CONSISTENCIA DE EJECUCIÓN: UNA PROPUESTA NO CACHE COHERENTE

Rafael B. García Jorge R. Ardenghi

Laboratorio de Investigación en Sistemas Distribuidos (LISiDi)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur – Bahía Blanca, Argentina
T.E.: +54 291–4595135 Fax: +54 291–4595136
{rbg,jra}@cs.uns.edu.ar

Resumen

La presencia de uno o varios niveles de memoria cache en los procesadores modernos, cuyo objetivo es reducir el tiempo efectivo de acceso a memoria, adquiere especial relevancia en un ambiente multiprocesador del tipo DSM dado el mucho mayor costo de las referencias a memoria en módulos remotos. Claramente, el protocolo de coherencia de cache debe responder al modelo de consistencia de memoria adoptado. El modelo secuencial SC, aceptado generalmente como el más natural, junto a una serie de modelos más relajados como consistencia de procesador PC, release RC, y más recientemente Java, asumen coherencia de cache. Existen, aunque en proporción mucho menor, otros modelos como el Dag y el location consistency LC que prescinden del requerimiento de coherencia. En este trabajo, analizadas las limitaciones que impone a nivel de hardware y software la coherencia, formulamos un nuevo modelo no cache coherente y un protocolo eficiente de cache para soportarlo. Este modelo, al cual referiremos como consistencia de ejecución EC, permite una ejecución secuencialmente consistente con programas paralelos libre de carrera, *data race free*, y en los casos de operaciones asincrónicas posibilita un comportamiento asimilable al del modelo Slow, lo cual lo tornaría válido para aplicaciones no sincronizadas.

Palabras Clave: DSM Memoria Compartida Distribuida, Modelos de Consistencia de Memoria, Coherencia de Cache.

VI Workshop de Procesamiento Distribuido y Paralelo

1. Introducción

El modelo de consistencia de memoria más aceptado en sistemas multiprocesador es el secuencial SC, dado que por ser una extensión de la semántica de ejecución en un sistema uniprocesador resulta el más intuitivo. Considerado el modelo más estricto a adoptar en un sistema, fue definido en forma no operacional por Lamport [Lam94] como sigue:

> *Un multiprocesador es Secuencialmente Consistente si el resultado de cualquier ejecución es el mismo que se tendría si las operaciones de todos los procesadores fuesen*

ejecutadas en un dado orden secuencial, y las operaciones de cada uno aparecen en dicha secuencia en el orden especificado por su programa.

Esta definición impone dos condiciones:

- Todos los accesos a memoria se ven realizados atómicamente en algún orden total, y
- Todos los accesos a memoria de cada procesador aparecerán en ese orden total siguiendo el orden del programa.

Mientras que en un esquema uniprocesador esto se logra con una lógica relativamente simple, pasar a un sistema multiprocesador lo torna complejo y restringe las optimizaciones a nivel de la arquitectura y de los compiladores, [SA95].

Se han desarrollado, trabajando desde el punto de vista del hardware, diversos modelos menos estrictos que el SC con el objetivo de admitir optimizaciones que no resultan de aplicación directa en sistemas siguiendo el modelo secuencial. Se puede mencionar entre ellos el modelo híbrido *weak ordering* WO propuesto por Dubois [MDB86], el modelo *processor consistency* tanto el de Goodman como el de Garachorloo, referidos como PCG y PCD respectivamente, [MAN93]. Estos modelos asumen de forma explícita o implícita coherencia de cache. Más recientemente se tiene Java el cual, como fue demostrado por Gontmakher [Gon00] a partir de la *Java Languaje Specification* JLS, también es cache coherente. Este requerimiento conlleva, a nivel del hardware, una sobrecarga que excede la requerida por la sincronización al forzar a que escrituras a una misma locación sean vistas en una misma secuencia en todos los procesadores. Con todo resulta menos restrictivo que la consistencia SC, en el sentido que este impone que todas las escrituras, en general, sean vistas en una dada secuencia. Además, dado que normalmente se adopta write back por cuestiones de escalamiento, se necesita ser el *owner* de un bloque para proceder con una escritura en un contexto cache coherente, lo cual induce invalidaciones que se traducen en pérdida de eficiencia del sistema.

Desde el punto de vista del software coherencia es un concepto difícil de visualizar trabajando en alto nivel, y resulta limitativo de las tareas de optimización que puedan llevar adelante los compiladores, como por ejemplo asignar variables a registros y reordenar operaciones de load.

En vista de esto se desarrolló el modelo Dag [RB96], un modelo de consistencia relajado de aplicación en sistemas DSM para programación multithread, y el LC [GS], en el que el estado de una locación de memoria se modela a través de un *partially ordered multiset* (pomset). Ambos modelos no son cache coherentes y, aunque desarrollados de manera independiente, resultan ser equivalentes, Frigo [Fri97].

Partiendo de un análisis de las limitaciones que presentan estos modelos, en cuanto a razonabilidad y una posible ineficiencia en los protocolos de cache propuestos, en este trabajo definiremos un modelo con el objetivo de que sea razonable y además eficiente.

El resto del trabajo se organiza de la siguiente forma. En la sección 2 analizaremos la coherencia de cache y alternativas de implementación. En la sección 3 abordaremos las limitaciones impuestas por la coherencia. En la sección 4 analizamos las limitaciones que presenta el modelo LC. En la sección 5 introducimos nuestro modelo de consistencia, al que denominamos consistencia de ejecución EC, y presentamos un posible protocolo de cache con las exigencias del modelo y de implementación eficiente. Por último en sección 6 formulamos conclusiones.

2. Coherencia de Cache

La problemática de la coherencia de cache en los sistemas multiprocesador con memoria compartida y cache privadas es un tema que ha sido abordado desde los comienzos de este tipo de sistemas. El problema tiene tres ejes, a saber: datos compartidos con acceso de lectura/escritura, migración de procesos, y actividad de entrada/salida. En nuestro trabajo enfocaremos la primer cuestión. Existe un amplio rango de soluciones propuestas, con grado variable de soporte de hardware y software a nivel de los procesadores, los controladores de cache, y de los módulos de memoria.

Al decir que un sistema es cache coherente, o simplemente coherente, se asume que:

1. Una lectura de un procesador a una locación X, posterior a una escritura del mismo procesador en esa locación X devolverá, de no mediar alguna escritura entre ambas operaciones, el valor de dicha escritura.

2. Una lectura de un procesador a una locación X posterior a una escritura realizada por otro procesador a la misma locación X devolverá el valor de dicha escritura, si ambas operaciones están suficientemente separadas y además no existe otra operación de escritura entre ambos accesos.

3. Las escrituras a una misma locación resultan serializadas. Esto es tomadas dos escrituras a una dada locación estas deberán ser vistas en el mismo orden por todos los procesadores.

El primer punto alude a respetar el orden del programa y el segundo a que en algún momento se verá el valor actualizado. El tercero, aunque más sutil, tiene la misma implicancia que el segundo. Si un procesador escribe en una locación un valor y luego un segundo procesador escribe otro valor en la misma locación, de no existir el requerimiento de serialización podría ser el caso de que un procesador viese en primer término el valor de la segunda y luego el de la primera, por lo que se mantendría desactualizado. Debemos agregar que la serialización de las escrituras es condición necesaria pero no suficiente. Requiere que a nivel del procesador se mantenga el orden del programa entre operaciones de lectura a una misma locación. Una arquitectura que no asegura este ordenamiento es el PowerPC de IBM.

Se tiene una serie de alternativas para resolver el problema de coherencia:

1. La más simple es no trabajar con caches privadas y asignar la cache a los módulos de memoria. La limitación es que si bien reduce el tiempo de acceso a memoria no mejora la cuestión de los accesos remotos en los sistemas DSM.

2. Dado que es importante retener el esquema de cache privada, una posibilidad es no emplearla para aquellas variables compartidas con posibilidad de lectura-escritura, por ejemplo locks, estructura de datos compartida, etc. Para las instrucciones y demás datos podría emplearse normalmente. El compilador debería marcarlas distinguiendo ambos tipos. La desventaja de esta alternativa es en principio la no transparencia, referida al programador o al compilador. El usuario debería declarar las variables como compartidas (o no compartidas) para lenguajes como Ada, Modula-2, o alternativamente un compilador multiprocessing, como Parafrase, podría clasificar los datos automáticamente. Además la detección en el procesador entre cacheables y no cacheables se realizará a nivel de páginas en un sistema con memoria virtual paginada. Esto provoca fragmentación interna, esto es más datos en condición de no cacheables de los que efectivamente son compartidos con atributo de lectura-escritura.

3. Dado que si todo los datos compartidos se definen como no cacheables la perfomance caería apreciablemente, se puede analizar que datos compartidos con atributo lectura-escritura se acceden siempre dentro de una sección crítica porque podrían ser cacheables. En cambio aquellos que protegen la sección crítica, por ejemplo los locks, deberían mantenerse no cacheables. Para mantener la coherencia todos los datos que han sido modificados en la sección crítica deberán invalidarse al salir de la misma, en lo que se conoce como *flush* de cache, lo cual asegura que no permanecerán en cache datos que podrían resultar desactualizados en un próximo acceso a la sección crítica. Claramente toda vez que se acceda a estos datos vía lock la coherencia está asegurada. Este esquema funciona para cache write through, con write back resultaría complejo. Además, dado que toda la cache debería ser *flushed*, la pérdida de eficiencia es clara, a menos de que se trate de una cache de tamaño reducido.

4. Una última alternativa es manejar dinámicamente la coherencia. Esto básicamente se puede llevar adelante mediante el *bus snoopy*, o con un esquema de directorio en sistemas que no emplean bus.

   En la primer alternativa, dada la capacidad de broadcast del bus, los controladores de cache verán todas las transacciones y responderán eventualmente con un cambio de estado del bloque en cache. Tiene la ventaja de ser distribuido y de que las escrituras a cada locación individual son vistas en todos los procesadores en un único orden secuencial por el efecto serializador del propio bus. Con una política write back se requiere obtener acceso exclusivo, ser owner, para proceder con una escritura. Si la memoria es el owner del bloque los procesadores podrán obtener copias de lectura solamente. El estado owner se alcanza generando una transacción *read/private* en el bus, el cual culmina con la invalidación del bloque en las cache de los demás procesadores. En caso de que un bloque se encontrase en estado modificado, primero debe actualizarse la memoria con la información en dicho bloque *commit* y luego invalidarse el bloque previo a la obtención del *ownership*.

   Se logran sistemas más escalables en configuraciones del tipo DSM, en donde la memoria compartida se distribuye entre los nodos y una red de interconexión escalable punto a punto reemplaza al bus, junto con un asistente de comunicación responsable de soportar el espacio de memoria compartida. Aquí no resulta de aplicación el *snoopy*, y el método más común de resolverlo es un esquema de directorio. La idea es mantener de forma explícita un directorio con la información necesaria para el manejo de los requerimientos. Así una entrada en este directorio, asociada a un bloque, poseerá indicación de que caches poseen una copia del mismo, y cual es el estado del bloque en éstas. Si bien el protocolo que debe entender con el estado del bloque en cache y las transiciones entre éstos es equivalente a un protocolo *snoopy*, se diferencian en el mecanismo para llevar adelante las operaciones fundamentales que se originan por un miss de cache o por un acceso de escritura a un bloque compartido. Ante un miss de cache se establecerá una comunicación con la entrada respectiva en el directorio usando la conexión punto a punto de la red. Este directorio normalmente está asociado al módulo de memoria que contiene al bloque. Aquí se determina si existen o no copias válidas y que acciones a seguir. Por ejemplo si se debe obtener una copia modificada desde otro nodo o, en caso de una operación de escritura, si se deben enviar invalidaciones y esperar en tal caso los respectivos *acknowledgments*. Además los cambios que se den en los bloques de cache generarán transacciones hacia la entrada respectiva del directorio para mantenerlo actualizado. Aquí es el propio directorio el que soporta la acción serializadora requerida por la coherencia. A estos sistemas DSM

se los refiere como *cache-coherent, nonuniform memory access*, CC-NUMA.

3. Limitaciones de la Coherencia

El asumir coherencia de cache en un sistema se origina en que es condición necesaria para soportar el modelo de consistencia secuencial. Además, tal como fue mencionado previamente, muchos de los modelos "derivados" del secuencial, en el sentido que con determinadas restricciones en el programa su ejecución resultará equivalente a la del modelo secuencial, asumen coherencia.

En nuestra opinión para un modelo relajado de consistencia este requerimiento puede no ser necesario y sí, como veremos, actúa en detrimento de la eficiencia del hardware y limita las optimizaciones del software.

Asumir coherencia obstaculiza el objetivo de obtener una visión *end-to-end*, [GS97]. En primer término la coherencia está definida con la granularidad del hardware, el tamaño de un bloque de cache, generando una división artificial entre tipos de datos de acuerdo a su tamaño. Además puede darse el caso de que existan en el código compilado referencias a memoria que no resultan visibles a nivel del código fuente, lo que haría imposible aplicar coherencia a ese nivel. Se debe remarcar que coherencia impone restricciones en el ordenamiento de las operaciones que van más allá del orden parcial impuesto por las operaciones de sincronización en un programa paralelo. La coherencia vincula además el ordenamiento de las operaciones de memoria en diferentes subprogramas, obstaculizando una visión modular a nivel del código fuente, y esto en un contexto donde los programas multithreaded son compilados por compiladores secuenciales que ignoran el requerimiento de coherencia. Normalmente los programadores se hacen cargo de ello mediante prueba y error, insertando declaraciones de variables "volatile" y deshabilitando optimizaciones del compilador hasta que el programa ejecute correctamente, todo lo cual trae aparejado ineficiencia en la ejecución del código resultante.

Otra cuestión es la pérdida de eficiencia debida al *false sharing* al aplicar coherencia de cache con bloques que incluyen varias palabras. Además, como se ha dicho previamente, el procesador deberá respectar el ordenamiento del programa entre operaciones de lectura a una misma locación, lo cual no siempre está asegurado. Ejemplo el PowerPC que soporta un modelo relajado de consistencia y cuenta con una instrucción de barrera SYNC para ordenar las operaciones a memoria. Llamativamente, intercalada entre dos lecturas a una misma locación, no fuerza el orden del programa entre ambas, demandando por ende una solución más sutil como por ejemplo emplear instrucciones del tipo *read-modify-write* para forzar este ordenamiento.

En línea con lo anterior, pensando en una arquitectura superscalar con ejecución fuera de orden, el tener que asegurar el orden del programa entre lecturas a una misma locación obliga a que el despacho de una operación de lectura se difiera hasta que las anteriores hayan definido su dirección, lo cual limita la concurrencia.

Veamos como el supuesto de coherencia da lugar a situaciones conflictivas y de pérdida de eficiencia a nivel de los lenguajes de programación paralelo relajados, tomando como ejemplo Java.

Java es un lenguaje que integra el soporte de concurrencia a través del *multithreading*. Incluye el compilador que traslada el fuente a *bytecodes*, los que son independientes de la plataforma, y una *Java Virtual Machine* JVM a cargo de ejecutar el *bytecode*.

En la *Java Language Specification* JLS capítulo 17, [JGS96], se describe el modelo de memoria en función de un *Abstract Memory System* AMS. Esta especificación resulta difícil de entender al punto de que en diversos artículos que analizan el modelo han llegado a interpre-

```
// p y q pueden ser aliased
int i = p.x
// escritura concurrente a p.x
// por otro thread
int j = q.x
int k = p.x
```

Figura 1: Ejemplo mostrando *reads kill*

taciones diferentes. Una cuestión interesante en tal sentido es que dado que el modelo resulta coherente, como fue demostrado por Gontmakher [Gon00], ciertas optimizaciones comunes del compilador resultan prohibidas, lo que en muchos casos no fue advertido inicialmente. Mucha gente no había percibido las implicancias en los compiladores de la coherencia en la JVM, resultado de lo cual realizaban optimizaciones que la violaban. Ejemplo de esto lo tenemos en la porción de código de la Fig. 1

Un compilador estándar eliminaría el *getfiel* correspondiente a k y lo reemplazaría rehusando el valor almacenado en i. Ahora bien, si p y q resultasen *aliased* y se diera el caso de que otro thread escribe en p/q.x entre el primer uso de p.x y el uso de q.x, el q.x adoptaría un valor más actual y no se cumpliría el requerimiento de coherencia, pues el segundo p.x vería un valor más viejo. Dan Scales en el Digital Western Research Laboratories realizó un estudio preliminar, "Impact of reads kill in Java" , referido al impacto de la coherencia en la performance. Los resultados sugieren que para programas de cómputo intensivo, con compiladores que manejen estas cuestiones, se tendrían ejecuciones entre el 20 % al 45 % más lentas.

Por último resulta claro que la coherencia de cache excluye la posibilidad de optimización a través de alocar variables en registros, dado que imposibilita que las modificaciones se propaguen, vale decir resulten visibles en los demás procesadores.

4. Limitaciones e Ineficiencias de la LC

El modelo introducido por Gao y Sarkar si bien tiene el mérito de no requerir coherencia, es a nuestro entender demasiado relajado como para que se pueda entender como razonable, y además creemos que el protocolo de cache propuesto no resulta de implementación directa.

En su trabajo de Master of Science en el MIT, Matteo Frigo [Fri97] elaboró un conjunto de requerimientos que entendía necesarios para que un modelo de consistencia fuese razonable, a saber:

1. Sea completo, en el sentido de que cualquier ejecución resulta válida en el modelo.

2. Sea monótono respecto al paralelismo, esto equivale a decir que al quitarle restricciones, por ejemplo removiendo un par acquire/release, toda ejecución válida en el original también lo será en el de mayor paralelismo.

3. Sea constructible, en el sentido de que el modelo se mantiene válido cuando se extiende la ejecución. Este principio trata de capturar la idea de que el modelo puede ser implementado exactamente, sin recurrir a uno más estricto.

Procesador 1	**Procesador 2**
L:= 1	...
...	acquire(L)
...	L := 2
...	release(L)
...	...
acquire(L)	...
A := L	...
release(L)	...
...	...
B := L	...

Figura 2: Ejemplo de ejecución que satisface LC pero no RC

4. Confina el determinismo, esto es luego de un nodo *join* posterior a accesos a memoria con condición de carrera, se alcance un comportamiento determinístico.

5. Classical, o lecturas no intrusivas, vale decir la incorporación arbitraria de operaciones de lectura en la ejecución no modifica la legalidad de los valores obtenidos en la ejecución original.

En su trabajo [GS] Gao y Sarkar realizaron una evaluación tendiente a demostrar las características de robusto y razonable del modelo LC, inspirados en parte en el trabajo de Frigo. En el se recogen las ideas de monótono, de lecturas no intrusivas, además se ejemplifica respecto a la naturaleza más relajada respecto al modelo RC, e incorpora el principio de equivalencia. Esto último implica que un programa *data race free* ejecutará de forma equivalente a lo que sería en un modelo RC, el que a su vez por la misma condición resulta equivalente al del SC.

Frigo en su trabajo de Tesis demuestra que el modelo LC es constructible pero no confina el no determinismo.

Por otro lado la propiedad de equivalencia fue cuestionada en el trabajo de Wallace, [CWA02], alentados en que la definición del modelo LC indica que los locks se corresponden con determinada/s variables, pero no fija nada respecto al reordenamiento admitido en las operaciones de coherencia. Resulta ser que de no restringir dicho reordenamiento, no se tendrá equivalencia entre el modelo LC y el RC, aún con programas *data race free*. En nuestra opinión, no necesariamente coincidente con la de los autores del modelo LC, resulta sensato demandar dentro del modelo reglas estrictas a este nivel.

En cualquier caso se puede concluir que no confina el no determinismo y que además alcanza un comportamiento aún más relajado que el del modelo Slow [HA90], el que a nuestro entender sería el modelo de consistencia relajado más razonable.

Para visualizar un poco esto veamos el ejemplo de la Fig. 2.

Si asumimos que el P_2 realiza el adquiere(X) antes que el P_1, podrá darse en la ejecución de P_1 que la primer lectura de X devuelva el valor 2, resultado de una lectura sincronizada, y que la segunda el valor 1, correspondiente a una lectura no sincronizada. Este comportamiento está en línea con la observación de Frigo en el sentido de no confinar el no determinismo, [Fri97]. Pero más aún, desde otro punto de vista, si asumimos que en la ejecución se respeta el orden del programa para lecturas a una misma locación, el modelo permite que una operación de lectura devuelva un valor que desde el punto de vista del propio procesador P_1 ha sido

sobrescrito. Dado que P_1 primero escribe en X un 1 y en una lectura posterior obtiene un valor 2, resulta claro que ha sido sobrescrito el valor de la primer lectura, luego no sería razonable que posteriormente una lectura regrese el valor 1. Esto implicaría un relajamiento que excede al modelo Slow [HA90], que entendemos el más relajado. Podemos mencionar en este punto que el modelo Slow bajo ciertas condiciones, como quedó demostrado en el trabajo de Himanshu Sinha [Sin93], resulta suficiente para soportar correctamente algoritmos iterativos asincrónicos.

La ineficiencia tiene que ver con la dificultad de implementar un protocolo de cache que satisfaga el modelo. Si bien cada bloque de cache se puede manejar con los bits de *valid* y *dirty*, y que además las operaciones de lectura/escritura se resuelven localmente, el problema se origina con las operaciones de sincronización. Al momento de un acquire(X) un bloque X en estado *clean* deberá invalidarse, no así si está *dirty*. Además previo a un release(X), si el bloque X está en estado *dirty* deberá actualizarse en memoria *commit*, esperar por el *acknowledgment* y pasarlo al estado *clean*.

5. Consistencia de Ejecución

El modelo no coherente LC [GS] resulta soportado por un esquema en donde para una lectura se tiene como valores legales, la última escritura previa según el ordenamiento $\prec$ del *pomset* o alternativamente cualquier otra escritura que no se encuentre ordenada en el *pomset*.

De nuestro análisis surgen dos cuestiones que a nuestro entender limitarían el rango de valores legales para una operación de lectura y posibilizarían un modelo más razonable y que encapsule el no determinismo.

El modelo de consistencia que proponemos corresponde al de un sistema multiprocesador con caches individuales en cada procesador y una memoria compartida. En tal caso, las escrituras en cada procesador a locaciones individuales resultan serializadas en las respectivas caches, con lo que manteniendo el ordenamiento de los *reads* a una misma locación no sería posible ver un valor de una escritura de un procesador que haya sido sobrescrita por una posterior del mismo procesador. Además, lógicamente, las escrituras de un procesador son inmediatamente visibles para dicho procesador. Ambas características son análogas a las del modelo Slow.

Una modificación que planteamos respecto al modelo LC está referida al comportamiento de la memoria para el caso de operaciones de escritura sincronizadas. Nuestra posición es que la semántica de un acquire, referido a un acceso excluyente para determinadas locaciones de memoria compartida, impone que las escrituras sean válidas en todo el sistema, vale decir este evento tendría efecto en las cache de cada uno de los procesadores del sistema. En lo referido a las operaciones de lectura en un procesador a una locación de memoria, y de manera análoga a la del modelo LC, el valor legal a retornar puede ser el de algún acceso de escritura asincrónica por parte de algún procesador (propia o el de una escritura conocida a través de la memoria global), o en su defecto el valor de la última escritura sincrónica, lo que ocurra último en la ejecución.

La idea a nivel de implementación es conservar la característica de que los acquire/release operan a nivel de determinadas locaciones, esto es no tienen un alcance indiscriminado, resultando en algún sentido similar al planteo del modelo de consistencia Scope, [LIKL96].

5.1. Protocolo de cache para el modelo EC

Se vio que la dificultad con el protocolo de cache propuesto para el modelo LC radica en que plantea un esquema de invalidaciones parciales, según las locaciones asociadas al acqui-

re/release.

En la comparación con el modelo Java se puede ver que los locks tienen dos propósitos. El primero es sincronizar el flujo de control entre threads, y el segundo manejar la consistencia de memoria entre threads. Aunque en el *bytecode* se tienen instrucciones lock y unlock, en el lenguaje de programación Java no existen lock y unlock de forma explícita. En cambio fragmentos de código fuente se pueden indicar como sincronizados, implicando instrucciones lock y unlock al comienzo y al final de los mismos. Más aún, si bien el lenguaje define una correspondencia entre objetos y locks a nivel de código fuente, no existe tal correspondencia entre objetos y locks a nivel del *bytecode*. Por ende, lock y unlock involucran a todas las variables, no solamente aquellas almacenadas en el objeto del método que se ha llamado.

Vamos a describir un posible protocolo de cache que pueda sustentar eficientemente el modelo propuesto y que resulte adecuado para un ambiente escalable del tipo DSM.

El planteo sería conservar a nivel de bloque de cache los bits de *valid* y de *dirty* y emplear los dos esquemas de escritura write back y write through, de acuerdo a si la escritura es asincrónica o sincrónica, respectivamente. En el caso asíncrónico, cada procesador podrá modificar el bloque de cache siguiendo la política write back, con el protocolo de lectura y escritura propio de este esquema.

Para el caso de escrituras sincrónicas la política de escritura en cache write through permite mantener actualizado en memoria global el valor de manera eficiente. Lo que planteamos, además del cambio a write through, es que se adopte una política de escritor único, esto se deba ser *owner* del bloque para las escrituras sincrónicas, obligando al *commit* de este bloque para las cache que lo tienen con el bit *dirty* en 1. Es de notar que en lo concerniente a las lecturas dentro de una zona crítica, entre un par de acquire/release, se procedería como es usual en cuanto al hit y al miss.

Podemos apreciar que proceder con la política de write through unido a la de *owner* para realizar una escritura sincrónica, permite aprovechar la flexibilidad del modelo de consistencia para alcanzar una implementación eficiente. Obsérvese en tal sentido que no se debe recurrir a invalidaciones, ni parciales como con el modelo LC, ni totales como ocurre en otros modelos.

Cuando un procesador haya ingresado en la zona crítica, los restantes procesadores podrían seguir trabajando concurrentemente, a menos de que requieran un acquire sobre las mismas locaciones. La única cuestión que actuaría limitando dicha concurrencia es que se origine un reemplazo en algún procesador de algún bloque que esté en condición de exclusivo. En tal caso la operación de reemplazo se verá frenada hasta que se libere esta condición.

Es aquí que podemos analizar la potencialidad referida al *false sharing*. Podemos ver que si el bloque en cuestión contenía otra información, además de la del dato sincronizado, el *false sharing* no tendría implicancias en cuanto permite la escritura concurrente. Esto debería estar soportado a nivel de la cache suministrando más de un bit *dirty* por bloque, como por ejemplo hace el ALPHA, con granularidad más fina por ejemplo de una palabra. Esta cuestión resulta más que interesante dada la tendencia a definir tamaños de bloque de varias palabras. De esa forma funcionaría correctamente la política de múltiple escritor para los accesos asincrónicos. Las actualizaciones parciales de los bloques se combinarían naturalmente en memoria, con los procesos de invalidación o reemplazo.

Por último, para las operaciones de sincronización global del tipo barrera, el planteo sería proceder a la invalidación sistemática *flush* de todas las caches individuales. Obviamente, en el caso de que los bloques tengan *dirty* en 1 se deberá proceder previamente con la respectiva actualización en memoria principal. Es solo para este caso, cuando se demanda una visión única de la memoria compartida por parte de todos los procesadores, que debemos recurrir a

una acción de invalidación total de las caches.

6. Conclusiones

En este artículo hemos argumentado respecto a las limitaciones que induce la coherencia de cache, tanto de hardware como de software. Partiendo del análisis de una de las propuestas más relajadas, la LC debida a Gao y Sarkar, y evaluados las limitaciones del modelo y los inconvenientes en cuanto a su implementación, hemos propuesto un nuevo modelo de consistencia superador del modelo LC. Se ha realizado un lineamiento de un protocolo de cache para el modelo que entendemos eficiente. Como trabajo futuro pensamos realizar estudios con el modelo y el protocolo, tendientes a dimensionar su capacidad tanto a nivel de programación como de la propia implementación.

Referencias

[CWA02] G. Tremblay C. Wallace and J. Amaral. On the tamability of the location consistency memory model. *Proc. PDPTA*, 2002.

[Fri97] Matteo Frigo. Magister of science tesis: The weakest reasonable memory model. *Massachusetts Institute of Technology*, 1997.

[Gon00] Alex Gontmakher. Java consistency, nonoperational characterizations for java memory behavior. *ACM, Transactions on Computer Systems Volume 18*, 2000.

[GS] G.R. Gao and V. Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Transactions on Computers*.

[GS97] G.R. Gao and V. Sarkar. On the importance of an end-to-end view of memory consistency in future computer systems. *Proc. Intl Symp.Highe Perfomance Computing*, 1997.

[HA90] P. Hutto and M. Ahamad. Slow memory: weakening consistency to enchance concurrency in distributed shared memoies. *Proc. 10th IEEE Intl. Conference on Distribute Computing Systems*, 1990.

[JGS96] B. Joy J. Gosling and G. Steele. The java laanguaje specification. *Addison-Wesley, Reading*, 1996.

[Lam94] Leslie Lamport. How to make multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1994.

[LIKL96] J. Singh L. Iftode and Princeton Kai Li. Scope consistency, a bridge between release consisency and entry consistency. *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[MAN93] R.John P. Kohli M. Ahamad, R. Bazzi and G.Ñeiger. The power of processosr consistency. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[MDB86] C. Scheurich M. Dubois and F. Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Annual Intl Symp. on Computer Architecture*, 1986.

[RB96] C. F. Joerg R.D. Blumofe, M. Frigo. An analysis of dag-consistent distributed shared-memory algorithms. *Proceedings of the Eichth Annual ACM Symmposium on Parallel Algorithms and Architectures*, 1996.

[SA95] y Koourosh Gharachorloo Sarita Adve. Shared memory consistency models: A tutorial. *WRL Technical Report 95/7 Digital, Palo Alto, California*, 1995.

[Sin93] Himanshu Shekhar Sinha. Mermera: non-coherent distributed shared memory for parallel computing. *Technical Report BU-CS-93-005, Computer Science Department, Boston University*, 1993.