Aproximaciones Hardware a la Memoria Transaccional

J. Rubén Titos, Manuel E. Acacio, José M. García
Dpto. Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
Campus de Espinardo 30100 Murcia
{rtitos,meacacio,jmgarcia}@ditec.um.es

Resumen

Este artículo presenta una descripción del estado del arte de los sistemas de memoria transaccional hardware. En particular, describimos cómo las diferentes propuestas actuales satisfacen las propiedades básicas de atomicidad y aislamiento, todo ello sin perder de vista aspectos relevantes como el anidamiento de transacciones, la virtualización o la semántica transaccional. Para dar respuesta a la falta de flexibilidad que caracteriza a las implementaciones hardware actuales, mostramos una primera aproximación hacia un sistema híbrido que soporta diferentes políticas de gestión de versiones y detección de conflictos (a nivel hardware), capaz de adaptarse dinámicamente a la cargas de trabajo que se ejecutan.

1. Introducción y Motivación

En los últimos años, la explotación del paralelismo a nivel de hilo o proceso como forma de mejorar el rendimiento ha pasado de ser una característica exclusiva de los servidores de gama alta o los super-computadores científicos, a estar presente en todos los segmentos del mercado. Fabricantes de microprocesadores como Intel o AMD han optado por los multiprocesadores en un chip (CMP, *Chip MultiProcessors*) como una forma practicable y realista de transformar el creciente número de transistores disponibles en rendimiento escalable. Hoy en día, los CMPs se han hecho omnipresentes y son empleados en todo tipo de plataformas, desde servidores hasta portátiles.

En la práctica, el éxito de los sistemas basados en CMPs está limitado por la dificultad de la programación multi-hilo. Escribir programas correctos y eficientes con los modelos de programación paralela convencionales es todavía una tarea considerablemente compleja, limitada sólo a unos pocos desarrolladores expertos. Estos modelos plantean un gran reto al programador puesto que exigen un difícil compromiso entre rendimiento y corrección. Explotar el máximo paralelismo presente es una tarea ardua y proclive a errores, mientras que programar de forma conservadora produce programas correctos más rápidamente a costa de degradar la escalabilidad. Así pues, a menos que se desarrollen nuevos modelos de programación que simplifiquen el desarrollo de aplicaciones paralelas, el potencial de rendimiento de los CMPs estará limitado a cargas de trabajo multiprogramadas y unas pocas aplicaciones del dominio de los servidores.

A este respecto, el modelo de programación de Memoria Transaccional (TM, *Transactional Memory*) puede reducir significativamente la dificultad de escribir programas concurrentes correctos. En particular, en respuesta a los problemas de rendimiento, escalabilidad y programabilidad que acarrea la programación con cerrojos del modelo tradicional de memoria compartida, la memoria transaccional ofrece una forma alternativa de sincronización más sencilla para el programador y más eficientemente implementada en hardware. Con TM, los programadores expresan *qué* regiones del código deben ser ejecutadas en exclusión mutua, en lugar de tener que especificar *cómo*

alcanzar dicha atomicidad usando cerrojos u otras construcciones de sincronización explícita. El inicio y fin de cada transacción en el código puede delimitarse mediante instrucciones `begin_xaction` y `end_xaction`, mediante bloques `atomic`, etc. Por su parte, el sistema TM ejecuta las transacciones concurrentemente de forma optimista, al tiempo que detecta y resuelve dinámicamente los conflictos que puedan surgir entre ellas. La principal ventaja de la memoria transaccional es la combinación de interfaz intuitiva e implementación eficiente: el esfuerzo de programación es similar al uso de cerrojos de grano grueso, al tiempo que se consigue la escalabilidad y rendimiento de los cerrojos de grano fino.

Un sistema TM ha de satisfacer en todo momento las propiedades de *atomicidad* y *aislamiento*. La primera requiere que una transacción se ejecute completamente o no se ejecute en absoluto, mientras que la segunda exige que el estado intermedio atravesado por una transacción parcialmente completada permanezca oculto al resto de transacciones. En este artículo exploramos brevemente el estado del arte de los sistemas de memoria transaccional hardware (HTMs), si bien hacemos alguna alusión a varias implementaciones híbridas.

Si bien Herlihy y Moss ya idearon a principios de los noventa un sistema de memoria transaccional en hardware [8], no ha sido hasta 2004 con la publicación de [7] que los HTMs han adquirido protagonismo. En comparación, la sobrecarga de partida de los sistemas TM software está entre un 40 y un 200 % por encima de las implementaciones hardware. Si el rendimiento es la principal ventaja de los HTMs, entre sus inconvenientes están el coste y la complejidad de implementación, debida a la modificación de elementos críticos (cachés, protocolo de coherencia), la virtualización, etc.

El resto del artículo se organiza como sigue. En la sección 2 repasamos los distintos modelos HTM que han aparecido en trabajos previos, clasificándolos en función de cómo implementan los mecanismos requeridos por la memoria transaccional. La sección 3 discute otros aspectos importantes de la memoria transaccional como son la semántica, el anidamiento y la virtualización de las transacciones. En la sección 4 analizamos algunos de los benchmarks que son empleados actualmente para evaluar los HTMs propuestos. En la sección 5 introducimos nuestra propuesta hacia sistemas HTM más flexibles. Finalmente, la sección 6 presenta las principales conclusiones del artículo y apunta la línea de investigación que estamos iniciando en este campo.

2. Implementaciones Hardware de la Memoria Transaccional

Garantizar las propiedades de atomicidad y aislamiento que un sistema TM debe satisfacer requiere la *detección y resolución de conflictos*, y la *gestión de versiones* de datos. Las políticas aplicadas a cada uno de estos tres mecanismos constituyen el espacio de diseño de los sistemas TM hardware.

2.1. Gestión de Versiones

Este mecanismo maneja el almacenamiento simultáneo de las dos versiones por las que atraviesan los datos modificados en una transacción: el nuevo valor será visible si la transacción hace *commit* mientras que el viejo valor es mantenido en caso de que la transacción aborte. Sólo una las dos versiones puede guardarse en la dirección de memoria del dato en cuestión, *in-situ*; la otra versión deberá ser almacenada en algún otro lugar, digamos, a un lado. Dependiendo de qué valor se guarda *in-situ* y cuál a un lado, la política de gestión de versiones se denomina ansiosa (*eager*), si el nuevo valor se guarda *in-situ* y el viejo a un lado, o perezosa (*lazy*), si el valor viejo permanece *in-situ* y el nuevo se guarda a un lado.

Los HTMs con gestión de versiones ansiosa guardan los viejos valores en un *undo-log*, que sólo se utiliza en caso de aborto para deshacer los cambios y restaurar la memoria a su estado anterior a la transacción. Ejemplos de HTM con política de versiones ansiosa son UTM [1], LogTM [9], y otras propuestas basadas en este último [12, 2].

En cambio, los HTMs con política de versiones perezosa utilizan un búfer de escritura o la

propia caché para mantener temporalmente el estado especulativo, hasta que las nuevas versiones son escritas a memoria o descartadas, según se haga commit o abort. Entre los sistemas que utilizan esta política encontramos TCC [7], LTM [1], VTM [10] y Bulk [6].
Una gestión de versiones ansiosa hace que los commits sean más rápidos que los abortos, pues los nuevos valores ya están en su posición de memoria correspondiente y sólo es necesario descartar el log. En cambio, con una política perezosa de gestión de versiones es más fácil quitarse de en medio una transacción abortada, ya que los valores en memoria siguen siendo los mismos que había al inicio de la transacción, y sólo hay que vaciar el búfer especulativo. En general, aunque es de esperar que los commits sean más frecuentes que los aborts, no hay una política de gestión de versiones que sea óptima en todos los casos.
Un problema que ha de afrontar la gestión de versiones perezosa es qué hacer cuando los datos escritos por la transacción desbordan la capacidad o asociatividad del búfer especulativo o caché. En TCC encontramos una primera aproximación: la transacción se detiene hasta obtener permiso de commit, y partir de ahí se ejecuta hasta su fin, escribiendo los resultados directamente en memoria compartida. Este comportamiento produce un efecto de serialización bastante negativo en el rendimiento, puesto que ninguna otra transacción se puede completar mientras el modo *overflow* está activo. Otras propuestas posteriores, como VTM o LTM, soportan transacciones de tamaño ilimitado al utilizar estructuras de datos en memoria virtual, si bien esto acarrea un considerable incremento en la complejidad de estos sistemas. En VTM, el estado especulativo se separa en dos partes, cacheado localmente y desbordado a memoria. En LTM, los datos especulativos que desbordan la caché se guardan en una tabla hash en memoria principal.

2.2. Detección de Conflictos

Un conflicto entre dos transacciones se produce cuando el conjunto de datos escritos por una transacción se solapa con el conjunto de datos leídos o escritos por otra transacción concurrente. Dicho de otra forma, dos transacciones concurrentes entran en conflicto cuando ambas acceden a una misma dirección de memoria, y al menos uno de los accesos es una escritura. Para dar soporte a la detección de conflictos, cualquier HTM debe seguir la pista de los datos leídos y escritos por cada transacción, lo cual se puede llevar a cabo con *granularidad* de bloque o de palabra. En el primer caso, sólo dos bits por bloque (R y W) son necesarios, pero pueden aparecer falsos conflictos cuando transacciones concurrentes acceden a diferentes palabras de un mismo bloque. Llevar cuenta de los conjuntos R y W con granularidad de palabra evita esta situación, pero requiere $2w$ bits por bloque para guardar esta información, donde w hace referencia al número de palabras por bloque. Muchas propuestas optan por aumentar la caché con bits R y W, para indicar si el bloque ha sido leído o escrito por la transacción actual. En cambio, Bulk [6] utiliza *firmas* para codificar mediante *hashing* las direcciones accedidas por un hilo, agrupando en sendos registros los conjuntos R y W.

Las políticas de detección de conflictos varían en función de *cuándo* se examina la información de los conjuntos R y W. Unos HTMs se basan en el protocolo de coherencia de caché para detectar conflictos conforme el hilo busca realizar una referencia a memoria, mientras que otros optan por aplazar la detección hasta que la primera de las transacciones en conflicto intenta el commit. Estas dos políticas se denominan *ansiosa o pesimista*, y *perezosa u optimista*, respectivamente. Sistemas como LTM, UTM, VTM y LogTM utilizan detección de conflictos ansiosa, mientras que TCC y Bulk se decantan por una política perezosa. La detección de conflictos ansiosa puede mejorar el rendimiento ya que puede resolver algunos conflictos utilizando detenciones en lugar de drásticos abortos. Por otro lado, una política perezosa puede mitigar el impacto de algunos conflictos encadenados y permite la implementación del chequeo de conflictos por lotes.

Independientemente de la política seguida, los HTMs que aumentan la caché con bits R/W deben afrontar de nuevo el problema del desbordamiento, ya sea por falta de capacidad o

asociatividad de la caché. Si bien recientemente se ha propuesto utilizar una caché de permisos R&W para reducir la frecuencia con la que se produce el desbordamiento [2], antes o después es necesario afrontar dicho problema. TCC lo hace de manera simplista: cuando algún bloque transaccional (bit R o W activo) ha de ser reemplazado, el sistema transita a modo *overflow*, tal como se comentó anteriormente. Otras propuestas posteriores refinan un poco más la solución: LogTM extiende el protocolo de directorio con estados pegajosos (*sticky*) a los que se transita cuando un procesador P expulsa un bloque transaccional. En estos estados, el directorio sigue reenviando las peticiones al procesador P, que pese a no tener ninguna información del bloque, puede inferir conflictos potenciales a partir de la petición. LTM, UTM y VTM utilizan estructuras de datos en memoria, que además de ser bastante complejas, incrementan la sobrecarga acarreada por la detección de conflictos en situaciones de desbordamiento. En Bulk, la información para la detección de conflictos está contenida en las firmas R y W. Cada firma codifica un super-conjunto de los conjuntos de lectura y escritura de la transacción, así que aunque existan bloques en el área de overflow en memoria (bloques escritos por la transacción), no es necesario acceder a dicha zona para comprobar si existen conflictos. Por contra, la codificación *hash* de la firma provoca en ocasiones la aparición de falsos positivos. En la figura 1 se puede ver una tabla que clasifica las diversas propuestas HTM de acuerdo con las características descritas.

2.3. Resolución de Conflictos

Una vez que se ha detectado un conflicto entre dos o más transacciones concurrentes, la política de resolución dictamina qué hacer para solventar dicha situación. Las políticas de resolución son independientes del momento en que se detecta el conflicto. Se puede optar por detener o abortar al peticionario/committer, según el escenario, o bien abortar a los demás. Generalmente, los HTM con política de detección optimista (como TCC y Bulk) resuelven el conflicto dejando que la transacción que intenta el commit gane el conflicto, abortando al resto. En cambio, existen varios HTMs propuestos cuya política de detección es ansiosa, que utilizan estrategias de resolución diferentes: LTM y VTM permiten que el hilo peticionario gane el conflicto, abortando a los demás; UTM y LogTM, sin embargo, bloquean la petición conflictiva para detener al peticionario, evitando interbloqueos de forma conservadora.

		Gestión de Versiones	
		Ansiosa	*Perezosa*
Detección de Conflictos	*Ansiosa*	UTM [1] LogTM [8] LogTM-SE [11] OneTM [2]	LTM [1] VTM [9]
	Perezosa	-	TCC [5] Bulk [4] SigTM* [3]

* Híbrido

Figura 1: Clasificación de los HTM propuestos.

3. Otros Aspectos de HTM

3.1. Anidamiento de Transacciones

Una de las ventajas del modelo de programación de memoria transaccional es que favorece la composición de software, es decir, la capacidad de que un módulo A invoque a B (que a su vez puede invocar a C, etc.) conociendo únicamente su interfaz y no su implementación. El método que prevalece en la actualidad está basado en cerrojos y es incapaz de ofrecer una composición transparente; para garantizar la seguridad y evitar los interbloqueos, los programadores a menudo necesitan saber qué cerrojos están (o pueden ser) ocupados tanto por el invocador como por el invocado.
Para facilitar la composición de software, los sistemas TM deben soportar el anidamiento de transacciones con el fin de permitir que un módulo invoque a otro sin conocer siquiera si éste utiliza transacciones. El método de implementación más directa y sencilla se denomina *anidamiento cerrado* y se basa en extender el aislamiento de una transacción interna hasta que la más externa hace commit, es decir, la transacción anidada se suma a la externa (*flattening*). Esto hace que, ante un conflicto,

todo el trabajo realizado desde el inicio de la transacción más externa haya de ser abortado. Otra opción más difícil de llevar a la práctica es el *anidamiento abierto*, en el cual una transacción interna libera el aislamiento y hace visibles sus cambios al hacer commit. Puesto que los cambios en la memoria causados por la transacción interna no se deshacen si la externa aborta, es necesario usar una *acción compensatoria* a un mayor nivel de abstracción, que deshaga el progreso conseguido. Este tipo de anidamiento puede incrementar el paralelismo, pero también conlleva un aumento de la complejidad en hardware y software.

3.2. Virtualización

Muchos de los sistemas HTM propuestos utilizan cachés hardware para manejar la gestión de versiones, y se ayudan del protocolo de coherencia para la detección de conflictos. No obstante, para que el paradigma de memoria transaccional sea útil a los programadores y pueda alcanzar amplia aceptación, es importante que las transacciones no estén limitadas por los recursos físicos ni ligadas a ninguna implementación hardware específica. Los sistemas TM deben garantizar una ejecución correcta a pesar de que las transacciones excedan su *quantum* de tiempo asignado por el planificador, desborden la capacidad o asociatividad de las cachés o incluyan más niveles de anidamiento de los que el hardware soporta. En otras palabras, un sistema TM debe virtualizar transparentemente el tiempo, el espacio y la profundidad de anidamiento. La virtualización de TM permite que las transacciones sobrevivan a los desbordamientos, paginación, cambios de contexto, migración de hilos y anidamiento extendido. Para conseguirla, no es posible basarse en estructuras ligadas a un procesador, sino que el estado transaccional (*log*, *R&W sets*) debe situarse en direcciones virtuales accesibles por el sistema operativo.
Un buen esquema de virtualización debe satisfacer ciertos requisitos de rendimiento y corrección: i) ser completamente transparente al usuario, ii) preservar la atomicidad y aislamiento bajo cualquier circunstancia, iii) no afectar al rendimiento del caso común, cuando la virtualización no se necesita, y iv) las transacciones virtualizadas no deben tener un efecto significativo en el rendimiento de otras transacciones concurrentes no virtualizadas.
Soportar mediante hardware la virtualización de transacciones es la forma óptima de cara al rendimiento. Sin embargo, la virtualización es por naturaleza un sistema de respaldo, que sólo se utiliza cuando los mecanismos hardware no son suficientes. Puesto que la mayoría de las transacciones no exceden los recursos hardware, es muy importante encontrar un equilibrio entre coste y rendimiento a la hora de diseñar el esquema de virtualización para un sistema de memoria transaccional.
Los primeros sistemas TM híbridos surgen con el fin de afrontar los retos de virtualización que tienen los HTMs. Las propuestas híbridas más recientes [5, 11] apuestan por implementar ciertos mecanismos en hardware, para reducir la sobrecarga de las transacciones software y aportar flexibilidad al sistema, al tiempo que se consigue una virtualización más sencilla. Por ejemplo, SigTM [5] utiliza firmas *hash* en hardware para acelerar la detección de conflictos, mientras que el resto de la funcionalidad transaccional, incluyendo la gestión de versiones, se implementa en software.

3.3. Semántica Transaccional

A la hora de definir una semántica transaccional clara, es obvio que las transacciones deben ser atómicas unas con respecto a otras. Sin embargo, su relación con el código no-transaccional es menos evidente, y como resultado de ello se han definido dos modelos de razonamiento sobre el ámbito de atomicidad [3]. En la semántica de *atomicidad fuerte*, las transacciones se ejecutan atómicamente unas con respecto a otras, así como respecto al código no transaccional. En cambio, en la semántica de *atomicidad débil*, las transacciones son atómicas sólo con respecto a otras transacciones, y por tanto su ejecución puede estar entrelazada con código no transaccional.
Un sistema de memoria transaccional necesita especificar si es fuerte o débilmente atómico, de forma similar a cómo un multiprocesador de memoria compartida tradicional indica

qué modelo de consistencia de memoria implementa. El programador debe conocer qué semántica de atomicidad implementa el hardware sobre el que desarrolla, pues un código que funciona correctamente bajo la semántica de atomicidad débil puede sufrir interbloqueo bajo atomicidad fuerte. De la misma forma, si el programador cree que el sistema es fuertemente atómico, pero en realidad sólo es débilmente atómico, un programa podría fallar debido a condiciones de carrera entre una transacción y código no transaccional.

4. *Benchmarks* Transaccionales

Hasta ahora, la mayoría de los estudios sobre sistemas de memoria transaccional vienen utilizando en su evaluación aplicaciones paralelas pertenecientes al paquete SPLASH-2. Estos *benchmarks* explotan el paralelismo de grano fino y han sido cuidadosamente optimizados con los años con el objetivo de evitar la sobrecarga de sincronización. Por esta razón, transformar sus regiones críticas protegidas con cerrojos en bloques transaccionales produce programas que pasan una pequeña parte de su tiempo de ejecución en transacciones de grano fino. Este comportamiento podría no ser representativo de los nuevos programas paralelos que se desarrollen con técnicas de memoria transaccional. No olvidemos que el principal atractivo de esta tecnología es la capacidad que tiene de ofrecer un buen rendimiento con código paralelo que usa con frecuencia transacciones de grano grueso.

Actualmente, la ausencia de software paralelizado bajo el modelo de memoria transaccional hace difícil encontrar un conjunto de aplicaciones paralelas representativas que actúe como marco de referencia común para la investigación en este área. De momento, el único conjunto de aplicaciones transaccionales disponible para toda la comunidad investigadora es STAMP (*Stanford Transactional Applications for Multi-Processing*) [5]. Esta nueva *suite* consta de programas paralelizados desde cero, que usan transacciones de grano grueso para ejecutar tareas concurrentes sobre estructuras de datos irregulares, como árboles o grafos. Los *benchmarks* que por ahora componen STAMP son: *delaunay*, que implementa el algoritmo Delaunay para generación de mallas; *genome*, que ordena un conjunto de genes para encontrar la secuencia original; *kmeans*, que agrupa objetos en k particiones de acuerdo con ciertos atributos; y *vacation*, que implementa un sistema de reservas de viaje.

5. Hacia una implementación HTM flexible

En un reciente estudio [4] se comparan tres HTMs con diferentes políticas de gestión de versiones y detección y resolución de conflictos. Además de mostrar que ningún diseño es superior a los demás para todas las aplicaciones evaluadas, el artículo identifica ciertos patrones de ejecución comunes que degradan el rendimiento, señala las características de los programas que los provocan, y pone de manifiesto que cada sistema se ve afectado por unas u otras *patologías* de rendimiento, en función de la forma en la que gestiona las versiones de los datos, detecta conflictos y los resuelve.

Estas interacciones entre sistema y carga de trabajo son el principal causante de la degradación de rendimiento que sufren unos y otros diseños [4]. Puesto que a priori se desconoce la naturaleza del código transaccional que va a ejecutar un HTM, sería deseable dotar a los HTMs de cierta flexibilidad de políticas, con el fin de evitar la aparición de estas interacciones indeseadas. A este respecto, nosotros apostamos por un enfoque híbrido en el que el hardware implementa varios mecanismos de gestión de versiones y resolución de conflictos, y los pone a disposición del software; es éste quien decide dinámicamente cuál es la política óptima a aplicar en cada momento, en función de las características de la aplicación en curso.

Nuestro trabajo toma varias ideas de otros HTMs y trata de combinarlas para conseguir un sistema de memoria transaccional híbrido del cual esbozamos algunas características a continuación. Nuestra meta es conseguir, de forma sencilla y con el mismo coste hardware de otros HTMs rígidos, un sistema flexible capaz de adaptar a petición del software la forma

en la que preserva la atomicidad y aislamiento de las transacciones. Para ello, nuestra aproximación soporta a nivel hardware la combinación de políticas de gestión de versiones (ansiosa y perezosa) y detección de conflictos (optimista y pesimista). Nosotros proponemos que el software utilice información obtenida tanto estática como dinámicamente para establecer la política adecuada en cada instante. En tiempo de compilación, es posible estimar parámetros como la longitud de las transacciones, su tamaño y nivel de anidamiento, e insertar pistas (*hints*) en el código, para informar acerca de lo que viene después. Posteriormente, la frecuencia de los conflictos será recopilada durante la ejecución gracias un contador accesible por software, que el hardware incrementa en cada conflicto y resetea periódicamente.

De esta forma, nuestra aproximación combinará las políticas ansiosas de gestión de versiones y detección de conflictos de LogTM-SE [12] con las perezosas de Bulk [6], sin necesitar para ello más hardware del que usa LogTM-SE. Además de las firmas *hash* R y W, nuestra propuesta hereda de LogTM-SE los registros base y puntero para soportar un *log* transaccional. Su novedad radica en que el *log* puede actuar como *before-image* o *after-image*: junto a la dirección virtual del bloque escrito, se guarda su antiguo o nuevo valor, según la política sea ansiosa o perezosa, respectivamente. Así pues, el espacio asignado al *log* en memoria virtual puede actuar bien como búfer de escritura especulativo o bien como *undo-log*. En función de la frecuencia de los conflictos, el software puede cambiar la gestión de versiones para agilizar aborts o commits, según convenga. Independientemente de la política, las escrituras al log serán a menudo aciertos de caché, puesto que éste es cacheable, privado al hilo, y habitualmente contendrá pocos bloques. Cuando se utiliza como *write buffer*, tanto las lecturas como las escrituras transaccionales tienen que chequear la firma W. En caso positivo, las lecturas obtienen del log el valor más actualizado, mientras que las escrituras lo recorren para sobreescribir la entrada. Creemos que esto no supone una sobrecarga excesiva, puesto que el log suele ser pequeño y estar cacheado, y con ayuda del compilador estos casos se pueden minimizar. Si la prueba es negativa, sólo las escrituras acceden al log para añadir una nueva entrada al final. Como en LogTM, los reemplazos de bloques transaccionales no tienen efecto en la gestión de versiones y no necesitan de ningún búfer.

Utilizando firmas, es posible soportar la detección de conflictos pesimista de LogTM-SE y la detección optimista de Bulk. Con política de detección pesimista, el sistema se basa en el protocolo de coherencia de caché para garantizar el aislamiento: se prohibe a cualquier elemento de procesamiento cachear un bloque que esté en el conjunto W de otro, ni tampoco cachear exclusivamente un bloque que esté en el conjunto R de otro núcleo. Como en LogTM, el problema de la victimización de bloques transaccionales se resuelve de forma elegante utilizando *sticky states*. Con esta política, el commit es una operación local muy rápida que sólo resetea las firmas y los punteros del log. Si se utiliza la detección optimista de Bulk, los conflictos se detectan cuando el hilo, tras conseguir permiso de commit, envía por *broadcast* su firma W. Los hilos receptores comprueban dicha firma contra las suyas propias, abortan en caso de conflicto o bien invalidan en sus cachés los bloques correspondientes. Toda la lógica requerida por la desambiguación/invalidación por lotes de Bulk está dentro de las necesidades de LogTM-SE, así que soportar la política de detección optimista no acarrea significativo hardware adicional.

6. Conclusiones y trabajo futuro

En este artículo hemos presentado un breve repaso del estado del arte de los sistemas de memoria transaccional hardware. Hemos clasificado las propuestas existentes en función de cómo soportan la atomicidad y el aislamiento de las transacciones, y atendido a aspectos como el anidamiento, la virtualización y la semántica transaccional. Finalmente, hemos introducido nuestra propuesta para un sistema flexible que soporte en hardware diferentes políticas de gestión de versiones y detección de conflictos. Además de afinar el diseño y eva-

luar el sistema propuesto, la línea de trabajo que seguiremos perseguirá solventar otras patologías de rendimiento que surjan como fruto de la interacción del sistema con la aplicación. Por ello, centraremos nuestro trabajo en la búsqueda de soluciones híbridas que combinen rendimiento y flexibilidad.

Referencias

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb 2005.

[2] Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34st Annual International Symposium on Computer Architecture*, Jun 2007.

[3] Colin Blundell, E Christopher Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

[4] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, Jun 2007.

[5] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[6] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, June 2006.

[7] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[8] Maurice Herlihy and Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–301, May 1993.

[9] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[10] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Jun 2005.

[11] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, Jun 2007.

[12] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.