Operaciones de Carga Fuera de Orden y sin Especulación en TSO

Alberto Ros[1], Trevor E. Carlson, Mehdi Alipour y Stefanos Kaxiras[2]

Resumen— **En el modelo de consistencia TSO (Total Store Order) implementado en la mayoría de los procesadores del mercado, no se permite la ejecución fuera de orden de las operaciones de carga (*load*) en un mismo hilo, sino que se debe garantizar el orden en que aparecen en el código. Se debe garantizar, por tanto, el llamado orden load→load. Con el fin de obtener un alto rendimiento, los procesadores actuales permiten ejecutar las operaciones de carga fuera de orden de forma especulativa. Si este reordenamiento es percibido por otros núcleos (*cores*), por ejemplo mediante una invalidación provocada por una escritura, las operaciones de carga son canceladas y ejecutadas de nuevo.**

Este artículo muestra por primera vez que no es necesario cancelar y ejecutar de nuevo las operaciones de carga que son percibidas por otros cores mientras están fuera de orden. En cambio, el reordenamiento puede ocultarse a los otros cores mediante extensiones en el protocolo de coherencia de cachés, por ejemplo, retrasando la escritura que generó la invalidación. La principal consecuencia de ésto es que podemos dar por válido todo valor leído por una carga, aunque ésta no se haya ejecutado en orden. Nuestra evaluación muestra que el coste de retrasar las escrituras cuando un reordenamiento va a ser percibido por otro core es mínimo. Además, aplicando este concepto a técnicas de retirada de instrucciones fuera de orden, se puede mejorar el tiempo de ejecución de las aplicaciones en un 10,2 % comparado con una retirada fuera de orden que no use el protocolo propuesto.

Palabras clave— **Arquitecturas multicore, protocolo de coherencia de cachés, modelo de consistencia, especulación, reordenamiento de lecturas, commit fuera de orden.**

I. Introducción

El modelo de consistencia de memoria de un procesador define el comportamiento de las operaciones de memoria, carga (*load*) y almacenamiento (*store*), ejecutadas por el procesador teniendo en cuenta el *orden* en el que aparecen en el código.

En el modelo de consistencia de memoria TSO (Total Store Order) implementado, por ejemplo, en los procesadores x86 fabricados por Intel y AMD [1] las reglas de reordenamiento de operaciones de memoria son las siguientes: load→load, store→store y load→store. Esto significa, por ejemplo, que si una carga $c1$ aparece en un programa antes que otra carga $c2$, $c1$ debe ser ejecutada antes que $c2$, por la regla load→load.

Sin embargo, con el fin maximizar el rendimiento, los procesadores superescalares con planificación dinámica ejecutan instrucciones fuera de orden y pueden transgredir *especulativamente* las reglas de orden del modelo de consistencia de memoria. Si estos reordenamientos son percibidos por otro núcleo (*core*), las operaciones, posiblemente ejecutadas incorrectamente son canceladas, volviendo a un estado previo donde se garanticen las reglas del modelo de consistencia, y ejecutadas de nuevo.

Inicialmente x=0 e y=0.

Core 0	Core 1
`ld ra,y`	`st x,1`
`ld rb,x`	`st y,1`

`ra==1` y `rb==0` es un resultado no válido en TSO.

TABLA I
Código de ejemplo.

Ejemplo: En el código que se muestra en la Tabla I, un core ejecuta dos operaciones de carga en registros (`ld`) mientras que el otro ejecuta dos operaciones de almacenamiento (`st`).

Supongamos que cuando el *Core* 0 intenta obtener el valor de `y`, éste no se encuentra en caché, por lo que la primera carga requiere una latencia alta para resolverse. La ejecución fuera de orden permite la ejecución desordenada de dos cargas de forma especulativa y, por tanto, el *Core* 0 puede ejecutar la segunda carga antes de que se resuelva la primera. Supongamos ahora que la segunda carga acierta en caché y lee el valor `0` de la dirección `x` en el registro `rb`. En este momento la segunda carga se considera *M-especulativa* [2] hasta que se resuelva la carga anterior (todas las anteriores en el caso general).

Imaginemos ahora que el *Core* 1 ejecuta las dos operaciones de almacenamiento y asigna tanto a `x` como a `y` el valor `1`. Posteriormente, el *Core* 0 realiza la carga de `y` en `ra` guardando el valor `1` escrito por el *Core* 1. El resultado `ra==1` y `rb==0` no es un resultado válido en TSO, ya que este resultado se puede obtener sólo si el *Core* 0 viola el orden load→load o el *Core* 1 viola el orden store→store.

Para solucionar esto, los procesadores actuales que implementan TSO cancelan y re-ejecutan las cargas M-especulativas cuando reciben una operación de invalidación de memoria a la dirección de donde la carga obtuvo el dato, es decir, cuando el reordenamiento ha sido detectado por otro core. En el ejemplo anterior, cuando el *Core* 1 ejecuta `st x,1` éste no tiene permiso de escritura sobre el bloque donde está `x` ya que ha sido leído previamente por el *Core* 0 y el protocolo de coherencia se encarga de enviar una invalidación al *Core* 0. Al recibir la invalidación sobre

[1]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: aros@ditec.um.es.

[2]Dept. of Information Technology, Uppsala University, e-mail: {trevor.carlson, mehdi.alipour, stefanos.kaxiras}@it.uu.se.

la carga M-especulativa, el *Core* 0 cancela la carga junto con todas las instrucciones que dependan de ella y las vuelve a ejecutar, obteniendo esta vez el valor correcto de `1` en `rb`.

Motivación: Idealmente, nos gustaría contar con una solución en la que una carga de memoria reordenada especulativamente (y sus instrucciones dependientes) no tengan que ser canceladas ante la llegada de una acción de coherencia a la vez que se garantize el modelo de consistencia del sistema. El beneficio, en este caso, sería eliminar la cancelación y re-ejecución. Sin embargo, esto no ocurre frecuentemente ya que en muy pocas ocasiones los reordenamientos de las cargas son detectados por otros cores. La importancia de esta propiedad radica en aprovechar el hecho de que las cargas no pueden ser canceladas una vez resueltas. En concreto analizaremos como aprovechar esta propiedad para el caso de la retirada (o *commit*) de instrucciones fuera de orden [3].

Propuesta: Este trabajo propone una nueva técnica para permitir ejecutar cargas de memoria en un orden no permitido por el modelo de consistencia sin necesidad de tener que re-ejecutar las cargas y sin violar el modelo de consistencia.

La idea en la que se basa nuestra propuesta es simple. El orden load→load se puede violar cuando un core percibe una carga ejecutada por otro core que no se ha resuelto en orden. Por tanto, proponemos que cuando una operación de almacenamiento está a punto de ver una violación del orden load→load debido al acceso a una carga desordenada en otro core, la capa de coherencia ocultará el desordenamiento retrasando la operación de almacenamiento hasta que las cargas queden ordenadas.

Para ello, es necesario extender el protocolo de coherencia para que permita retrasar los almacenamientos. Ésta es una operación frecuente en los procesadores actuales a nivel del procesador, ya que emplean un búfer de escritura donde se guarda temporalmente el valor de las operaciones de almacenamiento hasta que la memoria da permiso de escritura. Nosotros, sin embargo, implementamos este retraso a nivel del protocolo de coherencia, sólo si la invalidación implica el dato leído por una carga fuera de orden. Prestamos además en este trabajo especial atención a que el retraso de las escrituras no genere situaciones de bloqueo (*deadlock* o *livelock*), y comentamos la solución en cada caso.

Resultados: Mediante simulación mostramos que el retraso de las escrituras no afecta prácticamente al rendimiento del procesador, ya que no ocurre frecuentemente y además la latencia es ocultada por el búfer de escritura. Además, cuando empleamos commit fuera de orden con la posibilidad de retirar cargas desordenadas mejoramos el rendimiento del procesador en un 10,2 % en promedio y hasta un 28,3 % con respecto a un commit fuera de orden que no puede retirar cargas desordenadas.

II. Estado del arte

Existen soluciones en la literatura para retirar cargas especulativas fuera de orden. La más simple es relajar el modelo de consistencia de memoria de tal forma que acepte reordenamientos en las cargas [4], [3], [5]. Sin embargo, esta solución restringe el modelo de consistencia de memoria, excluyendo a la gran mayoría de procesadores del mercado.

Otras soluciones consisten en añadir mecanismos de *rollback* fuera del core con el fin de volver a un estado válido en caso de violación del modelo de consisntecia [6], [5]. Esta solución tiene un coste y una complejidad significativos y puede tener complicaciones para revertir el estado de la máquina ante operaciones de entrada/salida.

Un opción alternativa es esperar a realizar el commit fuera de orden hasta que todas las instrucciones de memoria anteriores estén ejecutadas y por lo tanto se puede garantizar que el reordenamiento no ha sido visto por otro core [3]. Esta opción tiene la desventaja de mitigar el beneficio del rendimiento del commit fuera de orden en el caso común de los reordenamientos para evitar el caso poco frecuente.

Por último, se puede restringir la arquitectura para reducir el tiempo de espera en la opción anterior implementando una red globalmente ordenada (por ejemplo, Gigaplane [7]) y no realizar el commit fuera de orden hasta que todas las instrucciones de memoria anteriores estén ordenadas en la red [3], [8], [9]. Esta opción es una mejora con respecto a la anterior, siempre que se limite la arquitectura a buses ordenados globalmente o a redes ópticas que proporcionen coherencia atómica [9]. No obstante, la mejora es que sólo tenemos que esperar a que las operaciones de memoria anteriores se ordenen, no a que se realicen. Desafortunadamente, incluso con esta mejora, no hay nada que se pueda hacer ante un reordenamiento que involucre direcciones de operaciones de memoria no resueltas.

Hasta donde sabemos, no se ha propuesto hasta la fecha una solución que permita el commit de cargas fuera de orden, que no requiera especulación y que además pueda aplicarse a cualquier arquitectura, incluyendo redes de interconexión no ordenadas y coherencia basada en directorios, las opciones arquitectónicas predominantes hoy en día. Además, incluso considerando redes ordenadas y coherencia basada en snooping, no existe ninguna solución para hacer commit de una carga cuando hay direcciones de operaciones de memoria anteriores no resueltas.

III. Ahora me ves, ahora no me ves

Nuestro objetivo es que una vez que una carga sea resuelta por el procesador, no pueda ser cancelada debido a una violación del modelo de consistencia. A la vez, queremos que el procesador pueda reordernar las cargas para optimizar su rendimiento. Para ello, tenemos que evitar que ocurra una la violación del modelo de consistencia.

Tomemos de nuevo el ejemplo de la Tabla I. Supongamos que la carga `ld ra, y` falla en caché, mientras

que la carga `ld rb, x` acierta en caché y guarda un `0` en `rb`. Como la carga `ld rb, x` está ya resuelta, no vamos a permitir que se cancele. De esta forma, se podría, por ejemplo, hacer commit fuera de orden de dicha instrucción.

Sin la opción de cancelar la carga `ld rb, x`, la única opción que nos queda para no violar el modelo de consistencia es forzar a que la carga anterior `ld ra, y` lea el valor `0` en `ra`. ¿Es esto posible? La respuesta es sí. La operación `ld ra, y` tiene que leer el valor de memoria antes de la operación `st y, 1` escriba el `1`.

Puede ocurrir que la operación `ld ra, y` se resuelva naturalmente anter que `st y, 1`. En este caso no se violaría la consistencia y la ejecución sería correcta. Sin embargo también puede ocurrir que la operación `st y, 1` se ejecute antes. En este caso, tenemos que evitar que se escriba un `1` en la dirección de memoria de y antes que la carga `ld ra, y` obtenga el valor `0`.

Para conseguir que el valor leido por cada carga se ajuste al orden load→load, proponemos modificar el protocolo de coherencia de caches. En un procesador actual, la invalidación generada por la escritura de la operación de almacenamiento en el *Core* 0 provocaría la re-ejecución de la operación de carga. Nuestra solución consiste en retrasar la escritura hasta que la operación de carga deje de ser M-especulativa, es decir, que todas las operaciones de carga anteriores se hayan resuelto *viendo* siempre en memoria valores anteriores a la escritura (en el caso de la Tabla I implicaría que el carga ha leido el valor `0`).

A. Bloqueo de escrituras

En los procesadores actuales, cuando un fallo de escritura emite una invalidación a una copia en caché, el procesador comprueba si esa copia ha sido leída por una operación de carga y ésta está en estado M-especulativo. En tal caso, cancela el efecto de la operación de carga y las instrucciones que dependen de ella, elimina el bloque de la caché y confirma (*ack*) al core que está realizando la escritura acerca de la invalidación del bloque. Cuando el core tiene todas las confirmaciones, obtiene el permiso de escritura y puede escribir el nuevo valor.

En nuestro protocolo de coherencia, al que llamamos *WritersBlock*, la escritura se bloquea en el caso en el que la copia a invalidar haya sido leída por una operación de carga desordenada (sin que otras cargas anteriores estén resueltas). El procesador no cancela la carga a pesar de invalidar el bloque de caché, pero en su lugar manda un mensaje de *nack* al directorio. El directorio transiciona el bloque al estado *WritersBlock* al recibir el mensaje de *nack*.

Esto tiene dos implicaciones. Primero, el core que pretende realizar la escritura no obtiene permiso para escribir, y por tanto el valor del bloque no se actualiza. Segundo, el directorio en estado *WritersBlock* bloquea todas las escrituras de otros cores a este bloque, garantizando que el bloque no sea escrito por ningún otro core.

En estas condiciones, nos aseguramos de que las cargas que están sin resolver en el core que sufrió la invalidación, se resuelvan con un valor válido según el orden load→load, siempre y cuando el orden store→store se mantenga. La pregunta que nos surge ahora es ¿cuándo debe un bloque salir del estado *WritersBlock*? La respuesta es cuando todas las operaciones de carga anteriores a la carga no ordenada estén resueltas, es decir, cuando la carga esté ordenada. En ese momento, el core deberá emitir un mensaje de *ack* al directorio. El directorio lo reenviará al core que inició la escritura y continúa esperando el permiso[1]. Éste último escribirá el nuevo valor en memoria y mandará un mensaje de desbloqueo al directorio. Cuando el directorio recibe este mensaje saca al bloque del estado *WritersBlock*.[2].

B. Evitando deadlocks (I)

En el protocolo descrito anteriormente es fácil mostrar como al retrasar las escrituras dejándolas bloqueadas en el directorio temporalmente se pueden producir deadlocks. Imagine que estamos en una situación en la que el bloque b está en estado *WritersBlock* en el directorio. La carga no ordenada que ha provocado que el bloque llegue a este estado referencia a la dirección $b1$ que pertenece, consecuentemente, al bloque b. Imagine también que en este mismo core, existe una carga anterior que no se ha resuelto todavía y que leerá el valor de la dirección $b2$, que también pertenece al bloque de memoria b. Si el procesador emite la petición de lectura para la carga $b2$, esta quedará bloqueada en el directorio al estar pendiente la resolución del permiso de escritura de b. El problema es que hemos creado un ciclo y, por tanto, un deadlock: la escritura no se puede resolver hasta que la carga a $b1$ se ordene; la carga a $b1$ no se orderna hasta que no se resuelva la carga a $b2$; la carga a $b2$ no se puede resolver hasta que la escritura no se resuelva.

La solución a este problema consiste en romper el ciclo. Proponemos, por tanto, resolver peticiones de lectura mientras estamos en estado *WritersBlock*. En general, como veremos más adelante, la solución para evitar cualquier deadlock en el protocolo *WritersBlock* radica en garantizar que las operaciones de carga se puedan resolver siempre.

En particular, en estado *WritersBlock* la caché compartida tiene siempre una copia válida del dato (debe ser enviada por el core que ha sufrido la invalidación en el caso en que la copia fuera exclusiva). Ante una petición de lectura el directorio obtiene el dato de la caché compartida, lo envía al peticionario y espera la confirmación de la recepción del dato para poder procesar otra lectura[3].

[1]El core invalidado no guarda la información del peticionario, así que tiene que enviarle el *ack* con indirección a través del directorio.

[2]Para más información acerca de las posibles condiciones de carrera en estado *WritersBlock* véase [10]

[3]Aunque se podrían procesar varias lecturas a la vez, consideramos sólo el caso de una lectura tal y como se hace en el protocolo base con el que nos comparamos (Sección V).

C. Evitando livelocks

Si una petición de lectura puede resolverse mientras el directorio está en estado *WritersBlock*, una vez que todas las invalidaciones se han realizado, y esta petición de lectura guarda la copia del bloque en caché, sería necesario enviar de nuevo las invalidaciones para los nuevos bloques cacheados. En este caso, es posible que el bloque nunca salga del estado *WritersBlock* ya que las nuevas invalidaciones pueden dar lugar a nuevos bloqueos.

La solución a este caso es el uso de lecturas no cacheables, de tal forma que una vez invalidadas las copias no se tengan que hacer más invalidaciones. Además, un dato no podrá ser leído por una carga si ésta no está ordenada, ya que éste sería un dato anterior a la escritura y tendría que ser invalidado por la escritura bloqueada, pero ya no se permiten más invalidaciones. Por tanto, para evitar livelocks, las peticiones de lectura que encuentran el bloque en estado *WritersBlock* se resuelven siempre con datos no cacheables, y además si la carga no está ordenada (hay cargas anteriores sin resolver), esta carga no puede resolverse usando el dato, sino que tendría que emitir la petición de lectura de nuevo.

D. Evitando deadlocks (II)

Incluso en el caso en el que se permitan resolver lecturas ante escrituras bloqueadas en el directorio, existe otra posibilidad de deadlock. Esto ocurre en el caso en el que una carga no ordenada y resuelta, no pueda dejar de ser no ordenada. Este caso existe si una carga anterior no puede lanzar la petición de lectura porque haya una petición de escritura (anterior o posterior) en trámite en el MSHR (miss status holding register).

Para solucionar este deadlock simplemente hay que permitir que las lecturas se puedan emitir y puedan reservar una entrada en el MSHR incluso en el caso en que haya un permiso de escritura ya solicitado para el mismo bloque. Esto no es necesario en los procesadores actuales ya que la escritura se resolverá siempre sin posibilidad de bloqueos, obteniendo el permiso de lectura junto con el de escritura, y resolviendo por tanto la carga. En particular, sólo permitimos que una carga pueda emitir la lectura estando una petición de escritura en vuelo si la carga es no M-especulativa, es decir, todas las cargas anteriores han sido resueltas. Esto es suficiente para garantizar la ausencia de deadlocks a la vez que minimiza el tráfico en la red.

E. Demostración de ausencia de deadlocks

En el protocolo de coherencia propuesto se puede dar la siguiente cadena de dependencias. Una operación de almacenamiento no se puede resolver hasta que no se tenga permiso de escritura. El permiso de escritura no se puede dar si éste implica la invalidación del bloque leído por una carga no ordenada. La carga no ordenada no se ordenará hasta que todas las cargas anteriores se resuelvan.

TABLA II
Parámetros del sistema.

Parámetros del procesador	
Ancho de emisión y retirada	4 instrucciones
Cola de instrucciones (IQ)	16 / 32 / 60 entradas
Búfer de re-ordenación (ROB)	32 / 128 / 192 entradas
Cola de *loads* (LQ)	10 / 48 / 72 entradas
Cola de *stores* (SQ)	16 / 36 / 42 entradas
Búfer de escritura (SB)	16 / 36 / 42 entradas
Tabla de lockdown (LDT)	32 entradas
Parámetros de la memoria	
Caché L1 (privada)	32 KB, 4 vías, 4 ciclos
Caché L2 (privada)	128 KB, 8 vías, 12 ciclos
Banco de caché L3 (compartida)	1024 KB, 8 vías, 35 ciclos
Tiempo de acceso a memoria	160 ciclos
Parámetros de la red	
Topología y Encaminamiento	Malla 2-D (16×8), X-Y
Tamaño de flit	16 bytes
Tamaño de mensaje en flits	5 (datos), 1 (control)
Latencia de un salto de red	6 ciclos

Siempre hay una carga en el procesador que es la carga más antigua no resuelta. Llamamos a esta carga *fuente de especulación*. Si garantizamos que esta carga nunca se pueda bloquear, podremos decir que *WritersBlock* no tiene dealocks. En efecto, esta carga siempre se puede resolver y obtener el dato, incluso si no guarda en caché el dato (por falta de recursos, o por garantizar la ausecia de livelock). Además el valor leído por esta carga será siempre válido, ya que está ordenada (todas las anteriores están resueltas). De este modo, podemos afirmar que *WritersBlock* es un protocolo libre de deadlocks.

IV. Commit fuera de orden

La principal ventaja de *WritersBlock* es que ofrece la garantía de que una carga resuelta nunca podrá ser cancelada por una violación del modelo de consistencia. Una de las aplicaciones de esta propiedad es la eliminación de la sexta condición para retirada de instrucciones fuera de orden propuesta por Bell y Lipasti [3], que impide retirar fuera de orden cargas M-especulativas. Gracias a *WritersBlock* esta restricción se puede eliminar obteniendo un mayor grado de instrucciones retiradas fuera de orden, con su correspondiente liberación de recursos (cola de instrucciones de carga –LQ– y búfer de re-ordenación –RoB–).

Lo único que tenemos que tener en cuenta cuando retiramos fuera de orden de una carga desordenada es que es necesario apuntar la dirección de esta carga en una estructura, la cual llamamos *tabla de lockdown* (LDT), mientras esté desordenada. En el caso de que una invalidación se reciba durante este periodo, se enviará un *nack* al directorio.

Necesitamos saber cuándo una carga deja de estar desordenada, es decir, cuando ésta se convierte en fuente de especulación. Para ello reservamos espacio para unos punteros en la LQ. Estos punteros apuntan a la entrada en la tabla de lockdown correspondientes a la dirección de la carga. Los punteros avanzan hacia la cabeza de la cola conforme se van retirando las cargas. Es posible, por tanto, que haya varios punteros en la misma entrada de la LQ. Cuando el puntero llega a la posición donde se encuentra la fuente de especulación, la entrada se elimina de

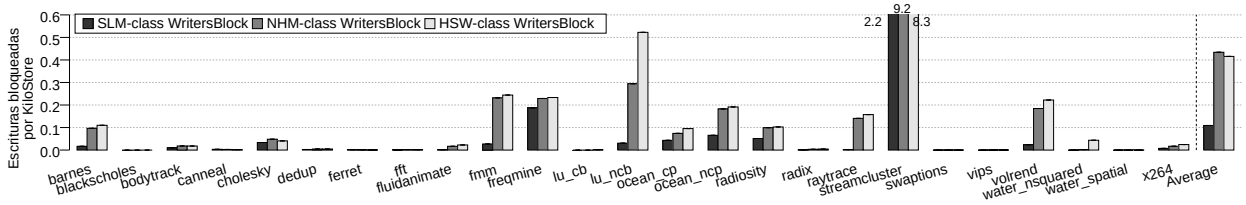

Fig. 1
Escrituras bloqueadas por KiloStore en *WritersBlock*

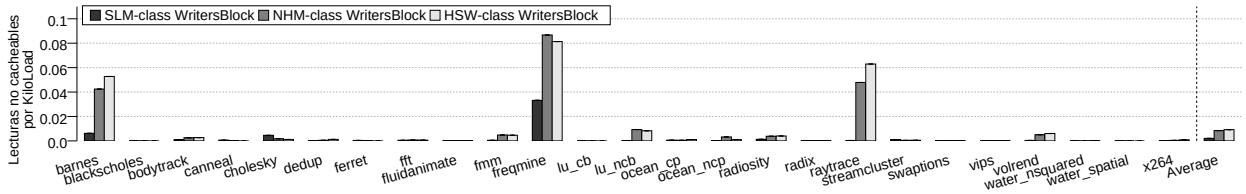

Fig. 2
Lecturas no cacheables por KiloLoad en *WritersBlock*

la tabla de lockdown (y los punteros de la LQ), enviando un *ack* en caso de que hubiera recibido una invalidación durante este tiempo.

V. Entorno de evaluación

Nuestra infraestructura de simulación esta basada en el simulador GEMS [11], que ofrece un modelo de tiempo detallado de la jerarquía memoria, conectada a un procesador de fuera de orden x86 que proporciona un modelo de consistencia TSO y es alimentado por datos generados por Sniper [12]. La red de interconexión está modelada con GARNET [13]. Para la evaluación utilizamos aplicaciones de las *suites* SPLASH-3 [14] y PARSEC 3.0 [15], con tamaño de entradas *simsmall* y mostramos los resultados para la región de código paralela.

Simulamos un procesador con 16 cores fuera de orden, y 16 bancos de caché L3 (compartida). El resto de parámetros se muestran en la Tabla II. Los parámetros con varios valores corresponden a tres configuraciones de procesadores que evaluamos con el fin de hacer un análisis de sensibilidad de *WritersBlock*. Estos tres parámetros corresponden a las configuraciones de los procesadores de bajo consumo Silvermont (*SLM-class*) y a los de mayor rendimiento Nehalem (*NHM-class*) y Haswell (*HSW-class*), respectivamente.

WritersBlock extiende al protocolo basado en directorio (estados MESI) proporcionado por GEMS con el soporte para el bloqueo de escrituras y entrega de copias de lectura no cacheables. El modelo del procesador soporta retirada de instrucciones fuera de orden como en Bell y Lipasti [3], salvo que no tenemos en cuenta la condición relacionada con las excepciones.

VI. Resultados

El propósito de esta evaluación es mostrar, en primer lugar, que el protocolo de coherencia *WritersBlock* no introduce sobrecarga alguna en el rendimiento, ya que las escrituras no se suelen retrasar y que el número de copias no cacheables enviadas también es bajo. Con esto, evaluamos las ventajas de *WritersBlock* cuando se emplea un procesador capaz de retirar instrucciones fuera de orden.

A. Escrituras bloqueadas

En *WritersBlock*, cuando una invalidación encuentra una carga M-especulativa, la solicitud de escritura se retrasa hasta que todas las operaciones de carga anteriores se ejecuten. Si el número de escrituras bloqueadas es alto, la latencia de las escrituras puede aumentar, afectando el rendimiento del procesador. La Fig. 1 muestra el número de peticiones de escritura bloqueadas por cada mil operaciones de almacenamiento (KiloStore) al variar el tipo de core. Los cores NHM-class y HSW-class sufren más peticiones de escritura bloqueadas debido a que pueden poseer en ejecución un mayor número de instrucciones de carga, pero el ratio es aceptable: 0,4 escrituras bloqueadas por mil operaciones de almacenamiento, siendo en el peor caso (*streamcluster*) menor que 1.

B. Lecturas no cacheables

Cuando las peticiones de escritura están en estado *WritersBlock*, los fallos de lectura se resuelven con datos no cacheables que pueden ser utilizados por la carga en caso de estar ordenada, es decir, si la carga es la fuente de especulación. Si el protocolo responde a muchas peticiones con datos no cacheables, el número de fallos de caché puede incrementarse y, por tanto, empeorar el rendimiento del procesador.

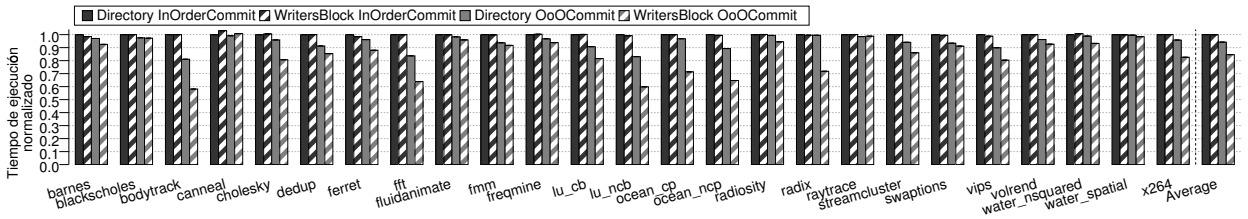

Fig. 3
Tiempo de ejecución normalizado

La Fig. 2 muestra el número de respuestas no cacheables por mil instrucciones de carga (KiloLoad). Una vez más, un procesador más agresivo tiene mas cargas en ejecución, lo que implica un mayor número de lecturas no cacheables. Sin embargo, el número de estas respuestas es muy bajo: 0,1 respuestas no cacheables por KiloLoad de media, siendo en el peor caso (*freqmine*) menor que 0,9.

C. Tiempo de ejecución

La Fig. 3 muestra el tiempo de ejecución para un protocolo de directorio y *WritersBlock* con y sin retirada de instrucciones fuera de orden. Las dos primeras barras muestran que *WritersBlock* no perjudica el rendimiento del protocolo de coherencia. Las otras dos barras muestran que *WritersBlock* puede mejorar el commit fuera de orden en un 10,2 % de media y hasta un 28,3 % al compararlo con un commit fuera de orden en un protocolo de directorio y hasta 15,4 % de media respecto a un commit en orden.

VII. Conclusiones

Este artículo presenta un protocolo de coherencia de caché que puede garantizar que una carga ejecutada fuera de orden no tenga que ser cancelada y re-ejecutada debido a una violación en el modelo de consistencia. En concreto, el valor cargado siempre respetará el orden load→load. Como consecuencia, las cargas no ordenadas y sus instrucciones dependientes pueden ser retiradas fuera de orden, mejorando el rendimiento de las técnicas de commit fuera de orden actuales.

Agradecimientos

Este trabajo es el resultado de la estancia 19981/EE/15 financiada por la Fundación Séneca–Agencia De Ciencia y Tecnología de la Región de Murcia bajo el programa "Jiménez De la Espada" para la movilidad, cooperación e internacionalización. Este trabajo ha sido co-financiado por el Ministerio de Economía y Competitividad (MINECO) y la Comisión Europea FEDER mediante el proyecto "TIN2015-66972-C5-3-R".

Referencias

[1] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, July 2010.

[2] Yuelu Duan, David Koufaty, and Josep Torrellas, "Scsafe: Logging sequential consistency violations continuously and precisely," in *22th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 249–260.

[3] Gordon B. Bell and Mikko H. Lipasti, "Deconstructing commit," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004, pp. 68–77.

[4] Linley Gwennap, "Digital leads the pack with 21164," *Microprocessor Report*, vol. 8, no. 12, pp. 249–260, Sept. 1994.

[5] Rafael Ubal, Julio Sahuquillo, Salvador Petit, Pedro Lopez, and José Duato, "Vb-mt: Design issues and performance of the validation buffer microarchitecture for multithreaded processors," in *16th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2007, p. 429.

[6] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, June 1997, pp. 199–210.

[7] Ashok Singhal, David Broniarczyk, Fred Cerauskis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, and Erik Hagersten, "Gigaplane^{TM}: A high performance bus for large smps," in *HotInterconnects Symp. IV*, Aug. 1996, pp. 41–52.

[8] Salvador Petit Marti, Julio Sahuquillo Borras, Pedro Lopez Rodriguez, Rafael Ubal Tena, and Jose Duato Marin, "A complexity-effective out-of-order retirement microarchitecture," *IEEE Transactions on Computers (TC)*, vol. 58, no. 12, pp. 1626–1639, Dec. 2009.

[9] Dana Vantrease, Mikko H. Lipasti, and Nathan Binkert, "Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 132–143.

[10] Alberto Ros, Trevor E. Carlson, Medhi Alipour, and Stefanos Kaxiras, "Non-speculative load-load reordering in tso," in *44nd Int'l Symp. on Computer Architecture (ISCA)*, June 2017.

[11] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sept. 2005.

[12] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[13] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. 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.

[14] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[15] Christian Bienia, *Benchmarking Modern Multiprocessors*, Ph.D. thesis, Princeton University, Jan. 2011.