Evaluación de un Protocolo de Directorio Basado en Lista de Compartidores para Manycores

Ricardo Fernández-Pascual, Alberto Ros y Manuel E. Acacio [1]

Resumen— **En la búsqueda de una solución eficiente y escalable al problema de la coherencia de cachés privadas en las futuras arquitecturas con un gran número de núcleos de procesamiento en el mismo chip (*manycores*), en este trabajo retomamos una clase de protocolos de coherencia que, a pesar de haber sido empleados en la década de los 90 en la construcción de diversos multiprocesadores de memoria compartida comerciales, han caído en el olvido en este nuevo contexto de las arquitecturas multinúcleo. En particular, evaluamos un protocolo de directorio basado en una lista de compartidores simplemente enlazada, comparando su rendimiento con dos protocolos con información de compartición centralizada: uno que emplea un vector de bits por cada línea de memoria y que claramente no es escalable desde el punto de vista de la cantidad de memoria que requiere el directorio, y otro que utiliza la estrategia de punteros limitados con un único puntero, y que tiene una sobrecarga de memoria similar al protocolo basado en lista enlazada. Los resultados de las simulaciones indican que para un número grande de núcleos de procesamiento, la versión basada en lista enlazada obtiene peor rendimiento que las otras dos anteriores como consecuencia del incremento en la ocupación del controlador de directorio que esta configuración provoca.**

Palabras clave— **Arquitecturas multinúcleo, protocolo de coherencia de cachés, código de compartición centralizado y distribuido, rendimiento.**

I. Introducción

CONFORME aumenta el número de núcleos de procesamiento integrados en el mismo chip en las arquitecturas multinúcleo, al ritmo que marca la famosa Ley de Moore, los mecanismos a través de los cuales dichos núcleos se comunican y sincronizan constituyen elementos claves del diseño de la arquitectura. Si la tendencia actual se mantiene, las arquitecturas multinúcleo que están por venir, con varias decenas de núcleos de procesamiento (*manycores*), seguirán empleando el modelo de memoria compartida e implementarán a nivel hardware un mecanismo que asegure la coherencia de los niveles privados de caché de cada núcleo[1]. De esta forma, comunicación y sincronización (esta última implementada normalmente usando posiciones de la memoria compartida) ocurrirán bajo el control del protocolo de coherencia de cachés.

El diseño de un protocolo de coherencia eficiente para una arquitectura con un número grande de núcleos de procesamiento no es algo nuevo. De hecho es algo que ya fue abordado en el contexto de los multiprocesadores de memoria compartida tradicionales. Entonces, los protocolos de coherencia más escalables —aquellos que se basan en el uso de un directorio repartido entre todos los nodos del multiprocesador— se clasificaban en dos categorías en función de cómo se almacenaba la información sobre los compartidores (código de compartición) para cada línea de memoria[2]. En una organización *basada en memoria*, la identidad de los compartidores de una línea de memoria se almacenaba en el nodo origen de la misma, en la entrada de directorio que tenía asignada y que estaba almacenada en memoria principal (de ahí el nombre). Por el contrario, en una organización *basada en caché*, el nodo origen almacenaba únicamente la identidad de uno de los compartidores, y todos ellos aparecían *enlazados* a través de una estructura de datos (listas enlazadas normalmente) con punteros almacenados en las cachés privadas. A día de hoy, las propuestas de protocolos de coherencia para arquitecturas multinúcleo con un número grande de núcleos de procesamiento caen dentro de la primera categoría. Obviamente, la denominación de organización de directorio *basada en memoria* no tiene sentido en este contexto, y a este tipo de protocolos los denominaremos protocolos basados en *código de compartición centralizado*. Por otro lado, a pesar de que los protocolos basados en caché fueron utilizados en varios multiprocesadores comerciales durante la década de los 90 ([3][4][5][6]), no han tenido repercusión en el contexto de las arquitecturas multinúcleo. Este tipo de protocolos, que denominaremos protocolos basados en *código de compartición distribuido*, ofrecían como principal ventaja con respecto a los anteriores una menor sobrecarga de memoria debida a la información de directorio[2]. Sin embargo también presentaban desventajas, como el hecho de que se alargaba la latencia de determinados fallos de caché, se requería introducir cambios en las cachés y se complicaba la gestión de los reemplazos de caché.

En este trabajo evaluamos el rendimiento de un protocolo basado en código de compartición distribuido en el contexto de una arquitectura multinúcleo. En concreto implementamos la versión más sencilla de protocolo de este estilo, que está basada en la utilización de una lista simplemente enlazada (y que llamamos *Lista*), y comparamos el rendimiento que obtiene contra el de dos organizaciones basadas en código de compartición centralizado. La primera emplea un código de vector de bits no escalable (será nuestra configuración *Base*) y la segunda utiliza un código basado en punteros limitados con un único puntero (la llamamos *1-puntero*). Los resultados de

[1]Dpto. de Ingeniería y Tecnología de Computadores, Univ. Murcia, e-mail: {ricardof, aros, meacacio}@um.es

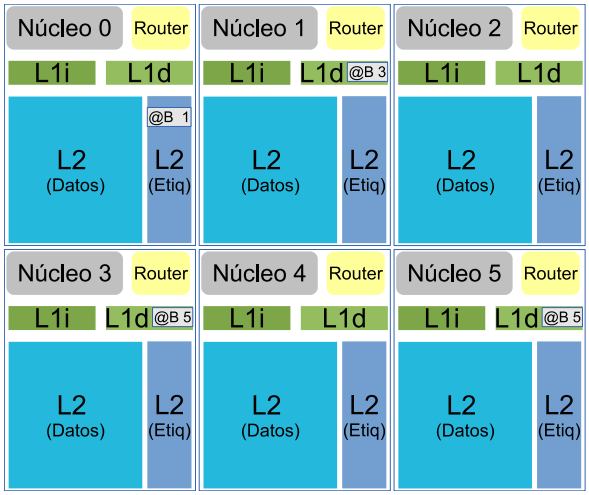


Fig.1. Ejemplo de lista para una línea de datos B compartida por los nodos 1, 3 y 5, y cuyo nodo origen es 0.

nuestras simulaciones indican que las tres configuraciones obtienen más o menos el mismo rendimiento para 16 núcleos de procesamiento. Sin embargo, cuando el número de núcleos crece hasta los 64, la versión *Lista* obtiene peor rendimiento que las otras dos anteriores como consecuencia del incremento en la ocupación del controlador de directorio que esta configuración provoca.

El resto del documento se organiza como sigue. En la Sección II describimos el funcionamiento del protocolo basado en código de compartición distribuido que hemos implementado. Después, en la Sección III comparamos la sobrecarga de memoria de las tres configuraciones evaluadas en este trabajo (*Base*, *Lista* y *1-puntero*). La Sección IV detalla el entorno de evaluación que asumimos y los resultados son mostrados en la Sección V. Finalmente, la Sección VI presenta las principales conclusiones que extraemos a la vista de los resultados.

II. Descripción del protocolo basado en lista simplemente enlazada

La principal diferencia del protocolo *Lista* respecto a un protocolo de directorio tradicional es que la información de compartición se almacena de forma distribuida entre el nodo origen (*home*) y todos los compartidores de la línea de caché. El conjunto de compartidores de una línea se representa mediante una lista simplemente enlazada, de forma que cada nodo necesita almacenar únicamente un puntero a un compartidor. En particular, el nodo origen, en la parte de etiquetas de la L2, almacena la identidad del primer compartidor, el cual, a su vez, almacena la identidad del siguiente compartidor junto con su copia de la línea de memoria en la caché L1, hasta llegar al último, que hace lo propio pero almacenando un puntero nulo (el valor nulo lo representamos almacenando la identidad del mismo compartidor). A modo de ejemplo, la Figura 1 muestra el caso de una arquitectura con 6 núcleos de procesamiento, en el que los núcleos 1, 3 y 5 comparten un bloque con datos cuya dirección es B. El nodo 0 constituye el origen para dicho bloque (*home*).

Debido a que la información de compartición se almacena de forma distribuida, la actualización de dicha información requiere un intercambio de mensajes entre los compartidores y el nodo origen que no es necesario en otros protocolos. En el protocolo *Lista* que describimos en este trabajo, la actualización de la lista de compartidores se inicia siempre desde el nodo origen, el cual permanece bloqueado (para esa línea) hasta que la actualización termina. De esta forma nos aseguramos de que nunca se realizan dos o más actualizaciones a la vez.

A. Resolución de un fallo de lectura

La resolución de un fallo de lectura cuando no existe aún ningún compartidor se realiza de forma casi idéntica a como lo hace un protocolo de directorio con información de compartición centralizada: una vez que la petición de lectura llega a la L2 en el nodo origen, éste envía los datos al peticionario, el cual contesta con un mensaje de *Unblock*. El nodo origen apunta la identidad del primer y único compartidor.

Si el dato no se encontrara en la L2, el nodo origen realizaría una petición a memoria y reenviaría los datos recibidos al peticionario después de copiarlos también en la L2. En este caso, el peticionario recibiría la línea en estado `E`.

En el caso de que ya existieran uno o más compartidores de la línea pedida, la identidad de uno de ellos estaría apuntada en el nodo origen. En ese caso, el nodo origen comunicaría la identidad de este compartidor junto con el mensaje de datos al nodo peticionario antes de sobreescribirla con el identificador del mismo. El peticionario apuntará al compartidor ya existente como su *siguiente compartidor* en la lista y envía el mensaje de *Unblock*. De esta forma, los compartidores quedan almacenados en la lista en orden inverso a cuándo se recibieron las peticiones en el nodo origen.

Si, por el contrario, la línea ha sido modificada previamente por un nodo, éste se encontrará en estado `M` y será el único que poseerá una copia válida de los datos. En ese caso, la L2 reenviará la petición de lectura al nodo en estado `M` y apuntará al peticionario como compartidor. El nodo en estado `M` pasará a estado `S` y enviará los datos y su propia identidad al peticionario, que apuntará al nodo que previamente estaba en estado `M` como su *siguiente compartidor* y envía el mensaje de *Unblock*.

No se necesita ningún mensaje adicional con respecto al protocolo *Base* para actualizar la lista de compartidores en este caso, ya que se aprovechan los mensajes de respuesta ya existentes.

B. Resolución de un fallo de escritura

La resolución de un fallo de escritura requiere la invalidación de todos los compartidores ya existentes. Esta invalidación se inicia en paralelo al envío del mensaje con los datos al peticionario.

Mientras que en un protocolo con información centralizada como el *Base* el nodo origen envía un mensaje de invalidación a cada compartidor (la informa-

ción completa sobre los compartidores se encuentra en la entrada de directorio asociada a la línea de memoria), lo cual permite que las invalidaciones se realicen en paralelo (aunque el envío de los mensajes será secuencial si la red de interconexión no ofrece soporte *multicast*), en un protocolo con información de compartición distribuida como *Lista* el nodo origen sólo puede enviarle el mensaje de invalidación a uno de los compartidores, el cual tendrá que reenviárselo al siguiente compartidor (si existe), hasta que se llegue al último compartidor. Por tanto, la latencia aumentará, especialmente cuando haya muchos compartidores. Por otro lado, mientras que en el protocolo *Base* cada uno de los nodos invalidados tiene que enviar un mensaje de confirmación al peticionario, en *Lista* sólo el último compartidor necesita enviar mensaje de confirmación.

El envío de los datos se realiza como en los fallos de lectura, según la línea se haya modificado o no.

Tanto en el protocolo *Base* como en el *Lista*, el peticionario no enviará el mensaje de *Unblock* al nodo origen hasta que no haya recibido tanto los datos como la confirmación de las invalidaciones, ya sean varios mensajes (en el caso de *Base*) o un mensaje (en el caso de *Lista*).

El número de mensajes utilizados por el protocolo *Lista* para resolver los fallos de escritura es el mismo o menor que en el protocolo *Base*, pero estos mensajes se procesan secuencialmente por los compartidores en lugar de en paralelo, por lo que la latencia es previsible que aumente.

C. Resolución de los reemplazos

Los reemplazos de bloques modificados se comportan exactamente igual en el protocolo *Lista* que en el *Base*. En ambos, el nodo que desea realizar el reemplazo envía una petición a la L2. La L2 contesta con un mensaje autorizando el reemplazo y finalmente el peticionario envía los datos a la L2. La petición previa al envío de los datos permite asegurar que la L2 podrá tratar el mensaje de datos y evita interbloqueos.

Sin embargo, el tratamiento de los reemplazos de bloques en estado compartido es la parte del protocolo *Lista* que se comporta de manera más diferente al protocolo *Base*. En el protocolo *Base*, los reemplazos de este tipo son silenciosos: la línea de memoria a reemplazar es simplemente descartada y no se envía ningún mensaje a la L2, por lo que ésta lo seguirá contando entre los compartidores hasta que decida invalidarlo. Es decir, el código de compartición puede ser inexacto y contener algunos nodos aunque estos no posean los datos realmente. Esto no es posible en el protocolo *Lista*, ya que la información sobre el siguiente compartidor se almacena junto con los datos, por lo que no se puede descartar el bloque sin haber comunicado esta información previamente. De no hacerlo, la lista de compartidores quedaría rota.

O bien la L2 o bien otro compartidor tienen apuntado al nodo que desea hacer el reemplazo como *siguiente compartidor*. Este nodo sería el *previo compartidor* del nodo que reemplaza. Para completar el reemplazo, es necesario actualizar el puntero del *previo compartidor* para que apunte al actual *siguiente compartidor* del nodo que reemplaza, desenlazando así a este último de la lista de compartidores. Debido a que este protocolo utiliza listas simplemente enlazadas, esta acción no se puede realizar sin recorrer la lista desde el principio.

Por ello, para iniciar un reemplazo, el nodo envía una petición a la L2 indicando quién es su actual *siguiente compartidor*. Si el nodo que reemplaza es el *siguiente compartidor* de la L2 (es decir, el primero de la lista), ésta actualiza su puntero y envía un mensaje de confirmación al peticionario indicándole que ya puede descartar los datos. En otro caso, la L2 reenvía la petición a su siguiente compartidor, el cual la reenvía al siguiente hasta llegar al *previo compartidor* del nodo que reemplaza, el cual actualiza su puntero y envía la confirmación para descartar los datos al peticionario. Una vez recibida la confirmación, el peticionario descarta los datos y envía un mensaje para desbloquear la L2.

El tratamiento de los reemplazos del protocolo *Lista* aumenta significativamente el número de mensajes que circulan por la red y el tiempo que la L2 permanece bloqueada.

III. Análisis de la sobrecarga de memoria

El mayor atractivo del protocolo *Lista* es su gran escalabilidad en cuanto a la cantidad de memoria necesaria para almacenar la información de compartición, lo cual implica que requiere menos área y, por tanto, una mayor escalabilidad en cuanto al consumo estático de energía. Mientras que la cantidad de memoria necesaria por entrada de directorio en un protocolo con vector de bits crece linealmente con el número de núcleos de ejecución (un bit por núcleo), el crecimiento es logarítmico para un protocolo basado en punteros como *Lista*. A su vez, *Lista* necesita un puntero adicional en cada entrada de L1, pero esto no es un problema si tenemos en cuenta que el número de entradas de L1 será siempre mucho menor que el número de entradas de L2.

La sobrecarga de memoria en todo el chip que supone almacenar la información de compartición con respecto a la memoria usada por las cachés (L1 y L2) usando cada código de compartición se refleja en la Figura 2. Podemos ver cómo el código de vector de bits (*Base*) es claramente no escalable, ya que requiere un bit por núcleo junto con cada etiqueta de la L2 (N bits, siendo N el número de núcleos). El código de punteros limitados con un puntero (*1-puntero*) es mucho más escalable, puesto que requiere solo un puntero ($\log_2 N$) más un bit *de desbordamiento* en cada entrada de la L2 para codificar el caso en el que hay más de un compartidor y por tanto será preciso realizar un *broadcast*. Por último, el código de compartición basado en listas enlazadas utilizado por el protocolo *Lista* requiere un puntero por cada entrada de la L2 ($\log_2 N$) y otro puntero ($\log_2 N$), en cada

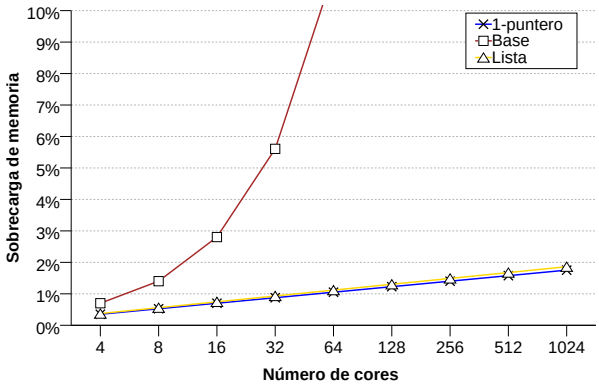


Fig.2. Sobrecarga de memoria de los tres protocolos evaluados.

TABLAI

Parámetros del sistema.

Parámetros de la memoria	
Tamaño de bloque	64 bytes
Caché L1 de datos e instr.	32KiB, 4 vías
Latencia de acceso a L1	1 ciclos
Caché L2 compartida	512KiB/celda, 16 vías
Latencia de acceso a L2	12 ciclos
Organización L1 y L2	Inclusiva
Información de directorio	Incluida en L2
Tiempo de acceso a memoria	160 ciclos
Parámetros de la red	
Topología	Malla 2-D (4×4) y (8×8)
Técnica de enrutamiento	Determinista X-Y
Tamaño de mensajes	5 flits (datos), 1 flit (control)
Tiempo de enlace	1 ciclo
Ancho de banda	1 flit por ciclo

entrada de la L1. Para codificar los punteros nulos en las cachés L1, se utiliza en cada nodo el identificador del propio nodo, aprovechando que no tendría nunca sentido que un nodo se apuntara a sí mismo como *siguiente compartidor*. Por otro lado, en la L2 no hace falta incluir el bit de válido para el puntero ya que el estado del bloque permite deducir su validez. Esto sucede tanto para el código *1-puntero* como para el código *Lista*.

IV. Entorno de evaluación

La evaluación del protocolo con código de compartición de listas enlazadas se ha realizado mediante los simuladores PIN[7] y GEMS 2.1[8], conectándolos de forma similar a como se propone en [9]. PIN obtiene todos los accesos a los datos accedidos por las aplicaciones y GEMS modela la jerarquía de memoria y calcula la latencia de acceso a memoria de cada petición del procesador. La red de interconexión se ha modelado con el simulador Garnet [10]. La arquitectura simulada corresponde a un multiprocesador en un único chip (*tiled*-CMP) tanto con 16 núcleos como con 64 núcleos. Los principales parámetros de evaluación se muestran en la Tabla I.

Para la evaluación de este artículo hemos implementado en GEMS un protocolo de tradicional de directorio con código de compartición de vector de bits (llamado *Base*), un protocolo que almacena como código de compartición un único puntero al propietario (*owner*) (llamado *1-puntero*), similar al *Magny-Cours* de AMD[11], y un protocolo con un código de compartición distribuido en las cachés mediante listas enlazadas (que hemos denominado *Lista*). En todos los protocolos la L2 es inclusiva con respecto a la L1. Por tanto, el código de compartición se puede guardar junto con las etiquetas de la caché L2.

Para la evaluación, hemos usado todas las aplicaciones de la suite SPLASH-2 con los tamaños recomendados[12]. Hemos tenido en cuenta la variabilidad en las aplicaciones paralelas tal y como se comenta en [13]. Para ello hemos realizado varias simulaciones para cada aplicación y configuración, insertando perturbaciones aleatorias en la latencia de cada acceso a memoria. Todos los resultados mostrados en este trabajo corresponden a la parte paralela de las aplicaciones evaluadas.

V. Resultados

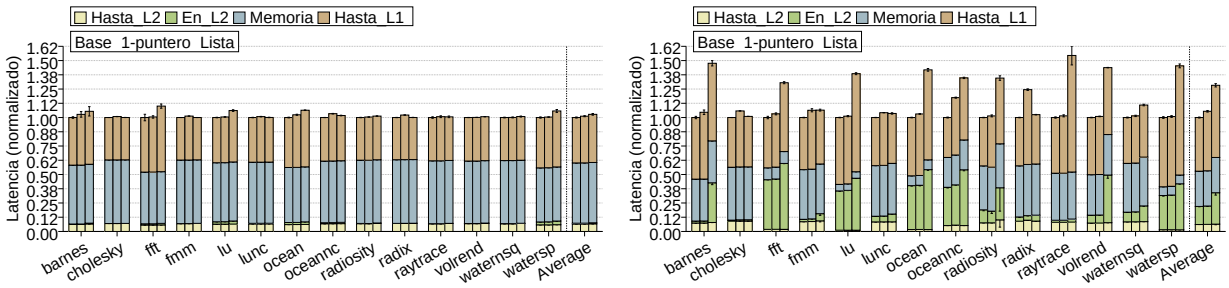
En esta sección describimos los resultados obtenidos al simular las aplicaciones utilizando los tres protocolos de coherencia mencionados anteriormente para configuraciones con 16 y 64 núcleos de ejecución. En primer lugar, nos centramos en la latencia de los fallos de caché y en cómo se distribuye esta latencia. Después, mostramos el tráfico en la red de interconexión generado por cada tipo de mensaje enviado para mantener la coherencia. Por último, consideramos el tiempo de ejecución de las aplicaciones utilizando cada protocolo de coherencia.

A. Latencia de los fallos de caché L1

El rendimiento de un multiprocesador está muy influenciado por la latencia de los fallos de caché. La decisión de diseño del código de compartición es un aspecto importante ya que puede influir en dicha latencia. La Figura 3 muestra la latencia normalizada de los fallos de las cachés de primer nivel para configuraciones de 16 y 64 procesadores. Esta latencia está separada entre el tiempo hasta llegar a la L2 (*Hasta_L2*), el tiempo de espera en la L2 (*En_L2*), el tiempo de acceso a la memoria (*Memoria*) y el tiempo desde que el fallo deja la L2 hasta que se resuelve (*Hasta_L1*).

Podemos observar que para la configuración de 16 procesadores (Figura 3(a)), la latencia de fallo no se ve muy afectada por el tipo de código de compartición usado. Lo único apreciable es un ligero incremento en el tiempo de respuesta de la L2 a la L1 (*Hasta_L1*) para los códigos *1-puntero* y *Lista*. Esto se debe a un aumento en la latencia de las escrituras, en el caso de *1-puntero* por incrementar el número de mensajes enviados y confirmaciones que se deben recibir y en el caso de *Lista* porque las invalidaciones que hay que enviar en un fallo de escritura se serializan tal y como se ha explicado.

Cuando observamos los resultados para 64 procesadores (Figura 3(b)) vemos que la latencia *Hasta_L1* aumenta en mayor medida debido al mayor número de núcleos. Sin embargo, el dato alarmante es el importante incremento en el tiempo de espera en la L2 (*En_L2*). Incluso en la configuración *Base* podemos apreciar que al pasar de 16 a 64 núcleos algunas aplicaciones ya sufren contención, pero el código basado



(a) 16 procesadores

(b) 64 procesadores

Fig.3. Latencia de los fallos de L1.

en lista exacerba esta contención. Esto es debido a que actualizar la lista de compartidores requiere más tiempo que actualizar los otros códigos de compartición centralizados. Para evitar errores en la lista, su actualización se realiza en exclusión mutua y por tanto la L2 permanece bloqueada para la línea de memoria cuyo código de compartición se está actualizando. Esto provoca mayor contención cuanto mayor es el número de procesadores que quieren actualizar la lista. Además, ante un reemplazo de un bloque compartido, la lista se debe actualizar, lo que supone un mayor tiempo de bloqueo en la L2.

B. Tráfico en la red

El diseño del código de compartición también influye en el tráfico generado por el protocolo de coherencia y, por tanto, en el consumo energético de la arquitectura. La Figura 4 muestra el tráfico normalizado medido en *flits* que pasa por la red de interconexión para configuraciones de 16 y 64 núcleos. Este tráfico se ha dividido en tráfico de datos debido a fallos de caché (*Datos*), tráfico de datos debido a reemplazos (*DatosReem*), tráfico de control debido a fallos de caché (*Control*), tráfico de control debido a reemplazos de datos privados (*ControlReemME*) y tráfico de control debido a reemplazos de datos compartidos (*ControlReemS*).

Como se puede apreciar en los resultados de la configuración de 16 núcleos (Figura 4(a)), el código de compartición *1-puntero* aumenta el tráfico debido a los mensajes de control, ya que este protocolo necesita hacer un *broadcast* del mensaje de invalidación cuando exista más de un compartidor. Por contra, *Lista* tiene el mismo tráfico de control (aunque las invalidaciones se envían en serie), pero incrementa considerablemente el tráfico debido a los reemplazos, sobre todo en el caso de los reemplazos de bloques compartidos que en los otros dos casos se pueden hacer de forma silenciosa. Este proceso de reemplazo, que además se realiza también en serie, es uno de los causantes del incremento de la contención en el directorio.

En el caso de 64 núcleos (Figura 4(b)), el tráfico de *1-puntero* se equipara al de *Lista*, ya que las operaciones de *broadcast* se hacen más costosas al aumentar el número de núcleos. Esto hace pensar que aunque *1-puntero* es tan escalable en términos de memoria como *Lista*, no lo es en cuestión de tráfico de red, y por tanto consumo energético en la red de interconexión, por lo que no será una solución factible para un mayor número de núcleos de ejecución.

Por último también se observa que el código basado en listas incrementa en gran medida el tráfico debido a reemplazos de datos compartidos, con lo que el manejo de estos reemplazos es uno de los puntos débiles de los protocolos que utilizan este código, ya que no solo aumenta el tráfico, sino que también causa más contención en el directorio, con la consiguiente degradación en el tiempo de ejecución, como mostramos a continuación.

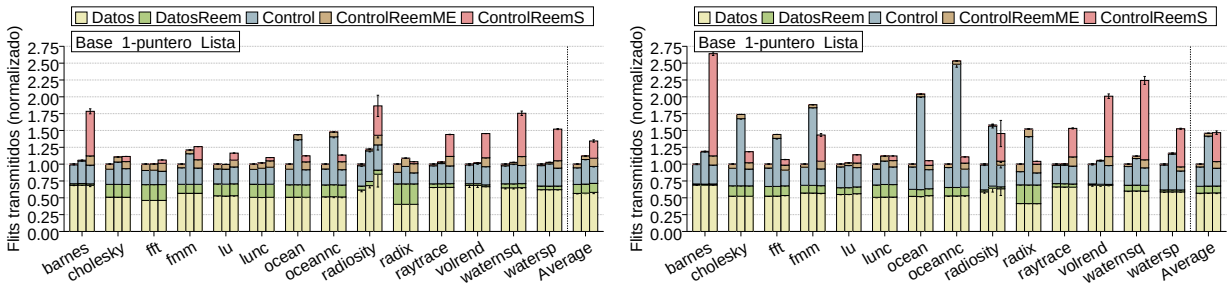
C. Tiempo de ejecución

Por último, mostramos cómo se ve afectado el tiempo de ejecución de las aplicaciones con los diferentes códigos de compartición evaluados. La Figura 5 muestra el tiempo de ejecución normalizado, de nuevo, para configuraciones de 16 y 64 núcleos.

La configuración de 16 núcleos (Figura 5(a)) no se ve muy afectada por el código de compartición en términos de tiempo de ejecución. Sin embargo, en la de 64 (Figura 5(a)) algunas aplicaciones sufren un aumento considerable en el tiempo de ejecución, sobre todo en el caso del código *Lista*. Estos incrementos se producen en *barnes*, *fft*, *lu*, *ocean*, *oceannc* y *volrend*, que están por encima de la media. Si nos fijamos de nuevo en la gráfica de la latencia de los fallos (Figura 3(b)), podemos apreciar que, precisamente, estas son las aplicaciones para las que aumenta considerablemente el tiempo de espera en la L2 con 64 núcleos.

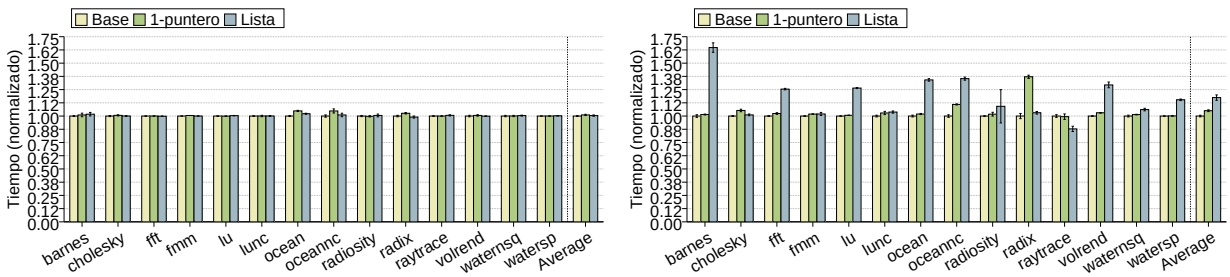
VI. Conclusiones

En este trabajo hemos evaluado el comportamiento de un protocolo basado en información de compartición distribuida de tipo lista simplemente enlazada (*Lista*) en el contexto de una arquitectura multinúcleo. Hemos visto que este tipo de protocolos escalan bien desde el punto de vista de la cantidad de memoria que se requiere para almacenar la información de directorio. Sin embargo, en cuanto al rendimiento se refiere, hemos podido comprobar que si bien para un número no muy grande de núcleos de ejecución el protocolo *Lista* es competitivo con respecto a otras alternativas basadas en código de compartición centralizado, conforme aumenta el número de núcleos de la arquitectura, su rendimiento se va



(a) 16 procesadores (b) 64 procesadores

Fig.4. Tráfico en la red de interconexión.



(a) 16 procesadores (b) 64 procesadores

Fig.5. Tiempo de ejecución.

viendo perjudicado. Según hemos podido comprobar esto se debe a una mayor latencia de los fallos de caché como consecuencia principalmente de una mayor ocupación del controlador de directorio, el cual permanece bloqueado por más tiempo, retrasando la atención a otros fallos a la misma línea. El principal causante de esto hemos visto que son las operaciones de reemplazo, las cuales aumentan en número (no hay reemplazos silenciosos) y tiempo que involucran al controlador de directorio.

A pesar de los resultados obtenidos, creemos que este tipo de protocolos presentan interesantes posibilidades que merece la pena explorar en el contexto de una arquitectura multinúcleo con un número grande de núcleos de ejecución (*manycore*). De estas forma, como trabajo futuro planteamos reducir el tiempo de bloqueo del directorio a través de una estrategia de reemplazo que minimice el tiempo que permanece ocupado el controlador de directorio.

Agradecimientos

This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2012-38341-C04-03"

Referencias

[1] Milo M.K. Martin, MarkD. Hill, and Daniel Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, July 2012.

[2] DavidE. Culler, JaswinderP. Singh, and Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, Inc., 1999.

[3] DavidB. Gustavson, "The scalable coherent interface and related standards proyects," *IEEE Micro*, vol. 12, no. 1, pp. 10–22, Jan. 1992.

[4] R.Clark and K.Alnes, "An SCI chipset and adapter," in *HotInterconnects Symp. IV*, Aug. 1996, pp. 221–235.

[5] Tom Lovett and Russell Clapp, "STiNG: A cc-NUMA computer system for the commercial marketplace," in *23rd Int'l Symp. on Computer Architecture (ISCA)*, June 1996, pp. 308–317.

[6] Radhika Thekkath, AmitP. Singh, JaswinderP. Singh, Susan John, and JohnL. Hennessy, "An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200," in *11th Int'l Parallel Processing Symp. (IPPS)*, Apr. 1997, pp. 8–17.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, VijayJanapa Reddi, and Kim Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2005, pp. 190–200.

[8] MiloM.K. Martin, DanielJ. Sorin, BradfordM. Beckmann, MichaelR. Marty, Min Xu, AlaaR. Alameldeen, KevinE. Moore, MarkD. Hill, and DavidA. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sept. 2005.

[9] Matteo Monchiero, JungHo Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi, "How to simulate 1000 cores," *Computer Architecture News*, vol. 37, no. 2, pp. 10–19, July 2009.

[10] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and NirajK. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[11] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes, "Blade computing with the AMD Opteron^{TM} processor ("Magny Cours")," in *21st HotChips Symp.*, Aug. 2009.

[12] StevenCameron Woo, Moriyoshi Ohara, Evan Torrie, JaswinderPal Singh, and Anoop Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, June 1995, pp. 24–36.

[13] AlaaR. Alameldeen and DavidA. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.