Infraestructuras de Barrera Eficientes para Sistemas Clusterizados MPSoC

José L. Abellán, Juan Fernández y Manuel E. Acacio[1]

Resumen— **La sincronización mediante barrera es una primitiva de programación clave para sistemas que implementan un modelo de programación de memoria compartida como los MPSoCs. Conforme aumenta el número de núcleos en estos sistemas, las implementaciones software no son capaces de proporcionar el rendimiento requerido haciendo imprescindible la aceleración hardware. En este artículo proponemos distintos esquemas de barrera para MPSoCs clusterizados haciendo uso de tecnología estándar y un *toolflow* industrial actual. Nuestros diseños presentan una sobrecarga en área mínima y un enorme potencial de rendimiento que es cuantificado utilizando un simulador para MPSoCs con soporte para OpenMP, comparándolo frente a la mejor implementación de barrera software.**

Palabras clave— **MPSoC Clusterizados, Barreras Hardware, Tecnología Estándar, OpenMP.**

I. Introducción y Trabajo Relacionado

La sincronización eficiente mediante barrera constituye un reto a medida que el número de núcleos de un sistema multi-procesador en un sólo chip (MPSoC) aumenta. Es por ello que ya no hay duda de que las implementaciones software no son capaces de proporcionar la escalabilidad requerida para estos sistemas y que el soporte hardware se torna esencial.

Las optimizaciones hardware para barrera a menudo consisten en controladores de memoria o de comunicación mejorados [3], que intentan reducir la sobrecarga del proceso de espera activa en los algoritmos de sincronización. Para ello, se centran en reducir la congestión a nivel de memoria y/o de la red de interconexión realizando la espera activa localmente sobre registros internos a cada núcleo. Sin embargo, son necesarios mensajes de sincronización que fluyen a través de la red principal de interconexión del sistema, típicamente una *Network-on-Chip* (NoC) [4], no siendo por tanto una solución óptima. Primero, el tráfico de datos de las aplicaciones y el de sincronización son muy diferentes, siendo por tanto muy complicado el idear una topología NoC que satisfaga ambos de manera eficiente. Por otro lado, la interferencia mutua entre ambos tipos de tráfico puede degradar el rendimiento de las aplicaciones y también enlentecer el proceso de sincronización.

Hay pocas soluciones para MPSoCs que proponen la adopción de infraestructuras de comunicación dedicadas para conducir el tráfico relacionado con la sincronización [11], [10]. Propuestas actuales [6], [1] hacen uso de tecnología de implementación no estándar para conseguir eficiencia y escalabilidad a medida que el número de núcleos aumenta. En este artículo por contra, haciendo uso de tecnología estándar exploramos diferentes infraestructuras que se basan en patrones de conectividad muy simples y protocolos de sincronización de barrera extremadamente rápidos. Sin embargo, la eficiencia y escalabilidad no son fáciles de conseguir cuando se usa este tipo de tecnología. Primero, la latencia de los enlaces de un chip aumenta conforme mejora la escala de integración [5]. Segundo, lo anterior también afecta negativamente a la frecuencia máxima de los controladores requeridos por los esquemas de barrera, haciendo los diseños más lentos.

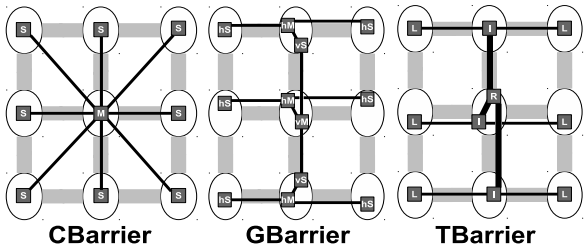

Fig. 1. Barreras Intra-clúster para un clúster de 9 núcleos.

Por otro lado, la mayoría de las plataformas MPSoC son escalables mediante clusterización y replicación de clústeres [7], [8], donde cada clúster puede potencialmente operar a una frecuencia distinta por motivos de eficiencia energética. Estas arquitecturas también serán consideradas en este artículo y añaden dos nuevas variables al espacio de diseño. Primero, la implementación hardware más eficiente a nivel de clúster podría no ser la misma entre clústeres donde se efectúa la sincronización global. Segundo, la transmisión de mensajes de sincronización en este último nivel es una comunicación globalmente asíncrona localmente síncrona (GALS) que no ha sido adecuadamente investigada antes.

Este trabajo presenta los resultados más importantes obtenidos en [2].

II. *Soporte hardware para Barreras*

Para el desarrollo de nuestras infraestructuras de barrera hacemos uso de tecnología estándar y de un *toolflow* industrial actual. Para minimizar las pérdidas de rendimiento antes descritas por el uso de esta tecnología, nuestros diseños han sido ideados utilizando patrones de conectividad muy simples y con mínimos requerimientos de ancho de banda para los enlaces o *Links* (1 bit de anchura en la mayoría de los casos). Así, las infraestructuras de barrera estudiadas a niveles intra- e inter-clúster son: CBarrier, GBarrier y la TBarrier.

[1]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mails: {jl.abellan,juanf,meacacio}@ditec.um.es

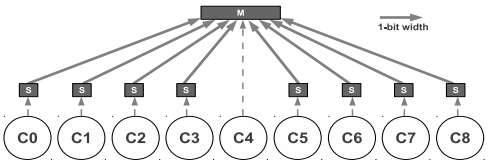

Fig. 2. Fase *gather* para CBarrier.

A. Barreras Intra-Clúster

Todas las barreras propuestas intra-clúster están basadas en el típico protocolo de dos fases *maestro-esclavo* [4]: la fase *gather* y la fase *release*. Durante la fase *gather*, un hilo denominado maestro espera a que todos los hilos denominados esclavos lleguen a la barrera utilizando para ello *flags* de estado. Éstos esperarán a que el maestro recopile todas las señales de todos los esclavos, en cuyo caso entra en ejecución la fase *release* completando la sincronización. Para ilustrar nuestras barreras hardware consideramos un clúster compuesto de 9 núcleos interconectados a través de una topología 2D-mesh.

A.1 La Arquitectura CBarrier

La CBarrier se muestra en la Fig. 1. Los *Links* son representados con líneas finas negras mientras que sus controladores se representan mediante cajas grises. Hay dos tipos de controladores: maestro (M) y esclavo (S). Este esquema se caracteriza por tener el mínimo número de etapas de sincronización, una propiedad deseable para la aceleración hardware.

El protocolo de CBarrier se basa en el intercambio de mensajes de un bit (señales) entre controladores maestro y esclavo. La Fig. 2 representa la fase *gather* cuando todos los núcleos alcanzan la barrera al mismo tiempo. La comunicación entre controladores se muestra con líneas sólidas, mientras que las escrituras locales se muestran con líneas punteadas. La fase *release* es exactamente la misma pero las notificaciones fluyen en el sentido opuesto.

A.2 La Arquitectura GBarrier

La GBarrier [1], mostrada en la Fig. 1, se ajusta perfectamente a la topología 2D-mesh asumida en este trabajo. Como se puede ver hay cuatro tipos de controladores: maestros o esclavos horizontales (prefijo h en la figura) o verticales (prefijo v). En este trabajo se ha implementado este esquema mediante tecnología estándar, en lugar de tecnología de *G-Lines* como en [1]. Para ello, las *G-Lines* se implementan como enlaces RC convencionales y reservando una línea diferente por cada controlador esclavo. Esto difiere de la implementación [1], donde las *G-Lines* son compartidas por todos los esclavos conectados a un mismo maestro. Además, simulamos la técnica *S-CSMA* utilizada en [1], que permite a un controlador maestro determinar el número de señales desde los esclavos emitidas de manera simultánea sobre una *G-Line*, programando el maestro para que muestree las señales de todos los enlaces de sus esclavos hasta que todas las señales esperadas sean recibidas. El protocolo de sincronización para GBarrier se ilustra en la Fig. 3 para la fase *gather*. La *release* es exactamente la misma pero las señales fluyen en el sentido contrario.

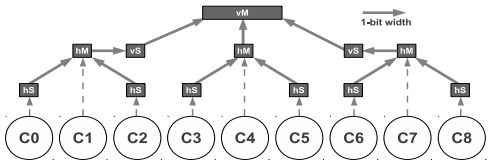

Fig. 3. Fase *gather* para GBarrier.

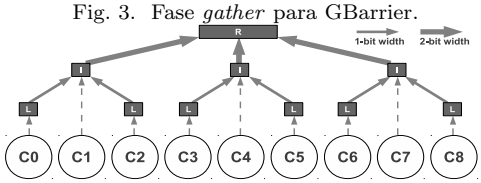

Fig. 4. Fase *gather* para TBarrier.

A.3 La Arquitectura TBarrier

Esta arquitectura, mostrada en la Fig. 1, proporciona la mejor escalabilidad teórica, debido a que necesita el menor número de mensajes intercambiados entre maestro y esclavos. Nuestra implementación utiliza tres tipos de controladores: nodos hoja (L), internos (I) y raíz (R). El protocolo de sincronización para la fase *gather* es ilustrado en la Fig. 4. El controlador R es responsable de contar el número de participantes. En esta fase, los controladores L envían un mensaje de un bit a su correspondiente controlador I. Cuando I ha recibido todos los mensajes esperados notifica al controlador R. Finalmente, R espera mensajes de todos los controladores I. Nótese que el enlace entre los controladores I y R es más ancho que entre el par L e I. En general, estos enlaces serán de anchos $log_2(Leaves)$ bits (e.g. 2 bits para el ejemplo de la figura) para soportar el tamaño de los mensajes necesarios para la fase *gather*. En la fase *release*, sólo enlaces de anchura un bit notificarán la finalización de la sincronización.

B. Barreras Inter-Clúster

El sistema MPSoC clusterizado abordado en este trabajo soporta velocidades de reloj distintas por cada clúster (ver por ejemplo STM p2012 [8]). La diferencia principal entre sincronización global (inter-clúster) y local (intra-clúster) es que los controladores globales podrían comunicarse utilizando dominios de reloj distintos. De ahí que hayamos diseñado barreras inter-clúster utilizando *Links* globales (*gLinks*). Para evitar meta-estabilidad en el controlador que recibe un mensaje en un dominio de reloj diferente utilizamos sincronizadores *brute-force*.

La Fig. 5 muestra CBarrier adaptada a la sincronización inter-clúster. Para comunicaciones entre maestro y esclavos, utilizamos dos sincronizadores

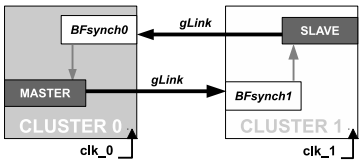

Fig. 5. CBarrier inter-clúster para 2 clústeres.

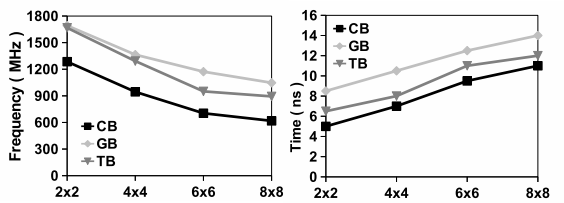

Fig. 6. Frecuencias óptimas y latencias para barreras intra-clúster.

brute-force (*BFsynch0* y *BFsynch1*). Esquemas similares para GBarrier y TBarrier pueden ser implementados simplemente añadiendo estos sincronizadores para cada *gLink*. Es importante apreciar que cada *BFsynch* tiene un bit de ancho excepto para la implementación TBarrier que, como se decía en la Sección II-A.3, podrían requerir hasta $log_2(Leaves)$ bits por *BFsynch*.

III. Evaluación

Como se explicó anteriormente, para implementar los mecanismos de sincronización se ha utilizado tecnología estándar STMicroelectronics 45nm haciendo uso de un *toolflow* industrial actual. Para ello se han utilizado herramientas tales como Synopsys Physical Compiler para la síntesis de los diseños, Cadence SoC Encounter para el proceso final de posicionamiento y rutado, así como también se ha procedido con la sintetización de tantos *clock trees* como dominios de reloj en el sistema (uno por clúster). Finalmente, se ha efectuado un procedimiento *sign-off* mediante Synopsys PrimeTime para validar la latencia de nuestros diseños. Nuestros esquemas han sido diseñados definiendo obstrucciones no rutables para simular el área de cada núcleo ($0.55\times0.55mm^2$). Además, se han definido *fences* para limitar el área donde cada uno de los controladores de barrera son posicionados. Tales obstrucciones y *fences* aseguran rutado de mínima longitud para los enlaces a fin de reducir su impacto en el área y rendimiento a medida que la longitud de éstos aumenta.

A. Barreras Intra-Clúster

Cada controlador de barrera ha sido implementado separando el retardo de las señales en los enlaces del tiempo de computación efectiva que éstos requieren para generar sus señales de salida. Para clústeres pequeños, el camino crítico estaría definido por el controlador de barrera más complejo (e.g. el maestro para CBarrier), pero a medida que la longitud de los enlaces aumenta para clústeres más grandes, los enlaces mismos podrían representar tal camino crítico. Por lo tanto, separar el retardo de los enlaces de los controladores es esencial para conseguir máximas frecuencias de operación. Así, si todos los núcleos llegan a la barrera al mismo tiempo, el tiempo teórico en ciclos de reloj para las barreras intra-clúster sería: 6 para CBarrier; 14 para GBarrier; y 10 para TBarrier.

La Fig. 6 muestra las máximas frecuencias para cada diseño intra-clúster dependiendo del tamaño del clúster. Como se puede observar, el número de etapas determina la frecuencia que se consigue, es

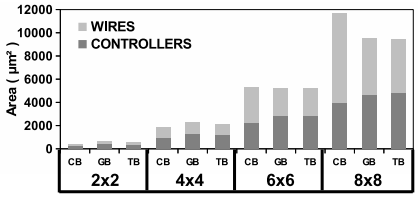

Fig. 7. Sobrecarga en área para barreras intra-clúster a 600MHz.

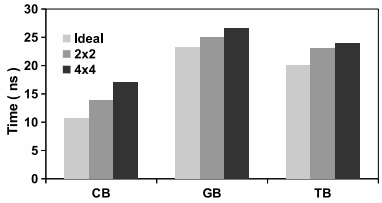

Fig. 8. Latencia para barreras inter-clúster operando a máxima frecuencia.

decir, a mayor número de etapas mayor frecuencia es obtenida. Por esta razón, las frecuenicas más altas son obtenidas por el diseño GBarrier. Además a medida que el tamaño de los clústeres es mayor, el tiempo de los caminos críticos obtenidos para controladores y enlaces será mayor, alcanzando frecuencias más bajas. Respecto a la latencia de las barreras, CBarrier se completa en el menor número de ciclos a pesar de operar a menores frecuencias.

La Fig. 7 ilustra la sobrecarga en área para nuestras barreras intra-clúster sintetizadas a 600MHz. El área dedicada a los enlaces (wires) constituye el factor dominante para los tres mecanismos y por eso CBarrier, que tiene los enlaces más largos, obtiene la mayor sobrecarga, que se empeora al considerar clústeres mayores.

B. Barreras Inter-Clúster

Para evaluar nuestras barreras inter-clúster, consideramos dos plataformas diferentes compuestas de 2×2 y 4×4 clústeres, asumiendo 4×4 núcleos por clúster como en [8]. Como se explicó anteriormente, el protocolo de sincronización para un sistema compuesto de múltiples clústeres/dominios de reloj se divide en dos niveles de sincronización. El primero, implementa la barrera a nivel intra-clúster y el segundo a nivel inter-clúster manejando distintos dominios de reloj. Por lo tanto, la máxima frecuencia en éste último nivel vendrá limitado por la máxima velocidad conseguida en el primer nivel y viceversa. En nuestra experimentación hemos obtenido que, considerando clústeres de 4×4 núcleos y la mejor implementación a nivel intra-clúster (CBarrier), la máxima frecuencia será de 950MHz (ver Fig. 6).

La Fig. 8 muestra la latencia de las barreras para clústeres de 2×2 y 4×4 operando a 950MHz. Estas latencias se corresponden a la operación inter-clúster sin añadir el tiempo intra-clúster. Mostramos también la latencia ideal para los esquemas de barrera operando a la máxima frecuencia sin considerar retardos de propagación de las señales. Como podemos observar, la implementación más eficiente es todavía CBarrier para estas configuraciones. El único caso en

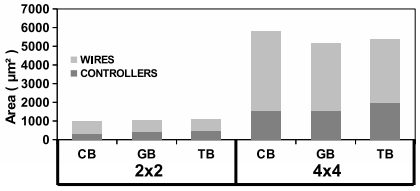

Fig. 9. Sobrecarga en área para las barreras inter-clúster.

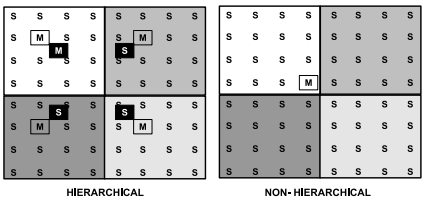

Fig. 10. Barreras clusterizadas basadas en esquema CBarrier.

el que CBarrier no sería la mejor opción es cuando los *gLinks* son tan largos que se anula el beneficio de tener el menor número de etapas de sincronización. Como se espera, el compromiso longitud-retardo de los *gLinks* es más pronunciado para la arquitectura CBarrier cuando se compara frente a las latencias ideales. Sin embargo, esto no es suficiente para que CBarrier sea superado por GBarrier o TBarrier. Utilizando la herramienta sign-off (i.e. Synopsys PrimeTime) hemos calculado que el retardo de los enlaces es función de sus longitudes de este modo: 0'7, 1'2 y 2'2 ns; para 2'2, 4'4 y 8'8 mm, respectivamente. Ya que nuestros diseños no tienen enlaces más largos que 8'8 mm, los valores anteriores explican una mínima penalización en latencia haciendo que el diseño inter-clúster CBarrier sea el más rápido.

La Fig. 9 muestra la sobrecarga en área para los tres diseños (no incluyendo el nivel intra-clúster). Como podemos ver, los *gLinks* confieren la mayor sobrecarga y por eso CBarrier es el más penalizado (tiene los enlaces más largos).

C. Sobrecarga de Clusterización

Hasta ahora, los esquemas de barrera han sido implementados a nivel intra- e inter-clúster que permite la ejecución de múltiples barreras en el sistema de manera simultánea (una por clúster). De hecho, cada uno de los controladores podría ser independientemente programado por software para habilitar distintos dominios de sincronización disjuntos. Ahora estudiamos la sobrecarga de tener una solución no jerárquica llamada *Flat*, en la que sólo una única barrera podría ser ejecutada en todo el sistema pero incluyendo el problema de tener distintos dominios de reloj (uno por clúster). Como caso de estudio, consideramos una plataforma de 64 núcleos dividida en cuatro clústeres de 16 núcleos con cuatro dominios de reloj (tamaño empleado en [8]).

De las Secciones III-A y III-B derivamos que el diseño más eficiente tanto a nivel inter-clúster como a nivel intra-clúster es CBarrier. Éste será el diseño adoptado por ambos esquemas clusterizados (ver *Jerárquico* y *Flat* en Fig. 10). Nótese que las cajas negras representan el primer nivel de los controladores inter-clúster. Como podemos ver en la figura,

TABLA I

Estadísticas de rendimiento para los dos diseños CBarrier.

	Freq.	**Lat.**	**Área (Enlaces & Contr.)**
Jerárquico	950MHz	22ns	$4.935\mu m^2$ & $5.922\mu m^2$
Flat	620MHz	17ns	$6.137\mu m^2$ & $7.977\mu m^2$

comparamos el diseño *Jerárquico* de CBarrier con su homólogo *Flat* que está compuesto de un único nivel en el que todos los controladores son esclavos y se conectan a uno global (el maestro M).

En la Tabla I detallamos los resultados de rendimiento en términos de máxima frecuencia, latencia de barrera y sobrecarga en área para ambos diseños de CBarrier. Como se puede observar, la máxima frecuencia es conseguida por el diseño *Jerárquico* debido al hecho de que para ambos escenarios los controladores maestros constituyen el camino crítico del rendimiento. Así, a mayor complejidad del maestro se conseguirá una frecuencia más baja. Por ello, como en el esquema *Flat* hay sólo un maestro para todos los esclavos del sistema, este diseño obtiene la frecuencia más baja (ver la tabla). Respecto a la latencia, el diseño *Flat* es el más eficiente porque el *Jerárquico* casi le dobla en número de etapas para la sincronización. Finalmente en cuanto a la sobrecarga en área, el esquema *Flat* es el peor diseño porque primero, requiere un mayor número de sincronizadores *brute-force* (uno para cada una de las comunicaciones entre esclavos que traspasan diferentes dominios de reloj), y segundo, la longitud media de todos sus enlaces es mayor que en el diseño *Jerárquico*.

IV. Simulación en Sistema MPSoC

Como experimentación final, calculamos el impacto de integrar nuestras mejores barreras hardware en una pila software real. Para ello, desarrollamos modelos SystemC de los dos componentes principales de nuestras barreras clusterizadas: los controladores locales y globales descritos en la Sección III-C. Anotamos estos modelos con las latencias extraídas de la caracterización presentada en la Tabla I. Los modelos son finalmente integrados en un simulador de sistema completo con precisión a nivel de ciclo que nos permite construir una instancia del sistema clusterizado presentado en la Sección III-C: cuatro clústeres y un total de 64 núcleos (16 por clúster). Todos los clústeres están interconectados a través de una red global NoC. En cada uno de ellos los 16 núcleos se comunican a través de una memoria de datos rápida multi-banco y multi-puerto denominada TCDM (Tightly-Coupled Data Memory). El número de puertos de memoria en la TCDM es igual al número de bancos para permitir accesos a diferentes bancos simultáneamente. Así, accesos libres de conflicto al TCDM se efectúan en sólo dos ciclos.

La pila software seleccionada está basada en el modelo de programación OpenMP. En éste, el paralelismo es especificado a muy alto nivel insertando directivas (anotaciones) a un programa secuencial escrito en lenguage C. Un compilador será el responsable de traducir estas directivas a hilos de ejecu-

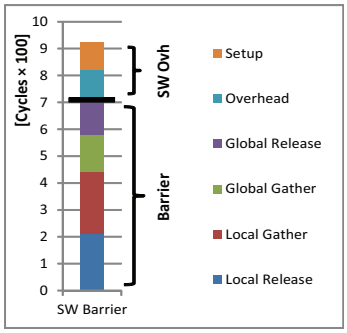

Fig. 11. Coste de la barrera software

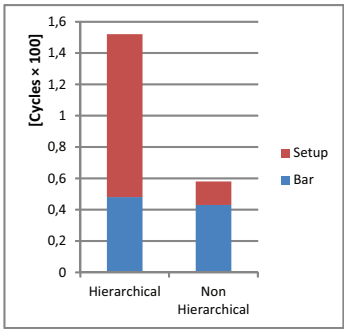

Fig. 12. Coste de las barreras hardware

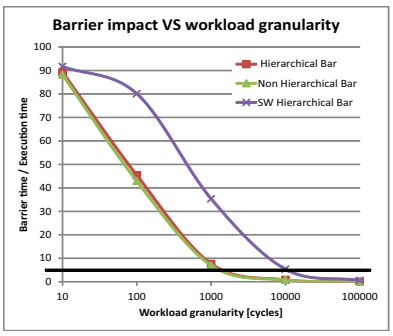

Fig. 13. Coste de Barrera / Workload

ción (hilos) para extraer paralelismo. Muchos de estos servicios de paralelización provistos por OpenMP son implementados dentro de una biblioteca *runtime* que es solicitada por los hilos. Nosotros reescribimos a bajo nivel estas primitivas para que se pueda hacer uso de las barreras de sincronización, de manera que se pueda seleccionar entre una implementación software o bien invocar a nuestros controladores de barreras hardware.

Para el caso de la implementación software, hemos considerado una variante de la barrera en árbol discutida en [9]. En la implementación software de la barrera, un único núcleo en el sistema actúa como el maestro global, un núcleo por clúster actúa como maestro local, mientras que el resto de núcleos actúan como esclavos locales (ver el esquema *Jerárquico* en la Fig. 10). De este modo, el proceso de sincronización para nuestra barrera software consistirá en: una fase *gather* local intra-clúster, en la que el maestro local recibe señales de sus esclavos; una *gather* global inter-clúster en la que el maestro global espera señales de los maestros locales; y finalmente, las correspondientes fases *release* global y local. Además, para prevenir que la actividad de espera activa de los hilos inyecte tráfico a la red de interconexión principal, distribuimos los *flags* de notificación y liberación de manera que cada núcleo efectúe la espera activa en una región de memoria local privada. En particular, nos beneficiamos de la naturaleza multi-banco de las memorias TCDM para asegurar que los esclavos dirigen sus transacciones de espera activa a un banco de memoria distinto.

Respecto a las barreras hardware, consideramos tanto el diseño *Jerárquico* como el *Flat* descrito en la Sección III-C. El número de hilos involucrado en una región paralela puede ser establecido por el programador mediante la cláusula `num_hilos` cuando se invoca a la directiva OpenMP `#pragma omp parallel`. Nuestras barreras son capaces de sincronizar un número de núcleos menor que el total, pero una fase denominada *Setup* es necesaria para programar los controladores adecuadamente. Para el diseño *Flat* esta fase consiste en la escritura de un registro mapeado a memoria del controlador maestro denominado `max_events`. En caso del *Jerárquico* y la barrera software esta fase es ligeramente más compleja. Basándonos en el número de núcleos en cada clúster (`CPC`) y el número de hilos participantes de la región paralela, es necesario averiguar el número de clústeres involucrados en la operación de sincronización (`FC`). Este número tendrá que ser escrito en el registro mapeado a memoria del controlador global. Todos los núcleos pertenecientes al primer grupo de `FC`-1 clústeres ejecutarán la barrera, de ahí que los correspondientes maestros locales tendrán que ser programados para efectuar la sincronización entre ellos. Por el contrario, no todos los núcleos pertenecientes al clúster `FC`-*th* estarán involucrados en la barrera, de ahí que el número apropiado de ellos tendrá que ser computado y escrito en el controlador maestro local.

La manera más natural para integrar esta fase de *Setup* en el modelo de ejecución OpenMP es permitir al hilo maestro realizar esta etapa de programación una vez que se cree una región paralela. Nosotros hemos insertado este conjunto de operaciones de escritura dentro de la función de biblioteca *runtime* (`parallel_start`). Estas operaciones son escrituras regulares a registros mapeados a memoria. Las correspondientes transacciones viajan a través de la NoC dirigidas a cada controlador de clúster activo. Una vez que cada controlador ha sido programado, el mecanismo hardware subyacente puede ser activado vía software mediante escrituras/lecturas a dos registros que cada controlador integra: `bar_reg_in`, que es usado cuando un núcleo quiere participar en la barrera (para empezar la fase *gather*); y `bar_reg_out`, que el controlador escribe para despertar el núcleo a fin de continuar con la ejecución de la aplicación (la finalización de la fase *release*).

Nuestro primer experimento compara el costo, en términos de ciclos, de los esquemas software y hardware para barrera. En la barrera software, para evitar medir el tiempo de espera debido a una llegada de hilo que no esté alineada, hemos medido el tiempo de barrera desde el hilo maestro, asegurando que todos los esclavos ya han entrado en la barrera. El desglose del costo de barrera es mostrado en la Fig. 11, mostrando aproximadamente 700 ciclos considerando el tiempo total para las fases *gather* y *release* locales y globales. También una sobrecarga de cien ciclos es requerida por el *runtime* más el costo de inicializar la barrera (fase *Setup*) que requiere 104 ciclos adicionales. En total, sincronizar 64 núcleos en la barrera software cuesta unos 900 ciclos.

En la Fig. 12 mostramos el costo de las dos implementaciones hardware para barrera. Como se puede ver, mientras que el tiempo de barrera por sí mismo no es muy diferente en ambos casos, la fase de *Setup*, tal y como se esperaba, es más costosa para el diseño *Jerárquico*. Es importante indicar que este coste de *Setup* solamente se paga cuando se abre una región paralela con un número de hilos diferente al previamente utilizado. En caso de que no varíe, el costo del *Setup* es drásticamente reducido (en torno a 15 ciclos).

Como segundo experimento, queremos estimar la granularidad de paralelismo que cada implementación habilita. Para este fin, utilizamos un benchmark sintético compuesto de un bucle que invoca una pequeña rutina ensamblador compuesta únicamente de instrucciones ALU para evitar contención en memoria. Esta rutina es anotada con una directiva `#pragma omp parallel` de OpenMP, que replica su ejecución entre todos los hilos participantes. Esta rutina puede ser parametrizada para generar distintas granularidades de tareas paralelas (desde 10 a 10.000 ciclos) para estudiar cómo el paralelismo se ve afectado por el tiempo de barrera. En Fig. 13 se ilustran los resultados de este último estudio, donde mostramos el porcentaje de tiempo de sincronización cuando la granularidad de la tarea paralela (i.e. ciclos de ejecución) se incrementa. Para tareas extremadamente pequeñas (10 ciclos), el tiempo de barrera es el factor dominante en todos los casos ($\geq 90\,\%$). Sin embargo, es posible ver que las barreras hardware consiguen una latencia un orden de magnitud menor con respecto a la software. Si cualitativamente establecemos que una sobrecarga del $5\,\%$ para sincronización es despreciable, es posible ver que este punto es alcanzado por las barreras hardware a una granularidad de miles de ciclos, mientras que este mismo punto es alcanzando por la barrera software en tareas de diez mil ciclos. Finalmente, desde una perspectiva software no existe apenas diferencia entre ambas barreras hardware por la sobrecarga de la pila software que tiende a ocultarla.

V. Conclusiones y Trabajo Futuro

Diseñar un esquema de barrera hardware con herramientas de diseño y tecnología estándar de forma eficiente es un verdadero reto, debido a la latencia creciente en las interconexiones entre componentes a medida que la escala de integración aumenta. Los esquemas hardware de barrera estudiados en este trabajo con un número menor de etapas proporcionan el mejor rendimiento a pesar de utilizar los enlaces más largos. Esto es a causa de que los retardos en las interconexiones no son tales como para contrarrestar el beneficio de tener un menor número de etapas de sincronización, aunque sí que se obtiene la mayor sobrecarga en área porque la herramienta de posicionamiento y rutado opera de manera más agresiva insertando repetidores para no disminuir la latencia de los enlaces. Sin embargo, desde la perspectiva software esto cambia ligeramente. Cuando se integran las barreras hardware en un sistema de simulación completo con una pila software basada en OpenMP, el soporte software para crear el paralelismo introduce una sobrecarga que tiende a ocultar las diferencias de rendimiento. Nuestros experimentos con OpenMP demuestran que ambas barreras hardware habilitan un grano de paralelismo un orden de magnitud más fino que una implementación pura software. Además, la barrera jerárquica permite sincronizar de manera independiente múltiples grupos de núcleos (uno por clúster) de manera concurrente. Esto es algo muy importante para por ejemplo el paralelismo anidado, o para cuando múltiples aplicaciones paralelas independientes se están ejecutando en el sistema. Ambos escenarios serán abordados como trabajo futuro.

Agradecimientos

El presente trabajo ha sido financiado mediante los proyectos "CSD2006-00046" y "TIN2009-14475-C04". José L. Abellán es beneficiario de la beca 12461/FPI/09 de la Comunidad Autónoma de la Región de Murcia (Fundación Séneca, Agencia Regional de Ciencia y Tecnología).

Referencias

[1] J. L. Abellán et al., "A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs". In *International Conference on Parallel Processing*, 2010.

[2] J. L. Abellán et al., "Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs". In *Design, Automaton & Test in Europe Conference*, 2012.

[3] M. Monchiero et al., "Efficient Synchronization for Embedded On-Chip Multiprocessors". In *IEEE Transactions on Very Large Scale Integration Systems*, vol.14, no.10, October 2006.

[4] O. Villa et al., "Efficiency and Scalability of Barrier Synchronization on NoC Based Many-core Architectures". In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008.

[5] D. Ludovici et al., "Capturing Topology-Level Implications of Link Synthesis Techniques for Nanoscale Networks-on-Chip", In *ACM Great Lakes symposium on VLSI*, 2009.

[6] J. Oh et al., "TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines", In *SIGARCH Comput. Archit. News*, vol.39, issue 3, pp.105-116, 2011.

[7] Plurality Ltd. "The HyperCore Processor". `www.plurality.com/hypercore.html`.

[8] ST Microelectronics and CEA., "Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology". 2010, `www.2parma.eu/images/stories/p2012_whitepaper.pdf`.

[9] A. Marongiu et al., "Supporting OpenMP on a multi-cluster embedded MPSoC". *Microprocessors and Microsystems* `www.sciencedirect.com/science/article/pii/S0141933111001001`

[10] J. Sartori and R. Kumar., "Low-Overhead, High-Speed Multi-core Barrier Synchronization". In *International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[11] C. Cascaval et al., "Evaluation of a Multithreaded Architecture for Cellular Computing". In *International Symposium on High-Performance Computer Architecture*, 2002.