Operaciones Atómicas Eficientes, Distribuidas, y no Especulativas para múltiples direcciones

Eduardo José Gómez-Hernández[1], Juan M. Cebrián[1], Rubén Titos-Gil[1], Stefanos Kaxiras[2], y Alberto Ros[1]

Resumen— **Las secciones críticas que leen, modifican y escriben (RMW) un pequeño conjunto de direcciones son comunes en aplicaciones paralelas y estructuras de datos concurrentes. Sin embargo, para escapar de las complejidades de los *locks* de grano fino, que requieren razonar sobre todas las posibles intercalaciones de hilos, los programadores suelen recurrir a *locks* de grano grueso para garantizar la atomicidad. Esto da lugar a que el conjunto de direcciones potencialmente conflictivas sea mucho mayor y, en consecuencia, a una mayor contención de los *locks* y a una serialización innecesaria. Como muchos han observado antes que nosotros, estos problemas se resolverían si las operaciones atómicas de RMW para múltiples direcciones estuvieran disponibles en los procesadores actuales, pero las propuestas académicas son poco prácticas debido a los potenciales *deadlocks* que aparecen por las limitaciones de recursos. La memoria transaccional es una alternativa que puede detectar conflictos en tiempo de ejecución con el objetivo de maximizar la concurrencia, pero tiene sobrecargas significativas en secciones críticas con contención alta.**

En este trabajo proponemos operaciones atómicas multidirección (MAD atomics). Los MAD atomics consiguen un bloqueo de grano fino, distribuido, eficiente y no especulativo para múltiples direcciones, basándose únicamente en el protocolo de coherencia y en un orden de bloqueo de direcciones de memoria predeterminado. A diferencia de trabajos anteriores, los MADs abordan el reto de permitir la modificación atómica sobre un conjunto de líneas de caché con direcciones arbitrarias, bloqueando simultáneamente todas ellas y evitando los *deadlocks*. Los MAD atomics requieren de muy poca área de silicio adicional por núcleo (alrededor de 68 bytes), superando en rendimiento y energía a implementaciones típicas basadas en *locks*. Nuestra evaluación utilizando gem5-20 muestra que MAD atomics mejoran el rendimiento en hasta 18× (3,4× de media), para las aplicaciones y estructuras de datos concurrentes respecto a una implementación base con *locks* ejecutada en 16 núcleos. Además, la mejora de rendimiento alcanza 2,7× de media cuando se compara con la implementación de memoria transaccional por hardware de Intel ejecutada en 16 núcleos.

Palabras clave— **Arquitecturas multi-núcleo, sincronización, secciones críticas, atomicidad, atómicos multidirección**

I. Introducción

La tendencia actual de aumentar el número de núcleos por procesador permite mejorar el rendimiento de las aplicaciones y estructuras de datos concurrentes. Lamentablemente, un mayor número de hilos provoca una mayor sobrecarga de sincronización, lo que da lugar a algoritmos menos eficientes cuando éstos se implementan con *locks* [1], [2].

Los *mutexes*[1], aunque son rápidos cuando hay poca contención, muestran una gran sobrecarga para los *locks* contendidos. En algunas implementaciones incluso requieren de la ayuda del sistema operativo. Desafortunadamente, el bloqueo de grano fino no es fácil de aplicar para cualquier cosa que no sean pequeñas secciones críticas. Además, un esquema de bloqueo de grano fino puede dar lugar a la necesidad de adquirir más de un *mutex* simultáneamente. Esto introduce la posibilidad de *deadlocks* si los *mutex* no se adquieren en un orden global.

Una alternativa que se ha considerado durante mucho tiempo como más escalable que los *mutex-locks* es el concepto de algoritmos no bloqueantes (*lock-free algorithms*). En estos, una instrucción de comparación e intercambio (CAS) comprueba si la condición para la ejecución atómica de una operación se mantiene y realiza alguna acción, almacenando el resultado. Aunque se han propuesto algoritmos no bloqueantes para varias estructuras de datos [3], escribir algoritmos no bloqueantes correctos que garanticen el progreso de todo el sistema (algoritmos sin *deadlocks*) es una tarea notoriamente difícil [4].

Una tercera alternativa de sincronización apareció más recientemente, la memoria transaccional por hardware (HTM) [5], [6]. Con HTM, las regiones críticas (*transacciones*) se ejecutan simultáneamente de forma especulativa, mientras que el hardware supervisa sus accesos a memoria y revierte la ejecución si surgen conflictos. En las implementaciones existentes de HTM, una transacción suele re-ejecutarse un número de veces antes de tomar el camino no especulativo, en el que la exclusión mutua se impone a través de un bloqueo global [7]. La desventaja de HTM es que no se adapta bien a las secciones críticas con contención alta debido a los altos ratios de abortos y a la frecuente serialización de hilos.

Las instrucciones atómicas RMW son la forma más eficiente de realizar una actualización atómica de una variable, ya que minimizan la serialización y no requieren la intervención del SO Nuestro objetivo es extenderlo a múltiples direcciones de una sección crítica. Estas nuevas operaciones atómicas MAD se ejecutan a nivel de líneas de caché, eliminando la falsa contención y permitiendo el acceso concurrente a datos disjuntos.

[1]Computer Engineering Department, University of Murcia, Spain, e-mail: {eduardojose.gomez, jcebrian, rtitos, aros} @um.es.

[2]Department of Information Technology, Uppsala University, Sweden, e-mail: stefanos.kaxiras@it.uu.se.

[0]Version traducida de: "Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations". 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)

[1]MUTually EXclusive: Elemento de un programa usado para negociar la exclusión mutua entre los hilos.

Varios trabajos anteriores propusieron la atomicidad multidirección [8], [9], [10], [11], pero no consideran escenarios de *deadlock* cuando se bloquean varias líneas de memoria ni proporcionan una solución segura y no especulativa. Hay dos razones por las que queremos seguir un enfoque no centralizado y no transaccional. En primer lugar, la centralización de la gestión de la atomicidad crea un cuello de botella que restringe la escalabilidad de instrucciones independientes y no conflictivas. En segundo lugar, deshacer trabajo múltiples veces resulta mucho más costoso en escenarios de alta contención que ejecutar con un orden predeterminado.

Evaluaremos MAD en el simulador multinúcleo gem5-20 [12]. Los resultados de rendimiento muestran una mejora de hasta 18× (con una media de 3,4×) en 16 núcleos respecto a una implementación base con *locks*. En comparación con una implementación de memoria transaccional por hardware similar a la de Intel (TSX), el beneficio de rendimiento alcanza 2,7× de media. Los atómicos MAD también reducen el número de instrucciones ejecutadas, quedándose en sólo un 10 % del número original de instrucciones, de media, para 16 núcleos, y un 5 % en 64 núcleos. Esto significa que los atómicos MAD no sólo son más rápidos que TSX, sino también más eficientes energéticamente.

II. Estado del Arte

Dijkstra presentó por primera vez el problema de adquirir varios *locks* con una analogía de cinco filósofos [13]. Su solución se basaba en un orden predeterminado para adquirir los *locks*. El enfoque de orden predeterminado elimina los bloqueos para adquirir cualquier número de *mutex-locks*, ya que los recursos de software son básicamente ilimitados: si los *locks* se toman en orden, no es posible ningún *deadlock*. Sin embargo, cuando se bloquean líneas de caché en las cachés locales de los núcleos, las limitaciones de recursos del sistema, como por ejemplo la asociatividad de la caché, ponen en peligro la ausencia de *deadlocks*. El problema se agrava si se tiene en cuenta que los atómicos de varias direcciones no pueden crear grupos atómicos personalizados a su conveniencia, ya que *todas* las direcciones solicitadas deben ser bloqueadas como un único grupo atómico.

Ros y Kaxiras [14], inspirados en los fundamentos teóricos establecidos por Coffman et al. [15], abordan el problema de la limitación de recursos asignando un orden global *subdirección* (llamado *orden léxico*) a cada línea de cache. El orden léxico viene dictado por un conjunto de bits de la dirección de la línea de caché, por ejemplo, los bits utilizados para indexar el conjunto de caché donde se coloca la línea. La Figura 1 muestra cómo se compara el orden léxico con el de las direcciones. Soluciones como la de Dijkstra siguen el orden de direcciones para establecer la dirección de bloqueo (Figura 1-a). Por otro lado, el orden léxico (Figura 1-b) tiene en cuenta el tamaño de la caché compartida más pequeña y la dirección de la línea de caché (eliminando los bits de desplazamiento dentro de la línea de caché). Utilizando el orden léxico, la Figura 1-c muestra un ejemplo de tres núcleos que bloquean diferentes conjuntos de direcciones y cómo se reordenan.

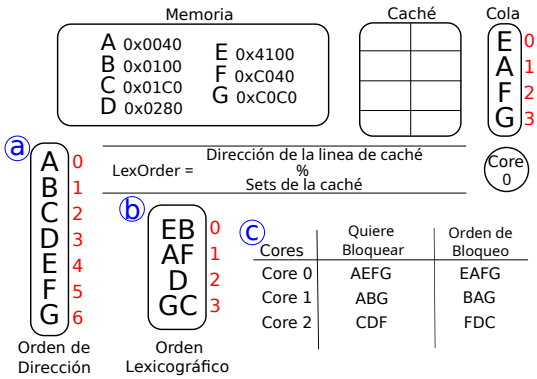

Fig. 1: Orden basado en dirección a orden léxico

Debido al orden léxico, los núcleos en espera no pueden formar un ciclo con las líneas de caché que ya han bloqueado y las líneas que quieren bloquear a continuación. El orden léxico garantiza que los conflictos se producirán en el mínimo orden común entre cualquier grupo atómico. Así, se evitan los *deadlocks* por recursos. Nuestra solución también se basa en el orden de léxico. Sin embargo, para las cachés privadas, Ros y Kaxiras resuelven los conflictos léxicos no permitiendo dos líneas de cache con el mismo orden de léxico en el mismo grupo atómico. Esto no es posible cuando se implementan operaciones atómicas de múltiples direcciones, ya que si el programa dicta que dos accesos deben realizarse atómicamente, el hardware no puede imponer una restricción de atomicidad, y *debería ser capaz* de mantener esos bloqueos al mismo tiempo. Por lo tanto, lo que necesitamos es un bloqueo no especulativo y *deadlock-free*, de grano fino para múltiples direcciones.

III. Atómicos MAD

Los atómicos MAD son un conjunto de instrucciones individuales capaces de actualizar atómicamente un pequeño número de posiciones de memoria. Lo ideal es encapsular una sección de código como la que se muestra en la Figura 2-a, en una única primitiva de hardware (Figura 2-b). La instrucción se descompondrá en varios micro-ops (Figura 2-c), que pueden adquirir los *locks* en un orden diferente con el fin de evitar *deadlocks* (Figura 2-d).

El encapsulamiento y el reordenamiento de la adquisición de *locks* también se aplican a las transacciones, o a cualquier otra primitiva de sincronización. Los atómicos MAD pueden coexistir en un programa con otras secciones críticas más grandes que utilizan locks/transacciones, ofreciendo así flexibilidad al programador. Los atómicos MAD utilizan una única macro-instrucción, decodificada en tiempo de ejecución en varias micro-operaciones. Esta implementación garantiza que no pueden interrumpirse durante la ejecución, y los cambios de contexto deben esperar hasta que la operación atómica MAD se complete y se retire. Por lo tanto, las interrupciones pueden retrasarse. Sin embargo, como hay una garantía de

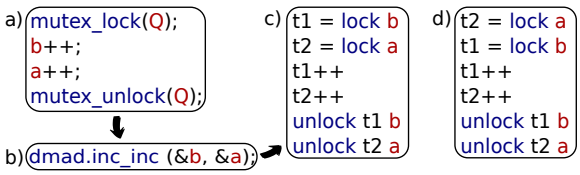

Fig. 2: a) Sección crítica con mutexes; b) Instrucción atómica MAD; c) Micro-operaciones generados del atómico MAD; d) Orden de ejecución de las instrucciones

progreso y las secciones críticas a las que se dirigen los atómicos MAD son cortas, los tiempos de espera no son extremadamente largos.

A. La Unidad de Reordenamiento Léxico (LexOU)

Las líneas de memoria involucradas en una operación atómica de múltiples direcciones necesitan ser bloqueadas en un orden predeterminado para evitar *deadlocks*. La unidad responsable de rastrear las líneas de caché y adquirir los bloqueos es la *Lexical reOrder Unit* (LexOU).

Cuando se ejecuta una micro-operación *lock*, la línea de memoria no se emite directamente al subsistema de memoria. En su lugar, las direcciones de las líneas de memoria se almacenan en un pequeño búfer implementado como una memoria direccionable por contenido (CAM) denominada *Lock Queue*. Las líneas de memoria no se bloquean hasta que todas las direcciones se insertan en la *Lock Queue*. Diferentes locks pueden dirigirse a la misma línea de caché. En este caso, sólo se almacena una única dirección de línea de memoria en la *Lock Queue* y un contador adicional indica el número de operaciones de bloqueo realizadas sobre esa línea de memoria.

Este contador se utiliza para desbloquear la línea de memoria después de que todas las operaciones de escritura a dicha línea de memoria se hayan completado. Las líneas de memoria se desbloquean utilizando la misma micro-op que se utiliza en los atómicos x86 (`stul` en la terminología de gem5-20).

B. Caso de Uso: Instrucciones MCAS

Basándonos en MAD atomics podemos derivar fácilmente instrucciones *compare_and_swap* para varias direcciones (MCAS). Nuestra implementación de la instrucción MCAS recibe los siguientes registros como entrada: *rax*, *rdx*, *rcx*, *rsi*, *rdi*, *r8*, *r9*, *r10*, *r11*, *r12*, *r13*, y *r14*, en ese orden específico. Cada conjunto de tres registros representa una operación CAS. El número final de registros utilizados depende de cuántas operaciones CAS deban realizarse atómicamente (seis para MCAS de aridad dos, nueve para MCAS de aridad tres, y así sucesivamente). Por ejemplo, el MCAS de aridad dos sigue este formato: *dmad_cas(*addr0, old0, new0, *addr1, old1, new1)*, donde add0 se almacena en rax, old0 en rdx, etc. Una vez decodificado, se traducirá en una secuencia de micro-ops similar a la de la Figura 2-c.

C. Otras Consideraciones

Aridad de atómicos MAD. El número máximo de direcciones que se pueden actualizar condicionalmente en la misma instrucción está limitado

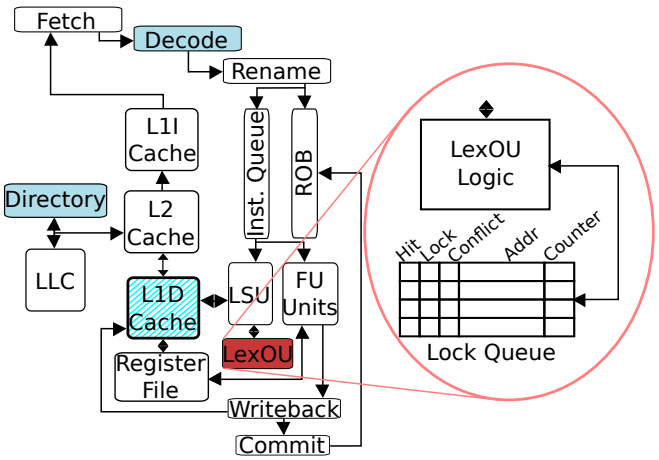

Fig. 3: Diseño micro-arquitectural. Las unidades modificadas aparecen en azul. Las estructuras nuevas aparecen en rojo

por el número de registros lógicos. Las operaciones MAD pueden requerir hasta tres registros (dirección de destino, valor antiguo y valor nuevo) por dirección. Dado que el X86_64 tiene hasta 16 registros disponibles para el programador [16], la aridad máxima para implementar MADs sin requerir instrucciones adicionales y hardware para saltarse la restricción de registros es de cinco.

Cambios Microarquitecturales. La Figura 3 destaca los cambios realizados para implementar los atómicos MAD. La etapa de decodificación incluye la adición de instrucciones MAD y sus correspondientes micro-ops. La caché L1D necesita llevar la cuenta de todas las líneas de caché bloqueadas. Dependiendo de la implementación real del hardware de bloqueo de una dirección, la L1D puede requerir modificaciones o no. Si se utiliza un bit por entrada de caché no se requieren modificaciones. En caso contrario, si se utiliza un único registro para rastrear la línea de caché bloqueada, este registro se amplía a una matriz de registros. La política de sustitución también se modifica para no seleccionar nunca como reemplazo las líneas de caché bloqueadas. Por último, cada conjunto del directorio requerirá un bit adicional para evitar los *deadlocks*.

IV. Limitación de Recursos

A. Cachés Privadas

La idea clave de nuestro enfoque es que podemos bloquear líneas de caché *exclusivas localmente en las cachés privadas, por lo general sin comunicarse con los directorios.*[2]. Si las direcciones que deseamos modificar atómicamente están todas presentes en la caché privada (*local*), no se requiere comunicación. *Sólo* cuando nos faltan una o más de las direcciones tenemos que considerar la interacción con las cachés compartidas y los directorios.

Por lo tanto, una propiedad necesaria para la caché privada es que debe ser capaz de albergar simultáneamente todas las líneas de caché que se bloquearán

[2]Suponemos directorios distribuidos en el caso general, por lo que diferentes direcciones pueden ser manejadas por diferentes directorios

durante una operación atómica de múltiples direcciones. De lo contrario, es posible que se produzca un *deadlock*. Como hemos mencionado, el bloqueo es estrictamente local: ningún otro núcleo, ni el directorio, suele ser informado cuando se bloquea una línea de memoria. Otros núcleos pueden ver retrasado su mensaje de invalidación o reenvío, ya que no pueden recibir una respuesta hasta que se libere el bloqueo.

Sin embargo, (a menos que la caché sea totalmente asociativa) las líneas de cache no se distribuyen libremente en la caché privada, sino que se asignan a un conjunto concreto de la caché. Por ejemplo, si tres líneas de cache están mapeadas en el mismo conjunto, pero la caché sólo tiene dos vías, después de bloquear dos de las tres, no hay más espacio en el conjunto para mantener la tercera. Esto conduce a un *deadlock* por limitación de recursos.

La limitación de recursos de la caché privada es un problema sencillo de resolver, ya que nos dirigimos a la operación atómica multidirección con baja *aridad*. La regla para las cachés privadas es que la aridad de la operación atómica debe ser menor o igual que la asociatividad de la caché privada. Dado que nuestra implementación actual de atómicos MAD soporta hasta cuatro direcciones, esto es compatible con la mayoría de los procesadores básicos. Por ejemplo, el ARM Cortex-A78 [17] tiene cuatro vías, el Skylake [18] de Intel tiene ocho vías y el Icelake [19] de Intel tiene doce vías para la caché de datos L1. En dispositivos más eficientes que utilizan cachés privadas de mapeo directo, no se podrían implementar operaciones atómicas de varias direcciones siguiendo la regla de la asociatividad. Afortunadamente, esta limitación puede resolverse con el uso de cachés víctimas totalmente asociativas [20]. En este caso, la aridad máxima de nuestros atómicos MAD es la asociatividad de la caché de víctimas, normalmente mayor que nuestra aridad máxima.

B. Directorios y Cachés de Datos Compartidas

Si la caché compartida (comúnmente el último nivel de la caché en el chip) es inclusiva con las cachés privadas, la caché compartida se convierte en otro recurso limitado. Además, las líneas de caché en las cachés privadas son rastreadas por el directorio. Si una línea de caché se almacena en una caché privada debe existir una entrada en el directorio para dicha línea de caché. Por lo tanto, la limitación de recursos en el directorio también puede provocar *deadlocks*. Dado que las cachés compartidas suelen tener un mayor número de entradas que el directorio (o al menos un número igual cuando la caché incluye información de directorio), nos centramos en el directorio, ya que nuestra solución cubre ambas.

Por ejemplo, si la caché privada ya tiene todas las líneas de cache, esto significa que las entradas de directorio correspondientes ya están presentes en el directorio y todo está bien. Cualquier intento de desalojar una de las entradas de directorio correspondientes se bloqueará intentando invalidar la línea de caché bloqueada en la caché privada hasta que se libere el bloqueo. Sin embargo, si la caché privada no tiene ya todas las líneas de caché que deseamos bloquear, debemos pedir las que faltan al directorio. Asumiendo que el directorio tiene al menos la misma asociatividad que la caché privada (una suposición válida para la mayoría de los sistemas), el directorio puede acomodar fácilmente cualquier combinación de direcciones que deseemos bloquear. Sin embargo, esto ignora el caso de que otra caché privada haya bloqueado silenciosamente un número de líneas de caché en un conjunto de directorios. Recordemos que una de las premisas básicas de nuestro trabajo es que no pedimos permiso para bloquear cuando tenemos una linea de caché privada como exclusiva.

El problema ahora es que la aparente asociatividad del directorio con respecto a una caché privada que quiere bloquear un número de entradas en un conjunto, se ha reducido a través de las acciones de otra caché privada. Además, como elegimos que el orden léxico sea igual al número de conjuntos en el directorio, las direcciones que mapean en el mismo conjunto del directorio tienen el mismo rango de orden léxico, y entran en conflicto entre sí.

Nuestro enfoque para evitar el mencionado tipo de *deadlocks* consiste en bloquear de forma conservadora los recursos del directorio. Cuando una operación atómica multidireccional desea bloquear más de una entrada de un conjunto del directorio todo el conjunto debe estar bloqueado. Suponemos que cada conjunto del directorio tiene un bit de bloqueo que puede ser activado cuando una solicitud llega al conjunto.

Si un atómico multidirección se comunica con el directorio para pedir una línea de caché que falta (o solicitar la exclusividad de una línea de caché) declara si quiere bloquear (o ya ha bloqueado) una o más líneas de caché del mismo conjunto. Si el atómico multidireccional opera sobre una sola entrada de un conjunto del directorio, no tiene que bloquear el conjunto sino que tiene que esperar por si alguien más ha bloqueado el conjunto. Si, por el contrario, el atómico multidirección opera sobre múltiples entradas de un conjunto del directorio, tiene que bloquear todo el conjunto y, por supuesto, tiene que esperar en caso de que el conjunto ya esté bloqueado.

Un atómico multidirección libera el bloqueo que tiene sobre un conjunto sólo después de que consigue modificar todas sus entradas en el conjunto. Mientras que un atómico multidireccional mantiene el bloqueo del conjunto, no experimentará ninguna interferencia de otros núcleos para este conjunto. A pesar de tener el bloqueo de un conjunto, un atómico multidirección puede tener que esperar a otros atómicos multidirección que hayan bloqueado silenciosamente las líneas de caché del conjunto. Sin embargo, basándose en el principio de adquirir todas las entradas necesarias antes de liberar el bloqueo, es imposible tener un *deadlock* con otro atómico.

C. Eviction Buffers

Un *eviction buffer* es un almacenamiento temporal para una línea de caché que está siendo expulsada de

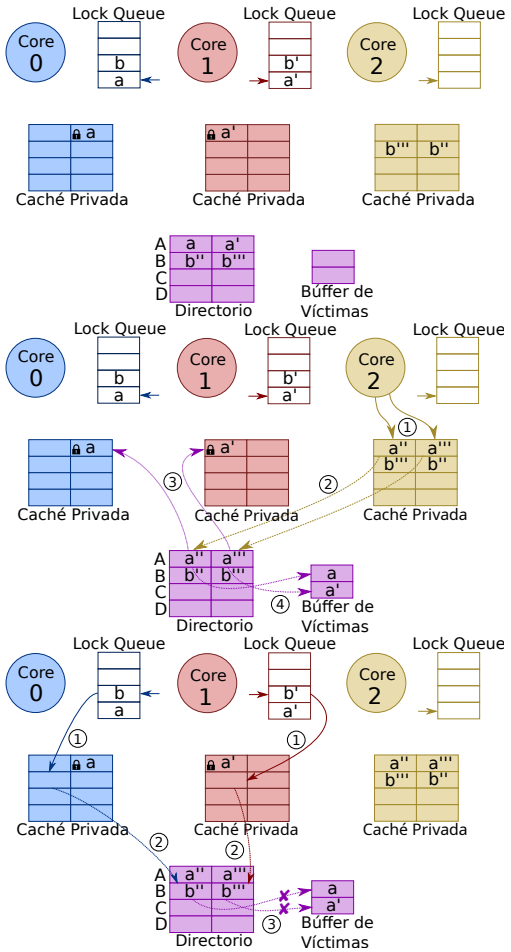

Fig. 4: Deadlock por recursos en los *eviction buffers*

la caché. Esto permite el uso inmediato de la entrada de la caché sin tener que esperar a las confirmaciones inducidas por el reemplazo. Cuando los *eviction buffer* están llenos, las líneas de memoria no pueden ser expulsadas hasta que se libere una nueva entrada.

Los *eviction buffers* de las cachés compartidas pueden contener líneas de caché bloqueadas en las cachés privadas. Esto da lugar al problema representado en la Figura 4. La Figura muestra un *deadlock* causado por la limitación de recursos en el *eviction buffer* del directorio. El núcleo 2 provoca dos reemplazos en el directorio, llenando las entradas *eviction buffers*. Para que las entradas reemplazadas salgan del *eviction buffer*, las líneas de caché rastreadas deben ser invalidadas en las cachés privadas. Sin embargo, las líneas de caché están bloqueadas por el núcleo 0 y el núcleo 1, que están en medio de una operación atómica de varias direcciones. Estos núcleos no pueden terminar de bloquear las líneas de caché requeridas por que el directorio no es capaz de reemplazar ninguna línea de caché, y por lo tanto no puede dejar espacio para que se rastreen nuevas líneas de caché (sin importar el conjunto donde estén indexadas). Por lo que sabemos, no hay ninguna solución publicada para este problema que sea adecuada para las operaciones atómicas multidirección.

Para evitar *deadlocks*, nuestra propuesta realiza sustituciones de directorios ïn situ"para los reemplazos de entradas privadas (es decir, sin utilizar el búfer de desalojo). Aunque las sustituciones ïn situ.ªumentan la latencia de las peticiones que no encuentran una entrada en el directorio, se trata de un escenario poco frecuente.

Tabla I: Configuración del Sistema

Procesador	
Num. Núcleos	1,2,4,8,16,32,64
Ancho del Issue	4 instrucciones
Ancho del Commit	8 instrucciones
Cola de Instructiones	128 entradas
Reorder buffer	224 entradas
Load queue	72 entradas
SQ + SB	56 entradas
Memoria	
Caché privada L1 I&D	32K/núcleo, 8 vías
Caché privada L2	256K/núcleo, 8 vías
Caché compartida L3	32M, 16 vías
Directorio	32768 conj., 16 vías
Tiempo acceso a Memoria	80ns

V. Metodología

A. *Entorno de Simulación*

Simulamos un procesador multinúcleo utilizando el simulador gem5-20 [12]. El sistema simulado ejecuta Ubuntu 16.04 con el kernel Linux 4.9.4. Los parámetros del procesador, tratan de imitar a un procesador Intel Skylake (Tabla I). Utilizamos Ruby para modelar la jerarquía de memoria y el protocolo de coherencia. Las latencias de ejecución y emisión se modelan según el hardware real, Fog [21].

Además de nuestro sistema de referencia, también modelamos un sistema HTM de mejor esfuerzo similar al de Intel TSX, basado en el soporte de memoria transaccional disponible en gem5-20.1 para arm TME. La política de resolución de conflictos de nuestro modelo TSX sigue la política de solicitante-ganador, tal y como se observa en las actuales implementaciones comerciales de Intel, que puede dar lugar a *livelocks* temporales que se resuelven volviendo a la exclusión mutua.

B. *Benchmarks*

Evaluamos nuestra propuesta utilizando un conjunto de *benchmarks* cuyas secciones críticas pueden traducirse a atómicos MAD. Recogemos estadísticas para la región de interés para dichos *benchmarks*, es decir, después de la fase de inicialización de la aplicación y antes de la entrada-salida.

B.1 **Operaciones Atómicas Multidirecciones**

Sustituir *locks*/HTM en secciones críticas por una MAD atomic evita los tiempos de adquisición de *locks*, elimina los bucles de reintento por los abortos y permite que todos los núcleos procedan en paralelo (si no se encuentran conflictos). **Bitcoin** es una aplicación que realiza transacciones de pago en bitcoin [22]. Emula las operaciones que se suelen realizar en la red bitcoin sobre un conjunto de monederos bitcoin. **Water-NS** y **Water-SP** son dos aplicaciones de la suite de *benchmarks* Splash-2 que modelan las fuerzas entre las moléculas de agua [23].

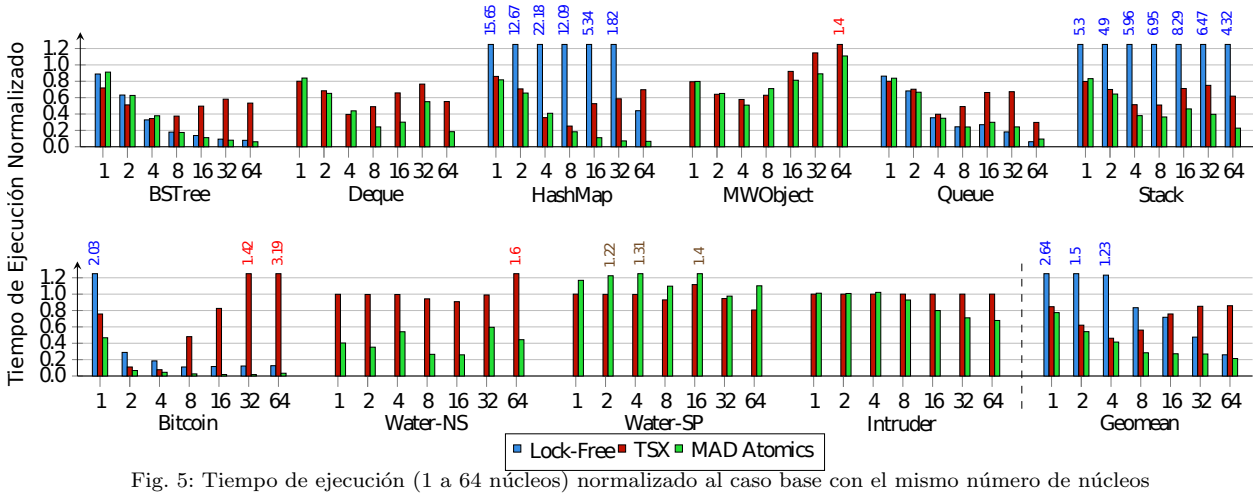

Fig. 5: Tiempo de ejecución (1 a 64 núcleos) normalizado al caso base con el mismo número de núcleos

B.2 **Operaciones MCAS**

También evaluamos MAD para varias estructuras de datos concurrentes ampliamente utilizadas en aplicaciones reales.

BSTree utiliza un árbol de búsqueda binario adecuado para almacenar datos jerárquicos. **Deque** implementa una cola de doble extremo (los elementos pueden ser añadidos y eliminados desde la parte delantera o trasera). **HashMap** implementa una matriz asociativa que asigna claves a valores utilizando una función hash para indexar los elementos. **MWObject** es una aplicación sencilla que incrementa un conjunto de cuatro variables [24], [25]. **Queue** es una aplicación que utiliza una cola para almacenar colecciones de objetos en orden FIFO. **Stack** implementa una cola de orden LIFO (el último elemento insertado es el primero en ser eliminado). **Intruder** es una aplicación de la suite STAMP que modela un sistema de detección de intrusiones en la red [26].

VI. Resultados

La Figura 5 muestra el tiempo de ejecución normalizado para las aplicaciones seleccionadas y las estructuras de datos concurrentes. MAD atomics supera a la implementación *mutex-locks* en todos los casos excepto en Water-SP. Water-SP es la única aplicación afectada por este punto de sincronización adicional, principalmente porque casi no tiene colisiones ni reintentos. Esta hipótesis se ve reforzada por el hecho de que TSX supera a MAD incluso para un hilo. Bitcoin es el que más se beneficia de los atómicos MAD, con una reducción del tiempo de ejecución de hasta un 98 % sobre *mutex-locks* (cuando se ejecuta en 32 hilos). La versión básica de Bitcoin utiliza una implementación que tiene un único *mutex-lock* para bloquear la tabla de transacciones. Esto muestra los enormes beneficios potenciales de los atómicos MAD en códigos escritos por programadores inexpertos que utilizan *mutex-locks* de grano grueso. De media, los atómicos MAD funcionan entre un 40 % y un 50 % mejor que los *mutex-locks* para el mismo número de hilos. A medida que aumenta el número de hilos, la sobrecarga de TSX debida a los conflictos adicionales supera las mejoras de rendimiento, lo que hace que sea menos escalable que MAD atomics.

La escalabilidad mejora para todas las aplicaciones con respecto a *mutex-locks*, especialmente para BSTree, HashMap, Bitcoin y Water-NS, que ahora pueden escalar fácilmente hasta 16 hilos (Figura 6). La escalabilidad de TSX está limitada a 4 hilos (8 para HashMap), y sólo supera ligeramente a los atómicos MAD para Water-SP. Para la mayoría de las aplicaciones, las secciones críticas son demasiado pequeñas y alta contención para obtener alguna ventaja de las implementaciones actuales de la memoria transaccional: debido a su política de solicitante-ganador, los abortos recurrentes inducidos por el conflicto en TSX eventualmente hacen que los hilos recurran a ejecutar las secciones críticas de forma no especulativa, en exclusión mutua. En resumen, MAD atomics es un enfoque mucho más sencillo y elegante para garantizar la consistencia sin bloqueos.

Por último, la Figura 7 muestra las instrucciones ejecutadas normalizadas para los diferentes *benchmarks*. Una reducción de las instrucciones ejecutadas (no necesariamente retiradas) se traduce en una reducción de la energía consumida por el procesador. Para 16 hilos, los atómicos MAD ejecutan, de media, sólo el 21 % del número de instrucciones original. Esta tendencia se mantiene hasta los 64 hilos. TSX le sigue de cerca, con un 41 % de instrucciones ejecutadas en comparación con la implementación base. Esto significa que MAD atomics no sólo es más rápido que TSX, sino también más eficiente energéticamente. Y esto sin incluir la energía extra que requiere el hardware de TSX.

VII. Conclusiones

Las operaciones atómicas multidirección eficientes, no especulativas y libres de bloqueo son una característica deseable y necesaria en los procesadores actuales. Este trabajo proporciona una implementación libre de *deadlocks* para las operaciones atómicas multidirección que tiene en cuenta las limitaciones de recursos reales del hardware. Esto se consigue mediante el protocolo de coherencia y estableciendo un orden de bloqueo predeterminado, con un coste hardware de menos de 68 bytes de almacenamiento extra por núcleo. Los atómicos MAD permiten una implementación eficiente de la primitiva MCAS y abren la

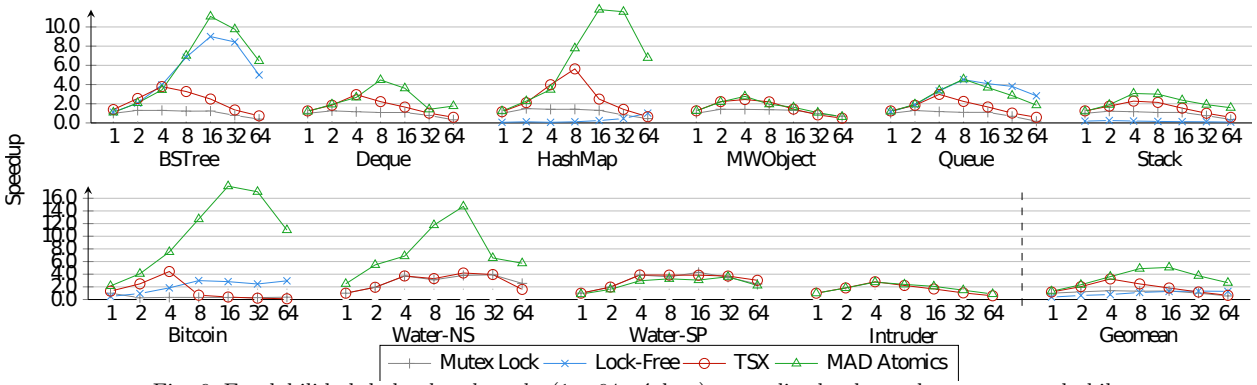

Fig. 6: Escalabilidad de los *benchmarks* (1 a 64 núcleos) normalizada al caso base con un solo hilo

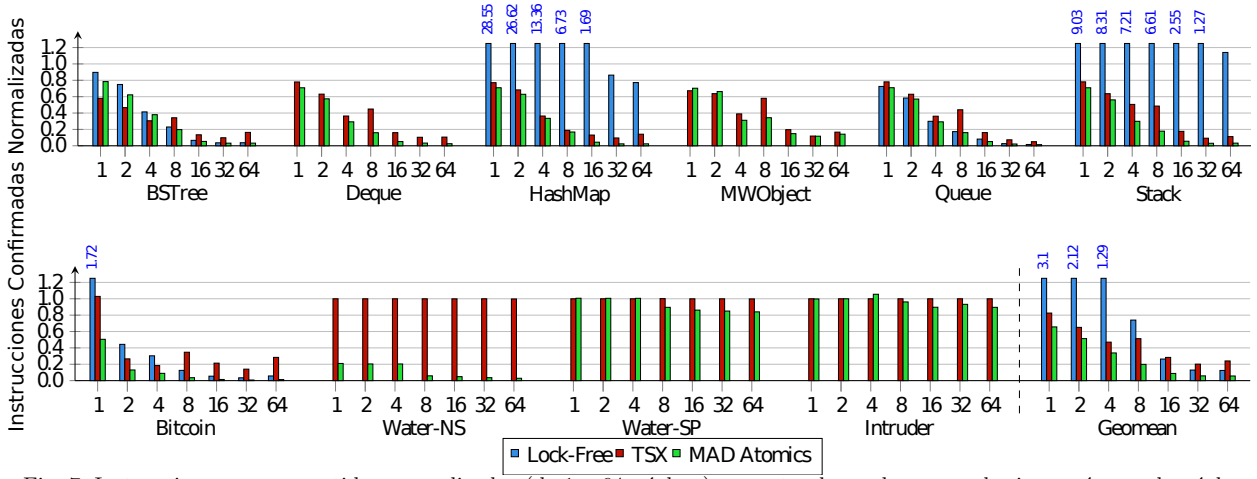

Fig. 7: Instrucciones comprometidas normalizadas (de 1 a 64 núcleos) respecto al caso base con el mismo número de núcleos

puerta a muchas otras primitivas atómicas multidirección. Como los atómicos MAD no son especulativos, ofrecen un mejor rendimiento que HTM bajo contención. Lo ideal es que ambas técnicas se utilicen conjuntamente cuando sea necesario.

Nuestros resultados muestran que para las aplicaciones evaluadas y las estructuras de datos concurrentes, los atómicos MAD superan a los *locks* software en hasta 18× (3,4× de media) y 2,7× de media en comparación con TSX, mejorando la escalabilidad desde un núcleo (bloqueos por software) hasta 16 núcleos. Las mejoras en rendimiento y en número de instrucciones ejecutadas se traducen directamente en un ahorro de energía, que alcanza 23× de media para 32 y 64 núcleos.

Agradecimientos

Este proyecto ha recibido financiación del *European Research Council (ERC)* bajo el programa *Horizon 2020* (Nº 819134), el proyecto *Vetenskapsradet* (Nº 2018-05254) y el *European Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC)* (Nº 801051).

Referencias

[1] Guancheng Chen and Per Stenstrom, "Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[2] Stijn Eyerman and Lieven Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 362–370, June 2010.

[3] Herb Sutter, "Writing lock-free code: A corrected queue," *Dr. Dobb's Journal*, October 2008.

[4] Herb Sutter, "The trouble with locks," *C/C++ Users Journal*, March 2005.

[5] Maurice Herlihy and J. Eliot B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 289–300.

[6] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun, "Programming with transactional coherence and consistency (TCC)," in *11th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2004, pp. 1–13.

[7] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar, "Performance evaluation of intel transactional synchronization extensions for high-performance," in *2013 Conf. on Supercomputing (SC)*, Nov. 2013, pp. 19:1–19:11.

[8] Douglas Macgregor, David S. Mothersole, and John Zolnowsky, "Method and apparatus for a compare and swap instruction," U.S. Patent 4584640, Apr. 1986.

[9] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek, "Multiple reservations and the oklahoma update," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 4, pp. 58–71, Nov. 1993.

[10] Ravi Rajwar and James R Goodman, "Transactional lock-free execution of lock-based programs," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 5–17.

[11] Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, and Smruti R. Sarangi, "A hardware implementation of the mcas synchronization primitive," in *2017 Design, Automation, and Test in Europe (DATE)*, Mar. 2017, pp. 918–921.

[12] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin

Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[13] Edsger W. Dijkstra, "Hierarchical ordering of sequential processes," *EDW-310, E.W. Dijkstra Archive, Center for American History, University of Texas at Austin.*

[14] Alberto Ros and Stefanos Kaxiras, "Non-speculative store coalescing in total store order," in *45th Int'l Symp. on Computer Architecture (ISCA)*, June 2018, pp. 221–234.

[15] Edward G. Coffman, Melanie Elphick, and Arie Shoshani, "System deadlocks," *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, June 1971.

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Number 325462-072US. May 2020.

[17] ARM, *ARM® Cortex®-X1 Core Technical Reference Manual*, Number r1p1. May 2020.

[18] Agner Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2020, Available at `https://www.agner.org/optimize/microarchitecture.pdf`.

[19] "Ice lake (client) - microarchitectures - intel," .

[20] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *17th Int'l Symp. on Computer Architecture (ISCA)*, June 1990, pp. 364–373.

[21] Agner Fog, "Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns," 2018, Available at `http://www.agner.org/optimize/instruction_tables.pdf`.

[22] Daniel Kondor, ," .

[23] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal 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.

[24] Steven Feldman, Pierre Laborde, and Damian Dechev, "A practical wait-free multi-word compare-and-swap operation.," 2013.

[25] Steven Feldman, Pierre Laborde, and Damian Dechev, "A wait-free multi-word compare-and-swap operation," *International Journal of Parallel Programming*, Aug. 2014.

[26] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Int'l Symp. on Workload Characterization (IISWC)*, Sept. 2008, pp. 35–46.