Optimización de un Protocolo de Directorio Basado en Lista Simple en 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 las cachés privadas en las futuras arquitecturas con un gran número de núcleos de procesamiento en el mismo chip (*manycores*), en este trabajo presentamos dos técnicas a través de las cuales un protocolo de directorio basado en una lista de compartidores simplemente enlazada puede alcanzar las prestaciones de uno que emplee vectores de bits no escalables. En particular, mostramos que la principal fuente de degradación del rendimiento con el protocolo basado en lista de compartidores simplemente enlazada está en los reemplazos de datos en estado compartido. A través de la técnica de *reemplazos oportunistas (RO)*, conseguimos aprovechar un reemplazo en curso para que otras cachés con reemplazos pendientes al mismo bloque de datos puedan realizarlos conforme el primero progresa, a la vez que mediante la técnica de *reemplazos concurrentes (RC)*, la caché compartida puede seguir atendiendo los fallos de lectura mientras está teniendo lugar un reemplazo. Usando ambas técnicas en conjunto conseguimos eliminar el problema de los reemplazos de datos en estado compartido, alcanzando una solución escalable al problema de la coherencia de cachés, al mismo tiempo que mantenemos el rendimiento que se alcanzaría con un directorio de mapa completo (no escalable).**

Palabras clave— **Arquitecturas multinúcleo, protocolo de coherencia de cachés, código de compartición distribuido, lista simple, reemplazos silenciosos, rendimiento, energía.**

I. Introducción

CONFORME aumenta el número de transistores que pueden integrarse en un solo chip, siguiendo la conocida Ley de Moore, se hace posible la realización de arquitecturas multiprocesador-en-un-solo-chip con cada vez más núcleos de procesamiento integrados. Lejos quedan ya los primeros diseños con dos núcleos por chip, como el pionero IBM POWER4[1]. La tecnología disponible a día de hoy ya permite la comercialización de arquitecturas multinúcleo con varias decenas de núcleos de procesamiento, como por ejemplo los procesadores con 72 núcleos Knights Landing[2] y Tile GX8072[3], de Intel y Tilera respectivamente, y la tendencia hacia diseños con un mayor número de núcleos (*manycores*) no se vislumbra que haya alcanzado aún su final[4].

En estas arquitecturas multinúcleo con un gran número de núcleos de procesamiento, los mecanismos a través de los cuales dichos núcleos se comunican y sincronizan constituyen elementos claves del diseño. Si la tendencia actual se mantiene, las arquitecturas multinúcleo que están por venir 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[5]. 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.

La mejor manera de asegurar la coherencia de las cachés privadas cuando el número de núcleos de procesamiento es grande es a través de un protocolo de coherencia de cachés basado en directorio. El directorio no es más que una estructura de memoria, implementada en la caché compartida de último nivel[1], que se utiliza para llevar cuenta de los compartidores de cada uno de los bloques de memoria que están almacenados en las cachés privadas. Obviamente, dados los recursos limitados dentro del chip, esta estructura de memoria debería ser lo más pequeña posible para no tener que detraer excesivos recursos de otros elementos importantes (como las cachés).

A día de hoy, las propuestas de protocolos de coherencia para arquitecturas multinúcleo con un número grande de núcleos de procesamiento emplean lo que denominamos *códigos de compartición centralizados*, es decir, toda la información sobre los compartidores de cada bloque de memoria se encuentra concentrada en el nodo origen (*home*). En estos diseños, para minimizar la cantidad de memoria dedicada a la estructura de directorio se suele recurrir al empleo de códigos de compartición comprimidos[6], que incrementan el número de mensajes por evento de coherencia, o más recientemente, al uso de funciones hash[7], que complican la codificación y decodificación de la información de compartición.

Una alternativa a lo anterior sería el empleo de estructuras de directorio basadas en un *código de compartición distribuido*. En este tipo de directorios, la identidad de los compartidores para cada bloque de memoria se reparte entre el nodo origen (mantiene un puntero a uno de los compartidores) y las cachés privadas (mantienen punteros a través de los cuales se enlazan todos los compartidores). El hecho de que el número de entradas en las cachés privadas sea bastante inferior al de la caché compartida hace que se pueda reducir la cantidad de memoria dedicada al directorio. Además, la cantidad de memoria de directorio dedicada a cada bloque se ajusta dinámicamente según el número de compartidores.

De entre los diseños de directorio que se podrían realizar utilizando estos códigos de compartición distribuidos, dos resultan especialmente interesantes. El

[1]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {rfernandez, aros, meacacio}@ditec.um.es

[1]Sin pérdida de generalidad, en este trabajo asumimos cachés de primer nivel privadas a cada núcleo y una caché de segundo nivel compartida por todos ellos, aunque físicamente distribuida

primero emplea una lista doblemente enlazada usando dos punteros en cada una de las entradas de las cachés privadas con copia de un bloque de datos. Esta implementación ya fue usada en la construcción de varios multiprocesadores comerciales durante la década de los 90 ([8][9][10][11]). Sin embargo, a pesar de sus ventajas, no ha tenido repercusión aún en el contexto de las arquitecturas multinúcleo. El segundo diseño reduce la cantidad necesaria de punteros en las cachés privadas usando una estructura de lista simple. En este trabajo mostramos cómo con la primera de las opciones (la basada en lista doblemente enlazada) se puede alcanzar el rendimiento de la mejor alternativa para un directorio centralizado (aunque no escalable) basada en el uso de vectores de bits completos. La segunda de las opciones (la basada en lista simple), sin embargo, se ve penalizada como consecuencia del mayor costo que los reemplazos de datos compartidos tienen. Para solucionarlo, sin embargo, proponemos y evaluamos en este trabajo dos técnicas: (i) *reemplazos oportunistas (RO)*, a través de la cual se consigue aprovechar un reemplazo en curso para que otras cachés con reemplazos pendientes al mismo bloque de dato puedan realizarlos conforme el primero progresa, y (ii) *reemplazos concurrentes (RC)*, a través de la cual el nodo origen puede seguir atendiendo los fallos de lectura mientras está teniendo lugar un reemplazo. A través de simulaciones detalladas de una arquitectura con 64 núcleos de procesamiento, demostramos que haciendo uso de estas dos técnicas, la versión basada en listas simples consigue igualar el rendimiento de la versión basada en listas doblemente enlazadas, y resulta, por lo tanto, preferible dado el menor costo de implementación que tiene (necesita la mitad de punteros en las cachés privadas).

El resto del documento se organiza como sigue. En la Sección II se explica el funcionamiento básico de los protocolos enlazados. En la Sección III describimos las dos técnicas que proponemos para mejorar las prestaciones del protocolo basado en listas simples (Reemplazos Oportunistas y Reemplazos Concurrentes). En la Sección IV evaluamos la sobrecarga de memoria que los protocolos enlazados introducen. La Sección V detalla el entorno de evaluación que asumimos y los resultados de las simulaciones son mostrados en la Sección VI. Finalmente, la Sección VII presenta las principales conclusiones que extraemos a la vista de los resultados.

II. Protocolos de coherencia con información de compartición distribuida

La principal diferencia de los protocolos considerados en este trabajo (*ListaSimple* y *ListaDoble*) respecto a un protocolo de directorio centralizado como *VectorBits* 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 o doblemente enlazada (según el protocolo), de forma que

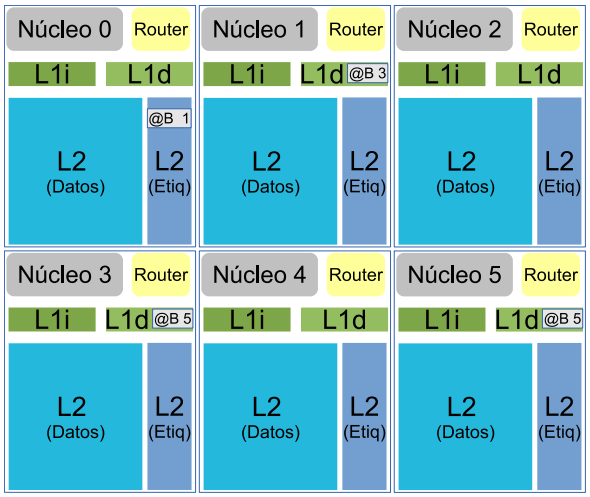

Fig.1. Ejemplo de lista para una línea de datos B compartida por los nodos 1, 3 y 5, y cuyo nodo origen es 0.

cada nodo necesita almacenar únicamente uno o dos punteros a compartidores. 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 propio nodo). En el caso de *ListaDoble*, los nodos también almacenan en la L1 otro puntero al compartidor previo de la lista, excepto el primero que almacena un puntero nulo.

A modo de ejemplo, la Figura 1 muestra el caso de una arquitectura con 6 núcleos que use el protocolo *ListaSimple*. En ella, 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 en estos protocols, la actualización de dicha información requiere intercambios de mensajes entre los compartidores y el nodo origen que no serían necesarios en otros protocolos.

A. Protocolo basado en lista simplemente enlazada

El protocolo *ListaSimple* sin optimizaciones utilizado en este artículo es una versión ligeramente mejorada del protocolo presentado en [12]. La nueva versión necesita un mensaje menos en el camino crítico de los reemplazos (aunque usa el mismo número de mensajes en total por cada reemplazo).

En la versión inicial del protocolo *ListaSimple*, 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, evitando carreras que podrían dar lugar a bloqueos o a pérdida de la integridad de la lista.

En muchos casos, *ListaSimple* se comporta prácticamente igual que un protocolo centralizado. Las diferencias principales se producen en el caso de las

lecturas y escrituras de bloques compartidos y los reemplazos de bloques compartidos. Los accesos y reemplazos a bloques no compartidos (no en estado S) no presentan diferencias.

En el caso de una petición de lectura para una línea que ya posea al menos un compartidor, se realiza una inserción del peticionario al comienzo de la lista de compartidores. En este caso, la identidad del primer compartidor de la lista está apuntada en el nodo origen. Cuando éste recibe la petición de lectura, responde al nodo peticionario comunicándole la identidad de ese primer compartidor junto con el mensaje de datos. El peticionario escribe esta información en su puntero al *siguiente compartidor* y envía el mensaje de *Unblock*. Cuando el nodo origen recibe este último mensaje, actualiza su puntero para que apunte al nuevo compartidor. De esta forma, los compartidores quedan almacenados en la lista en orden inverso a cuándo se recibieron las peticiones en el nodo origen.

También es posible que la petición de lectura se refiera a una línea que se encuentra en estado Modificado o Exclusivo y para la que la única copia de los datos está en una L1. En ese caso, el nodo origen reenviará la petición de lectura al nodo propietario de los datos. Cuando el propietario recibe esta petición, pasa a estado compartido (S) y contesta al peticionario con los datos y su identidad para que el peticionario lo apunte como *siguiente compartidor*. El peticionario contestará al nodo origen con un mensaje de *Unblock* que actualizará su puntero como en el caso anterior.

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

Por su parte, 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. Pero mientras que en un protocolo con información centralizada el nodo origen envía un mensaje de invalidación a cada compartidor (la información 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 *ListaSimple* 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 centralizada cada uno de los nodos invalidados tiene que enviar un mensaje de confirmación al peticionario, en *ListaSimple* 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.

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.

La diferencia del protcolo *ListaSimple* respecto a *VectorBits* que, como se muestra después, resulta más importante para el rendimiento es la forma en la que se tratan los reemplazos de bloques en estado compartido. En el protocolo centralizado, 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 *ListaSimple*, 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 solicitando el reemplazo. Cuando la L2 puede atender la petición, se bloquea y envía un mensaje autorizando el reemplazo al peticionario. Cuando éste lo recibe, descarta el bloque de datos y responde a la L2 con un mensaje 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 la transacción termina. En otro caso, la L2 envía una petición a su siguiente compartidor, el cual la reenvía al siguiente hasta llegar al *previo compartidor* del nodo que reemplaza (es decir, hasta que el puntero de *siguiente compartidor* es igual al peticionario), el cual actualiza su puntero y envía un mensaje para desbloquear la L2.

Obsérvese que no se puede enviar la identidad del *siguiente compartidor* del nodo que reemplaza en el primer mensaje, ya que otra petición podría modificar la lista ante de que el reemplazo se empiece a procesar.

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 perma-

nece bloqueada.

B. Protocolo basado en lista doblemente enlazada

El protocolo *ListaDoble* se diferencia de *ListaSimple* en que cada compartidor no sólo alamacena en su caché L1 la identidad del siguiente compartidor de la lista, sino también la del anterior. Esto permite que los reemplazos de bloques compartidos se puedan realizar en muchos casos sin que la caché compartida tenga que intervenir, a diferencia de lo que ocurre con *ListaSimple*. Estos reemplazos no son silenciosos en *ListaDoble* como en *VectorBits*, pero normalmente requieren menos mensajes que los de *ListaSimple* y, lo que es más importante, no necesitan que la caché compartida se bloquee en ningún momento. De hecho, con este protocolo es posible que un nodo se elimine de la lista al mismo tiempo que otro se añade, o que varios nodos se eliminen a la vez. Otro aspecto muy interesante desde el punto de vista de la escalabilidad es que el número de mensajes necesarios no crece con el número de compartidores del bloque ni de nodos del sistema.

ListaDoble opera de forma muy similar a *ListaSimple* en la mayoría de las transacciones. Las diferencias aparecen en dos casos: en las peticiones de lectura a una línea que ya esté siendo compartida (en estado S) por otro nodo y en los reemplazos de líneas compartidas.

En el caso de las lecturas, las diferencias se deben a la necesidad de actualizar el puntero al previo compartidor del nodo siguiente al peticionario. Para ello, se utilizan dos mensajes extra que están fuera del camino crítico del fallo. El proceso se desarrolla como sigue: el peticionario envía un mensaje a la caché compartida, cuyo puntero apunta en ese momento al actual primer compartidor de la lista. Al igual que en *ListaSimple*, la caché contesta con los datos y además incluye en el mensaje la identidad del primer compartidor. Cuando el peticionario recibe este mensaje, apunta como siguiente compartidor al que era primero. Es ese momento el peticionario ya puede realizar la lectura. Además, envía un mensaje a su nuevo siguiente compartidor para pedirle que actualice su puntero al compartidor previo. Cuando recibe la contestación a este último mensaje, envía un mensaje de *Unblock* a la caché compartida que actualiza su puntero y finaliza la transacción.

La forma de realizar los reemplazos de líneas compartidas es la ventaja de *ListaDoble* con respecto a *ListaSimple*, puesto que *ListaDoble* no necesita recorrer la lista de compartidores desde el principio para poder actualizar el puntero al nodo siguiente del compartidor que precede en la lista al reemplazante. Cuando un nodo que no es el primero de la lista necesita reemplazar, envía directamente una petición a su predecesor. En dicha petición se incluye la identidad de su actual siguiente compartidor. Cuando éste recibe el mensaje, comprueba que sigue siendo efectivamente el predecesor del nodo que quiere reemplazar (puesto que otras transacciones han podido modificar la lista mientras el mensaje estaba en tránsito) y, si es así, actualiza su puntero al siguiente compartidor y envía dos mensajes: uno al peticionario para indicarle que ya puede descartar los datos y los punteros, y otro a su nuevo siguiente compartidor para pedirle que actualice su puntero al previo compartidor. El compartidor previo permanecerá semibloqueado hasta que su nuevo nodo siguiente le indique que ya ha actualizado su puntero para evitar bloqueos y actualizaciones incorrectas de la lista. Mientras esté semibloqueado sólo atenderá a peticiones de lectura de su propio procesador y a peticiones de actualización de su puntero previo, pero no atenderá a otras peticiones de reemplazo, invalidaciones, o peticiones de escritura de su procesador.

En caso de que el nodo que necesite reemplazar sea el primero de la lista, enviará su petición directamente a la L2, que actualizará su puntero y enviará un mensaje al peticionario para indicarle que puede descartar los datos y punteros asociados, y otro mensaje al hasta ahora segundo compartidor (si lo hay) para que actualice su puntero al previo compartidor. La L2 permanecerá bloqueada hasta que reciba la contestación al último mensaje.

Como ya se ha mencionado, los reemplazos a líneas compartidas en *ListaDoble* pueden ocurrir concurrentemente con inserciones de nuevos compartidores y con otros reemplazos.

III. Optimización de los reemplazos del protocolo *ListaSimple*

Como se mostró en [12], la principal razón de la pérdida de tiempo y gasto adicional de energía de *ListaSimple* con respecto a *VectorBits* es la gestión de los reemplazos de bloques compartidos. En particular, estos reemplazos mantienen bloqueada la L2 mientras se recorre la lista de compartidores para actualizar el puntero de compartidor previo al que reemplaza e introducen un número de mensajes por cada reemplazo que crece con el número de compartidores. En esta sección se explican dos técnicas que reducen las desventajas de estos reemplazos y que se pueden utilizar conjuntamente o de forma independiente.

A. Reemplazos oportunistas (RO)

Hemos observado que en algunas aplicaciones se produce con cierta frecuencia la situación en la que varios compartidores del mismo bloque de datos solicitan expulsarlo de la caché L1 casi al mismo tiempo. En la versión básica de *ListaSimple*, todos estos reemplazos son atendidos secuencialmente por la L2 y, como se ha descrito anteriormente, cada uno de ellos requiere que la lista se recorra desde el principio hasta encontrar el nodo anterior al que está reemplazando, lo cual requiere un tiempo significativo. Por ello, cuando esto ocurre, las peticiones de reemplazo de algunos nodos tardan en ser atendidas. Mientras uno de esos nodos está esperando a que se atienda su petición de reemplazo, lo único que hace (respecto a ese bloque de caché) es propagar a través de la lista los mensajes relativos a los reemplazos de los otros

nodos.

Con los *reemplazos oportunistas* se trata de hacer que, cada vez que se recorre la lista para realizar un reemplazo, se aproveche para eliminar de la misma todos aquellos nodos que estén esperando para ser eliminados. De esta forma, se ahorran los correspondientes recorridos que se tendrían que realizar después.

El funcionamiento es como sigue. Después de que un nodo envíe un mensaje a la L2, se queda esperando a que ésta le conteste autorizando el reemplazo (igual que se ha descrito en la sección II-A). Si mientras está esperando la autorización recibe un mensaje relativo a un reemplazo de otro compartidor (estos mensajes los envía inicialmente la L2 y son propagados a través de la lista por las L1, como se describe en la sección II-A), en lugar de propagar el mensaje (o actualizar su puntero y contestar a la L2 en caso de que coincida con el nodo que está reemplazando), contesta a quien se lo ha enviado, que es su nodo previo, con un mensaje pidiéndole que cambie su puntero para que apunte a su siguiente nodo. A continuación el nodo previo vuelve a reenviar su mensaje a su nuevo nodo siguiente (o actualiza su puntero y contesta a la L2 si resulta que el nuevo nodo siguiente es el nodo que está reemplazando). De esta manera el nodo que estaba esperando para reemplazar ya está desconectado de la lista, aunque su petición de reemplazo aún sigue pendiente y será atendida en algún momento por la L2. Cuando esto ocurra, el nodo finalizará el reemplazo contestando con un mensaje de cancelación que desbloquea a la L2 sin tener que recorrer la lista de nuevo.

B. Reemplazos concurrentes (RC)

El efecto que los reemplazos compartidos tienen en el tiempo de ejecución de *ListaSimple* no se debe, principalmente, al incremento del tráfico de la red ni tampoco al incremento del tiempo que el nodo reemplazante dedica a la realización del remplazo. Esto es así debido a que el incremento en tráfico está repartido a lo largo de la ejecución de la aplicación (es decir, no se crean momentos de máxima congestión) por lo que puede ser absorbido fácilmente por la red, y a que los reemplazos no están en el camino crítico de los fallos que los inician porque todos los protocolos considerados asumen el uso de un buffer de postescritura.

Lo que más afecta al rendimiento es que la L2 permanece bloqueada mientras se realiza el reemplazo, que puede ser un tiempo largo. Mientras la L2 está bloqueada, no puede atender otras peticiones, por lo que los reemplazos de datos compartidos aumentan la latencia de todos los fallos (incluso los de otros nodos) debido a que estos tienen que esperar, una vez que su petición llega a la L2, a que ésta se desbloquee. Es decir, el hecho de que los reemplazos involucren a la L2 aumenta su contención. Hemos comprobado que este aumento de latencia afecta especialmente a los fallos de lectura.

Analizando el protocolo *ListaSimple*, nos podemos dar cuenta de que, para garantizar la ausencia de carreras y la corrección, no es estrictamente necesario que la L2 atienda todas las peticiones de manera secuencial. En particular, resulta fácil modificar el protocolo para que la L2 atienda inmediatamente las peticiones de lectura incluso aunque esté bloqueada por la realización de un reemplazo compartido siempre que el reemplazante no sea el primero de la lista (caso en el que, por otra parte, no es necesario recorrer la lista de compartidores). Esto es posible porque, como ya se ha mencionado, los nuevos compartidores se insertan siempre al principio de la lista. Una vez que la L2 ha reenviado al nodo que figure el primero en la lista la petición de reemplazo para buscar al nodo previo del que reemplaza para que actualice su puntero, cualquier inserción al principio de la lista resulta inocua ya que es imposible que el nuevo nodo fuera el predecesor del que reemplaza.

La implementación de esta mejora requiere añadir un nuevo estado intermedio en el controlador de coherencia que combina los estados intermedios que tratan los reemplazos y los fallos de lectura sobre líneas en estado S. Con esto, el protocolo nunca hará esperar a una petición de lectura debido a un reemplazo. Sólo una petición de lectura puede ejecutarse concurrentemente a un reemplazo, el resto de peticiones se ejecutan secuencialmente, y varios reemplazos o varias peticiones de lectura también se ejecutan secuencialmente.

IV. Análisis de la sobrecarga de memoria

El mayor atractivo de los protocolos con información de compartición distribuida es su gran escalabilidad en cuanto a la cantidad de memoria necesaria para almacenar la información de compartición. Al requerir menos área, mejorarán además su escalabilidad en cuanto al consumo estático de energía.

La Figura 2 refleja la sobrecarga de memoria que supone almacenar la información de compartición con respecto a la memoria usada por las cachés (L1 y L2). La cantidad de memoria necesaria por entrada de directorio en *VectorBits* crece linealmente con el número de núcleos de ejecución (almacena N bits, siendo N el número de núcleos). En cambio, *ListaSimple* es mucho más escalable ya que su crecimiento es logarítmico puesto que requiere solo un puntero ($\log_2 N$) por cada entrada de la L2, que indica el inicio de la lista, y otro puntero ($\log_2 N$) en cada entrada de la L1, que apunta al siguiente compartidor. Si tenemos en cuenta que el número de entradas de L1 será siempre mucho menor que el número de entradas de L2 (normalmente un orden de magnitud), el área vendrá determinada principalmente por el puntero en la L2. Es por esto que la sobrecarga de memoria de *ListaDoble*, que requiere dos punteros por cada bloque en la L1, es muy similar a la de *ListaSimple*.

V. Entorno de evaluación

La evaluación de este trabajo se ha realizado mediante los simuladores PIN[13] y GEMS 2.1[14], co-

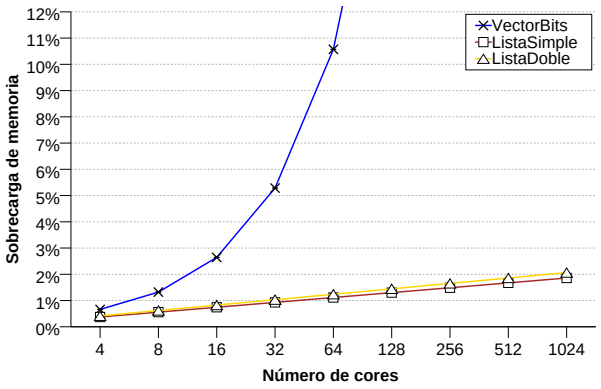

Fig.2. Sobrecarga de memoria de los protocolos evaluados.

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 ciclo
Caché L2 compartida	256KiB/celda, 16 vías
Latencia de acceso a L2	6 ciclos más latencia de red
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 de 8×8 nodos
Técnica de conmutación	Wormhole
Técnica de enrutamiento	Determinista X-Y
Multicast/Broadcast	No soportado
Tamaño de mensajes	4 flits (datos), 1 flit (control)
Tiempo de enrutamiento	1 ciclo
Tiempo de conmutador	1 ciclo
Tiempo de enlace	2 ciclos
Tamaño del buffer	6 flits
Ancho de banda	1 flit por ciclo

nectándolos de forma similar a como se propone en [15]. PIN obtiene todos los accesos a los datos accedidos por las aplicaciones, así como la sincronización entre los hilos de la application. 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 SiCoSys [16]. La arquitectura simulada corresponde a un multiprocesador en un único chip (*tiled*-CMP) 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 *BitVector*), un protocolo con un código de compartición distribuido en las cachés mediante listas enlazadas (que hemos denominado *SingleList*) y que hemos optimizado posteriormente tal y como se describe en la sección III, y un protocolo que implementa una lista doblemente enlazada (denominado *DoubleList*).

Además, hemos usado una gran variedad de aplicaciones de los conjuntos de aplicaciones SPLASH-2[17] y PARSEC 2.1[18]. *Barnes*, *Cholesky*, *FFT*, *Ocean*, *Radix*, *Raytrace*, *Volrend* y *Water-NSQ* pertenecen a SPLASH-2 y emplean los mismos tamaño de entrada recomendados en el artículo[17]. *Bodytrack*, *Canneal*, *Streamcluster* y *Swaptions* pertenecen a PARSEC 2.1 y usan tamaño de entrada *sim-medium*. Hemos tenido en cuenta la variabilidad en las aplicaciones paralelas tal y como se describe en [19]. Todos los resultados mostrados en este trabajo corresponden a la parte paralela de las aplicaciones evaluadas.

VI. Resultados

En esta sección describimos los resultados obtenidos al simular las aplicaciones utilizando los tres protocolos de coherencia mencionados anteriormente, junto con las dos optimizaciones al protocolo de lista simplemente enlazada, para una configuracion con 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, mostramos tanto el tiempo de ejecución de las aplicaciones como su consumo dinámico y estático para cada protocolo de coherencia analizado.

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 de los fallos de caché L1 normalizada con respecto a *VectorBits*. Esta latencia se muestra dividida en tiempo de espera en la L1 (*En_L1*), tiempo hasta llegar a la L2 (*Hasta_L2*), tiempo de espera en la L2 (*En_L2*), tiempo de acceso a la memoria (*Memoria*) y tiempo desde que el fallo deja la L2 hasta que se resuelve (*Hasta_L1*).

En muchos tramos la latencia se mantiene constante en todos los protocolos analizados. Sin embargo, hay dos tramos en los que la latencia se ve afectada: *En_L2* y *Hasta_L1*. Los protocolos de código de compartición distribuido, tanto *ListaSimple* como *ListaDoble*, aumentan ligeramente el tiempo de *Hasta_L1*. Esto se debe a los fallos de escritura, ya que en los protocolos basados en listas la invalidación de los compartidores se realiza en serie, mientras que en *VectorBits* se realiza en paralelo.

En el caso del tiempo *En_L2*, el protocolo *ListaSimple* sufre un incremento importante. Esto es debido a que la actualización de la lista de compartidores requiere más tiempo que la actualización de un código de compartición centralizado basado en un vector de bits. En el caso de la implementación inicial de *ListaSimple* la actualización de la lista de compartidores, ya sea por lecturas, escrituras o reemplazos, se realiza en exclusión mutua y por tanto la L2 permanece bloqueada para la línea de memoria que se está actualizando. Esto provoca una gran contención sobre todo debido a los reemplazos de líneas compartidas, que además crece con el número de núcleos que quieren actualizar la lista. Como se puede observar,

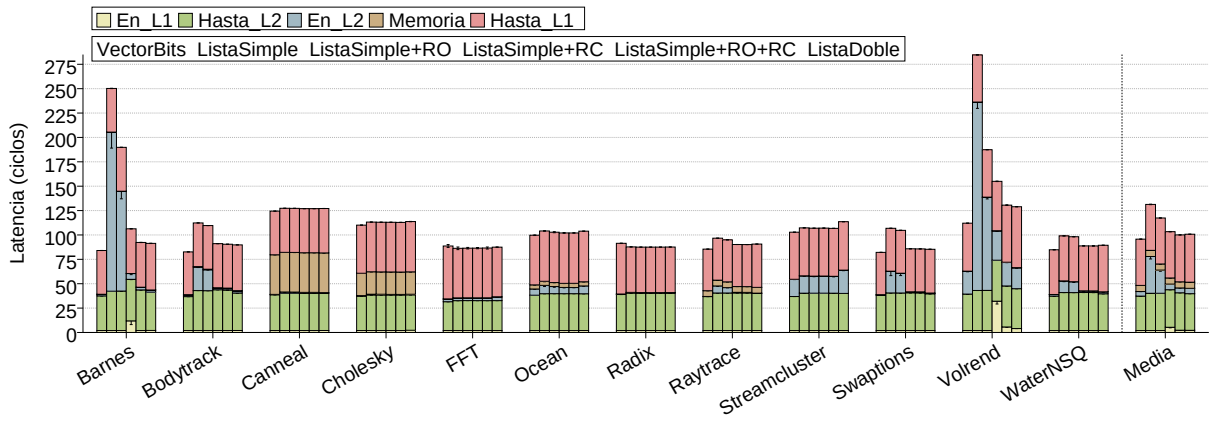

Fig.3. Latencia de los fallos de L1.

las dos técnicas propuestas en este artículo mejoran considerablemente esta latencia, dejándola en valores similares al protocolo *VectorBits*.

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, tal y como veremos más adelante. La Figura 4 muestra el tráfico que pasa por la red medido en *flits* y normalizado con respecto a *VectorBits*. 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 en estado modificado o exclusivo (*ControlReemME*) y tráfico de control debido a reemplazos de datos en estado compartido (*ControlReemS*).

Como cabe esperar, el tráfico debido a datos es similar en todos los protocolos. Sin embargo, esto no siempre sucede para los mesajes de control. En el caso del tráfico de *Control* debido a fallos de cache, *ListaSimple* requiere un tráfico de similar a *VectorBits* ya que, los fallos de lectura requieren el mismo número de mensajes y, aunque las invalidaciones de los fallos de escritura se envían en serie, el número de mensajes enviados es parecido. Pero, por contra, el tráfico de *Control* generado en *ListaDoble* se incrementa comparado con el resto de protocolos. Esto se produce porque cuando en *ListaDoble* hay un fallo de lectura y se añade un nuevo compartidor a la lista doble, se debe actualizar el puntero al nodo previo del primer comparidor, requiriendo por tanto mensajes de control extra.

La gran diferencia se aprecia en el tráfico de control debido a los reemplazos, y más en particular, para el caso de los reemplazos de bloques en estado compartido. En *VectorBits* todos los reemplazos de bloques compartidos se hacen de forma silenciosa. Sin embargo, en los protocolos distribuidos, el bloque reemplazado contiene información de compartición y, de reemplazarse sin actualizar la lista, los nodos siguientes quedarían desconectados. La actualización de la lista ante un reemplazo requiere varios mensajes de control para recorrerla hasta el nodo anterior al reemplazado. Como esta operación se realiza en serie y en exclusion mutua, es uno de los causantes del incremento de la contención en el directorio, con la consiguiente degradación en el tiempo de ejecución, como mostramos en el siguiente apartado.

Las propuestas presentadas en este artículo ayudan en gran medida a reducir el tiempo que la L2 está bloqueada debido a la actualización de la lista de compartidores, pero no tanto a la hora de reducir el tráfico en la red, ya que aunque los reemplazos oprtunistas reducen el número de mensajes debido a que evitan que se tenga que recorrer repetidamente la lista de compartidores cuando se producen varios reemplazos a la vez, esta reducción sólo resulta apreciable en *Barnes*, *Bodytrack* y *Volrend*. La técnica de reemplazos y fallos de lectura concurrentes no reduce el tráfico generado por el protocolo de listas.

C. Tiempo de ejecución

El tiempo de ejecución de las aplicaciones bajo cada protocolo depende de la latencia de los accesos a memoria. La Figura 5 muestra el tiempo de ejecución normalizado, de nuevo, respecto a *VectorBits*.

Algunas aplicaciones sufren un aumento considerable en el tiempo de ejecución, sobre todo en el caso de *ListaSimple*. Estos incrementos con respecto a *VectorBits* se producen en *Barnes*, *Bodytrack*, *Swaptions*, *Oocean*, *Raytrace*, *Streamcluster*, *Volrend* y *WaterNSQ*. Si nos fijamos de nuevo en la gráfica de la latencia de los fallos (Figura 3), podemos apreciar que, precisamente, estas son las aplicaciones para las que también aumenta el tiempo de espera en la L2 debido a contencion en la actulización de la lista.

Las dos propuestas presentadas en este trabajo ayudan a la reducción de esta latencia, y por tanto, a la reducción del tiempo de ejecución de las aplicaciones como se aprecia en la Figura 5. La combinación de las dos propuestas da como resultado un tiempo de ejecución comparable al de un protocolo con vector de bits, pero con unos requerimientos de memoria muy inferiores y más escalables. Por tanto, nuestras optimizaciones convierten a un protocolo de listas

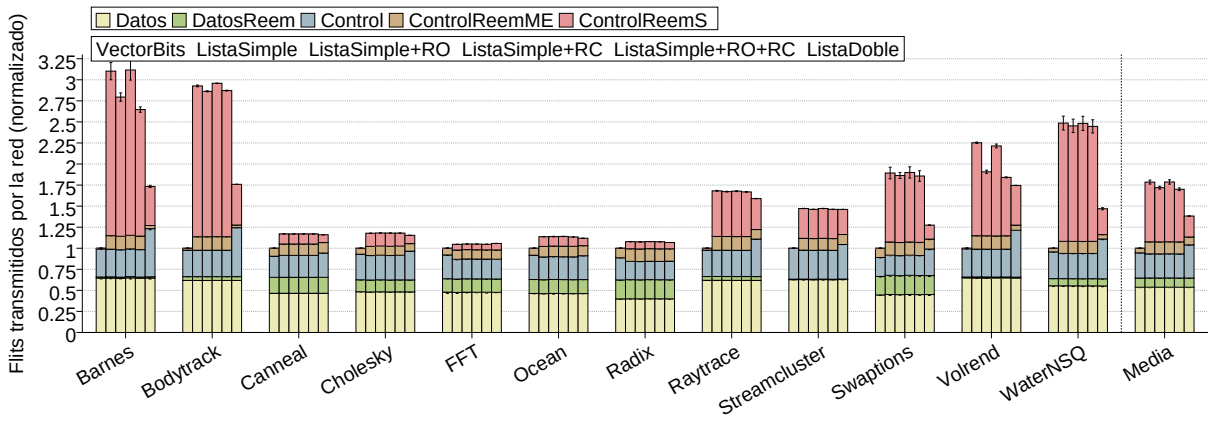

Fig.4. Tráfico en la red de interconexión.

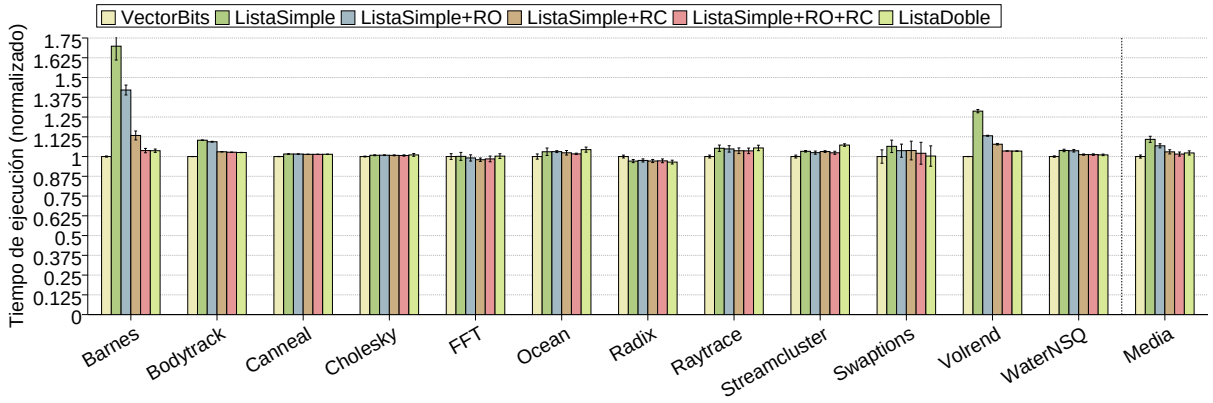

Fig.5. Tiempo de ejecución.

simplemente enlazada en una decisión de diseño muy atractiva para futuros multiprocesadores en un único chip.

D. Consumo de energía

Un aspecto muy importante del rendimiento de un multiprocesador es su consumo de energía. Esta sección analiza tanto el consumo dinámico como estático de las propuestas presentadas en este trabajo.

La Figura 6 muestra el consumo de energía dinámica de las aplicaciones para cada procolo estudiado, normalizado respecto a *VectorBits*. El consumo ha sido dividido en acceso a datos (L1 y L2), acceso a metadatos (L1 y L2) y consumo de la red de interconexión. En primer lugar podemos apreciar que el consumo a la parte de los metadatos (acceso a la parte de las etiquetas y bits de control) es despreciable. Además, el consumo por acceso a datos es muy similar en los diferentes protocolos analizados.

La diferencia está en el consumo debido al tráfico en la red. En los protocolos basados en códigos de compartición distribuidos se produce un incremento notable en el consumo dinámico. En el caso de *ListaSimple* el incremento es más pronunciado que en el caso de *ListaDoble*. La propuesta de reemplazos oportunistas ayuda a reducir ligeramente este consumo.

Pero no sólo el consumo dinámico dicta el consumo de un multiprocesador. El consumo estático representa cada vez una fracción más importante del consumo total de la energía del sistema. La Figura 7 muestra dicho consumo para las cachés de primer y segundo nivel, normalizado respecto a *VectorBits*. *VectorBits* es el protocolo que más energía estática consume, ya que la L2 posee un código de compartición de 64 bits. Los protocolos de lista simplemente enlazada reducen este consumo ya que requieren menos área para almacenar la informacón de directorio. Además como las optimizaciones propuestas en este trabajo reducen el tiempo de ejecución de una lista simple, el consumo estático durante la ejecución de las aplicaciones se reduce. De hecho, combinando las dos propuestas presentadas en este trabajo, la energía estática consumida es la menor de todos los protocolos. Esto demuestra lo adecuado que es el uso de listas simplemente enlazadas con reemplazos oportunistas y concurrentes en sistemas multiprocesador.

VII. Conclusiones

En este trabajo hemos realizado una evaluación exhaustiva de dos protocolos de coherencia basados en estructuras de directorio con códigos de compartición distribuidos. El primero, que denominamos *ListaDoble*, emplea una estructura de lista doblemente enlazada mediante dos punteros en cada una de las

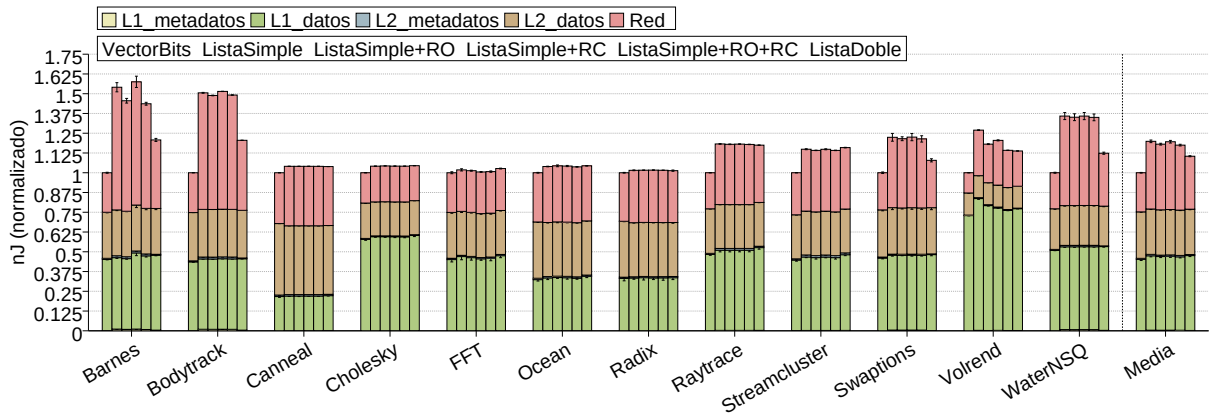

Fig.6. Consumo de energía dinámica.

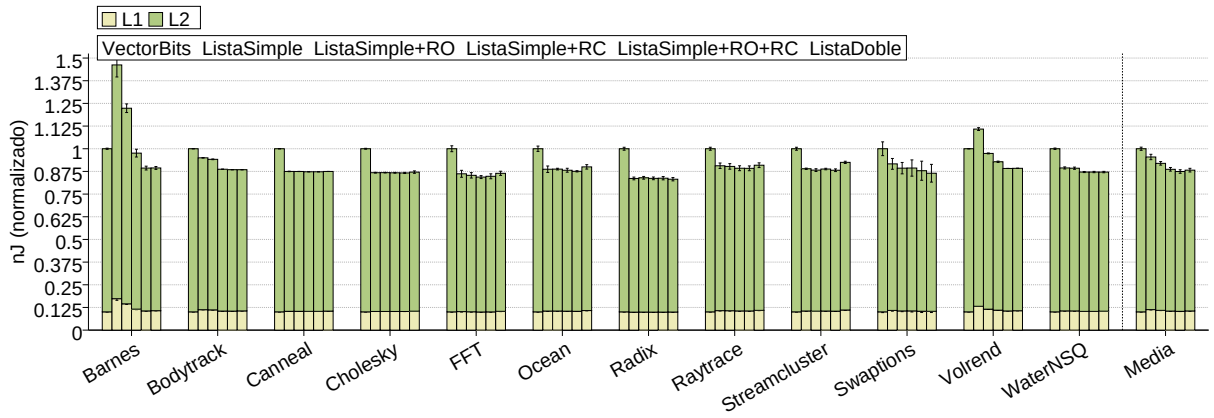

Fig.7. Consumo de energía estática (leakage).

entradas de las cachés privadas con copia de un bloque de datos. Esta implementación, que ya fue usada en la construcción de algunos multiprocesadores comerciales durante la década de los 90, no ha tenido, sin embargo y a pesar de sus ventajas, repercusión aún en el contexto de las arquitecturas multinúcleo. El segundo protocolo, que denominamos *ListaSimple*, reduce a la mitad la cantidad necesaria de punteros en las cachés privadas mediante el uso de una estructura de lista simple.

Además, también mostramos cómo con la primera de las opciones (la basada en lista doblemente enlazada) se puede alcanzar el rendimiento de la mejor alternativa para un directorio centralizado (aunque no escalable) basada en el uso de vectores de bits completos. La segunda de las opciones (la basada en lista simple), sin embargo, se ve penalizada como consecuencia del mayor costo que los reemplazos de datos compartidos tienen. Para solucionarlo, sin embargo, hemos propuesto y evaluado dos técnicas: (i) *reemplazos oportunistas (RO)*, a través de la cual se consigue aprovechar un reemplazo en curso para que otras cachés con reemplazos pendientes al mismo bloque de dato puedan realizarlos conforme el primero progresa, y (ii) *reemplazos concurrentes (RC)*, a través de la cual el nodo origen puede seguir atendiendo los fallos de lectura mientras está teniendo lugar un reemplazo. A través de simulaciones detalladas de una arquitectura con 64 núcleos de procesamiento, demostramos que haciendo uso de estas dos técnicas, la versión basada en listas simples consigue igualar el rendimiento de la versión basada en listas doblemente enlazadas, y resulta, por lo tanto, preferible dado el menor costo de implementación que tiene.

A la vista de los resultados obtenidos, creemos que estos protocolos de directorio basados en listas representan una alternativa más viable que los protocolos actuales, basados en el uso de códigos de compartición centralizados, de cara a la consecución de estructuras de directorio escalables que puedan hacer posible la implementación de arquitecturas multinúcleo con cientos de núcleos de procesamiento integrados.

Agradecimientos

Este trabajo ha sido co-financiado por el Ministerio de Economía y Competitividad (MINECO) y la Comisión Europea FEDER mediante el proyecto "TIN2012-38341-C04-03" y por la Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia mediante el proyecto Jóvenes Líderes en Investigación "18956/JLI/13".

Referencias

[1] J.M. Tendler, J.S. Dodson, J.S. Fields, H.Le, and B.Sinharoy, "POWER4 system microarchitecture," *IBM*

Journal of Research and Development, vol. 46, no. 1, pp. 5–25, Jan. 2002.

[2] David Kanter, "Knights landing details," *Real World Technologies 2014*, Jan. 2014.

[3] Matthew Mattina, "Architecture and performance of the tilera TILE-Gx8072 manycore processor," Invited presentation at 21st HotInterconnects Symp., 2013.

[4] Shekhar Borkar, "Thousand core chips: a technology perspective," in *44th Design Automation Conference (DAC)*, June 2007, pp. 746–749.

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

[6] ManuelE. Acacio, José González, JoséM. García, and José Duato, "A two-level directory architecture for highly scalable cc-NUMA multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 16, no. 1, pp. 67–79, Jan. 2005.

[7] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 169–180.

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

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

[10] 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.

[11] 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 Symp. on Parallel Processing (IPPS)*, Apr. 1997, pp. 8–17.

[12] Ricardo Fernández-Pascual, Alberto Ros, and ManuelE. Acacio, "Characterization of a list-based directory cache coherence protocol for manycore cmps," in *3rd Workshop on On-chip Memory Hierarchies and Interconnects (OMHI 2014)*, Aug. 2014, pp. 254–265.

[13] 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.

[14] 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.

[15] 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.

[16] Valentín Puente, JoséA. Gregorio, and Ramón Beivide, "SICOSYS: An integrated framework for studying interconnection network in multiprocessor systems," in *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, Jan. 2002, pp. 15–22.

[17] 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.

[18] Christian Bienia, Sanjeev Kumar, JaswinderPal Singh, and Kai Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[19] 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.