundo lugar, si en la distribución sigue habiendo una cantidad alta de tareas grandes (una de cada 4 es de tamaño grande) parece que tiene sentido mantener una partición grande para ellas y evitar recurrir a la unión de particiones con una frecuencia que resultaría excesivamente alta.

Lo que realmente significa una mayoría de tareas pequeñas es que de cada cuatro tareas que llegan, dos van a ser pequeñas, una mediana y otra grande. Y esto en relación a la distribución de medianas, supone que una de las medianas se ha sustituido por una de las pequeñas, de cada grupo de 4, en media. Lo cual no puede tener un gran impacto ni en la $FLLTideal$ para este tipo de distribución ni en el área media que la representa, como puede verse en los resultados obtenidos (pasa de 0,25 para medianas a 0,225 para las pequeñas, debido a que el tamaño de las grandes tiene mucho más peso en el cálculo que el tamaño de las pequeñas).

Una vez hecha esta reflexión, deja de resultarnos tan sorprendente el resultado y encontramos que tiene sentido. Más adelante presentaremos varias pruebas realizadas que confirman estos resultados teóricos.

Para una carga de trabajo donde prácticamente todas las tareas son pequeñas resulta más eficiente aumentar el número de particiones. Para este caso obtenemos un $\overline{a}_{L_{pp}} = 0,15 \cdot A$, que se acerca mucho al tamaño medio de las tareas pequeñas.

El mismo resultado se obtendría también para una distribución del tipo L_p donde las tareas grandes fueran del mínimo tamaño posible. En este caso tendríamos un $\overline{a}$ medida en tiempo real más parecida a la de L_{pp} que a la de L_p y el parámetro D_p lo indicaría subiendo por encima del valor ≈ 1 y se determinaría un cambio a 6 particiones, lo que como veremos en los resultados

presentados, resulta más conveniente que mantener 4 particiones. Hacemos esta aclaración porque el valor teórico obtenido para L_p es de $P = 4,4$, lo que, como decíamos, significa que no está completamente claro que dividir el área en 4 particiones sea lo más adecuado siempre para este tipo de distribuciones.

Podríamos también definir un perfil donde casi todas las tareas sean grandes. Si aplicamos la definición anterior al tamaño de tareas grandes, obtendremos un resultado de $P = 2$, nada sorprendente pues si prácticamente todas las tareas que llegan pueden alcanzar un tamaño del 50 % de la FPGA lo que intuitivamente haríamos es dividir la FPGA en dos mitades y ejecutar una tarea en cada una de ellas. Sin embargo, este tipo de distribución correspondería a una carga de trabajo muy elevada y la $FLLT_{ideal}$ será mucho más baja que en cualquiera de los otros casos, incluido el de tareas grandes. Si además observamos que no todas las tareas que lleguen ocuparán media FPGA y que muchas tareas de las grandes pueden ser de tamaño inferior o igual al 25 % de la FPGA, nos damos cuenta de que no merece la pena añadir más casuísticas al algoritmo que ejecuta la UAD ya que la solución de 3 particiones propuesta para un tipo de distribución de mayoría de tareas grandes (siempre que los tamaños de las 3 particiones resultantes sean los adecuados) es más que suficiente para estos raros casos. De hecho, si tenemos 3 particiones y una de ellas es de tamaño el 50 % del área de la FPGA, si alguna de las dos tareas es de tamaño mayor al 25 % de la FPGA puede ejecutarse en dicha partición y la otra en alguna de las dos que quedan libres. Si no cabe en ninguna de las otras dos, se unirán, lo que no supondrá un sobrecoste al sistema ya que nunca estarán ocupadas por tareas pequeñas cuyo fin de ejecución haga esperar a las más grandes.

4.4.1.4. Verificación con resultados experimentales

A continuación presentamos el resultado de ejecutar diferentes tipos de distribuciones con el caso base, es decir, con una FPGA dividida en 4 particiones de diferente tamaño. Las pruebas se han realizado simulando una FPGA de 50*50 CBRs. La tabla 4.2 muestra las características de los lotes de tareas utilizados y los resultados obtenidos de su ejecución con 4 particiones.

TABLA 4.2: Ejecución de diferentes tipos de distribuciones con 4 particiones

Lote	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$	$\overline{\mathbf{a}}$ (%**A**)	$\mathbf{D_p}$ **4P**	**R** (%$\mathbf{V_{ej}}$)
L1	1	0,25	0,5	0,25	25,00	1	**100**
L2	0,90	0,25	0,5	0,25	23,23	1,11	**100**
L3	0,79	0,5	0,25	0,25	19,77	1,26	**100**
L4	0,75	0,5	0,25	0,25	13,00	1,92	**60**
L5	0,69	0,8	0,1	0,1	12,01	2,07	**56**
L6	0,70	0,25	0,25	0,5	30,02	0,78	**85**

El lote L1 es un lote ideal, para el cual se obtiene un D_p exactamente igual a 1. El siguiente lote, L2, es un lote que corresponde a una distribución de medianas, con una frecuencia de llegada de tareas alta y cercana a la ideal y con un área media ligeramente superior a la de las tareas medianas (por efecto del peso del área de las tareas grandes en el cálculo de la media) y que se ejecuta sin problemas y en su totalidad con una distribución en 4 particiones. Para este caso el valor de D_p es ligeramente superior al valor teórico de 1.

El lote L3 es un lote del tipo de mayoría pequeñas en el que muchas de las tareas grandes tienen tamaños cercanos al 50 % del área de la FPGA, por lo que el área media resultante está más cercana al tamaño de las grandes (de hecho es el máximo de las medianas) y su ejecución en 4 particiones resulta

también muy eficiente. El valor de D_p está también por encima de 1.

En el caso del lote L4, donde el tamaño de las tareas grandes es bastante menor que el 50 % del área de la FPGA, vemos que el área media resultante (13 %) está en el margen inferior del tamaño de las medianas y por debajo del área teórica del 15 % obtenida para L_{pp}. En este caso la frecuencia de llegada de tareas es relativamente alta y D_p es cercano al doble del teórico 1 que indica una buena distribución del área de la FPGA en particiones. Algo similar sucede con el lote L5 donde prácticamente no hay tareas grandes (y estas no llegan a ocupar la mitad de la FPGA en ningún caso).

Por último el lote L6 representa una distribución de tareas grandes. El área media es muy grande, la tercera parte de la FPGA, y el valor de D_p para este tipo de distribución es inferior a 1.

Con estos resultados podemos confirmar que para las distribuciones presentadas solamente serían necesarios los cambios en las particiones para los lotes L4, L5 y L6, que pertenecen a las situaciones en que D_p se aleja mucho de su valor de 1 teórico. Observamos también que la variación de D_p por debajo de su valor teórico y el real es asimétrico y que es más importante examinar el área media del conjunto de tareas a la hora de tomar decisiones que los valores de β, δ y γ.

Presentamos también la tabla 4.3 que resume el estudio teórico del número de particiones adecuado para diferentes tipos de distribuciones de tipo gaussiano y no gaussiano.

Podemos observar que con un modelo de adaptación dinámica que maneje distribuciones del área de la FPGA en 3, 4 y 6 particiones se pueden cubrir la mayoría de casos. No así casos extremos, como la situación en que todas

TABLA 4.3: Cálculo del número de particiones necesario para diferentes distribuciones

$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$	$\overline{\mathbf{a}}\,(\,\%\mathbf{A})$	**P**
0,25	0,5	0,25	25,00	4,00
0,5	0,25	0,25	22,50	4,40
0,25	0,25	0,5	32,50	3,07
0,2	0,4	0,4	30,00	3,33
0,4	0,2	0,4	28,00	3,57
0,2	0,4	0,2	22,00	4,54
0,8	0,1	0,1	15,00	6,66
1	0	0	10,00	10,00
0	1	0	20,00	5,00
0	0	1	50,00	2,00
0,33	0,33	0,33	26,66	3,75

las tareas son pequeñas, aunque sí incluye (gracias a la unión de particiones) situaciones en que la totalidad de las tareas sean grandes (usando 3 particiones y uniendo las dos más pequeñas cuando sea necesario). También podemos observar que el paso a 6 particiones solamente merece la pena cuando el número de tareas realmente grandes (que ocupen más del 25 % de la FPGA) sea muy pequeño (o nulo) en relación al resto de tareas.

A continuación presentamos la tabla 4.4 con los resultados de ejecutar los lotes anteriores con el número de particiones adecuado para sus características.

Los resultados presentados en esta tabla nos muestran que una distribución en particiones adecuada a las características de la carga de trabajo permite ejecutar el total de la carga asignada, es decir, obtener un buen rendimiento del algoritmo en prácticamente cualquier situación posible.

En la siguiente sección presentamos detalladamente la forma en que se hará el reparto del área de la FPGA para cada uno de estos tres casos: $P = 3$, $P = 4$

TABLA 4.4: Ejecución de diferentes tipos de distribuciones con 4 particiones

Lote	**Tipo**	**Nº particiones**	**R** (%$\mathbf{V_{ej}}$)
L1	L_m (ideal)	4	**100**
L2	L_m	4	**100**
L3	L_p	4	**100**
L4	L_{pp}	6	**100**
L5	L_{pp}	6	**100**
L6	L_g	3	**100**

y $P = 6$.

4.4.2. Forma y tamaño de las distintas distribuciones en particiones

- **Distribución de tareas pequeñas:** Para un L_p que contenga una cuarta parte de tareas que ocupan hasta media FPGA se ha obtenido un $P = 4,4$ que daría como resultado una distribución en particiones del área de la FPGA en 4 particiones de diferente tamaño, correspondientes a las del caso base representadas en la figura 4.5. En este caso la partición más grande P_3 ocupa la mitad de la FPGA, las particiones P_2 y P_1 ocupan el 20 % y la partición pequeña, P_0 ocupa el 10 % del área de la FPGA.

- **Distribución de tareas medianas:** Para L_m se ha obtenido también un $P = 4$ que daría como resultado una distribución del área de la FPGA en 4 particiones de diferente tamaño, correspondientes a las del caso base representadas en la figura 4.5.

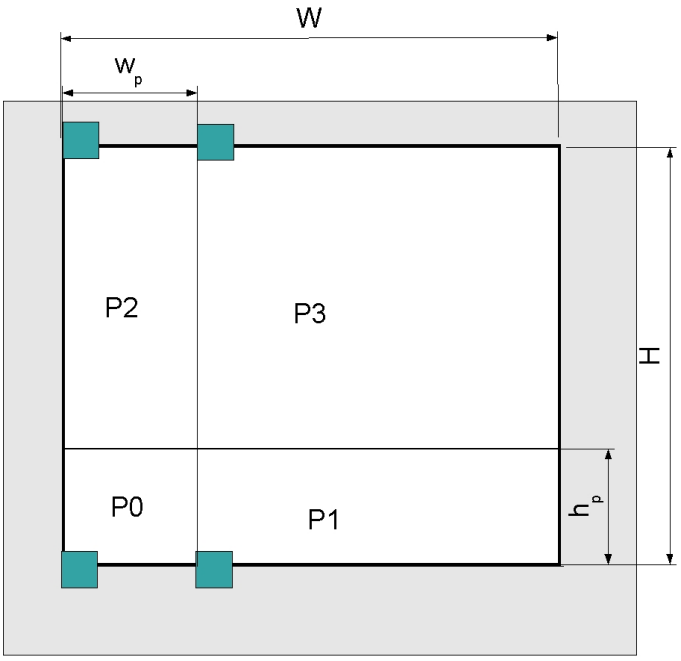

FIGURA 4.5: Particiones adecuadas para una distribución de tareas pequeñas y medianas

- **Distribución de tareas grandes:** Para L_g se ha obtenido un $P = 3$ que daría como resultado una distribución en particiones del área de la FPGA en 3 particiones de diferente tamaño, correspondientes a las representadas en la figura 4.6. En este caso la partición más grande, P_2, ocupa la mitad de la FPGA y la otra mitad está dividida en dos partes de igual tamaño. Esta distribución en particiones permite ejecutar una tarea de tamaño 50 % del área de la FPGA (y hasta 2 si unimos las otras dos particiones) y hasta 3 tareas grandes, siempre que las otras dos no superen el 25 % del tamaño de la FPGA, sin por ello perjudicar la planificación de tareas más pequeñas que pueden llegar intercaladas con las grandes.

- **Distribución casi total de tareas pequeñas:** Para L_{pp} (y L_p con el

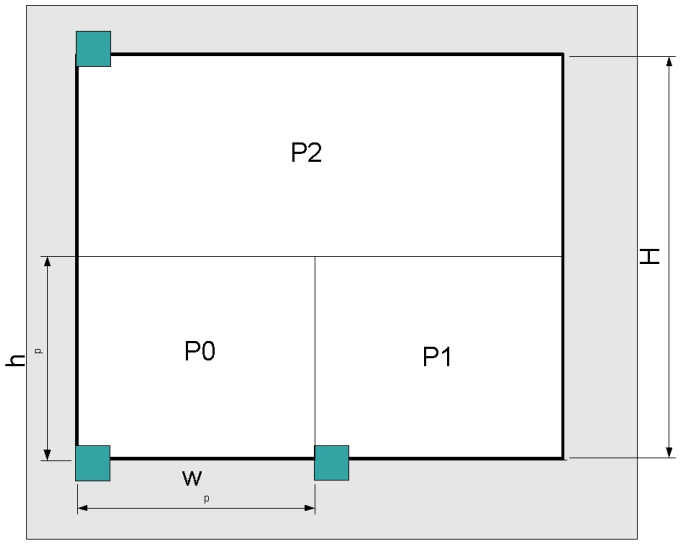

FIGURA 4.6: Particiones adecuadas para una distribución de tareas grandes

100 % de tareas grandes del mínimo tamaño) se ha obtenido un $P = 6,6$ que daría como resultado una distribución del área de la FPGA en 6 particiones. Los tamaños de estas particiones podrían ser todos iguales entre sí (16,66 % del tamaño de la FPGA), es decir, que obtendríamos una división en particiones que resultarían demasiado pequeñas para cualquier tarea grande y muchas medianas que llegaran. Otra opción, que nos parece más acertada, es dividir el área de la FPGA en 6 particiones de diferente tamaño, según muestra la figura 4.7, donde 4 de las particiones pueden ser muy pequeñas (12,5 % del área de la FPGA) mientras que las otras dos pueden ser algo mayores (25 % del área de la FPGA), lo que evita tener que realizar unión de particiones en muchos casos y no disminuye el número de particiones disponibles para las tareas pequeñas.

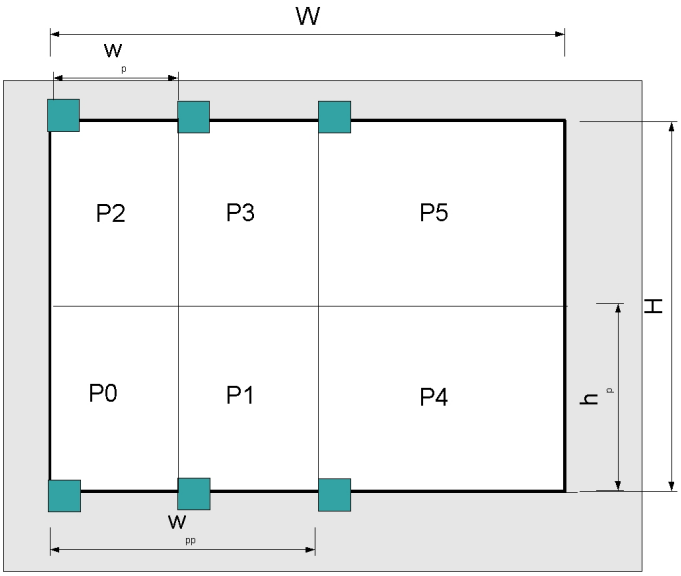

FIGURA 4.7: Particiones adecuadas para una distribución con casi todas las tareas pequeñas

4.4.3. Resultados experimentales

Mostramos aquí un conjunto de pruebas cuyo objetivo es comprobar la validez de estos tres tipos de distribuciones propuestas, realizadas con una serie de lotes de tareas que se ajustan a los modelos teóricos de distribuciones expuestos anteriormente. Primero presentaremos los resultados obtenidos con lotes de tareas medianas y grandes y después los resultados obtenidos para lotes con tareas fundamentalmente pequeñas.

4.4.3.1. Ejecución de lotes de tareas medianas y grandes

La tabla 4.5 presenta las comparativas entre la ejecución en 4P y 3P de lotes de tareas medianas y grandes con diferentes cargas de trabajo.

Los lotes L1, L2 y L3 corresponden a distribuciones de tareas medianas. Su área media es más cercana al tamaño máximo de las medianas que a cualquiera de los otros dos. El valor de D_p teórico calculado para estos lotes de tareas

TABLA 4.5: Comparativa 4P - 3P

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ **(%A)**	$\mathbf{D_p}$ **4P**	$\mathbf{D_p}$ **3P**	**R 4P (%)**	**R 3P (%)**
L1	0,92	22,54	1,11	1,48	**100**	**76,91**
L2	0,90	23,23	1,11	1,50	**100**	**76,00**
L3	0,77	21,03	1,19	1,58	**100**	**87,96**
L4	0,76	26,93	0,93	1,24	**100**	**92**
L5	0,75	28,56	0,87	1,16	**83,88**	**100**
L6	0,69	28,75	0,86	1,15	**89,62**	**100**
L7	0,70	29,73	0,84	1,12	**85**	**100**
L8	0,55	29,73	0,84	1,12	**91**	**100**
L9	0,44	29,73	0,84	1,12	**100**	**100**

indica que su ejecución será mejor con una distribución en 4 particiones que en 3 particiones, lo que viene confirmado por los resultados obtenidos.

El lote L4 corresponde a una distribución particular, que hemos denominado *lote plano*, ya que el porcentaje de tareas de cada tipo es exactamente igual. Corresponde a la distribución reflejada en la última línea de la tabla 4.3, para la cual se obtenía un valor de $P = 3,75$. Vemos que el valor de D_p es más cercano al teórico cuando se calcula para una distribución en 4 particiones que para una de 3 particiones y los resultados experimentales confirman que su ejecución efectivamente es mejor con 4 particiones, aunque la diferencia en rendimiento no es excesiva.

Cuando la distribución se aleja del perfil de tareas medianas, como es el caso del resto de los lotes de tareas, vemos que al aumentar el área media (por efecto de una mayor presencia de tareas grandes en el lote) los valores calculados de D_p se acercan más al valor ideal cuando se calcula para 3 particiones que para 4 particiones. La ejecución de todos los lotes excepto uno de ellos demuestran que la distribución del área de la FPGA en 3 particiones es mucho más adecuada

para este tipo de distribuciones.

Examinaremos ahora en detalle lo que sucede con los lotes L7, L8 y L9. De hecho se trata exactamente del mismo lote de tareas (mismo área media y mismo volumen asignado) y lo único que los distingue es la frecuencia de llegada de tareas. Cuanto más se acerca α al valor ideal que corresponde a la situación de carga de trabajo que llena el espacio de la FPGA prácticamente la totalidad del tiempo, más diferencia encontramos en el rendimiento entre la ejecución en 4 particiones y la ejecución en 3 particiones.

Estos resultados confirman que para un sistema infrautilizado, como el L9, con un valor de $\alpha = 0,44 < 0,5$, la frecuencia de llegada de tareas es tan baja que cualquiera de las dos distribuciones en particiones daría buenos resultados. Esto ya se comentó en la sección 4.1.2 y quedó reflejado en el esqueleto de algoritmo para la UAD presentado, en el cual solamente se calcula el valor de D_p para sistemas que tangan un $\alpha \geq 0,5$.

4.4.3.2. Ejecución de tareas fundamentalmente pequeñas

A continuación presentamos la tabla 4.6, que muestra la comparativa de la ejecución de lotes de tareas de tipo casi todas pequeñas en 4P y 6P.

En esta tabla todos los lotes de tareas, excepto el L6, son de tareas muy pequeñas, donde ninguna supera el 25 % del tamaño de la FPGA. Vemos que el área media de los lotes L1, L2, L3, L4 y L5 están ligeramente por debajo del 15 % teórico obtenido para L_{pp}. En todos estos casos se observa que los valores de D_p mas cercanos a 1 corresponden a una distribución en 6 particiones y que la ejecución de estos lotes de tareas con 4 particiones es claramente peor que para 6 particiones, a excepción del lote L5 donde es exactamente

TABLA 4.6: Comparativa 4P - 6P

Lote	$\boldsymbol{\alpha}$	$\bar{\mathbf{a}}$ (%**A**)	$\mathbf{D_p}$ **4P**	$\mathbf{D_p}$ **6P**	**R 4P**(%)	**R 6P**(%)
L1	0,69	13,88	1,80	1,20	**80,57**	**100**
L2	0,58	14,16	1,76	1,17	**84,82**	**100**
L3	0,62	14,00	1,78	1,19	**83,12**	**100**
L4	0,55	14,00	1,78	1,19	**96**	**100**
L5	0,46	14,00	1,78	1,19	**100**	**100**
L6	0,79	19,77	1,26	0,84	**100**	**85,27**
L7	0,13	3,65	27,39	18,26	**100**	**100**

igual para ambas distribuciones en particiones. Los lotes L3, L4 y L5 son exactamente el mismo conjunto de tareas pero con frecuencias de llegada de tareas decrecientes. Se pone de nuevo de manifiesto que para una frecuencia de llegada correspondiente a un sistema infrautilizado no merece la pena realizar cambios en las particiones.

El lote L6 corresponde a una distribución de tipo L_p, donde la mitad de las tareas son de tamaño pequeño. Sin embargo hay un 25 % de tareas de tamaño cercano al 50 % de la FPGA. El área media de las tareas de este lote es prácticamente el máximo de las tareas medianas. En el estudio teórico habíamos obtenido un valor de $P = 4,4$ para este tipo de distribuciones. Los resultados muestran que la ejecución con 6 particiones es peor que con 4 ya que estas tareas tan grandes aparecen con bastante frecuencia (una de cada cuatro es grande) y necesitan 4 de las particiones pequeñas para ejecutarse. La unión de estas cuatro particiones del esquema en 6 particiones hace que solamente queden 2 particiones libres para ejecutar el resto de tareas, mientras que en el esquema de 4 particiones aún quedarían 3 libres, y de ahí el bajo rendimiento en la ejecución en 6 particiones para este lote.

El lote L7 corresponde a la simulación de 12 ejecuciones de JPEG en una versión de síntesis donde algunas tareas se desdoblan (JPEG paralelo). La síntesis se realizó para una Virtex-2 XCV2P30, de tamaño mucho menor que una Virtex-5, y aún así el resultado es un conjunto donde el 100 % de tareas son pequeñas. Estas 12 ejecuciones solapadas de JPEG paralelo dan un buen resultado tanto con un esquema de 4 particiones como con uno de 6 particiones ya que su frecuencia de llegada es muy baja. El tamaño medio de las tareas, 3,65 %, es menos de la mitad del tamaño máximo de las tareas pequeñas, 10 %, y de ahí los exageradamente altos valores obtenidos para D_p que indican que en caso de aumentar la frecuencia de llegada de tareas sería imprescindible aumentar el número de particiones a un número mayor incluso que 6.

Por tanto, hemos comprobado de forma experimental que las distribuciones en particiones obtenidas son adecuadas para una mayoría de distribuciones de tareas.

4.5. Observación del sistema en tiempo real

4.5.1. Variación de D_p en tiempo real

Los resultados obtenidos hasta el momento han sido realizados a partir de cálculos teóricos sobre un conjunto de tareas que se conoce de antemano. Cuando el valor de D_p se calcule en tiempo real, mostrará variaciones respecto a este cálculo teórico debido a que estaremos tomando subconjuntos del lote sobre el que se realiza el cálculo. En las tablas 4.5 y 4.6 podemos también observar que cuando las tareas de un lote no se ajustan a las condiciones

ideales (los tamaños de las tareas tendrían que ser exactamente iguales a los tamaños de las particiones para que no se desaprovechara área de FPGA) los valores de D_p para el lote completo se desvían de los teóricos.

Mostraremos un ejemplo de la variación de D_p cuando se calcula en tiempo real y pasaremos después a analizar resultados empíricos con diferentes tipos de distribuciones que nos indicarán la variación en tiempo real de D_p y de ellos obtendremos las pautas para determinar las condiciones en que el algoritmo de la UAD deberá decidir los cambios de particiones.

4.5.2. Ejemplo de variación de D_p en tiempo real

Tomaremos el fragmento del lote de tareas que componen el lote ideal $L1$ mostrado en la tabla 4.1. Este lote, examinado en su totalidad, corresponde al caso en que $D_p = 1$. Sin embargo al analizar los valores calculados para D_p en tiempo real, observamos que oscilan entre $0,91$ y $1,11$, es decir aproximadamente un $10\,\%$ del valor teórico para el lote. La figura 4.8 nos muestra una gráfica con los valores calculados a lo largo de la ejecución de este lote, para una distribución en 4 particiones y con una ventana de 16 tareas (por ello el primer valor de D_p se obtiene en $u.t = 24$).

La tabla 4.7 nos muestra las 10 primeras tareas de $L1$. El valor de D_p para este subconjunto de tareas es $D_p = 1,09$. Veremos sin embargo que en tiempo real este valor varía por encima y por debajo de su valor teórico, dependiendo del subconjunto de tareas presentes en la ventana de tareas en el momento de realizar el cálculo.

Vamos a ir calculando el valor de D_p como lo haría la UAD, tomando las m

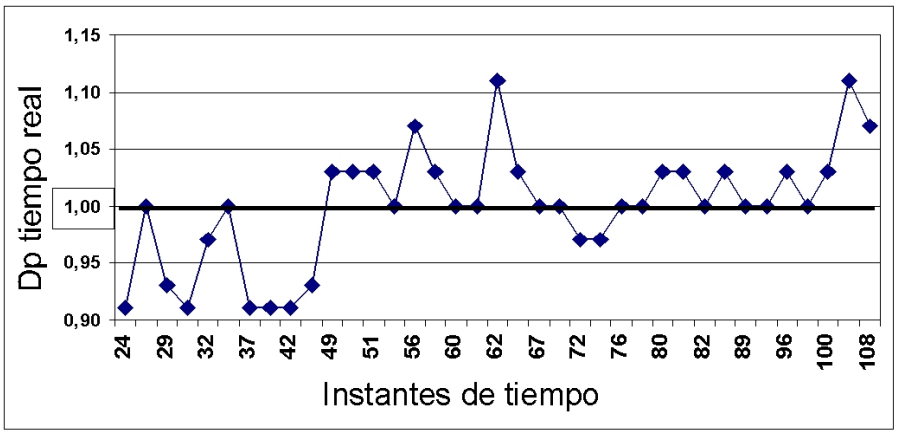

FIGURA 4.8: Oscilaciones del valor de D_p en tiempo real I

últimas tareas que han llegado al sistema. Como se trata de un ejemplo sencillo, cogeremos un número de tareas igual al número de particiones, es decir, cuatro. En la figura 4.9 se ve que estas 4 primeras tareas son exactamente una grande, dos medianas y una pequeña de tamaños iguales a las particiones existentes, y que por tanto la primera vez que se calcula D_p se obtiene el valor 1.

Tarea	w_i	h_i	tll_i	tej_i	$tmax_i$
T1	35	35	1	5	11
T2	15	35	1	6	13
T3	35	15	1	8	17
T4	15	15	1	7	15
T5	35	35	6	10	26
T6	15	35	7	5	17
T7	15	15	8	5	18
T8	35	15	9	9	27
T9	15	35	12	6	24
T10	15	15	13	12	37

$D_p = 1$

FIGURA 4.9: Dp para la primera ventana de tareas

Al desplazar la ventana de tareas, sale del cálculo la tarea $T1$ y entra la $T5$, dando de nuevo como resultado un $D_p = 1$. Para la tercera ventana volvemos a obtener un valor $D_p = 1$ ya que de nuevo tenemos un subconjunto de tareas

TABLA 4.7: Parte de un lote de tareas ideal

Tarea	$\mathbf{w_i}$	$\mathbf{h_i}$	$\mathbf{tll_i}$	$\mathbf{tej_i}$	$\mathbf{tmax_i}$
T1	35	35	1	5	11
T2	15	35	1	6	13
T3	35	15	1	8	17
T4	15	15	1	7	15
T5	35	35	6	10	26
T6	15	35	7	5	17
T7	15	15	8	5	18
T8	35	15	9	9	27
T9	15	35	12	6	24
T10	15	15	13	12	37

que representa la distribución L_m de forma perfecta.

Puesto que el orden de llegada de tareas no tiene por qué ser grande-mediana-mediana-pequeña, lo que sucede es que llega un momento en que al desplazar la ventana de tareas, el grupo con el que se realiza el cálculo puede no corresponder exactamente a la distribución L_m. Esto sucede al tomar la cuarta ventana de tareas, como muestra la figura 4.10.

Tarea	w_i	h_i	tll_i	tej_i	$tmax_i$
T1	35	35	1	5	11
T2	15	35	1	6	13
T3	35	15	1	8	17
T4	15	15	1	7	15
T5	35	35	6	10	26
T6	15	35	7	5	17
T7	15	15	8	5	18
T8	35	15	9	9	27
T9	15	35	12	6	24
T10	15	15	13	12	37

Dp = 1,14

FIGURA 4.10: Dp para la cuarta ventana de tareas

Al salir la tarea $T4$ de la ventana y entrar la $T8$ volvemos a tener una

distribución perfecta de L_m y de nuevo obtenemos un $D_p = 1$. En la siguiente ventana volvemos sin embargo a obtener un valor de $D_p = 1,39$. Al realizar el último cálculo de D_p para este fragmento del lote ideal $L1$ obtenemos de nuevo un valor de D_p alejado del valor 1, como muestra la figura 4.11.

Tarea	w_i	h_i	tll_i	tej_i	$tmax_i$
T1	35	35	1	5	11
T2	15	35	1	6	13
T3	35	15	1	8	17
T4	15	15	1	7	15
T5	35	35	6	10	26
T6	15	35	7	5	17
T7	15	15	8	5	18
T8	35	15	9	9	27
T9	15	35	12	6	24
T10	15	15	13	12	37

Dp = 1,66

FIGURA 4.11: Dp para la última ventana de tareas

Este sencillo ejemplo nos muestra dos aspectos importantes que hay que tener en cuenta a la hora de determinar cambios en las particiones a partir del valor obtenido para el cálculo de D_p:

1. **Oscilación del valor de D_p**: el parámetro D_p obtenido en tiempo real oscila por encima y por debajo del valor real de D_p para un conjunto de tareas. Es necesario determinar empíricamente, a partir de distribuciones de tareas conocidas, cuál es el margen de variación de D_p por encima de 1 que hace necesario un aumento en el número de particiones. De forma análoga, aunque no simétrica, es necesario también determinar de manera empírica el valor real de D_p por debajo de 1 que indica la necesidad de disminuir el número de particiones.

2. **Cambio en la distribución**: para determinar un cambio en las parti-

ciones no es suficiente con que se dé la condición $D_p > 1$ o $D_p < 1$ una única vez, ya que esto podría deberse al orden en que llegan las tareas que se están considerando. Podríamos decir, a una *casualidad*. Experimentalmente hemos comprobado que al menos debemos esperar tres veces a que el valor de D_p calculado en tiempo real muestre la misma tendencia de desviación para realizar el cambio en las particiones, ya que si no fuera así, las oscilaciones en el valor de D_p calculado en tiempo real llevarían a un frecuente cambio de particiones innecesario.

Si al examinar un conjunto de tareas se observa un desequilibrio y al salir una tarea del cálculo y entrar otra nueva observamos que esta tendencia no se repite, y que no se vuelve a repetir en las siguientes dos veces que lo realicemos, no parece que la tendencia observada corresponda a un verdadero cambio en la distribución sino a una desviación del mismo (los conjuntos de tareas que llegan al sistema no tienen por qué ajustarse a ninguna de las distribuciones de forma exacta).

El hecho de que en un subconjunto pequeño de tareas se observe que no se está aprovechando adecuadamente el área de dispositivo (o que hay un exceso de área solicitada) no tiene mayor importancia si la situación no se repite dando lugar a retrasos en la ejecución de las siguientes tareas. De ahí la importancia de incluir un criterio acumulativo para la condición de desequilibrio, que se pondrá de manifiesto si las situaciones se reiteran en ventanas de tiempo contiguas.

4.5.3. Valores críticos de D_p

A continuación presentamos las pruebas realizadas para determinar los valores empíricos de D_p que obligan al cambio en las particiones. Ya hemos visto que en caso del lote ideal D_p oscila entre los valores $0,9$ y $1,11$. Es de esperar que para una distribución de medianas no ideal, D_p presente mayores oscilaciones aún. En la tabla 4.8 los lotes L1, L2 y L3 deberían ejecutarse en 4 particiones (ideal, L_m y L_p con tareas de tamaño cercano al 50 % de la FPGA). Se ha calculado D_p con una ventana de 16 tareas. Los lotes están compuestos por conjuntos de 65 tareas.

TABLA 4.8: Distribuciones y variación de D_p en tiempo real I

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ **(%A)**	$\mathbf{D_p}$ **teor.**	**Max** $\mathbf{D_p}$ **t real**	**Min** $\mathbf{D_p}$ **t real**
L1	1	25,00	1	1,11	0,91
L2	0,90	23,23	1,11	1,31	0,93
L3	0,79	19,77	1,26	1,77	1,09
L4	0,76	26,93	0,93	1,39	0,85
L5	0,87	22,00	1,13	1,52	0,87

Podemos observar que ningún valor de D_p rebasa el 1,8 ni es inferior a 0,8. Se pone de manifiesto también la ya mencionada falta de simetría en cuanto al impacto de la presencia de tareas pequeñas y grandes tanto en el rendimiento del algoritmo como de los cálculos en tiempo real.

Hemos incluido el lote L4 que corresponde a una distribución tipo plana, donde la tercera parte de las tareas son de cada uno de los tamaños. Observamos que este lote es el que presenta una mayor desviación entre el máximo valor de D_p y el mínimo, ya que la presencia de tareas grandes es más elevada que en una distribución de medianas, como también lo es la de pequeñas y

esto hace que el cálculo de D_p en tiempo real oscile más que en distribuciones donde hay una clara predominancia de alguno de los tipos de tareas.

Ahora mostraremos los resultados obtenidos al ejecutar lotes de tareas que corresponden a distribuciones cuya ejecución con una distribución en 4 particiones es peor que si se ejecutan con 3 o con 6 particiones. Mostraremos los resultados de los valores máximos y mínimos de D_p a lo largo de su ejecución tanto con 4 particiones (4P) como con 6 (6P) o con 3 (3P) y una gráfica donde se muestre la evolución temporal de este parámetro. Empezaremos analizando los resultados para lotes de tareas donde es necesario pasar a 3 particiones, tabla 4.9. Las primeras columnas nos muestran las características de los lotes y después el valor teórico de D_p para 4 particiones y a continuación los valores máximo y mínimo obtenidos en tiempo real. Las últimas tres columnas muestran el valor teórico de D_p para cada lote con 3 particiones y su variaciones en tiempo real.

TABLA 4.9: Distribuciones y variación de D_p en tiempo real II

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ (%**A**)	**4P**	**Máx**	**Min**	**3P**	**Máx**	**Min**
L1	0,70	29,73	0,84	0,89	0,63	1,12	1,34	0,89
L2	0,69	28,75	0,87	1,16	0,59	1,16	1,55	0,90
L3	0,87	22,00	1,13	1,52	0,87	1,51	1,99	1,27

En estos ejemplos podemos observar que para 4P tenemos un valor teórico bastante bajo e inferior a $0,9$, lo que se traduce en tiempo real en valores mínimos muy bajos. Sin embargo los valores de D_p obtenidos para estos mismos lotes de tareas con una distribución en 3P se ajustan a los márgenes de variación que hemos obtenido para lotes de tareas medianas cuando se están

ejecutando en 4P (inferiores a $1,8$ y por encima de $0,8$).

La figura 4.12 muestra las variaciones del valor de D_p en tiempo real para L2. Hemos tomado este lote para mostrar la gráfica porque es el que mayor irregularidad presenta en los valores máximo y mínimo.

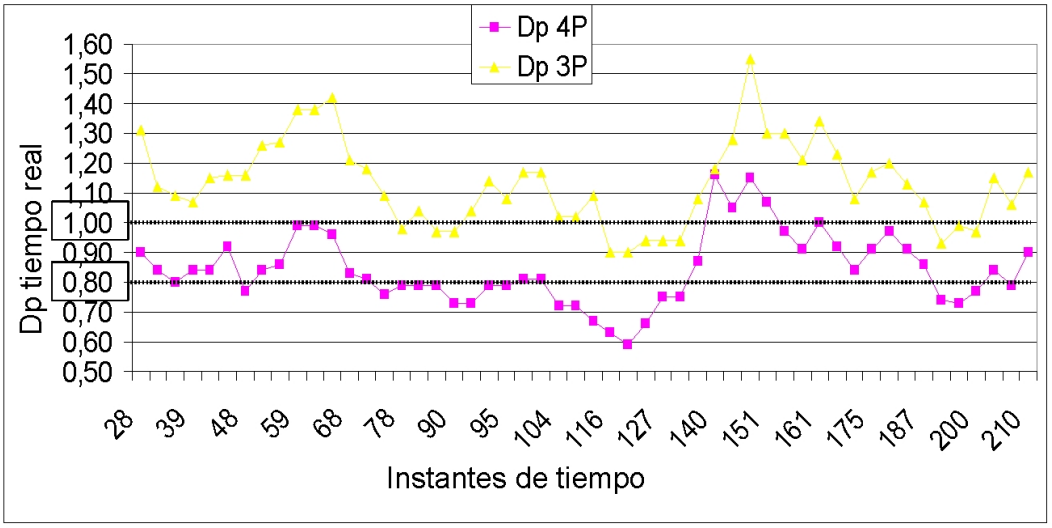

FIGURA 4.12: Oscilaciones del valor de D_p en tiempo real II

Observamos que con la ejecución en 3P, el valor de D_p oscila en torno al valor de 1, excepto en un momento dado en que hay una secuencia de tareas de menor tamaño que llegan cercanas en el tiempo y sube hasta el valor $1,55$. El valor calculado en tiempo real para la ejecución en 3P corresponde a la línea superior de la gráfica 4.12.

La gráfica de D_p para 4P, la línea inferior de la gráfica 4.12, se mantiene claramente por debajo de 1 y en numerosas ocasiones baja por debajo de $0,8$. Uniendo estas observaciones a las realizadas con los lotes de tareas de tipo L_m examinadas anteriormente, podemos deducir que el límite $D_p < 1$ medido en tiempo real se traducirá en $D_p \leq 0,8$ tres veces seguidas, para evitar

que el impacto de las oscilaciones del valor de D_p lleven a tomar decisiones precipitadas y asegurar que se trata de una tendencia real del conjunto de tareas y no de un momento particular de la ejecución en que varias tareas grandes han llegado bastante cercanas en el tiempo. En la gráfica podemos observar también que D_p para el caso 4P empieza tomando valores bajos y va disminuyendo progresivamente, llegando a estar por debajo del valor $0,8$ en alguna ocasión y mostrando una tendencia a la baja hasta que realmente se puede confirmar que se trata de una distribución que no es adecuada para 4P.

Los valores de D_p en tiempo real para una distribución en particiones adecuada nunca son inferiores a $0,8$. Su valor máximo es $1,52$, mostrando la misma tendencia que habíamos visto para lotes de tareas de tipo L_m donde en ningún caso el valor de D_p superaba el valor de $1,8$.

Sin embargo, para una distribución en particiones no adecuada, los valores de D_p claramente oscilan por encima del valor 1 llegando a superar el valor $1,8$ en numerosas ocasiones y dando un máximo valor de D_p muy alto, $1,99$, como se puede ver en la tabla 4.9.

Con los resultados obtenidos hasta el momento podríamos concluir que:

1. La situación $D_p > 1$ se alcanza en tiempo real cuando la distribución genera tres veces seguidas un valor de $D_p \geq 1,8$, condición que indica la necesidad de aumentar el número de particiones, ya sea de 3 a 4 o de 4 a 6.

2. La situación $D_p < 1$ se alcanza en tiempo real cuando la distribución genera tres veces seguidas un valor de $D_p \leq 0,8$, indicando la necesidad de disminuir el número de particiones, ya sea de 4 a 3 o de 6 a 4.

Para la confirmación definitiva de estas conclusiones es necesario realizar pruebas comparativas de la oscilación de los valores de D_p en 4P y en 6P. A continuación presentamos una tabla con resultados de ejecutar lotes de tareas de tipo L_{pp} (L1 y L2) y de tipo L_m (L3) en 4 y 6 particiones y los resultados en tiempo real obtenidos para ellas. La tabla 4.10 muestra las características de los lotes junto a los valores teóricos obtenidos para 4P y sus variaciones en tiempo real y a continuación el valor teórico de D_p para 6P y sus variaciones en tiempo real.

TABLA 4.10: Distribuciones y variación de D_p en tiempo real IV

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ (%**A**)	**4P**	**Máx**	**Min**	**6P**	**Máx**	**Min**
L1	0,62	14,00	1,78	2,22	1,56	1,19	1,45	1,06
L2	0,53	14,50	1,72	2,00	1,55	1,15	1,27	1,03
L3	0,90	23,23	1,11	1,31	0,93	0,74	0,86	0,61

Los dos primeros lotes corresponden a distribuciones donde las tareas son bastante pequeñas y su ejecución es más adecuada en 6P. Vemos que si se ejecutan en 4P la UAD determinaría un cambio a 6P. Asimismo, vemos que cuando este tipo de distribuciones se estuviera ejecutando en 6P la UAD nunca determinaría un cambio en las particiones.

El último lote, L3 corresponde a una distribución tipo L_m. Si se estuviera ejecutando en 4P y con los criterios obtenidos para el cambio en las particiones, nunca cambiaría, lo cual es correcto. Si una distribución de este tipo se estuviera ejecutando en 6P la UAD determinaría el cambio de 6P a 4P.

La figura 4.13 muestra los valores de D_p obtenidos para 4P con un lote L_g, un lote L_m y un lote L_{pp} en tiempo real y los valores de D_p en relación a los

límites propuestos para los cambios en las particiones.

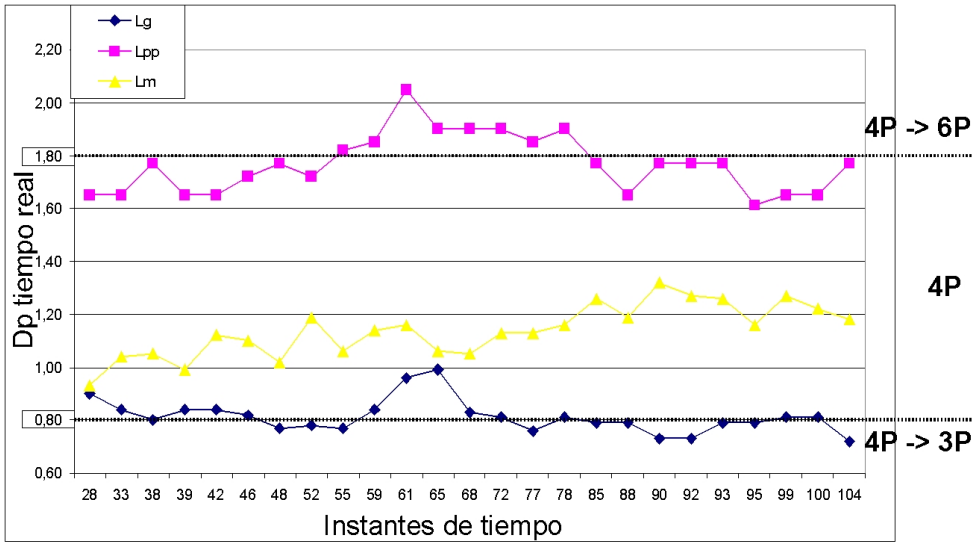

FIGURA 4.13: Límites del valor de D_p en tiempo real para determinar cambios

La tabla 4.11 resume el conjunto de cambios a realizar en función de los valores calculados para D_p en tiempo real.

TABLA 4.11: Distribuciones y variación de D_p en tiempo real

$\mathbf{D_p}$	**3P**	**4P**	**6P**
$\leq 0,8$	-	3P	4P
$\geq 1,8$	4P	6P	-

4.5.4. Ventana de observación del sistema

Todas las pruebas experimentales mostradas hasta ahora y los resultados presentados en el capítulo 6 se han realizado con una ventana de 16 tareas. En esta sección explicaremos por qué se utiliza una ventana de tareas en lugar de una temporal y por qué se ha utilizado ese número de tareas.

Al tratarse de un sistema donde no se sabe cuándo llegará la siguiente tarea ni cómo será ésta, es muy difícil definir el intervalo de tiempo durante el cual debe observarse el sistema para sacar conclusiones acertadas acerca de las características de la distribución de tareas. Si la frecuencia de llegada de tareas es baja, tendremos un conjunto muy pequeño de tareas para tomar decisiones y en un caso así una sola tarea tiene mucho peso en el valor de D_p calculado. Por el contrario, si la frecuencia de llegada de tareas es alta es posible que estemos tardando excesivo tiempo en detectar la necesidad de un cambio.

Por estas razones hemos optado por observar el sistema en tiempo real a través de una ventana de tareas deslizante de manera que la muestra obtenida para analizar las características de la carga de trabajo contenga siempre el mismo número de tareas y sea igualmente significativa en todas las posibles circunstancias. Una FIFO de cierto tamaño almacenará los datos de cada nueva tarea que llega al sistema y su contenido se utilizará para calcular α, $\overline{a}$ y $\overline{tej}$.

Si lo que queremos es observar si un grupo de tareas se parece a alguna de las distribuciones definidas, con el propósito de cambiar las particiones si éstas no se ajustan a la carga de trabajo, parece lógico que el número de tareas a observar sea múltiplo del número de particiones.

Esto nos lleva a que m, el número de tareas almacenadas en la FIFO será múltiplo del número de particiones:

$$m = k \cdot P \tag{4.16}$$

Si lo que queremos es que el sistema actúe de forma dinámica pero no precipitada, tendremos que elegir un valor de k que no sea excesivamente

pequeño ni grande. Descartamos $k = 1$ y $k = 2$ ya que darían un número muy bajo de tareas a observar (obtendríamos valores entre 3 y 12 tareas).

Así que podemos ver qué sucede si tomamos un valor $k = 3$. Para el caso de $P = 4$ estaríamos observando 12 tareas. Para $P = 3$ solamente 9 tareas. Para $P = 6$ ya serían 18. Con $k = 4$ estaremos examinando las últimas 12, 16 o 24 tareas que han llegado al sistema. Este número empieza a parecer razonable y puede servirnos como consideración inicial.

Un $k = 5$ daría números de 15, 20 y 30, lo que en algunos casos puede resultar excesivo, especialmente si estamos trabajando con 6 particiones, de las cuales 4 son muy pequeñas, y empiezan a llegar un número significativo de tareas grandes, que podrían ocasionar serios problemas de rendimiento en el sistema.

De momento nos quedaremos con $k = 4$ como un número razonable para definir el tamaño de la ventana de tareas utilizada para observar el sistema en función del número de particiones que se estén utilizando.

A continuación vamos a presentar una serie de pruebas experimentales realizadas para confirmar la intuición de que utilizar un valor de $k = 4$ es acertada. Para ello recurriremos a varios lotes de tareas que corresponden a diferentes tipos de distribuciones y examinaremos los valores de D_p obtenidos en tiempo real con diferentes valores de k en relación al valor teórico de D_p para el lote completo y a partir de ahí comentaremos y justificaremos las decisiones tomadas para la implementación de la FIFO en la UAD.

La figura 4.14 nos muestra los valores de D_p obtenidos para $k = 3$, $k = 4$ y $k = 5$ para un lote de tareas tipo L_m. Para este lote completo obtenemos un $D_p = 1,11$. Debido al diferente número de tareas almacenadas en la FIFO para

realizar los cálculos, la UAD empieza a obtener valores para D_p en distintas unidades de tiempo. Hemos representado los valores obtenidos a partir de la unidad en tiempo en la que D_p empieza a calcularse para todos los valores de k.

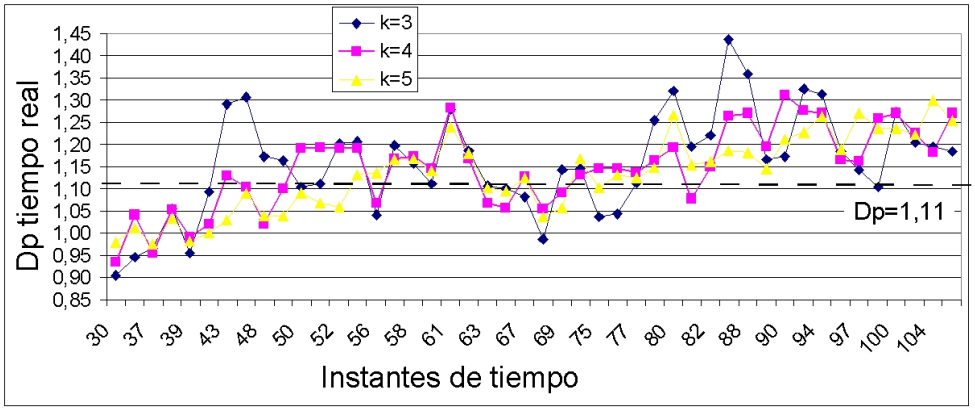

FIGURA 4.14: Variación de D_p en función de k I

Esta figura nos muestra que cuanto más alto es el valor de k menor es la oscilación de los valores de D_p medidos en tiempo real con respecto al valor teórico del lote completo. Para el valor de $k = 3$ la desviación en algunos momentos es excesiva y dará un valor de D_p muy distorsionado. Los valores de D_p obtenidos para $k = 5$ son los que más se ajustan al valor teórico (y si aumentáramos el valor de k se ajustarían aún más). También podemos observar que para $k = 4$ los valores obtenidos no son mucho peores que los obtenidos para $k = 5$ y sí bastante mejores que los obtenidos para $k = 3$. Además, la desviación del valor ideal no es simétrica y es mayor cuando la oscilación de D_p esté por encima del valor teórico que cuando esté por debajo.

La tabla 4.12 muestra los resultados obtenidos para el lote tipo L_m del que hemos presentado el gráfico. Para este lote se han realizado las ejecuciones con

una distribución en 4 particiones, por lo que los tamaños de las ventanas son de 12, 16 y 20 tareas respectivamente. La última fila de la tabla nos muestra la máxima desviación del valor de D_p obtenido en tiempo real del valor teórico, en porcentaje. Vemos que la diferencia entre $k = 4$ y $k = 5$ es muy pequeña, mientras que la diferencia de pasar de un $k = 3$ a $k = 4$ sí que es considerable.

TABLA 4.12: Variación de D_p con k, L_m

$\mathbf{L_m}$	$\mathbf{k = 3}$	$\mathbf{k = 4}$	$\mathbf{k = 5}$
Máx D_p	1,44	1,31	1,30
D_p téor.	1,11	1,11	1,11
Min D_p	0,90	0,93	0,97
Desv (%)	**29,73**	**18,01**	**17,11**

La tabla 4.13 muestra los resultados obtenidos para el lote tipo L_g, donde podemos apreciar la misma tendencia respecto al valor de D_p obtenido en tiempo real con diferentes valores de k. Como es un lote de tareas de tipo grande, los tamaños de las ventanas para estos mismos valores de k son ahora de 9, 12 y 15 tareas, ya que se han ejecutado en una distribución en 3 particiones.

TABLA 4.13: Variación de D_p con k, L_g

$\mathbf{L_g}$	$\mathbf{k = 3}$	$\mathbf{k = 4}$	$\mathbf{k = 5}$
Máx D_p	1,68	1,54	1,36
D_p téor.	1,12	1,12	1,12
Min D_p	0,75	0,83	0,86
Desv (%)	**50,00**	**37,50**	**21,43**

En este caso la desviación máxima es muy alta ya que el número de tareas en la ventana de tareas es bastante menor que en los casos estudiados para el lote de tipo L_m, llegando a ser de 15 tareas el máximo.

Por último realizamos el mismo experimento con un lote de tareas tipo L_{pp} ejecutado en una distribución en 6 particiones. La tabla 4.14 muestra los resultados obtenidos para el lote tipo L_{pp}, donde podemos apreciar la misma tendencia respecto al valor de D_p obtenido en tiempo real con diferentes valores de k. Como es un lote de tareas tipo pequeño, los tamaños de las ventanas para estos mismos valores de k son ahora de 18, 24 y 30 tareas. Vemos que hay una gran diferencia entre el número de tareas que se está utilizando para tomar decisiones con cada tipo de distribución diferente.

TABLA 4.14: Variación de D_p con k, L_{pp}

L_{pp}	**k = 3**	**k = 4**	**k = 5**
Máx D_p	1,38	1,37	1,34
D_p téor.	1,19	1,19	1,19
Min D_p	1,08	1,09	1,12
Desv (%)	**15,96**	**15,13**	**12,60**

El paso de $k = 3$ a $k = 4$ en este ejemplo no da una mejora significativa en cuanto a disminución del error y sin embargo el aumento de $k = 4$ a $k = 5$ sí que lo hace.

Estos experimentos ponen también de manifiesto que utilizar el mismo valor de k para todas las distribuciones en particiones no parece una buena idea ya que supone una clara desventaja a la hora de tomar decisiones acertadas a las distribuciones que trabajan con un menor número de particiones. Una dificultad añadida es el hecho de que el tamaño de la FIFO debería variar también cuando se haga un cambio en la distribución en particiones.

Por todo ello consideramos acertada la realización de estas pruebas ya que ponen de manifiesto lo adecuado de utilizar una ventana de tareas de tamaño

fijo y que considere un número de tareas suficiente para no cometer un error excesivo y suficientemente pequeño para que la toma de decisiones no se aplace demasiado.

Para poder decidir el número adecuado de tareas que deben formar parte de la ventana de tareas presentamos a continuación la tabla 4.15 que resume los resultados anteriores y que presenta el error cometido en función del número de tareas en la ventana.

TABLA 4.15: Desviación de D_p para diferentes tamaños de ventana de tareas

	9	**12**	**12**	**15**	**16**	**18**	**20**	**24**	**30**
Desv(%)	50,00	37,50	29,73	21,43	18,01	17,11	15,96	15,13	12,60

En esta tabla podemos observar que si el objetivo es cometer el menor error posible obviamente tendríamos que coger un número de tareas lo mayor posible. Sin embargo hay que tener en cuenta que son datos tomados en tiempo real y que no es conveniente tampoco esperar demasiado a detectar una situación de no adecuación de la distribución de particiones ya que podría empezar a rechazarse un número elevado de tareas por el simple motivo de no estar realizando una buena gestión del área del dispositivo.

Para un número de tareas razonable, de 16, vemos que el error cometido está en torno al 20 %. Vemos también que para reducir sensiblemente el error habría que pasar a examinar prácticamente el doble de tareas. Con 30 tareas en la ventana podríamos reducirlo a aproximadamente el 12 %. También vemos que el mismo número de tareas puede dar un error en la apreciación de D_p bastante diferente dependiento del lote y tipo de tareas que lo forman (caso de 12 tareas que se ha repetido con $P = 4$ y $k = 3$ y con $P = 3$ y $k = 4$).

Lo que sí queda claro es que tomar menos de 16 tareas no parece muy acertado porque el error cometido será grande. Por encima de 16 tareas las mejoras que podrían resultar significativas serían pasar a 20 tareas o a 30 o algún número intermedio entre 16 y 30.

Si estuviéramos trabajando con 3 particiones y manejamos una ventana de 20 tareas la situación podría ser muy mala. Por un lado, el algoritmo dispone de menos opciones para ubicar tareas cuantas menos particiones haya, por lo que si está trabajando con una distribución en $3P$ y empiezan a llegar tareas pequeñas y a mayor frecuencia tiene pocas opciones de gestionarlas y pronto empezará a rechazarlas. Además si la ventana de tareas es grande, como el peso del área de las tareas grandes es mucho mayor que el de las pequeñas en la media, D_p tardará mucho en subir su valor y el retraso en la toma de decisiones será muy perjudicial para el algoritmo.

En el estudio teórico de las distribuciones vimos que para que el área media de una distribución se acerque al tamaño de las tareas pequeñas es necesario que haya un 80 % de tareas pequeñas en el conjunto de tareas. Es decir, que en la ventana de tareas no se apreciaría la presencia de las tareas pequeñas hasta que no hubiera cerca de 16. Si hay 3 particiones solamente se podrán ejecutar 3 tareas simultáneamente. Esto quiere decir que de estas 16 tareas que hay en la ventana probablemente se habrán rechazado casi la mitad de ellas (supongamos que 3 se están ejecutando y hay 2 tareas esperando en cada cola porque sus tiempos máximos se lo permiten; esto significaría que de estas 16 ya se han rechazado 7 para cuando D_p supere el valor de $1,8$ por primera vez). Si además consideramos que la diferencia en el error cometido es poco significativa (en nuestros ejemplos sería poco más de 1 %), no merece la pena

utilizar un número tan elevado de tareas en la ventana.

Si la ventana es de 16 tareas, tendrá que haber cerca de 12 para que D_p supere el valor que decide un aumento en las particiones por primera vez. Realizando los mismos cálculos anteriores, probablemente se habrán rechazado la cuarta parte de las tareas pequeñas, que en este caso son 3 tareas.

Existe una ventaja adicional para utilizar una ventana de 16 tareas, que está relacionada con la implementación de la UAD. Para determinar D_p tenemos que calcular el área media de las tareas que están en la ventana. Este cálculo se simplifica enormemente si utilizamos un número de tareas que sea una potencia de 2 ya que la división por 16 se puede realizar en hardware con un sencillo desplazamiento de bits. Lo mismo sucederá con el cálculo del tiempo medio de ejecución y de cualquier otro valor medio que se quisiera calcular (por ejemplo el de los tiempos máximos de las tareas).

Por todos los motivos expuestos, consideramos que una ventana de tareas de tamaño fijo que contenga los datos de las últimas 16 tareas que han llegado al sistema proporciona una información suficientemente fiable como para tomar decisiones acertadas con la agilidad necesaria, además de resultar una opción que puede facilitar mucho la implementación de la UAD (o parte de ella) en hardware.

Por último presentamos la tabla 4.16 con las desviaciones obtenidas para el parámetro α en tiempo real, a partir de los mismos casos para los que hemos presentado las variaciones de D_p.

Podemos observar que el valor del parámetro α presenta unas variaciones en tiempo real mucho mayores que D_p respecto a su valor teórico para el lote completo. Esto es debido a que la llegada de tareas no se realiza de forma

TABLA 4.16: Desviación de α para diferentes tamaños de la venta de tareas

	9	**12**	**12**	**15**	**16**	**18**	**20**	**24**	**30**
Máx.	1,23	1,17	1,25	1,08	1,19	0,91	1,18	0,91	0,93
Teórico	0,70	0,70	0,90	0,70	0,90	0,62	0,90	0,62	0,62
Mín.	0,62	0,63	0,62	0,63	0,70	0,58	0,76	0,58	0,57
Desv(%)	75,71	67,14	38,88	54,28	32,22	46,77	32,22	46,77	50,00

uniforme y hay intervalos de tiempo en que las tareas llegan muy seguidas (incluso en grupos de 3 o 4 tareas) y otros en que se distancian más. Esto hace que el valor fluctúe mucho más que el D_p con el tamaño de la ventana de tareas.

Vemos que la máxima desviación es por exceso, es decir, en valores por encima del teórico, y que el límite inferior para estos lotes nunca está por debajo del $\alpha = 0,5$ que tomamos como referencia para un sistema infrautilizado (lo cual es correcto para todos ellos).

Lo que sucedería si en algún momento α tomara un valor inferior a $0,5$ es que D_p no se calcularía en ese momento. Si α se mantuviera por debajo de $0,5$, D_p seguiría sin calcularse, pero esto no sería un problema ya que al estar llegando pocas tareas al sistema, la distribución en particiones que se esté utilizando no es crucial para el rendimiento del algoritmo, ya que dispone de la habilidad de unir particiones cuando sea necesario y podría planificar el total del conjunto incluso con una distribución en particiones no adecuada, como se vió en los resultados de la sección 4.4.3.

Por el contrario, si α vuelve a subir, el valor de D_p se volverá a calcular y el sistema no pierde información acerca de las características del conjunto de tareas que han llegado recientemente (cada nueva tarea entrante se guarda

en la FIFO, independientemente del valor de α). Si esto sucediera, podría retrasarse la toma de una decisión, pero como vemos no sería en situaciones donde la carga de trabajo es alta y prácticamente no daría lugar a pérdidas en el rendimiento.

En la figura 4.15 mostramos la ejecución correspondiente a un lote de tareas con un valor global de $\alpha = 0,52$. Se trata de un lote de tipo L_g que se está ejecutando en 4P. La UAD no tendría ningún problema con este tipo de conjuntos de tareas de baja frecuencia de llegada para detectar la necesidad de un cambio en las particiones ya que como vemos en la figura, el valor de α (línea inferior de la gráfica) solamente se encuentra por debajo de 0,5 dos veces a lo largo de toda la ejecución. Se dejaría de calcular D_p en $t = 41$ y $t = 42$ pero esto no impide detectar un poco más adelante que D_p está rozando el valor límite de 0,8, ni realizar el cambio de particiones cuando D_p baja por debajo de 0,8 tres veces seguidas en $t = 105$.

De hecho, el valor teórico $\alpha < 0,5$ se convierte en $\alpha < 0,35$ en tiempo real, si suponemos una desviación por exceso del cálculo de su valor del 30 % (de la tabla 4.16, para una ventana de 16 tareas). Es decir, que la UAD solamente ignorará la no adecuación de la distribución en particiones en situaciones donde realmente la carga de trabajo sea muy inferior a la capacidad de la FPGA.

Por todas las razones expuestas, hemos realizado todas las pruebas experimentales de la UAD con una ventana de 16 tareas y los resultados obtenidos son buenos.

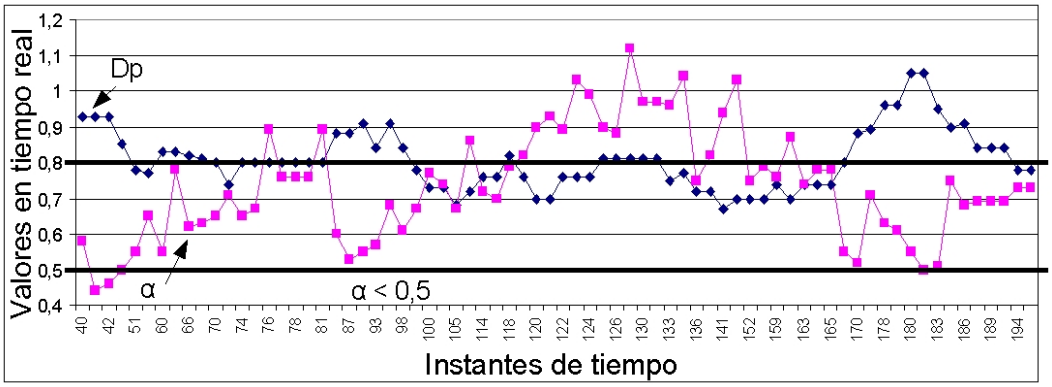

FIGURA 4.15: Variación de α y D_p, en un sistema en el límite de infrautilización

4.6. Conclusiones

Resumimos aquí las conclusiones a las que hemos llegado con el estudio anterior y que determinan la manera en que se implementará la UAD:

1. **Tipos de distribuciones básicas**: con 4 distribuciones básicas (prácticamente todas las tareas pequeñas, tareas pequeñas, tareas medianas y tareas grandes) es suficiente para describir un amplio rango de situaciones posibles en el sistema.

2. **Distribuciones en particiones**: utilizando 3 tipos de distribuciones en particiones con 3, 4 y 6 particiones podemos asegurar la correcta ejecución de prácticamente la totalidad de posibles cargas de trabajo.

3. **Parámetros de medida**: con dos parámetros fácilmente medibles en tiempo real, α y D_p, podemos detectar las situaciones en que será necesario cambiar el número de particiones. El parámetro D_p solamente se calculará cuando $\alpha \geq 0,5$ y el cambio de particiones se realizará conforme a la tabla 4.17.

TABLA 4.17: Cambios de particiones en función de D_p

$\mathbf{D_p}$	**3P**	**4P**	**6P**
$\leq 0,8$	-	3P	4P
$\geq 1,8$	4P	6P	-

4. **Ventana de tareas**: se utilizará una ventana de 16 tareas para realizar la observación del sistema.

En el siguiente capítulo proporcionaremos detalles acerca del procedimiento para llevar a cabo el cambio en las particiones con el mejor rendimiento posible, seguido de un ejemplo detallado del funcionamiento del algoritmo de la UAD.

Capítulo 5

Implementación de la Adaptación Dinámica

5.1. Introducción

Para completar la descripción del funcionamiento de la UAD nos queda explicar en detalle cómo se hace efectivo un cambio en las particiones. El cambio en las particiones se realiza en un momento dado en que hay una serie de tareas ejecutándose y otras tareas esperando en las colas, asignadas a particiones que tienen un determinado tamaño y se detecta la necesidad de cambiar el tamaño y número de las particiones. Tenemos que analizar cuál es la mejor manera de realizar este cambio en las particiones afectando en la menor medida posible al conjunto de tareas ya planificadas.

Asimismo analizaremos en detalle cuáles son los valores de los parámetros que manejarán la UAD, la UPT y la ULT cuando se determine un cambio en las particiones.

Lo que físicamente determina las particiones de la FPGA son los accesos al bus, es decir, los lugares donde se configuran las bus macros, que deberán coincidir con los lugares donde se ubican las conexiones al bus en las tareas en el momento de sintetizarlas. Esto quiere decir que el cambio en las particiones implicaría una reubicación de las bus macros en la FPGA, acción que conllevaría la reconfiguración del bus en la FPGA. Es necesario también el aumento o disminución del número de colas que maneja la UPT, así como la actualización de las variables que almacenen los tamaños de las particiones.

En primer lugar examinaremos las ubicaciones de las bus macros para cada distribución de particiones y la posibilidad de configurarlas todas desde un principio para evitar la reconfiguración del bus cada vez que haya un cambio de particiones. Después analizaremos cómo se puede realizar un cambio en la distribución de las particiones sin afectar a las tareas que ya se están ejecutando y retrasando en la menor medida posible la ejecución de las tareas ya planificadas en las colas correspondientes a la anterior distribución.

5.2. Definición de parámetros

La figura 5.1 muestra los valores de los parámetros para cada una de las distribuciones en particiones.

1. Cuando el cambio de particiones sea de P a $P' = 4$, $w'_p = 0,30 \cdot W$ y $h'_p = 0,30 \cdot H$.

2. Para el cambio a $P' = 3$ particiones, $w'_p = 0,5 \cdot W$ y $h'_p = 0,5 \cdot H$.

3. Para el cambio a 6 particiones, $P' = 6$ junto con $w'_p = 0,25 \cdot W$, $h'_p =$

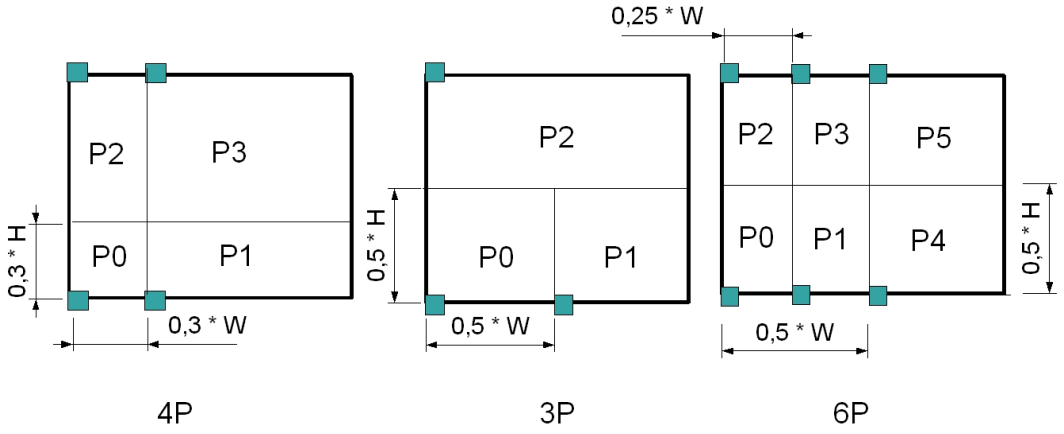

FIGURA 5.1: Distribuciones en particiones utilizadas

$0,5 \cdot H$ y $w'_{pp} = 0,5 \cdot W$ para que se realice la división en 6 particiones de forma correcta.

Respecto a la ubicación de las macros de acceso al bus de comunicaciones de cada partición, m_{P_i}, tendremos un número de accesos al bus igual a P y con localizaciones que dependen de los valores que toman w_p, h_p y w_{pp} para cada tipo de distribución.

Si el número de particiones es P, solamente se utilizarán los accesos al bus m_{P_0} a $m_{P_{P-1}}$. La partición menor siempre es la P_0 en todas las distribuciones posibles y la mayor la que tenga mayor índice. La numeración de las particiones se ha elegido de manera que coincida el mayor número posible de bus macros. En el caso de utilizar 3 particiones hay un bus macro que no se usa, mientras que en la caso de 6 particiones deben aparecer dos bus macros más y un solo parámetro que indique la localización de las nuevas bus macros (estarán a la misma distancia del origen de coordenadas pero una de ellas en la parte inferior de la FPGA y la otra en la superior) a la que hemos llamado w_{pp}.

5.2.1. Cambio de particiones

Examinaremos ahora la manera en que se puede ejecutar el cambio en las particiones. Hemos tenido en cuenta dos aspectos:

1. **El coste en tiempo de esperar a que terminen de ejecutarse todas las tareas que están en la FPGA** en el momento de decidir el cambio puede llegar a ser muy alto y variable, ya que depende de los tiempos de ejecución de las tareas que se están ejecutando. Ya que solamente se plantean los cambios en las particiones cuando tenemos una afluencia alta de tareas al sistema, podría dar lugar al rechazo de una gran cantidad de tareas.

2. **El coste en tiempo de reconfigurar el bus de comunicaciones** también es alto y por las mismas razones expuestas podría dar lugar al rechazo de una gran número de tareas.

Por ello hemos buscado una manera de realizar el cambio en las particiones que evite estas dos situaciones, lo cual no es complicado ya que disponemos de información acerca de cómo se solapan las particiones de la antigua distribución con la nueva y en el momento de decidir el cambio podríamos mover las tareas que estén esperando en las colas a las nuevas colas y conforme las tareas que están ejecutándose en la FPGA vayan terminando su ejecución, ir enviando tareas de las colas a ejecutar en las partes de la FPGA que han quedado libres conforme al nuevo mapa de particiones de la FPGA.

Asimismo, evitar la reconfiguración del bus de comunicaciones cada vez que se hace un cambio en las particiones es fácil: se pueden configurar todas

las bus macros necesarias para todos los casos simultáneamente en la FPGA desde el principio. De esta forma todas las bus macros estarán disponibles y solamente se utilizarán las que en cada momento se necesiten.

5.2.2. Posiciones de las bus macros

A continuación mostramos la figura 5.2 con las bus macros necesarias para las particiones en cada una de las distribuciones: 4P, 3P y 6P. Están numeradas de izquierda a derecha empezando por la esquina inferior izquierda.

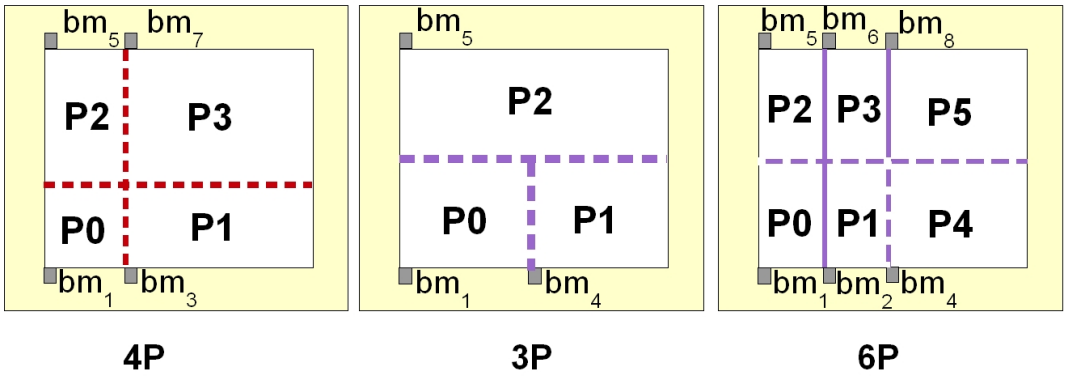

FIGURA 5.2: Ubicación de las bus macros para las distintas distribuciones

Observamos que algunas bus macros son compartidas en las tres distribuciones posibles (bm_1 y bm_5) y las otras son específicas para alguna de las distribuciones. Las líneas discontinuas rojas indican la división de la FPGA para 4 particiones. Las líneas lila discontinuas nos muestran la división para 3 particiones, y como son comunes a 3P y 6P, son del mismo color que las líneas que indican al división en 6P (las lila tanto continuas como discontinuas).

La figura 5.3 une la información desglosada en la figura 5.2 y nos muestra la configuración del bus que se puede realizar una vez al principio de la ejecución

y que sirve para cualquiera de las tres posibles distribuciones en particiones que gestiona nuestro algoritmo. Se trata de configurar cuatro bus macros en la parte inferior de la FPGA y otras cuatro situadas en las mismas posiciones horizontales pero en la parte superior de la FPGA.

Observamos que las particiones P_1 para 3P y P_4 para 6P coinciden y comparten la bus macro bm_4. La partición P_0 de la distribución 3P es la unión de las particiones P_0 y P_1 de la distribución 6P, por lo que comparten una de las bus macros, la bm_1. En la parte superior de la FPGA la partición P_2 comparte bus macro en todos los tipos de distribuciones, la bm_5. Las particiones P_3 de 4P y 6P utilizan diferentes bus macros (bm_7 y bm_6) y la partición P_5 no comparte su bus macro (bm_8) con ninguna otra ya que la partición P_2 de 3P ocupa la mitad superior de la FPGA y utiliza la misma bus macro que la P_2 de las otras dos distribuciones.

Se trata en total de 8 bus macros que se sitúan en las siguientes posiciones:

1. $bm_1 = (0, 0)$ se utiliza para la partición P_0 en cualquiera de las distribuciones en particiones.

2. $bm_2 = (0, 25 \cdot W, 0)$ se utiliza para la partición P_1 en la distribución 6P.

3. $bm_3 = (0, 30 \cdot W, 0)$ se utiliza para la partición P_1 en la distribución 4P.

4. $bm_4 = (0, 50 \cdot W, 0)$ se utiliza para la partición P_1 en la distribución 3P y para la P_4 en la distribución 6P.

5. $bm_5 = (0, H)$ se utiliza para la partición P_2 en cualquiera de las distribuciones en particiones.

6. $bm_6 = (0, 25 \cdot W, H)$ se utiliza para la partición P_3 en la distribución 6P.

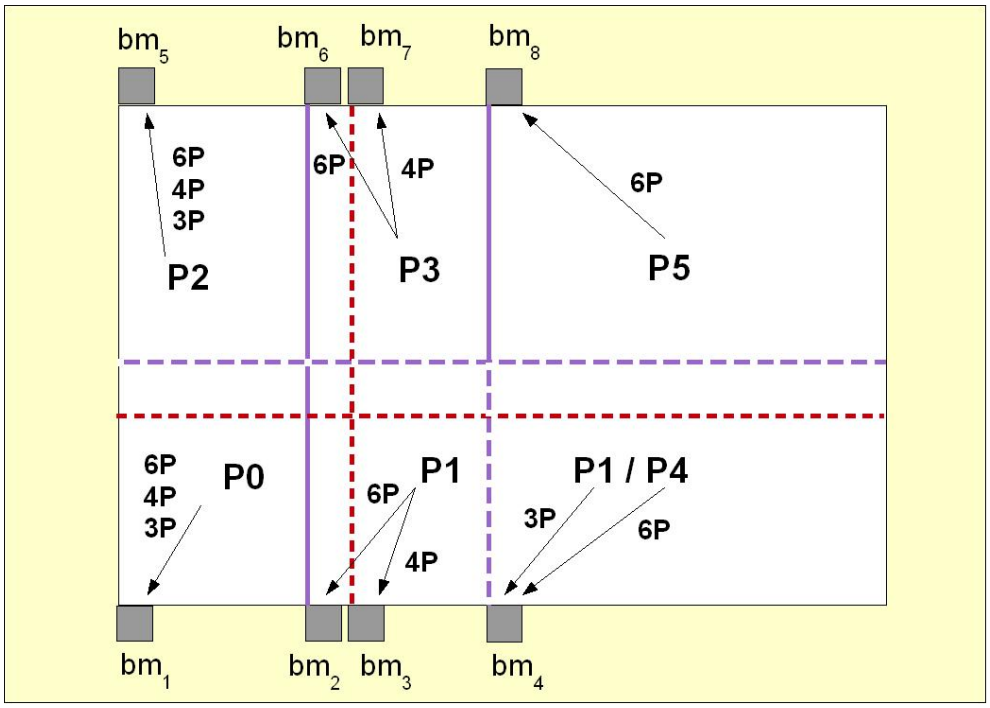

FIGURA 5.3: Ubicación de las bus macros válidas para todas las distribuciones

7. $bm_7 = (0, 30 \cdot W, H)$ se utiliza para la partición P_3 en la distribución 4P.

8. $bm_8 = (0, 50 \cdot W, H)$ se utiliza para la partición P_5 en la distribución 6P.

Con este procedimiento podemos realizar el cambio de forma gradual, haciendo que se solapen las distintas distribuciones en particiones durante la fase del cambio y reduciendo de forma muy significativa la penalización en rendimiento que conlleva realizar el cambio.

Por ejemplo, si se cambia de 3 particiones a 4 particiones y la tarea de P_0 con 3P termina su ejecución la primera, podrá pasar a ejecutarse una de las tareas que están esperando en las colas en la partición P_0 de 4P utilizando la misma bus macro, la bm_0, para acceder al bus de comunicaciones y sin interferir en la ejecución de las otras dos tareas que continúan ejecutándose con la distribución

3P. Si a continuación termina la tarea que se estaba ejecutando en la partición P_2 de la distribución 3P, podrá empezar a ejecutarse una tarea de las colas que quepa en la partición P_2 de la distribución 4P utilizando la bm_5 y sin interferir en la ejecución de la tarea que queda en la partición P_1 de la distribución 3P. Por último, cuando la última tarea que se está ejecutando en P_1 de 3P termine su ejecución, pasarán a estar disponibles las particiones P_1 y P_3 de la distribución 4P y las tareas que se ejecuten en ellas podrán utilizar las bus macros bm_3 y bm_7 respectivamente. En resumen, el cambio en las particiones se puede realizar de forma gradual y sin necesidad ni de reconfigurar el bus de comunicaciones ni de retrasar la ejecución de muchas de las tareas que esperan en la colas.

En la siguiente subsección presentamos la forma en que se ha implementado la gestión de las particiones para el cambio gradual en la distribución de particiones de la FPGA.

5.2.3. Implementación del cambio gradual

La figura 5.3 nos muestra cómo algunas particiones de distinto tipo coinciden y otras se solapan. Para poder determinar qué tareas asignadas al nuevo tipo de distribución pueden lanzarse a ejecutar conforme van terminando las tareas que ocupaban particiones de la antigua distribución, la ULT dispone de tres tablas de correspondencias que le indican, para cada tipo de distribución cuáles son las particiones que se solapan con las de las otras distribuciones. Las tablas 5.1, 5.2 y 5.3 nos muestran las correspondencias entre particiones que solapan con los distintos modelos de particiones utilizados.

TABLA 5.1: Tabla de correspondencias para 4P

4P	**3P**	**6P**
P_0	P_0	P_0, P_1
P_1	P_0, P_1	P_1, P4
P_2	P_0, P_2	P_0, P_1, P_2, P_3
P_3	P_0, P_1, P_2	P_1, P_3, P_4, P_5

TABLA 5.2: Tabla de correspondencias para 3P

3P	**4P**	**6P**
P_0	P_0, P_1, P_2	P_0, P_1
P_1	P_2, P_3	P_1, P_4
P_2	P_2, P_3	P_2, P_3, P_5

Para llevar a cabo el cambio gradual, la ULT dispone de otras tres tablas de ocupación en las que registra los tiempos en que las particiones de cualquier distribución quedarán libres. Cada vez que envía una tarea a ejecutar a la FPGA, consulta la tabla de correspondencias de la distribución que se está utilizando en ese momento y marca los tiempos de fin de ejecución para todas las demás distribuciones, escribiendo en ellas la cantidad $t+tej_i$ (si corresponde a un tiempo posterior al ya escrito). De esta manera, cuando un cambio en las particiones es notificado por la UAD, la ULT consulta la tabla correspondiente a la nueva distribución y a través de un sencillo cálculo comprueba si una partición de la nueva distribución de particiones está libre para ejecutar una tarea de las colas.

A continuación mostramos un ejemplo sencillo de funcionamiento de las tablas de correspondencias. Partiremos de un ejemplo en que la FPGA está vacía y estamos en la unidad de tiempo 0, funcionando con una distribución en

TABLA 5.3: Tabla de correspondencias para 6P

6P	**4P**	**3P**
P_0	P_0, P_2	P_0
P_1	P_0, P_1, P_2, P_3	P_0
P_2	P_2	P_2
P_3	P_2, P_3	P_2
P_4	P_2, P_3	P_1
P_5	P_3	P_2

4 particiones. Llega una tarea que tiene un tiempo de ejecución de 10 unidades de tiempo y que se asigna a la partición P_3. La tabla de ocupación para 4P se actualiza conforme muestra la tabla 5.4.

TABLA 5.4: Tabla de ocupación para 4P I

4P	P_0	P_1	P_2	P_3
tfin	0	0	0	10

La tabla de ocupación equivalente para 3 particiones se actualizará según muestra la tabla 5.5.

TABLA 5.5: Tabla de ocupación para 3P I

3P	P_0	P_1	P_2
tfin	10	10	10

Y la tabla de ocupación equivalente para 6 particiones quedaría según se muestra en la tabla 5.6.

Con esta información, si en $t = 4$ se notificara un cambio a 6 particiones y al mismo tiempo hubiera llegado una tarea pequeña para ejecutar, con tiempo

TABLA 5.6: Tabla de ocupación para 6P I

6P	P_0	P_1	P_2	P_3	P_4	P_5
tfin	0	10	0	10	10	10

de ejecución de 8 unidades de tiempo, la ULT dispondría de las particiones que cumplan la condición $t - tfin \geq 0$ para ejecutarla mientras termina la tarea anterior. Así, podría ejecutarla en la partición P_0 de la nueva distribución sin perturbar la ejecución de la otra tarea. Actualizaría las tablas de ocupación con la información de la nueva tarea según muestran las tablas 5.7, 5.8 y 5.9.

TABLA 5.7: Tabla de ocupación para 4P II

4P	P_0	P_1	P_2	P_3
tfin	12	0	0	10

TABLA 5.8: Tabla de ocupación para 3P II

3P	P_0	P_1	P_2
tfin	12	10	10

Este sencillo procedimiento permite a la ULT disponer de información actualizada de la ocupación de particiones para cualquiera de las distribuciones en todo momento y realizar el cambio en las particiones de forma gradual y con la menor penalización posible.

TABLA 5.9: Tabla de ocupación para 6P II

6P	P_0	P_1	P_2	P_3	P_4	P_5
tfin	12	10	0	10	10	10

5.3. Ejemplo

En esta sección mostraremos de forma detallada cómo funciona la Adaptación Dinámica. Utilizaremos un lote de 161 tareas, cuya composición se muestra en la figura 5.4.

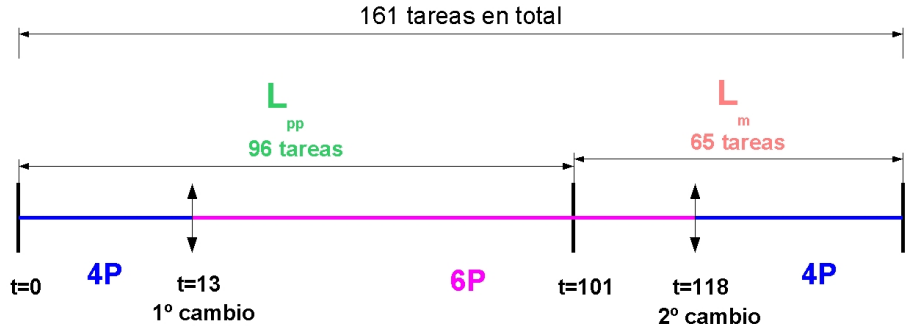

FIGURA 5.4: Composición del lote utilizado para el ejemplo

Las primeras 96 tareas corresponden a una distribución tipo L_{pp}, que se están ejecutando con 4P (tramo azul a la izquierda). La UAD detecta que la distribución en particiones no es adecuada y decide el cambio a 6P (tramo rosa). Las últimas 65 tareas del lote corresponden a una distribución tipo L_m. Tras el cambio en el tipo de distribución (paso de L_{pp} a L_m en $t = 101$), la UAD detecta que es más adecuado volver a una distribución en 4P y decide hacer un nuevo cambio (tramo azul a la derecha). La tabla 5.10 muestra el perfil de este lote de tareas.

En la tabla podemos ver que el conjunto del lote de tareas se parece más a

TABLA 5.10: Perfil del lote de tareas con dos cambios de particiones

$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$	$\overline{\mathbf{a}}$	$\overline{\mathbf{tej}}$	**Vas**
0,83	0,39	0,36	0,25	17,40	6,34	456004

una distribución plana que a ninguna de las otras definidas, si solamente nos fijamos en los parámetros β, δ y γ. Sin embargo el área media de las tareas del lote es la de una distribución L_{pp}. Esto significa que con esta carga de trabajo ninguna de las distribuciones en particiones conseguiría un máximo aprovechamiento de la FPGA y hemos elegido un ejemplo así para poner de manifiesto la necesidad de completar el algoritmo básico con la funcionalidad de Adaptación Dinámica, o lo que es lo mismo, con las posibilidad de utilizar diferentes distribuciones en particiones a lo largo de una ejecución para lograr el máximo rendimiento de nuestro algoritmo.

Si la versión básica del algoritmo ya supera en rendimiento a un algoritmo complejo como el FF, con la posibilidad de realizar la adaptación dinámica nuestro algoritmo consigue aumentar aún más esta diferencia en el rendimiento, como veremos en el capítulo 6 de Resultados Experimentales.

Pasamos ahora a analizar en detalle la ejecución de este lote de tareas y después presentaremos un resumen de la mejora en rendimiento obtenida gracias a la implementación de la Adaptación Dinámica.

La ejecución empieza con una distribución 4P. Sin embargo el conjunto de 96 primeras tareas que llegan al sistema corresponde a una distribución L_{pp}, por lo que sería más adecuado utilizar 6P para esta carga de trabajo. La figura 5.5 refleja la situación en este primer tramo de ejecución del lote-ejemplo.

La UAD detecta con bastante rapidez la desviación existente en D_p. La

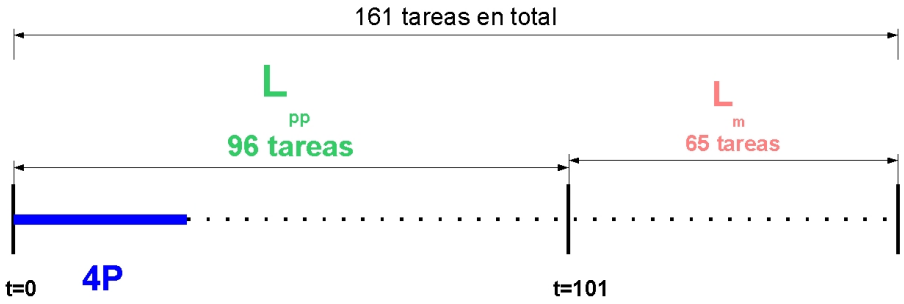

FIGURA 5.5: Primer tramo de la ejecución: L_{pp} en 4P

figura 5.6 nos muestra la variación de D_p en las 20 primeras unidades de tiempo de ejecución de este lote. En $t = 9$ han llegado las primeras 16 tareas al sistema y el primer valor de D_p calculado ya supera el valor umbral de 1,8.

FIGURA 5.6: Valores de D_p antes del primer cambio

Como el valor de D_p se mantiene por encima del umbral, la UAD decide realizar un cambio a 6P en $t = 13$. En este momento la FPGA está llena y hay una tarea esperando en las colas, que había llegado en $t = 9$. Es una tarea pequeña con tiempo de ejecución 7 $u.t.$ y que tiene que ejecutarse antes de $t = 23$ (el tiempo máximo utilizado para todas las pruebas realizadas es

el doble del tiempo de ejecución de las tareas, contado a partir del momento de su llegada). En $t = 13$ los contenidos de las tablas de ocupación son los mostrados en las tablas 5.11 y 5.12.

TABLA 5.11: Tabla de ocupación para 4P, primer cambio

4P	P_0	P_1	P_2	P_3
tfin	20	16	20	19

TABLA 5.12: Tabla de ocupación para 6P, primer cambio

6P	P_0	P_1	P_2	P_3	P_4	P_5
tfin	20	16	20	19	20	19

Puesto que la tarea que espera en las colas es de tamaño pequeño, podría ejecutarse en cualquiera de las particiones de 6P. La ULT le asigna P_1, que es la de menor tiempo de espera, al comprobar que $16 + 7 \leq 23$ y por tanto cumple el tiempo máximo de espera permitido para esta tarea. Cambia esta tarea a la cola Q_1 de 6P.

A partir de $t = 13$ la ejecución prosigue con una distribución en 6P, según muestra la figura 5.7.

La siguiente tarea no llega al sistema hasta $t = 14$, momento en el que se le asignará una cola conforme a la nueva distribución en 6P. La tupla que define a esta tarea es (15, 15, 14, 5, 24). La UPT comprueba el estado de las colas para 6P. Comprueba que en Q_3 cumple la condición $19 + 5 = 24 \leq 24$, y la escribe en Q_3 de la nueva distribución.

Las tres siguientes tareas llegan juntas en $t = 17$. Dos de ellas son de tamaño mediano y una de ellas es grande (de $25 \cdot 25$ CBRs). Han transcurrido

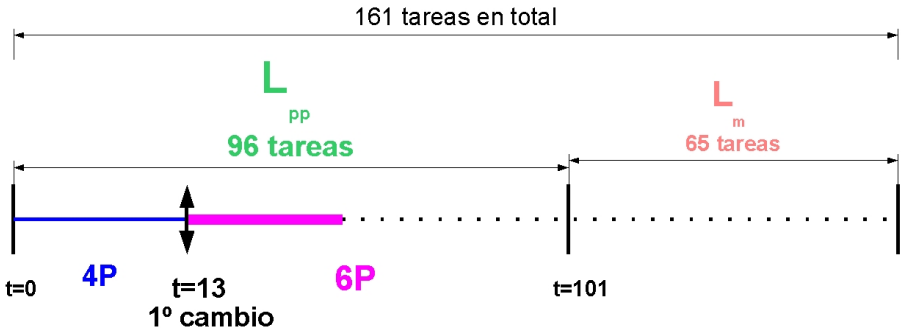

FIGURA 5.7: Segundo tramo de la ejecución: L_{pp} en 6P

4 unidades de tiempo desde que se realizó el cambio y todas ellas son asignadas a diferentes colas de la nueva distribución de particiones, ya que cumplen con su tiempos máximos.

Este es un ejemplo en el que se ha podido realizar el cambio gradual en las particiones sin penalización alguna.

La ejecución prosigue con una distribución 6P, con D_p por encima del valor $0,8$ y por debajo de $1,8$.

Sin embargo, en $t = 101$, la distribución de tareas cambia de L_{pp} a L_m. Es el momento en que llega una tarea de tamaño grande, del 42 % del área de la FPGA y se tiene que recurrir a la unión de las 4 particiones más pequeñas, por lo que empiezan a acumularse retrasos en el sistema.

La figura 5.8 nos muestra la parte de la ejecución en que nos encontramos ahora.

Si no se repitiera esta situación, el sistema volvería a equilibrarse y no habría pérdidas en el rendimiento. Sin embargo, siguen llegando tareas de un tamaño mayor al de las particiones 6P y en $t = 106$ una tarea grande debe ser rechazada. El algoritmo empieza a perder eficacia.

La figura 5.9 nos muestra la evolución de D_p en un tiempo anterior y

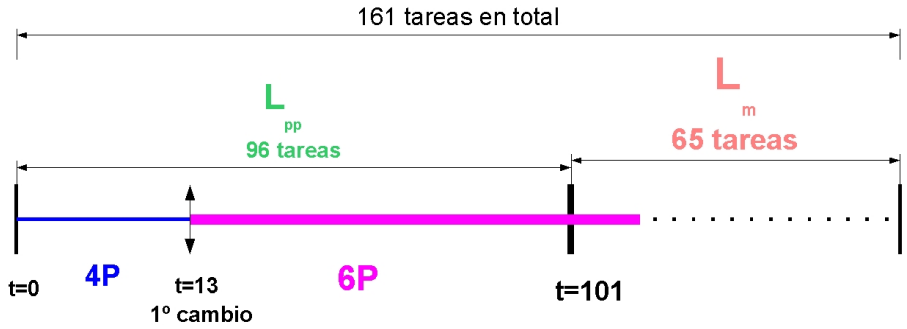

FIGURA 5.8: Tercer tramo de la ejecución: L_{pp} cambia a L_m y se está utilizando una distribución en 6P

posterior al segundo cambio en la distribución de tareas.

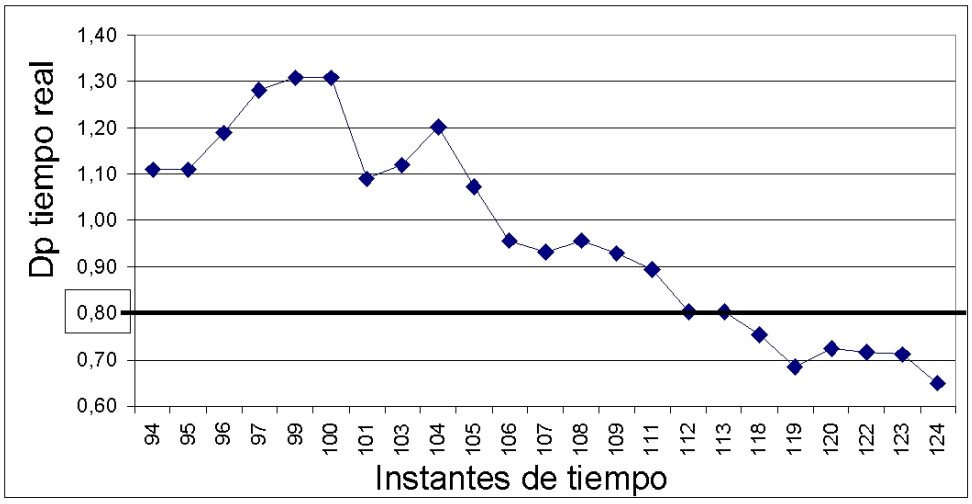

FIGURA 5.9: Valores de D_p en el segundo cambio

Observamos que en $t = 101$, momento en que llega la tarea grande, D_p baja su valor. Después llegan una tarea de tamaño mediano y otra pequeña y D_p vuelve a subir, pero cuando llega la siguiente tarea grande, en $t = 106$, su valor baja otra vez (ya que ahora hay dos tareas grandes en la ventana de 16 tareas y su peso en el área media es grande) y sigue bajando hasta que en $t = 112$ (en este momento llega otra tarea grande) toca el límite de $0,8$ por

primera vez. A partir de aquí sigue bajando y la UAD determina un cambio a 4P en $t = 118$.

En este momento hay tres tareas ejecutándose en la FPGA: una que ocupa $P_1 + P_3 + P_4 + P_5$ y otras dos en las dos particiones restantes. Las tablas de ocupación se muestran en las tablas 5.13 y 5.14.

TABLA 5.13: Tabla de ocupación para 6P, segundo cambio

6P	P_0	P_1	P_2	P_3	P_4	P_5
tfin	122	123	120	123	123	123

TABLA 5.14: Tabla de ocupación para 4P, segundo cambio

4P	P_0	P_1	P_2	P_3
tfin	123	123	123	123

En ese momento hay una tarea de tamaño mediano esperando en las colas, cuyo tiempo máximo de ejecución permite asignarle la partición P_2 en la nueva distribución 4P. Las siguientes tareas que llegan tienen también tiempos máximos que les permiten esperar en las colas hasta $t = 123$, así que ninguna es desechada. De nuevo se ha podido realizar el cambio gradual en las particiones sin tener que rechazar ninguna tarea.

El resto de la ejecución se termina con la distribución en 4P y sin que se rechacen tareas, con un buen aprovechamiento del área de la FPGA, según muestra la figura 5.10.

La tabla 5.15 compara el rendimiento que se habría obtenido con el algoritmo básico con el rendimiento obtenido con el algoritmo que implementa la

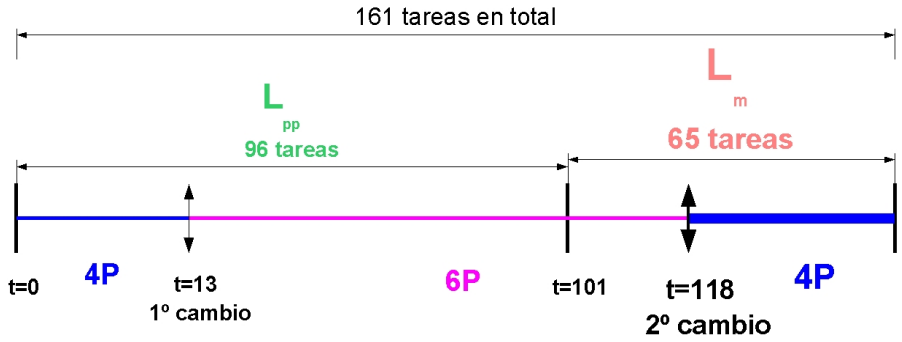

FIGURA 5.10: Último tramo de la ejecución: L_m en 4P

adaptación dinámica (AD) y ejecuta este lote con 4P-6P-4P. El volumen de la carga de trabajo de este lote de tareas es de 456,004 $CBRs \cdot \#u.t.$

TABLA 5.15: Comparativa de rendimientos

	Básico	**AD**
# u.t.	218	218
$\overline{\mathbf{a}}(\%\mathbf{A})$	70,15	77,73
$\mathbf{V_{ej}}$	382.269	423.636
$\mathbf{R}(\%)$	**83,83**	**92,90**

Esta tabla nos muestra que aunque el rendimiento del algoritmo básico es bastante bueno, el rendimiento con la adaptación dinámica es mucho mejor. Se ha desechado un mínimo de volumen de trabajo asignado (recordemos que antes de realizar el cambio se descartaron unas pocas tareas, que suponen un porcentaje bastante bajo de la carga de trabajo) y que el algoritmo, gracias a los cambios en las particiones realizados, ha logrado planificar la ejecución de prácticamente toda la carga de trabajo asignada. Para este caso, la mejora en el rendimiento con la adaptación dinámica es del $9,1\,\%$.

Este ejemplo nos muestra dos cambios de particiones en los que no ha sido necesario rechazar tareas al cambiar el número y tamaño de las particiones.

Esta situación no siempre se da. A partir de las pruebas realizadas, encontramos también ejemplos donde una de las tareas (y no siempre la última que llegó antes de decidir el cambio) tiene que ser rechazada. En general serán una o dos tareas las que se tienen que rechazar debido al cambio en las particiones, y en todos los casos estudiados y que se presentarán en el capitulo 6 de Resultados Experimentales, observamos que se puede alcanzar un buen aumento en el rendimiento del algoritmo al permitir el cambio en el número y tamaño de las particiones a lo largo de su ejecución.

5.4. Algoritmo de la UAD

A continuación mostramos el algoritmo que se utliza en la Unidad de Adaptación Dinámica. Este algoritmo, que se ejecuta en paralelo con el algoritmo de la Unidad de Planificación de Tareas, es también de complejidad constante.

Algoritmo 10 Algoritmo para la UAD

```
Escribir-tarea-en-FIFO(T_i);
FLLT_real = 16/(tll_m − tll_1);
ā=Calcula-area-media();
t̄ej = Calcula-tej-medio();
FLLT_ideal = A/ā · t̄ej;
α = FLLT_real/FLLT_ideal;
if (0,5 ≤ α) then
  D_p = A/ā · P;
  if (D_P ≥ 1,8) then
    contmasP = contmasP + 1;
    if (contmenosP ≥ 1) then
      contmenosP = contmenosP − 1;
    end if
  end if
  if (D_P ≤ 0,8) then
    contmenosP = contPmenos + 1;
    if (contmasp ≥ 1) then
      contmasP = contmasP − 1;
    end if
  end if
  if (0,8 < D_P < 1,8) then
    if (contmasp ≥ 1) then
      contmasP = contmasP − 1;
    end if
    if (contmenosP ≥ 1) then
      contmenosP = contmenosP − 1;
    end if
  end if
end if
if (contmasP = 3) then
  Aumentar-numero-de-particiones(ā);
end if
if (contmenosP = 3) then
  Disminuir-numero-de-particiones(ā);
end if
```

Algoritmo 11 Subrutina Aumentar-numero-de-particiones($\overline{a}$)

$D'_p = 0$;
if $(P = 3)$ **then**
  $P' = 4$;
  $D'_p = A/\overline{a} \cdot P'$;
  if $(D'_p \leq 1,8)$ **then**
    $cambio = true$;
    $w'_p = 0,3$,$h'_p = 0,3$, $P = P'$;
  end if
end if
if $(P = 4)$ or $(D'_p > 1,8)$ **then**
  $cambio = true$;
  $P' = 6$,$w'_p = 0,25$,$h'_p = 0,5$,$w'_{pp} = 0,5$,$P = P'$;
end if

Algoritmo 12 Subrutina Disminuir-numero-de-particiones($\overline{a}$)

$D'_p = 0$
if $(P = 6)$ **then**
  $P' = 4$
  $D'_p = A/\overline{a} \cdot P'$;
  if $(D'_p \geq 0,8)$ **then**
    $cambio = true$;
    $w'_p = 0,3$,$h'_p = 0,3$, $P = P'$;
  end if
end if
if $(P = 4)$or $(D'_p < 0,8)$ **then**
  $cambio = true$;
  $P' = 3$,$w'_p = 0,5$,$h'_p = 0,5$, $P = P'$;
end if

Capítulo 6

Resultados experimentales

En este capítulo presentamos los resultados de los diferentes experimentos realizados, tanto con la versión básica del algoritmo como de la versión con la funcionalidad de Adaptación Dinámica.

Todas las pruebas se han realizado con un simulador del entorno de planificación, que incluye la UPT, la ULT, la UAD, un vector de P elementos que representa la FPGA y un archivo con la entrada de datos que contiene las tuplas que definen a cada tarea.

Se han utilizado dos tamaños de FPGA diferentes: una de 20*20 CBRs y otra de 50*50 CBRs. Los tamaños de las particiones utilizadas (expresados en CBRs) se detallan en la tabla 6.1.

TABLA 6.1: Tamaños de particiones

FPGA	$\mathbf{P_0}$	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$
$20x20$	25	75	75	225
$50x50$	225	525	525	1225

Aunque el rendimiento del algoritmo, según hemos explicado en el capítulo 3, se mide como la relación entre el volumen ejecutado y el volumen asignado, presentaremos los resultados experimentales especificando tres parámetros, relacionados entre sí, para una mayor claridad. Los parámetros presentados en los resultados son:

1. **Rendimiento**: relación entre el volumen ejecutado y el volumen asignado, expresado en porcentaje según la fórmula $R = \frac{V_{ej}}{V_{as}} \cdot 100$.

2. **% Área media utilizada**: representa el porcentaje de área de FPGA utilizada en media durante la ejecución completa del lote de tareas.

3. **Unidades de Tiempo**: número total de unidades de tiempo, medido desde el momento en que llega la primera tarea del lote a ejecutarse en la FPGA hasta que termina la ejecución de la última tarea en la FPGA. Para un mismo volumen de trabajo ejecutado, a menor valor de unidades de tiempo, el valor del área media utilizada será mayor.

6.1. Experimentos con el algoritmo básico 4P

En esta sección presentaremos los resultados obtenidos para la simulación de una FPGA de 20*20 CBRs. Hemos creado lotes de tareas que representan distribuciones artificiales de tareas. Se han realizado estas pruebas con el objetivo de comprobar la eficiencia de nuestro algoritmo básico de 4 particiones (ABP) en comparación con otros algoritmos complejos, como First Fit, cuyo funcionamiento se ha descrito en detalle en el capítulo 3. Por esta razón todas

las pruebas presentadas en esta sección se refieren a una distribución del área de la FPGA en 4 particiones.

6.1.1. *Benchmark* artificiales

En primer lugar presentamos la tabla 6.2 con los detalles de las características de lotes de tareas artificiales que hemos construido para comaprar el rendimiento de FF con nuestro algoritmo. Todos ellos están compuestos por 52 tareas.

TABLA 6.2: Características de los *benchmark* artificiales

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ (%A)	$\overline{\overline{\mathbf{tej}}}$	V_{as}	$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$
L1	0,96	25,00	5,00	26.000	0,25	0,50	0,25
L2	1,03	25,00	5,00	26.000	0,25	0,50	0,25
L3	1,16	25,00	5,00	26.000	0,25	0,50	0,25
L4	0,94	24,52	5,00	25.500	0,29	0,46	0,25
L5	0,96	25,00	5,00	26.000	0,25	0,50	0,25
L6	0,96	25,00	5,00	26.000	0,25	0,50	0,25
L7	1,10	29,08	5,00	30.250	0,15	0,52	0,33
L8	1,22	32,21	5,00	33.500	0,13	0,46	0,41

A continuación pasaremos a comentar en detalle la información contenida en la tabla. Lo haremos lote por lote, explicando las diferencias entre algunos de ellos, que no se ven reflejadas en la tabla ya que dan lugar - aparentemente - a idénticos conjuntos de tareas y sin embargo contienen diferencias que, como veremos, generan resultados distintos.

Una característica común a todos estos *benchmark* artificiales es que tienen una frecuencia de llegada de tareas muy alta, por encima del 90 % de la ideal, y los primeros están muy cercanos a casos ideales. A partir de estos primeros

lotes prácticamente ideales hemos introducido variaciones en parámetros como el orden de llegada de tareas y desviaciones del caso ideal.

- **L1**: Se trata de un lote de tareas de tipo L_m casi ideal (el parámetro α vale prácticamente 1).
- **L2**: Es igual que el lote L1 (el mismo conjunto de tareas) pero con una frecuencia de llegada de tareas más alta (se han cambiado los tiempos de llegada de las tareas para que estén más cercanos).
- **L3**: También es igual al lote L1 pero se ha aumentado la frecuencia de llegada de tareas aún más.
- **L4**: Este lote muestra una ligera desviación respecto al ideal y contiene una proporción ligeramente mayor de tareas de tipo pequeño. Por ello el área media es algo menor a la de los anteriores y los valores de D_p también varían ligeramente. Esto es debido a que se ha representado una situación en la que llega un pequeño pico de tareas pequeñas, es decir, un grupo de 6 tareas pequeñas que llegan prácticamente seguidas, en medio de un lote de tareas de tipo mediano cercano al ideal.
- **L5**: Se ha construido con las mismas tareas que L1 y se han mantenido los tiempos de llegada de las tareas. Lo que se ha cambiado es el orden en que llegan al sistema. Podríamos decir que es como el conjunto L1 pero después de barajarlo y manteniendo los mismos tiempos de llegada para que la frecuencia de llegada de tareas no cambie.
- **L6**: Los tamaños de las tareas y sus tiempos de llegada son los mismos pero mientras que en L1 y los anteriores lotes los tiempos de ejecución

son uniformes, aquí el tiempo de ejecución medio es igual al de los otros lotes pero se ha obtenido asignando tiempos de ejecución dispares en las tareas (con una relación 1:10 entre ellos).

- **L7**: En este lote de tareas se ha aumentado la proporción de tareas grandes hasta sobrepasar ligeramente el 30 %, siendo la proporción de tareas pequeñas menor que en los lotes anteriores.

- **L8**: Toma como base el lote anterior y se incrementa la proporción de tareas grandes a costa de las pequeñas.

Los lotes de tareas cuyas características hemos mostrado y comentado se han diseñado a propósito para poder estudiar el impacto de determinados factores en el rendimiento de nuestro algoritmo. Estos factores y los lotes de tareas a partir de los cuales se pueden realizar las observaciones son:

1. **Frecuencia de llegada**: Los resultados correspondientes a los lotes L1, L2 y L3 nos mostrarán la efectividad de nuestro algoritmo en comparación con FF respecto al incremento de la frecuencia de llegada de tareas por encima de su valor ideal (es decir, cuando la carga de trabajo asignada sea superior al límite teórico de la capacidad de la FPGA).

2. **Pico de tareas pequeñas**: Los resultados correspondientes a los lotes L1 y L4 (con casi la misma frecuencia de llegada de tareas) nos mostrarán la efectividad de nuestro algoritmo para resolver situaciones puntuales de variación de la carga de trabajo con respecto a la distribución ideal. Hemos probado el caso de un pico de tareas pequeñas porque es el que

más perjudica al rendimiento de nuestro algoritmo frente al de FF, como se explicó en detalle en el capítulo 3.

3. **Orden de llegada de las tareas**: La comparación en la ejecución de los lotes L1 y L5 nos mostrará la sensibilidad de nuestro algoritmo frente a FF respecto al orden de llegada de las tareas. Este aspecto también se comentó con detalle en el ejemplo del capítulo 3.

4. **Uniformidad en los tiempos de ejecución**: La comparación entre los lotes L1 y L6 nos dará una idea de la sensibilidad de nuestro algoritmo frente a la dispersión en los valores de los tiempos de ejecución de las tareas.

5. **Sistema sobrecargado**: La comparación entre los lotes L3, L7 y L8 nos dará una idea de la eficiencia de nuestro algoritmo para resolver situaciones de sobrecarga de trabajo. Observamos que en los dos últimos lotes, L7 y L8, el volumen de trabajo asignado es un 20 % mayor que en los demás lotes y la frecuencia de llegada de tareas es también bastante superior a la ideal. Este efecto se ha conseguido disminuyendo los tiempos entre la llegada de tareas y aumentando considerablemente el tamaño de éstas.

La tabla 6.3 nos muestra los resultados de la ejecución de nuestro algoritmo y del algoritmo FF para estos *benchmark*.

En la figura 6.1 vemos que el rendimiento de nuestro algoritmo básico es claramente superior al de FF en todos los casos que hemos simulado.

TABLA 6.3: Resultados para los *benchmark* artificiales

	R (%)		# **u.t.**		**Área** (%)	
Lote	**ABP**	**FF**	**ABP**	**FF**	**ABP**	**FF**
L1	100,0	78,8	69	69	94,2	74,2
L2	100,0	58,2	69	64	94,2	59,0
L3	75,5	45,2	61	57	80,4	51,5
L4	100,0	75,4	69	70	91,1	68,8
L5	100,0	53,8	69	69	94,2	50,7
L6	98,7	78,4	69	70	91,7	73,8
L7	85,1	60,3	72	69	85,1	60,3
L8	76,5	57,5	71	69	90,7	69,7

6.1.1.1. Conclusiones del análisis de los resultados para *benchmark* artificiales

En primer lugar vamos a resumir unas conclusiones generales, aplicables a todos los lotes de tareas y después comentaremos en detalle los aspectos mencionados anteriormente, relacionados con grupos específicos de lotes de tareas.

1. **Rendimiento**: Como ya se indicó en el ejemplo de funcionamiento explicado en detalle en el capítulo 3, el algoritmo FF presenta una clara desventaja frente al nuestro siempre que haya una presencia de tareas de tamaño relativamente grande y solamente obtendrá resultados iguales o mejores que el nuestro en casos bastante extremos donde tengamos prácticamente la totalidad de tareas de tamaño pequeño (se mostrarán este tipo de resultados más adelante en este capítulo). De la primera columna podemos deducir que para una gran cantidad de situaciones y siempre que aparezcan tareas de tamaño superior al 25 % del tamaño de

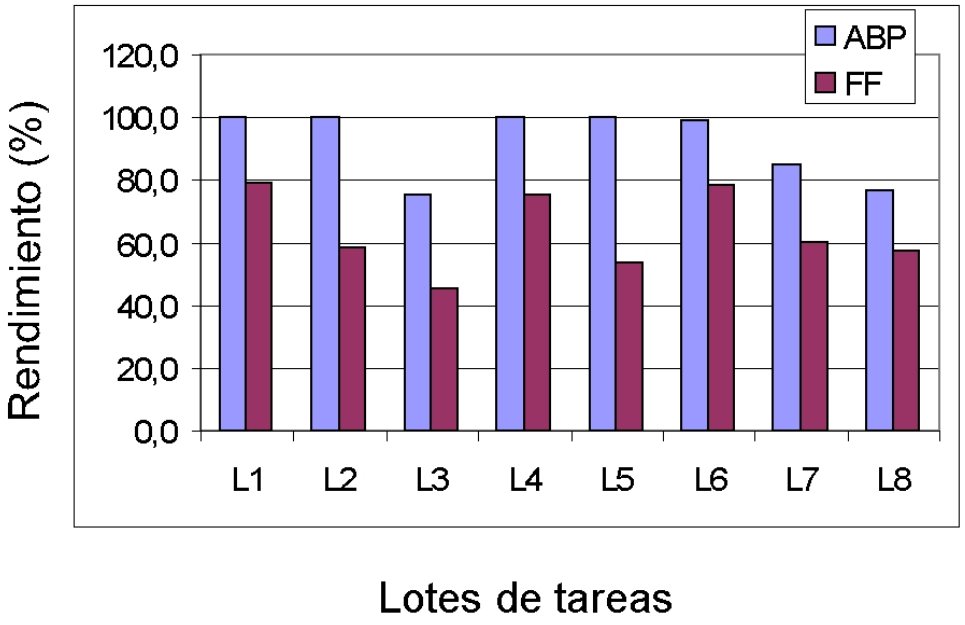

FIGURA 6.1: Comparación del Rendimiento de ABP con FF

la FPGA, nuestro algoritmo básico de complejidad constante que divide el área de la FPGA en 4 particiones de diferentes tamaños presenta un rendimiento superior al algoritmo FF, lo que también se ve reflejado con claridad en la figura 6.1.

2. **Tiempo de ejecución del lote**: Si observamos las unidades de tiempo necesarias para ejecutar los lotes completos observamos que nuestro algoritmo está ligeramente por encima de FF en la mitad de los casos. Esto no es debido a una mala gestión por parte de nuestro algoritmo sino al hecho de que ejecuta una mayor cantidad de trabajo, como puede observarse en la columna del Rendimiento. Por ello podemos decir que realmente esta columna analizada en conjunto con la tercera nos indica que nuestro algoritmo puede ejecutar una cantidad de trabajo mayor que FF en el mismo tiempo o un tiempo ligeramente superior. El caso con mayor

diferencia es el del lote L2 en que la diferencia de tiempos es del $7,8\,\%$ a favor de FF mientras que la diferencia de volumen de trabajo ejecutado es de $41,8\,\%$ a nuestro favor. En algunos casos, como el representado por el lote L5, la cantidad de trabajo ejecutada por nuestro algoritmo es casi el doble que la ejecutada por FF mientras que el tiempo empleado es el mismo. El hecho de poder ejecutar la misma o superior carga de trabajo en tiempos parecidos se ve reflejada en la tercera columna, que muestra el área media de FPGA utilizada en cada ejecución.

3. **Área media utilizada**: A pesar de la sencillez de planteamiento de nuestro algoritmo y de no ser un algoritmo avaricioso en uso de área, vemos que la idea de equilibrar y organizar la carga de trabajo adecuadamente en el espacio a través de la clasificación por tamaños (particiones) y en el tiempo a través de los tiempos máximos de ejecución (tiempos de espera en cada cola) resulta ser más eficaz en el uso de área que el planteamiento del algoritmo de FF, dando en todos los experimentos realizados un porcentaje superior en uso de la FPGA.

Pasamos ahora a analizar los aspectos concretos que determinan las diferencias entre los lotes de tareas:

1. **Frecuencia de llegada**: Los resultados correspondientes a los lotes L1, L2 y L3 prueban una mayor efectividad de nuestro algoritmo en comparación con FF respecto al incremento en la frecuencia de llegada de tareas. En el lote L1 el valor de α es cercano al ideal y representa una situación de carga de trabajo de la FPGA al borde de la saturación. Vemos que en este caso FF no es capaz de ejecutar la totalidad del trabajo asignado,

debido, como ya hemos dicho, a que en dicho lote hay un 25 % de tareas de tamaño grande. Observamos también que la pérdida de rendimiento del algoritmo FF con el aumento en la frecuencia de llegada de tareas es mucho más rápido que para nuestro algoritmo, que permite un ligero margen por encima de $\alpha = 1$. Para L3, donde la sobrecarga del sistema es evidente (α es un 16 % superior al límite teórico establecido para tener la FPGA llena el 100 % del tiempo) nuestro algoritmo tampoco es capaz de ejecutar la totalidad de la carga de trabajo asignada, pero aún así muestra un rendimiento muy superior al de FF.

2. **Pico de tareas pequeñas**: Comparando los resultados correspondientes a los lotes L1 y L4, observamos que nuestro algoritmo básico es eficaz resolviendo situaciones puntuales, como un breve aumento de tareas pequeñas, que hemos llamado pico de tareas.

3. **Orden de llegada de las tareas**: La comparación en la ejecución de los lotes L1 y L5 prueba que nuestro algoritmo es menos sensible al orden de llegada de tareas que FF, ya que clasifica las tareas y las mantiene esperando en la cola que les corresponde siempre que su tiempo de espera cumpla con los límites de tiempo establecidos para la tarea. Esto hace que el orden de llegada de tareas afecte en menor medida a los resultados de la ejecución que en el caso de FF en el que el orden de llegada de las tareas tiene un gran impacto en la fragmentación del área libre de la FPGA.

4. **Uniformidad en los tiempos de ejecución**: La comparación entre los lotes L1 y L6 nos indica la importancia de este aspecto en el rendimiento

del algoritmo. Si los tiempos de ejecución de las tareas son parecidos, nuestro algoritmo hará una mejor gestión del área de la FPGA que si son muy dispares. Aunque el rendimiento de nuestro algoritmo aun en el caso de mayor disparidad entre los tiempos de ejecución de las tareas es superior al de FF, en este ejemplo observamos que el rendimiento de nuestro algoritmo se ve más afectado por este hecho que el algoritmo FF. Esto puede explicarse por el hecho de que si una de las tareas tiene un tiempo de ejecución mucho más alto que las otras la cola asociada a la partición que le corresponda mostrará un tiempo de espera mucho más alto, obligando entonces a que otras tareas que deberían asignarse a dicha cola se envíen a otras colas correspondientes a un tamaño mayor, por lo que el uso del área de la FPGA es peor y el rendimiento del algoritmo disminuye (esto se ve claramente reflejado en la segunda columna para los lotes L1 y L6 donde el uso medio de área de nuestro algoritmo es menor cuando los tiempos de ejecución son más dispares). Aún así conviene destacar que nuestro algoritmo ejecuta una mayor cantidad de trabajo que el algoritmo FF y en el mismo tiempo.

5. **Sistema sobrecargado**: La comparación entre los lotes L3, L7 y L8 nos indica que la eficiencia de nuestro algoritmo para resolver situaciones de sobrecarga de trabajo es mejor que la del algoritmo FF. En los dos últimos lotes, L7 y L8, el parámetro γ que indica la proporción de tareas grandes en el lote, está por encima del 30 %, causando un deterioro en el rendimiento del algoritmo FF mucho más acusado que en el nuestro. Estos dos últimos lotes de tareas confirman además la necesidad

de mejorar el rendimiento de nuestro algoritmo añadiendo la capacidad de ajustar el tamaño y número de particiones a diferentes situaciones. Estos dos ejemplos de conjuntos de tareas nos han servido también para darnos cuenta que una proporción de tareas grandes superior al 40 % en un conjunto de tareas era un indicativo claro de la necesidad de cambiar el esquema básico de particiones, ya que el rendimiento obtenido con la versión básica empieza a estar por debajo de límites aceptables (un rendimiento ligeramente superior al 75 % indica que aproximadamente una de cada cuatro tareas es rechazada).

6.1.2. *Benchmark* sintéticos

Puesto que los lotes presentados en la sección anterior corresponden a situaciones creadas artificialmente, hemos creído conveniente construir un lote de tareas a partir de datos de la síntesis de tareas reales. Una explicación detallada de las aplicaciones utilizadas y cómo se han sintetizado se puede obtener en [NHCB02]. Nos hemos basado en esta única publicación dada la dificultad de encontrar información detallada y completa (es decir, número de CLBs utilizados y tiempos de ejecución) acerca de la síntesis de aplicaciones reales.

La composición del *benchmark* sintético se muestra en la tabla 6.4, dando un total de 106 tareas. Hemos incluido también la aplicación JPEG sintetizada en una Virtex-2 XCV2P30, en una versión en que algunas subtareas se ha dividido en otras más pequeñas e idénticas (JPEG paralelo), dando lugar a un conjunto de 16 tareas de tamaño pequeño.

Las aplicaciones utilizadas son de tamaños muy diversos y tiempos de eje-

TABLA 6.4: Composición del *benchmark* sintético

Aplicación	**Nº ejecuciones**
parallel JPEG	2 (16 sub-tareas)
Sobel	14
Multiplicación de matrices	3
Closure	2
Homogeneous	12
Image thresholding	59

cución muy dispares, variando del tamaño 15*15 y tiempo de ejecución $1ms$ hasta tamaños de $11x15$ y tiempo de ejecución $13\,ms$ o tamaños de 4*2 y tiempo de ejecución $22\,ms$. La tabla 6.5 nos muestra las características de este conjunto de tareas, donde se especifica el número de ejecuciones de cada aplicación en el *benchmark*.

TABLA 6.5: Características del *benchmark* sintético

$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ (%A)	$\overline{\mathbf{tej}}$	V_{as}	$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$	D_p3P	D_p4P	D_p6P
0,63	20,51	6,56	24.694	0,49	0,23	0,28	1,62	1,22	0,81

Podemos observar que no se corresponde exactamente con ninguna de las distribuciones teóricas estudiadas. Del estudio de los parámetros α, β y γ sería una distribución de tipo pequeñas con una proporción de grandes ligeramente superior al 25 % definido para este tipo de distribución. Por ello el área media de las tareas es del máximo del tamaño de tareas medianas aunque no sean las que predominen en el lote. Esto nos da una distribución muy interesante ya que nos permitirá comprobar el grado de fiabilidad del modelo teórico a la hora de ser puesto en práctica en un entorno real de trabajo.

Analizando los valores obtenidos para el parámetro D_p, indicador de la adecuación del tipo de distribución de particiones a la carga de trabajo, observamos que aunque la mayoría de tareas son de tamaño pequeño, al haber una proporción bastante alta de tareas grandes (cercana al 30 %), una distribución en 6 particiones no sería adecuada (recordemos que un D_p teórico de $0,8$ supone que en tiempo real se alcanzarán valores muy bajos, del orden de $0,6$ y que indican que hay una clara infra-utilización del área de la FPGA). Además se tendría que recurrir a la unión de particiones con excesiva frecuencia.

Una distribución en 3 particiones tampoco lo sería, debido a la gran proporción de tareas pequeñas en el conjunto, que obliga a tener un mayor número de particiones disponible para poder ejecutar un mayor número de ellas simultáneamente.

El valor obtenido para 4 particiones está también lejos del ideal pero es el más cercano de todos. La tabla 6.6 nos muestra los resultados obtenidos para la ejecución de este *benchmark* sintético con nuestro algoritmo básico, es decir con 4 particiones. Para este caso $u.t = 1\,ms$.

TABLA 6.6: Resultados de la ejecución en 4P

	FF	**ABP**
% área media usada	45,00	55,10
Unidades de tiempo	104	112
Rendimiento	75,8	100

6.1.2.1. Conclusiones del análisis de los resultados para *benchmark* sintéticos

Los resultados presentados en la tabla 6.6 nos muestran la misma tendencia en cuanto a la comparación de rendimiento entre nuestro algoritmo y el algoritmo FF que hemos observado en la sección anterior con los *benchmark* artificiales. Aun a pesar de la alta presencia de tareas de tamaño pequeño (situación ventajosa para FF frente a nuestro algoritmo). La inevitable presencia (puesto que se trata de aplicaciones del mundo real), de tareas grandes en el conjunto tiene una influencia negativa mucho mayor en el rendimiento de FF. La carga de trabajo ejecutada es claramente inferior a la ejecutada por nuestro algoritmo, dando un peor uso del área, aunque en cualquiera de los dos casos el área de la FPGA no es utilizada de forma intensiva. Además el algoritmo FF utiliza más tiempo para ejecutar una menor carga de trabajo.

Este ejemplo, por supuesto, podría ser ampliado con más experimentos, pero no deja de ser un indicativo de la tendencia que podemos esperar en el comportamiento de sistemas de computación de propósito general que incluyan una FPGA como elemento procesador, donde se ejecuten tareas diversas de tamaños muy diferentes y tiempos de ejecución dispares. Aun en casos que no se parecen en absoluto a los artificiales, nuestro algoritmo sigue mostrando una mayor eficiencia a la hora de gestionar el DHWR.

6.2. Experimentos con Adaptación Dinámica

En esta sección presentaremos los resultados obtenidos de la ejecución de la versión del algoritmo que realiza la Adaptación Dinámica.

Hemos simulado una FPGA de 50*50 CBRs. En algunos casos hay un solo cambio en las particiones y en otras hay dos y se han probado todas las combinaciones de cambios en las particiones relevantes (hemos omitido 3P ->6P y 6P ->3P por considerar que son situaciones bastante excepcionales aunque el algoritmo las realizaría si fuera necesario de una vez y sin tener que pasar por la distribución en 4P).

En primer lugar presentamos la tabla 6.7 que resume las características de los *benchmark* que hemos utilizado para probar la funcionalidad de Adaptación Dinámica y comparar la eficiencia de las dos versiones del algoritmo y con el algoritmo FF.

TABLA 6.7: Características de los *benchmark* para Adaptación Dinámica

Lote	$\boldsymbol{\alpha}$	$\overline{\mathbf{a}}$ (%A)	$\boldsymbol{\beta}$	$\boldsymbol{\delta}$	$\boldsymbol{\gamma}$	**Tipo**	**Nº tareas**	**Cambios**
AD1	0,69	13,88	0,49	0,32	0,19	L_p	104	4P ->6P
AD2	0,58	14,16	0,49	0,30	0,21	L_p	103	4P ->6P
AD3	0,62	12,36	0,50	0,25	0,25	L_p	96	4P ->6P
AD4	0,69	28,75	0,25	0,25	0,50	L_g	98	4P ->3P
AD5	0,75	28,56	0,25	0,25	0,50	L_g	102	4P ->3P
AD6	0,92	22,54	0,25	0,50	0,25	L_m	103	3P->4P
AD7	0,77	21,03	0,25	0,50	0,25	L_m	94	3P->4P
AD8	0,87	22,00	0,25	0,25	0,50	L_m	100	3P ->4P
AD9	0,83	17,40	0,39	0,36	0,25	L_p->L_m	161	4P->6P->4P

Los tres primeros lotes de tareas, AD1, AD2 y AD3 corresponden a diferentes representaciones de distribuciones de pequeñas, L_p, con distintas cargas de trabajo correspondientes a sistemas equilibrados y frecuencias de llegada de tareas también diferentes. El caso del lote AD3 es el único que muestra una distribución igual a la teórica mientras que los otros dos muestran ligeras

variaciones respecto al mismo. Todas ellas corresponden a distribuciones para las cuales una distribución en 6 particiones es mucho más adecuada que el caso base. Para probar la eficacia de la Adaptación Dinámica, hemos supuesto que inicialmente se están ejecutando en un modelo 4P y que el sistema detecta la necesidad de cambiar a 6P, lo que sucede en algún punto de la ejecución, de ahí la información mostrada en la última columna de la tabla.

Los siguientes, AD4 y AD5, corresponden a lotes de tareas grandes, también con cargas de trabajo equilibradas aunque superiores a los lotes anteriores, cuya ejecución en un modelo 3P es más efectiva que en 4P. Se ha simulado la ejecución inicial de estos lotes con 4P y el módulo de Adaptación Dinámica detecta esta situación y la corrije pasando a un modelo en 6P.

Los tres siguientes lotes, AD6, AD7 y AD8, corresponden a situaciones inversas a la anterior, en que un lote de tareas de tipo mediano (y cuya ejecución es más efectiva en 4P que en 3P) se está ejecutando con una distribución 3P y el algoritmo detecta la situación y la corrije pasando a 4P.

Por último el lote de tareas AD9 contiene un mayor número de tareas (y una mayor carga de trabajo que todos los anteriores) porque simula una distribución de pequeñas que se está ejecutando inicialmente en 4P. El sistema detecta y corrije la situación pasando a 6P, pero más adelante la distribución cambia de perfil y pasa a ser de tipo medianas, con lo cual la distribución 6P no es la adecuada. El sistema detecta este cambio en la distribución y vuelve al modelo de ejecución en 4 particiones.

A continuación presentamos la tabla 6.8 con los resultados obtenidos de ejecutar estos lotes de tareas. Para estas pruebas hemos realizado tres ejecuciones de cada lote:

1. **Versión básica del algoritmo (ABP)**: en la que el lote completo se ejecuta con la distribución en 4 particiones para los lotes AD1 a AD4 y AD9 y con la distribución en 3 particiones para AD6 a AD8.

2. **Versión con Adaptación Dinámica (AD)**: en la que se habilita la funcionalidad de AD y se produce un cambio en la distribución de particiones (dos en el caso de AD9) a lo largo de la ejecución del lote (los cambios que se producen para cada caso están reflejados en la tabla 6.7).

3. **Algoritmo FF**: además de comparar la eficiencia de nuestras dos versiones del algoritmo basado en particiones, hemos obtenido también los resultados de la ejecución del algoritmo FF para estos lotes de tareas.

TABLA 6.8: Resultados para los *benchmark* de AD

	R (%)			**#u.t.**			**área(%)**		
Lote	**ABP**	**AD**	**FF**	**ABP**	**AD**	**FF**	**ABP**	**AD**	**FF**
AD1	80,57	94,57	73,24	134	133	132	51,85	60,65	50,63
AD2	84,82	99,48	76,42	154	152	153	48,00	57,03	47,27
AD3	83,89	94,03	75,59	126	125	127	52,20	58,97	49,67
AD4	89,62	97,46	64,67	236	234	232	55,24	60,59	45,41
AD5	83,88	97,53	62,42	220	223	218	57,21	65,62	46,20
AD6	76,91	97,82	65,59	186	185	182	67,49	86,30	59,34
AD7	83,96	97,53	69,25	195	193	190	61,79	72,51	54,80
AD8	75,93	97,68	55,43	179	178	177	64,90	83,95	55,43
AD9	83,83	92,90	59,83	218	218	218	70,15	77,73	52,57

En todos los casos la versión del algoritmo con Adaptación Dinámica consigue un rendimiento mayor que el algoritmo sin esta funcionalidad. Además es destacable que con la Adaptación Dinámica se consigue ejecutar más del 90 %

de la carga de trabajo asignada al sistema. La figura 6.2 nos muestra la comparativa de rendimiento de la versión básica (ABP), la versión con adaptación dinámica (AD) y FF.

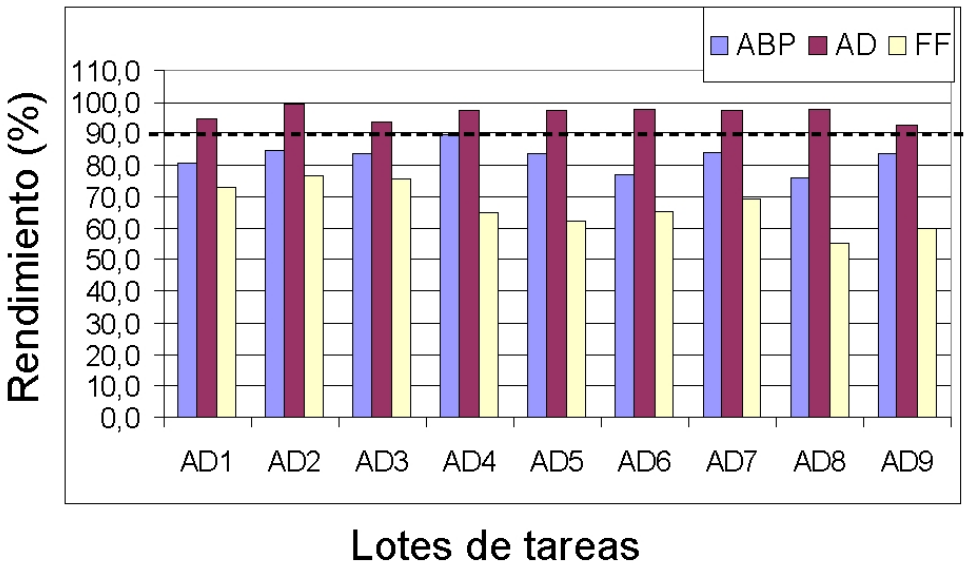

FIGURA 6.2: Comparativa de rendimientos

La mejora de rendimiento de la versión con AD frente al algoritmo sin ella es bastante variable, como se explicó en detalle en el capítulo 4, ya que depende de las tareas que se están ejecutando en el momento de tomar la decisión de cambiar la distribución en las particiones, así como de los tiempos máximos de ejecución de las tareas que están esperando en las colas en ese momento, de las siguientes tareas que vayan llegando y de la duración de los cambios en las distribuciones (cuanto más tiempo permanezca estable un tipo de distribución después de realizar el cambio mayor será la mejora en el rendimiento que se obtiene).

La mejora de rendimiento con la funcionalidad de AD varía desde el 8 % en el peor de los casos para el lote AD4, porque la versión básica sin AD ya

presenta un rendimiento bastante alto, cercano al 90 %, o superior al 20 % para los lotes AD6 y AD8 que son los que menor rendimiento obtienen con la versión básica. La figura 6.3 nos muestra el incremento en el rendimiento de nuestro algoritmo cuando se utiliza la Adaptación Dinámica para los lotes de tareas presentados.

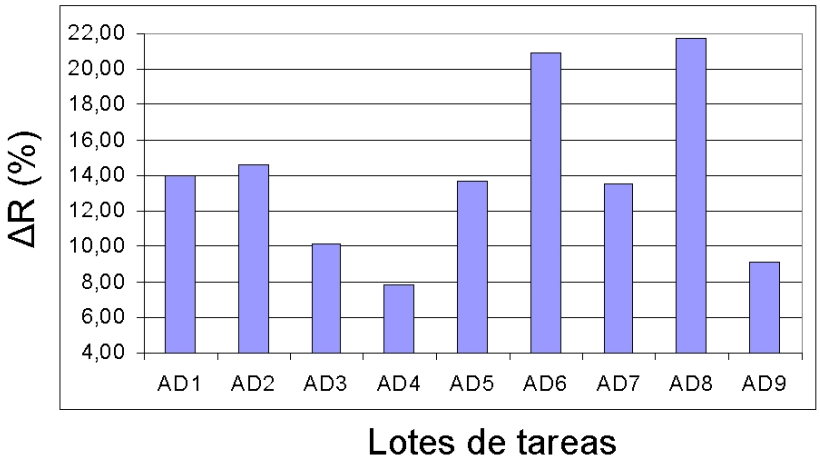

FIGURA 6.3: Mejora del Rendimiento con Adaptación Dinámica

6.2.1. Conclusiones del análisis de los resultados para *benchmark* de AD

De las pruebas realizadas y cuyos resultados se muestran en la tabla 6.8 podemos sacar las siguientes conclusiones respecto a la funcionalidad de Adaptación Dinámica:

1. Incluso sin la funcionalidad de Adaptación Dinámica y utilizando una distribución en particiones que no es la más adecuada para las características de un conjunto de tareas, nuestro algoritmo presenta un mejor rendimiento que el algoritmo FF, aunque la diferencia en el rendimiento

es algo menor que para los otros *benchmark* con los que se han realizado las pruebas de las secciones anteriores.

2. La funcionalidad de Adaptación Dinámica supera en rendimiento a la versión básica del algoritmo y garantiza la ejecución de prácticamente la totalidad de la carga de trabajo (rendimiento superior al 90 %). Podemos decir por tanto que nuestro algoritmo en su versión completa, es decir, con la capacidad de adaptarse dinámicamente es claramente mejor que el algoritmo FF para cualquier situación que pueda presentarse a lo largo de la ejecución.

3. La funcionalidad de Adaptación Dinámica dota a nuestro algoritmo de una estabilidad de rendimiento de la que carece la versión básica, ya que proporciona la flexibilidad necesaria para ajustar la distribución en particiones a cualquier situación y garantiza la ejecución de prácticamente cualquier carga de trabajo, sea cual sea su comportamiento.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

De todo lo expuesto en este trabajo de investigación podemos obtener algunas conclusiones acerca de los objetivos alcanzados a través del diseño de nuestro entorno dinámico para la gestión de DHWR basado en la división de una FPGA en particiones de diferente tamaño.

La primera de ellas es que con **algoritmos sencillos** y **fáciles de implementar** como los presentados en el capítulo 3, ambos de **complejidad constante** se puede **igualar e incluso mejorar sustancialmente la eficiencia** a la hora de gestionar el dispositivo respecto a algoritmos complejos avariciosos en uso de área como por ejemplo FF. Estos algoritmos utilizan una **estructura de datos muy sencilla** para representar el área libre en la FPGA y realizan unos **sencillos y rápidos cálculos** tanto para asignar una ubicación a cada nueva tarea como para examinar la adecuación de la división en particiones y decidir si es necesario realizar un cambio y cuál debe ser dicho

cambio.

Un aspecto muy importante de nuestros algoritmos de complejidad constante es que están **basados en las posibilidades reales de la tecnología existente**, como se explica en detalle en el apéndice A y que ha sido implementado sobre una Virtex-2 XCV2P30, como se detalla en el apéndice B.

Además nuestro algoritmo de selección de una ubicación presenta una ventaja importante frente a los otros algoritmos complejos y es el hecho de que en el momento de llegada de cada nueva tarea **se puede calcular con rapidez y exactitud el tiempo de espera** para dicha tarea. **En el caso de que tenga que ser rechazada, esta información se comunica de forma inmediata al resto del SO** y no es necesario esperar a la expiración de su tiempo máximo como ocurre con el uso de algoritmos complejos, retrasando así la comunicación de rechazo de tarea, lo que limita las posibilidades para su re-planificación.

El **rendimiento de la versión básica del algoritmo es superior al de FF** para cualquier conjunto de tareas donde existan tareas de tamaño grande, y la limitación de uso de área debido a la división en particiones **solamente resulta menos eficiente en casos extremos** en que todas las tareas sean de tamaño pequeño y con una frecuencia de llegada alta. En los resultados presentados se observa también que el algoritmo básico es menos sensible al orden de llegada de tareas y presenta un mejor rendimiento en situaciones de sobrecarga del sistema (responde mucho mejor a los aumentos en la frecuencia de llegada de tareas). Además puede absorber sin perjuicio de su rendimiento picos de tareas de tamaño pequeño que puedan presentarse a lo largo de la ejecución.

La **funcionalidad de Adaptación Dinámica añade una considerable mejora al rendimiento del algoritmo básico**, hasta el punto de que permite **garantizar la ejecución de más del 90 % de la carga de trabajo** para un amplio rango de tipos de distribuciones de tareas y siempre que el sistema no esté sobrecargado. Como ya se ha comentado, existe una excepción, que es el caso en que todas las tareas a ejecutar sean de tamaño pequeño y con una frecuencia de llegada alta. **El rendimiento del algoritmo aumenta una media del 15 %** con respecto a la versión básica, como se demostró en las pruebas presentadas en el capítulo 6.

Este dato es muy significativo si tenemos en cuenta que la complejidad del algoritmo que examina la adecuación de la distribución de las particiones a la carga de trabajo y toma las decisiones acerca de los cambios a realizar es **también de complejidad constante**.

Además, y gracias a una **estratégica situación de los accesos al bus de comunicaciones**, se evita su reconfiguración al realizar un cambio en las particiones, lo que **elimina la penalización temporal de realizar el cambio**.

Otro aspecto también muy importante es la utilización de las **tablas de correspondencias de particiones**, que permiten realizar el **cambio de particiones de forma gradual** y **minimizar el coste en volumen de trabajo rechazado** asociado a dichos cambios.

Resumiendo todo lo expuesto podemos concluir **que una división del área de la FPGA en particiones, siempre que se realice de forma dinámica, permite encontrar de manera rápida y eficiente una ubicación para la ejecución de tareas y garantiza la ejecución de prácti-**

camente la totalidad de cualquier carga de trabajo asignada para su ejecución HW, implementando de forma sencilla, realista y eficaz la multitarea HW sobre DHWR.

7.2. Trabajo futuro

Como líneas de investigación que se derivan del presente trabajo mencionaremos tres:

1. **Incluir distribuciones con mayor número de particiones**: puesto que las capacidades de las FPGAs van en aumento, parece evidente que una división del área en 6 particiones puede resultar insuficiente en algunos casos y que sería conveniente incluir en el entorno el estudio de la división de la FPGA en un número mayor de particiones. Esto permitiría ejecutar un mayor número de tareas simultáneamente aunque plantea nuevos problemas a la hora de gestionar las comunicaciones entre tareas y la entrada / salida de datos.

2. **Refinar el estudio en tiempo real** incluyendo los parámetros β, δ y γ en la observación del sistema, para que la decisión de realizar un cambio no se refleje únicamente en el número de particiones sino también en los tamaños de dichas particiones. Se podrían así manejar distribuciones en particiones con el mismo número de particiones pero diferentes tamaños. Por ejemplo, si todas las tareas de una carga de trabajo tienen tamaños inferiores al 25 % pero superiores al 10 %, lo más conveniente sería dividir el área de la FPGA en 4 particiones de igual tamaño.

3. **Implementación de un prototipo 2D**: aunque ya se implementó en su momento un prototipo en una FPGA de la familia Virtex-2, se trataba de una adaptación del entorno propuesto a 1D. Puesto que recientemente han aparecido en el mercado FPGAs que se pueden reconfigurar en 2D, sería interesante implementar el algoritmo en una de ellas: posiblemente la Virtex-5.

Apéndice A

Modelo de reconfiguración parcial

A.1. Reconfiguración parcial en 2D

Las FPGAs más recientes de Xilinx, desde la Virtex-4, tienen un modelo de reconfiguración parcial dinámica muy similar. Ambas son reconfigurables parcialmente en 2D.

Las FPGAs se configuran a través de un conjunto de bits o mapa de bits de configuración. Este mapa de bits contiene comandos y datos de configuración. Los bits se agrupan en *frames*. Un *frame* es la mínima cantidad de información que se puede leer o escribir en la FPGA y cada uno de ellos configura una parte de la FPGA. Su tamaño determina por tanto la mínima porción de FPGA reconfigurable dinámicamente.

La unidad mínima básica reconfigurable en las Virtex 4 es un *frame* de 16 CLBs de altura y una columna de ancho. De esta forma se pueden reconfigurar áreas de la FPGA de un número múltiplo de 16 CLBs de alto y varias columnas de ancho. Esto quiere decir que nuestro modelo de gestión del área de la FPGA

es idóneo para la tecnología actual.

Cada *frame* está compuesto por 41 palabras y lleva asociada una dirección de 32 bits que lo identifica dentro de la FPGA. De estos bits, hay uno que indica si la zona a configurar está en la parte superior (*top*) o inferior (*bottom*) de la FPGA. Las columnas se numeran de izquierda a derecha, mientras que las filas se numeran a partir de la fila central de la FPGA, en imagen especular como muestra la figura A.1. Además la dirección del frame contiene información acerca del tipo de bloque, y la fila y la columna donde empieza el frame.

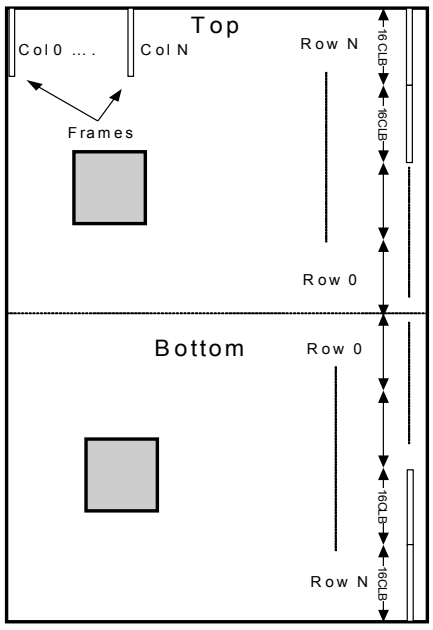

FIGURA A.1: Numeración de filas y columnas en la Virtex-4

Existen dos registros de configuración, uno donde se escribe la dirección del frame (FAR = frame address register) y otro donde se escribe el contenido del frame (FDRI = frame data register input).

La reconfiguración parcial dinámica se realiza a través de un único interfaz, el ICAP (Internal Configuration Access Port) que en la actualidad supone el

cuello de botella de estos sistemas.

A.2. Técnicas de diseño modular

Inicialmente Xilinx propuso dos técnicas para la reconfiguración parcial dinámica [Xil03], una basada en módulos (generalmente llamado diseño modular) y otra basada en diferencias. El método que ha sido utilizado por la mayoría de los investigadores es el diseño modular, y es el que vamos a explicar a continuación porque es el modelo de reconfiguración parcial dinámica en el que se apoya nuestro trabajo.

El método basado en diferencias requiere del diseñador una minuciosa tarea de revisión manual y no es por tanto ni popular ni adecuado para aplicaciones de cierta complejidad.

La reconfiguración modular divide el área de la FGPA en zonas o áreas que pueden ser estáticas o dinámicas. Las áreas estáticas se reconfiguran una sola vez para realizar una determinada tarea (en nuestro caso, partes del S.O) y las áreas dinámicas son las que van a ser reconfiguradas con frecuencia (en nuestro caso las particiones para ejecución de tareas).

Los recursos de las tareas que se van a ejecutar en un área no pueden utilizar recursos de otras áreas, por ejemplo para rutado de señales. Esto permite que se pueda realizar la multitarea hardware ya que asegura que el área destinada a ejecutar una tarea (partición) estará libre por completo para dicha tarea (lo cual a su vez garantiza la reubicabilidad de tareas) y además al reconfigurar esa parte de la FPGA no interfiere con el funcionamiento del resto.

La reserva de un área de la FPGA para uso dinámico es fácil de realizar y

basta con asignar un determinado valor a una variable:

$$Routing - in - Dynamic - area = 0$$

La comunicación entre las diferentes áreas se realiza a través de *bus macros*, situadas en los bordes de las áreas definidas. Algunos autores han hecho interesantes propuestas para implementar buses en el chip capaces de interconectar las áreas entre sí y con los pines de entrada / salida [BA05], [BMG06].

Las técnicas iniciales propuestas por Xilinx se basaban en la reconfiguración por columnas y esto únicamente permitía la reconfiguración en 1D.

Algunos autores han publicado ingeniosos trabajos en los cuales presentaban la posibilidad de realizar reconfiguración parcial dinámica en 2D con las Virtex-2, como es el caso de [HSKB06] y otros han incluido ya en sus publicaciones la extensión de la técnica de reconfiguración modular para FPGAs reconfigurables en 2D como la Virtex-4 [SBB+06].

Xilinx "tomó nota" de las ideas propuestas por los autores mencionados en el párrafo anterior y las integró en su primera propuesta de modelo de reconfiguración parcial dinámica, dando nacimiento en marzo de 2006 a su reciente flujo de diseño EAPR (Early Access Partial Reconfiguration), [EAP06], y que presentaron en una ponencia invitada en el FPL'06, [LBM+06]. Liberado ya de las rígidas restricciones de reconfiguración de columnas completas, el EAPR permite la reconfiguración de módulos rectangulares de tamaños arbitrarios. Asimismo Xilinx ha introducido cores de bus macros pre-rutados, gracias a los cuales es posible permitir el paso de señales de rutado de áreas estáticas a través de las áreas dinámicas.

A.3. Tareas reubicables

Explicaremos ahora cómo se pueden generar tareas reubicables para las FPGAs de Xilinx de última generación: Virtex-4 y Virtex-5.

La herramienta de desarrollo ISE de Xilinx permite generar un mapa de bits parcial, es decir, la asignación de *slices* y rutado necesarios para realizar una determinada tarea, constreñidos a un área rectangular (por medio de los archivos *.ucf) y de ubicación concreta dentro del dispositivo. La ubicación en el dispositivo viene determinada por una dirección de *frame*, que marca la posición del primer *frame* a reconfigurar dentro del dispositivo.

Una vez se conoce la ubicación definitiva de la tarea, tenemos que calcular la nueva dirección de *frame*. Esto supone calcular la nueva columna donde se ubicará la tarea. En definitiva, la realización de una suma del *frame* inicial con una cantidad fija, que será el número de columnas que separan los accesos al bus entre dos particiones contiguas, como se muestra en la figura A.2.

Si la tarea fue inicialmente compilada para ejecutarse en la partición P0 y se asigna la partición P1, será necesario sumarle la cantidad $offset$:

$$nueva = base + offset \tag{A.1}$$

Sin embargo, si la tarea ha sido compilada inicialmente para ejecutarse en P0 pero se le asigna la partición P2 nos encontramos con dos problemas:

- **No basta con sumar un offset de fila:** ya que las particiones P2 y P3 se encuentran en la mitad superior de la FPGA y la numeración de filas es simétrica a partir de la línea central que separa las dos mitades

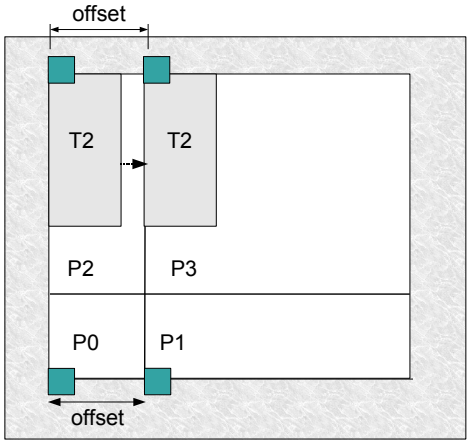

FIGURA A.2: Reubicación de tarea

de la FPGA.

- **La posición de los accesos al bus ya no es correcta:** puesto que no tendrá acceso a las bus macros tal como están situadas en las particiones P2 y P3.

La solución para este problema consiste en la generación de dos mapas de bits diferentes para cada tarea, según la mitad de la FPGA en que se vayan a ejecutar, de forma que se opte por la versión de compilación *inferior* para las particiones P0 y P1 y la versión *superior* para las particiones P2 y P3, según se muestra en la figura A.3.

El coste de esta opción supone un aumento en la memoria necesaria para almacenar los mapas de bits de las tareas.

Como es frecuente en los sistemas de computación, nos encontramos ante el dilema tiempo vs. memoria. En este caso consideramos que los perjuicios

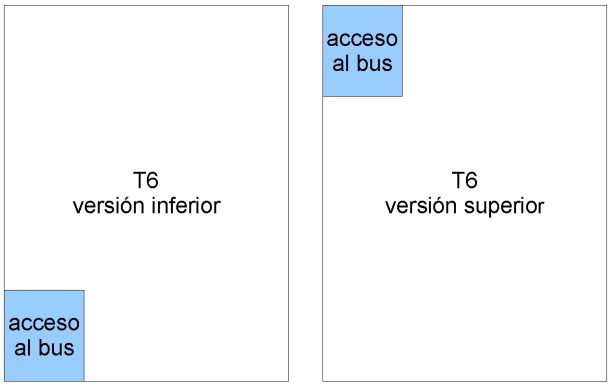

FIGURA A.3: Dos compilaciones de la misma tarea

derivados de reconfigurar el bus de comunicaciones superan a los de un mayor uso de memoria, elemento que en la actualidad no representa un sobre-coste excesivo en ningún sistema.

El problema del acceso al bus se podría resolver también cambiando la ubicación de la línea de bus a la que acceden P2 y P3 según la figura A.4. El bus dejaría de ser periférico y se convertiría en dos líneas paralelas unidas entre sí.

Esta opción deja dos bloques de particiones separadas, lo que limita en gran medida la capacidad de adaptación a diferentes circunstancias, que requieren la unión de particiones o la redistribución del área de ejecución en particiones de diferente tamaño y/o número, obligando en estos casos a reconfigurar el bus.

Consideramos que dado que uno de los cuellos de botella de este tipo de sistemas en la actualidad es el tiempo de reconfiguración, esta opción por el momento debe descartarse.

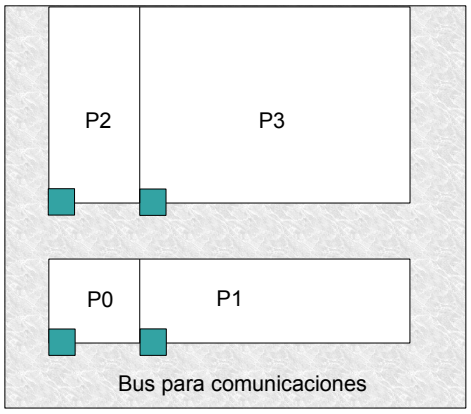

FIGURA A.4: Posible modificación de la ubicación del bus

A.4. Bus de comunicaciones

Explicaremos ahora el modelo de bus de comunicaciones en el que se basa nuestro trabajo.

Es evidente que resulta imprescindible algún tipo de bus que permita la comunicación entre tareas y de las tareas con los pines de entrada/ salida de la FPGA.

El bus de comunicaciones consta de los siguientes elementos:

- **Líneas de datos:** se pueden utilizar las líneas largas de la FPGA
- **Bus macros:** configuradas en posiciones fijas, adyacentes a las particiones, según muestra la figura A.5
- **Conexión a los pines de e/s:** la asignación de pines de e/s se hace de acuerdo con el tamaño de las particiones y su ubicación en la FPGA.

Para la implementación de un bus de comunicaciones efectivo se ha utilizado ya en [GMR+08] un bus tipo Wishbone, en un sistema de características parecidas al propuesto en esta tesis, por su popularidad y frecuente uso para conectar cores IP en un chip, independientemente de la tecnología utilizada (FPGA, ASIC etc.). Una descripción detallada de dicho bus se puede encontrar en [Her02].

La figura A.5 muestra el modelo de FPGA utilizado hasta ahora junto al modelo de bus propuesto.

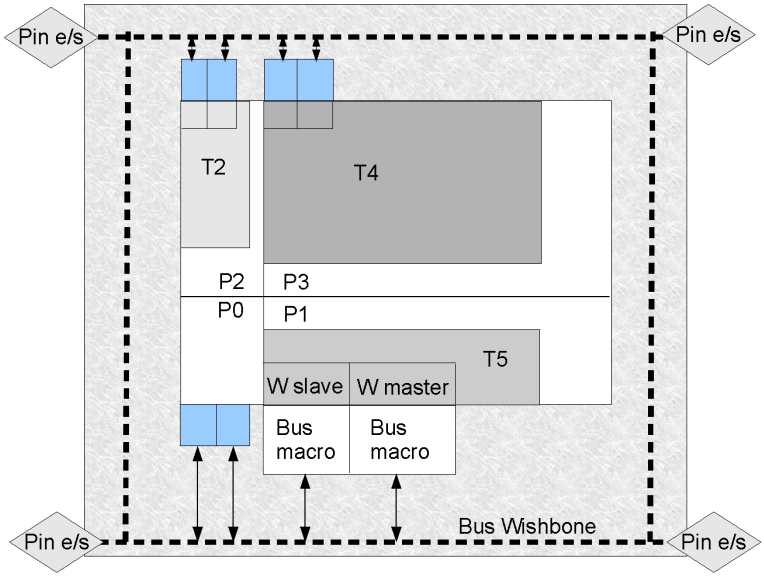

FIGURA A.5: Modelo de bus Wishbone para nuestro sistema

A.4.1. Resumen bus Wishbone

El bus Wishbone tiene una arquitectura maestro / esclavo y soporta conexiones punto a punto, bus compartido, *crossabr switch* y conexión orientada a flujo de datos.

Cada tarea está identificada por una etiqueta. Existe una tabla en un espacio de memoria compartida que contiene la relación entre direcciones de las tareas (la dirección de memoria donde está almacenado su mapa de bits), los puertos de e/s que están asignados a dicha tarea y su etiqueta identificativa.

Las tareas y módulos de SO disponen de un interfaz maestro / esclavo, mientras que los periféricos pueden disponer de maestro, esclavo o ambos, según sean sus necesidades. El esquema del bus, adaptado a nuestro entorno, puede verse en la figura A.6.

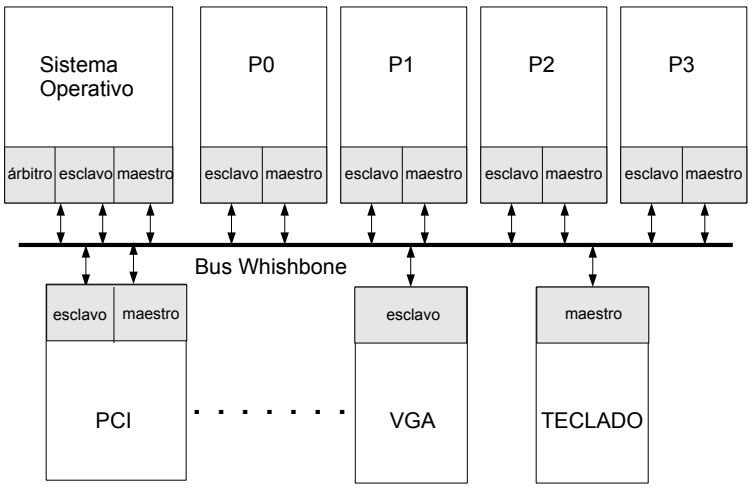

FIGURA A.6: Esquema del bus Wishbone

Los interfaces tipo master son los únicos que pueden iniciar una transacción mientras que los tipo slave sirven para recibir datos. El hecho de que cada tarea disponga de ambos interfaces supone que pueden comunicarse entre sí sin necesidad de que intervenga el S.O, lo cual resulta muy eficiente.

Cada mapa de bits de las tareas debe, por tanto, disponer de un área dedicada al Wishbone slave y otra al Wishbone master, en la zona que corresponda

a las bus macros definidas en la FPGA. Estos módulos son idénticos para cada tarea y solamente hay que mezclar el mapa de bits de la tarea en sí con estos módulos fijos para generar los mapas de bits de las tareas con acceso al bus.

Apéndice B

Descripción de la implementación del prototipo

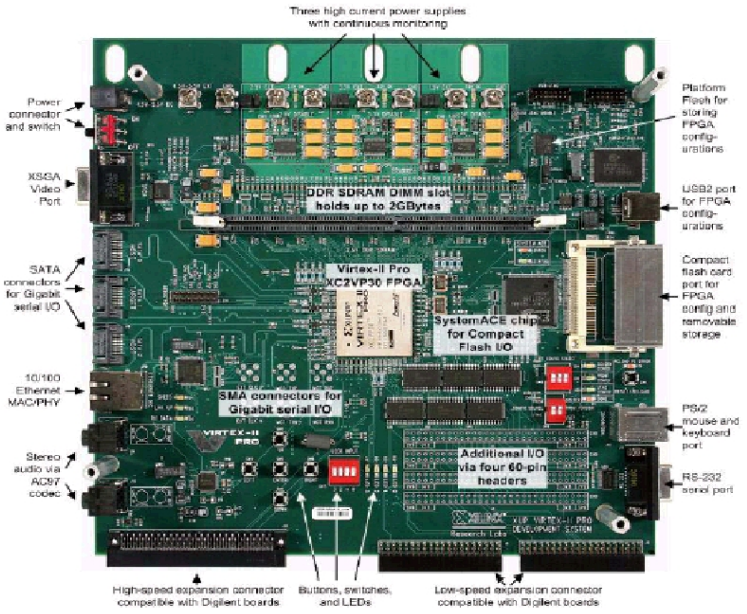

FIGURA B.1: Fotografía de la placa XUP (Xilinx)

B.1. Descripción del entorno y herramientas

B.1.1. Descripción del hardware utilizado

El prototipo se ha implementado sobre una placa de Xilinx XUPV2P (Xilinx University Virtex-II Pro), de la que mencionamos los siguientes elementos utilizados en el prototipo:

- **FPGA XCV2P30:** La Virtex-II Pro es una FPGA que además de los bloques lógicos e interconexiones programables que caracterizan a toda FPGA, dispone también de dos procesadores PowerPC 405 empotrados junto a la lógica de la FPGA, multiplicadores de 18x18 bits y varios módulos de memoria RAM distribuidos por columnas, además de un DCM (digital clock manager).

  Soporta reconfiguración parcial dinámica. Aunque su organización interna es 2D, la reconfiguración parcial dinámica se realiza en 1D, lo que significa que la mínima unidad programable dinámicamente es una columna de CLBs. No obstante, existen trabajos de investigación como [SBB+06] donde se permite la reconfiguración parcial en 2D.

  Cada CLB se compone de cuatro *slices*, cada uno de los cuales a su vez dispone de dos LUTs de cuatro entradas y dos elementos de memoria (flip-flops) y elementos de lógica y acarreo.

- **Slot para DDR SDRAM:** permite conectar hasta 2GB de memoria.

- **Conectores RS232 DB9 y VGA:** el primero es un puerto serie utilizado para conectar la FPGA con el terminal de vídeo. De esta manera

se puede monitorizar la ejecución del software. El segundo se utiliza para representar la situación de las particiones de la FPGA.

- **Reloj de sistema de 100 MHZ**
- **Controlador System ACE y conector de Compact Flash:** utilizado para cargar la configuración de la FPGA y los bitstreams de las tareas a ejecutar en la DDR RAM.
- **4 LEDs:** conectados a los pines de entrada/salida de la Virtex-II Pro y utilizados para monitorizar la ejecución de la aplicación.
- **5 pulsadores:** conectados también a los pines de e/s de la FPGA, que se utilizan para reiniciar la ejecución de la aplicación.

B.1.2. Herramientas de desarrollo

- **Embedded Development Kit de Xilinx (EDK 8.1):** Herramienta que incorpora *cores* de Xilinx necesarios para utilizar algunos dispositivos del sistema (como los procesadores Power Pc, el controlador de memoria DDR RAM, etc.) al código C que implementa el algoritmo planificador y que se ejecuta en uno de los procesadores del chip. Esta herramienta genera un mapa de bits que será escrito en la memoria Flash y desde ella (a través del chip system ACE) se configurará la FPGA y se cargarán los mapas de bits de las tareas a ejecutar en la memoria DDR RAM, de la cual serán leídas durante la ejecución del algoritmo para su configuración y ejecución en la FPGA.
- **Xilinx Platform Studio (XPS):** Interfaz gráfica del EDK.

- **Impact:** es una herramienta de Xilinx Platform Studio que permite descargar el *bitstream* generado con el EDK en la FPGA. Este *bitstream* contiene la configuración de la FPGA, que incluye todos los dispositivos periféricos necesarios para el correcto funcionamiento del sistema diseñado.

- **Xilinx Microprocessor Debugger (XMD):** herramienta del EDK que se utiliza para introducir el código del algoritmo en la DDR y permite insertar puntos de ruptura en el código para su depuración.

B.1.3. Configuración de la Virtex II Pro

La FPGA Virtex-II Pro se ha configurado con el EDK para disponer de los siguientes elementos:

- **Procesadores:** Un procesador Power Pc empotrado en el dispositivo. El procesador se sitúa en un área fija y determinada por los *cores* de Xilinx. Este dispositivo será el que ejecute el código de la aplicación, cargado en la memoria DDR RAM.

- **Área de ejecución de tareas:** Para poder implementar la reconfiguración parcial dinámica en una Virtex-II Pro es necesario definir de antemano las áreas del dispositivo que serán reconfiguradas dinámicamente, de forma que el resto del hardware pueda ser utilizado para los demás elementos que componen el sistema. En este caso se definen cuatro particiones en el área de la FPGA para ejecución de las tareas. Se trata de particiones no contiguas debido a la restricción de área disponible

(por la presencia del Power Pc) y que se extienden a toda la altura del dispositivo debido a la restricción de reconfiguración por columnas. Los anchos de las columnas son 4, 8, 16 y 24, como muestra la figura B.2.

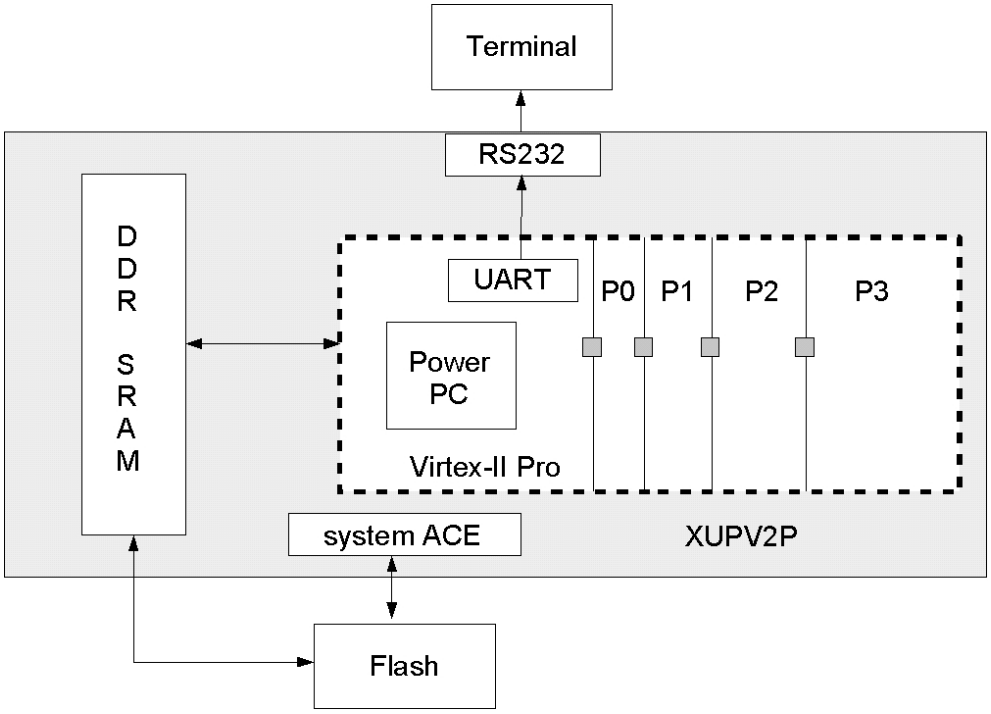

FIGURA B.2: Esquema del prototipo

- **UART:** dispositivo que se implementa en el propio hardware de la FPGA y se utiliza para comunicaciones asíncronas serie de datos entre la FPGA y el terminal.

- **LEDS:** se incorporan dispositivos que controlan el encendido de los LEDs de la placa, para monitorizar el funcionamiento del sistema.

- **Pulsadores:** controlan los botones de que dispone la placa, para reiniciar la ejecución de la aplicación.

- **Controlador de DDR RAM:** se incluye también un controlador de memoria DDR RAM para que el procesador pueda acceder al código de la aplicación y ejecutarlo y para que la aplicación pueda acceder a dicha memoria y leer los *bitstreams* de las tareas a ejecutar y los cargue en la FPGA.

La Virtex-II Pro dispone de un único interfaz para reconfiguración llamado ICAP.

B.1.4. Diseño

El algoritmo se ha implementado en lenguaje de programación C y consta de los siguientes archivos, que se corresponden con las unidades funcionales del algoritmo básico, a excepción de xhwicap-cf.c y de xhwicap-parse.c que son proporcionados por Xilinx para el adecuado manejo de los *bitstreams*. Omitimos los archivos **.h* por no considerarlos relevantes a la exposición de la implementación del algoritmo.

- **main.c:** Este archivo contiene el cuerpo principal del programa y las inicializaciones de las variables.

- **planificador.c:** Este módulo examina las tareas que llegan al sistema y les busca una partición en la que quepan (en este caso solamente tiene que comparar el ancho de la tarea con el ancho de las particiones, ya que todas ocuparán el espacio correspondiente a columnas completas en la FPGA). Les asigna un tiempo de reconfiguración, ya que solamente una tarea puede ser reconfigurada a la vez y además es necesario asegurarse

de que la partición está vacía y de que la tarea anterior ha terminado de reconfigurarse. Una vez hechas las comprobaciones, les asigna una partición o bien las rechaza.

- **lanzador.c:** Este módulo examina las colas para ver si hay tareas planificadas que aún no han sido reconfiguradas. En caso de haberlas, busca la siguiente planificada para reconfigurar y lee el *bitstream* de la memoria DDR RAM y a través de una rutina cuyos parámetros son el puntero a la posición de memoria donde está almacenado el bitstream y el tamaño del bitstream, da la orden al ICAP para reconfigurarla. En este módulo se utilizan otras rutinas proporcionadas por Xilinx e incluidas en los ficheros antes mencionados.

  En el momento en que la ubicación de una tarea se ha seleccionado, se genera su mapa de bits absoluto a partir del código reubicable de la tarea y se carga en la FPGA, según el procedimiento explicado en [GMR+08].

- **xhwicap-cf.c y xhwicap-parse.c:** Contienen la rutina que carga un bitstream parcial desde la Compact Flash a DDR RAM y devuelve el tamaño del bitstream en palabras de 32 bits: $intXHwIcap_CF2Mem(Xuint8* filename, Xuint32*baseaddr)$ y otras funciones proporcionadas por Xilinx para manejo del ICAP.

Bibliografía

[ABF+07] A. Ahmadinia, C. Bobda, S. Fekete, J. Teich, and J. Van der Veen. Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices. *IEEE Trans. Comput.*, 56(5):673–680, 2007.

[ABK+04] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich. Task Scheduling for Heterogeneous Reconfigurable Computers. In *Proceedings of the 17th symposium on Integrated circuits and system design*, pages 22–27, 2004.

[ABT04] A. Ahmadinia, C. Bobda, and J. Teich. A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. In *Proceedings of ARCS'04*, pages 125–139, 2004.

[BA05] C. Bobda and A. Ahmadinia. Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Devices. *IEEE Design and Test*, 22(5):443–451, 2005.

[BD01] G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. In *Proceedings of the 11th International Conference*

on Field-Programmable Logic and Applications, pages 182–191, London, UK, 2001. Springer-Verlag.

[BdNSSL06] P. S. Brandao do Nascimiento, M. Stelita, J. L. Seixas, and M. E. Lima. Mapping of Image Processing Systems to FPGA Computers Based on Temporal Partitioning and Design Space Exploration. In *Proceedings of SBCCI'06*, pages 50–55, 2006.

[BKS00] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design and Test - Special Issue on Reconfigurable Computing, January-March 2000*, 17(1):68–83, 2000.

[BMG06] C. Bieser and K. D. Müller-Glaser. Rapid Prototyping Design Acceleration Using a Novel Merging Methodology for Partial Configuration Streams of Xilinx Virtex-II FPGAs. In *Proceedings of IEEE International Workshop on Rapid System Prototyping*, pages 193–199, 2006.

[BS99] K. Barzagan and M. Sarrafzadeh. Fast Online Placement for Reconfigurable Computing. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 300, Washington, DC, USA, 1999. IEEE Computer Society.

[BTAS07] S. Bhunia, M. Tabib-Azar, and D. Saab. Ultralow-Power Reconfigurable Computing with Complementary Nano-

Electromechanical Carbon Nanotube Switches. In *Proceedings of ASP-DAC*, pages 86–91, 2007.

[CG05] C. Chit and M. Glesner. An FPGA Implementation of the AES-Rijndael in Previous OCB/ECB Modes of Operation. *Microelectronics Journal*, 36(2):139–146, 2005.

[CH02] K. Compton and S. Hauck. Reconfigurable Computing: a Survey of Systems and Software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[DE01] O. Diessel and H. ElGindy. On Dynamic Task Scheduling for FPGA-based Systems. *International Journal of Foundations of Computer Science*, 12(5):645–669, 2001.

[DEM+00] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. *IEE Computer and Digital Techniques, 2000.*, 147(3):181–188, 2000.

[Die98] O. Diessel. *On Scheduling Dynamic FPGA Reconfigurations - a Partial Rearrangement Approach.* PhD thesis, Dpt. of Computer Science and Software Engineering, University of Newcastle, Australia, 1998.

[DP05] K. Danne and M. Platzner. A Heuristic Approach to Schedule Periodic Real-Time Tasks on Reconfigurable Hardware. In *Proceedings of the International Conference on Field Programmable*

Logic and Applications (FPL05), pages 568–573, 24- 26 August 2005.

[DW99] O. Diessel and G. Wigley. Opportunities for Operating Systems Research in Reconfigurable Computing. In *Technical report ACRC-99-018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia*, 1999.

[DWM+05] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon. Demystifying 3D ICs: the Pros and Cons of Going Vertical. *Design and Test of Computers, IEEE*, 22(6):498–510, Nov.-Dec. 2005.

[EAP06] Xilinx: Early Access Partial Reconfiguration User Guide for ISE 8.1.01 i, 2006.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, EC-12(6):747–755, 1963.

[Est02] G. Estrin. Reconfigurable Computer Origins: the UCLA Fixed-plus-variable F+V Structure Computer. *IEEE Ann. Hist. Comput.*, 24(4):3–9, 2002.

[GMR+08] A. González, H. Mecha, S. Román, D. Mozos, and J. Septién. Synthesis of Relocatable Tasks and Implementation of a Task Communication Bus in a General Purpose HW System. In *Proceedings of ERSA*, pages 307–308, 2008.

[GP02] J. Geoffrey and C. Pretty. Software Defined QCIF Simple Profile MPEG-4 for Portable Devices using Dynamically Reconfigurable DSP. *Comput. Stand. Interfaces*, 24(5):453–472, 2002.

[Har01] R. Hartenstein. Coarse Grain Reconfigurable Architecture (Embedded Tutorial). In *Proceedings of ASP-DAC '01*, pages 564–570, New York, NY, USA, 2001. ACM.

[Hau98] S. Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–638, Apr 1998.

[Her02] R. Herveille. Whishbone B.3 Revision Specification; http://www.opencores.org/projects.cgi/web/wishbone/wishbone, 2002.

[HSKB06] M. Hubner, C. Schuck, M. Kuhnle, and J. Becker. New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Microelectronic Circuits. In *Proceedings of the IEEE ISVLSI*, page 97, 2006.

[HV04] M. Handa and R. Vemuri. An Efficient Algorithm for Finding Empty Space for Online FPGA Placement. In *Proceedings of DAC'04*, pages 960–965, June 2004.

[JTR+05] M. Janiaut, C. Tanougast, H. Rabah, Y. Berviller, C. Mannino, and S. Weber. Configurable Hardware Implementation of a Conceptual Decoder for a Real-time MPEG-2 Analysis. In *Proceedings of FPL*, pages 386–390, 2005.

[KCMO06] A. Kinane, A. Casey, V. Muresan, and N. O'Connor. FPGA-based Conformance Testing and System Prototyping of an MPEG-4 SA-DCT Hardware Accelerator. *Proceedings of FPL*, pages 317–318, 2006.

[KD06] S. Koh and O. Diessel. CommA: A Communications Methodology for Dynamic Module Reconfiguration in FPGAs. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 273–274. IEEE Computer Society, 2006.

[KKK+04] H. Kalte, M. Koester, B. Kettelhoit, M. Porrmann, and U. Rückert. A Comparative Study on System Approaches for Partially Reconfigurable Architectures. In *Proceedings of ERSA'04*, pages 70–76, 2004.

[KKP06] M. Koester, H. Kalte, and M. Porrmann. Relocation and Defragmentation for Heterogeneous Reconfigurable Systems. In *Proceedings of ERSA'06*, pages 70–76, 2006.

[KPK05] M. Koester, M. Porrmann, and H. Kalte. Task Placement for Heterogeneous Reconfigurable Architectures. In *Proceedings of FPT'05*, pages 43–50, 2005.

[KTR08] I. Kuon, R. Tessier, and J. Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[LBM+06] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of Xilinx FPGAs. In *Proceedings of FPL*, pages 1–6, Aug. 2006.

[LCH00] Z. Li, K. Compton, and S. Hauck. Configuration Caching Management Techniques for Reconfigurable Computing. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 22–36, 2000.

[Leo08] P. H. W. Leong. Recent Trends in FPGA Architectures and Applications. In *Proceedings of DELTA 2008*, pages 137–141, Jan. 2008.

[LH01] Z. Li and S. Hauck. Configuration Compression for Virtex FPGAs. In *Proceedings of IEEE Symposium on FCCM '01*, pages 147–159, 2001.

[MD96] E. Mirsky and A. Dehon. Matrix: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, 1996.

[Mei03] B. Mei. Exploiting Loop-Level Parallelism on Coarse Grained Reconfigurable Architectures using Modulo Scheduling. In *Proceedings of DATE '03*, page 10296, 2003.

[MLJ98a] P. Merino, J. C. López, and M. Jacome. Designing Dynamically Reconfigurable Systems: a High Level Approach. In *Proceedings of DCIS '98*, pages 458–463, 1998.

[MLJ98b] P. Merino, J. C. López, and M. Jacome. A Hardware Operating System for Dynamic Reconfiguration of FPGAs. In *Proceedings of FPL*, pages 431–435, 1998.

[MM06] H. Masanori and K. Michitaka. A Multi-Context FPGA Using a Floating-Gate-MOS Functional Pass-Gate and its CAD Environment. In *Proceedings of APCCAS*, pages 1803–1806, Dec. 2006.

[MTAB07] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. *VLSI Signal Processing Systems*, 47(1):15–31, 2007.

[MVVL02] J. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins. Enabling Hardware-software Multitasking on a Reconfigurable Computing Platform for Networked Portable Multimedia Appliances. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, 2002.

[NB04] J.Ñoguera and R. M. Badía. Power-Performance Trade-Offs for Reconfigurable Computing. In *Proceedings of CODES+ISSS '04*, pages 116–121, 2004.

[NHCB02] A.Ñayak, M. Haldar, A.Ñ. Choudhary, and P. Banerjee. Accurate Area and Delay Estimators for FPGAs. In *Proceedings of DATE*, pages 862–869, 2002.

[OPF06] J. Oliveira, A. Printes, and R.C.S Freire. FPGA Architecture for Static Background Subtraction in Real-time. In *Proceedings of SBCCI*, pages 26–31, 2006.

[Poz06] D. S. Poznanovic. The Emergence of Non-von Neumann Processors. In *Reconfigurable Computing: Architectures and Applications*, pages 243–254. SpringerLink, 2006.

[PRMC07] E. Pérez, J. Resano, D. Mozos, and F. Catthoor. Memory Hierarchy for High-performance and Energyaware Reconfigurable Systems. *Computers and Digital Techniques, IET*, 1(5):565–571, Sept. 2007.

[RF78] S. S Reddi and E. A. Feustel. A Restructurable Computer System. *IEEE Trans. Comput.*, 27(1):1–20, 1978.

[RMC05] J. Resano, D. Mozos, and F. Catthoor. A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-Time the Reconfiguration Overhead of Dynamically Reconfigurable Hardware. In *Proceedings of DATE '05*, pages 106–111, Washington, DC, USA, 2005. IEEE Computer Society.

[RMVC05] J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A Reconfiguration Manager for Dynamically Reconfigurable Hardware. *IEEE Computers Design and Test*, 22(5):452–460, 2005.

[RSÉHB08] F. Rivera, M. Sánchez-Élez, R. Hermida, and N. Bagherzadeh. Scheduling Methodology for Conditional Execution of Kernels

onto Multicontext Reconfigurable Architectures. *Computers and Digital Techniques IET*, 2(3):199–213, May 2008.

[SBB+06] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular Dynamic Reconfiguration in Virtex FPGAs. *IEE Computers and Digital Techniques*, 153(3):157–164, 2006.

[SC01] B. Salefski and L. Caglar. Re-Configurable Computing in Wireless. In *Proceedings of DAC'01*, pages 178–183, 2001.

[SLM00] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA Coprocessors. In *Proceedings of FPL'00*, pages 121–130, 2000.

[SM06] K. Stevens and O. A. Mohamed. Single-chip FPGA Implementation of a Pipelined, Memory-based AES Rijndael Encryption Design. In *Proceedings from Canadian conference on Electrical and Computer Engineering*, pages 1296–1299, 2006.

[SWP04] C. Steiger, H. Walder, and M. Plaztner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004.

[TSMM06] J. Tabero, J. Septién, H. Mecha, and D. Mozos. Task Placement Heuristic Based on 3D-adjacency and Look-ahead in Reconfigurable Systems. In *Proceedings of ASP-DAC*, pages 396–401, 2006.

[TSMM08] J. Tabero, J. Septién, H. Mecha, and D. Mozos. Allocation Heuristics and Defragmentation Measures for Reconfigurable Systems Management. *Integration*, 41(2):281–296, 2008.

[Und04] K. Underwood. FPGAs vs CPUs: Trends and Peak Floating Performance. In *Proc. of the Twelfth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 171–180, 2004.

[ver08] IEEE Standard Verilog Hardware Description Language, 2008. http://www.verilog.com/IEEEVerilog.html.

[vhd06] IEEE VHDL Analysis and Standarization Group, 2006. http://www.vhdl.org/vasg.

[WK02] G. Wigley and D. Kearney. The Management of Applications for Reconfigurable Computing using an Operating System. In *Proceedings of CRPIT '02*, pages 73–81, 2002.

[WK07] M. Watanabe and F. Kobayashi. A 0.35 um CMOS 1,632-Gate-Count Zero-Overhead Dynamic Optically Reconfigurable Gate Array VLSI. In *Proceedings of ASP-DAC '07*, pages 124–125, Washington, DC, USA, 2007. IEEE Computer Society.

[WP02] H. Walder and M. Platzner. Non-Preemptive Multitasking on FPGAs: Task Placement and Footprint Transform. In *Proceedings of ERSA'02*, pages 24–30, 2002.

[WP03] H. Walder and M. Platzner. Online Scheduling for Block-Partitioned Reconfigurable Devices. In *Proceedings of DATE '03*, pages 10290–10295, 2003.

[WSP03] H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proceedings of IPDPS '03*, page 178.2, 2003.

[Xil03] Xilinx Corp. Two Flows for Partial Reconfiguration: Module Based or Difference Based; Application Note XAPP290, Version 1.1, 2003. http://www.xilinx.com/support/documentation/-application_notes/xapp290.pdf.

[Xil09] Xilinx: our history http://www.xilinx.com/company/history.htm, 2009.

Índice de figuras

Índice de tablas