# Sistemas de Memoria Transaccional Hardware: Políticas, Conflictos y su Influencia en el Rendimiento

***José Rubén Titos Gil*** rtitos@ditec.um.es

## MEMORIA TESIS DE MASTER

Master en "Tecnologías de la Información y Telemática Avanzadas" – Curso 2006/07
Dpto. Ingeniería de la Información y las Comunicaciones
Dpto. Ingeniería y Tecnología de Computadores
Facultad de Informática. Universidad de Murcia.
Campus de Espinardo. 30100 Murcia. Spain.

# Resumen Extendido

La dificultad de la programación multihilo sigue siendo uno de los mayores obstáculos que los programadores encuentran a la hora de explotar de forma efectiva los recursos computacionales de los ya omnipresentes chips multinúcleo. La Memoria Transaccional es una prometedora abstracción sobre el modelo de programación de memoria compartida, ya que tiene la capacidad de aliviar muchos de los retos que acarrea el uso de cerrojos en la programación paralela.

Los sistemas de memoria transaccional hardware (HTM) implementan los mecanismos necesarios para soportar eficientemente la semántica de transacciones, en particular, la gestión de versiones de datos y la detección y resolución de conflictos. Los diseños HTM actuales aplican políticas predeterminadas e invariables sobre dichos mecanismos, tratando de favorecer el comportamiento más comúnmente esperado en las aplicaciones. Muchas de estas propuestas asumen que las transacciones completadas exitosamente serán mucho más frecuentes que las abortadas en las cargas de trabajo transaccionales del futuro.

La primera parte de este trabajo presenta las diferentes políticas de gestión de versiones y detección de conflictos que componen el espacio de diseño de un sistema de memoria transaccional hardware, y señala la influencia que la elección de cada política tiene en el grado de impacto de los conflictos sobre el rendimiento del sistema. Para cada decisión de diseño del sistema, se alude a propuestas concretas para comentar las bondades e inconvenientes de la aproximación, siguiendo un orden evolutivo-cronológico a la hora de presentar las diferentes soluciones. Así, esta parte del trabajo introduce los sistemas de memoria transaccional hardware a un lector no experimentado hasta cubrir de forma breve el estado del arte en este área.

En la segunda parte esta tesis, tratamos de mostrar cuantitativamente cómo un grupo importante de aplicaciones desarrollado bajo el modelo de memoria transaccional está destinado por su naturaleza a experimentar muchos conflictos. Llevamos a cabo la primera caracterización de conflictos en un conocido sistema de memoria transaccional hardware (LogTM) que utiliza *benchmarks* verdaderamente transaccionales (STAMP). Lo que tratamos de mostrar con nuestra evaluación es que las transacciones abortadas pueden llegar a ser bastante frecuentes en algunos tipos de cargas de trabajo transaccionales, y que este hecho ha de ser considerado a la hora de afrontar los parámetros de diseño de los sistemas TM hardware, presentados en la primera parte del trabajo.

Con este estudio pretendemos demostrar que para que el modelo de memoria transaccional libere a los programadores del difícil compromiso entre programabilidad y prestaciones, los sistemas HTM deben ser capaces de obtener buenos niveles de rendimiento incluso en presencia de frecuentes conflictos y transacciones abortadas. Así, los sistemas HTM deben encontrar la manera de soportar políticas más flexibles tanto en los mecanismos de gestión de versiones y de detección de conflictos, además de esquemas más sofisticados de recuperación ante transacciones abortadas. Para dar respuesta a la falta de flexibilidad que caracteriza a las implementaciones hardware actuales, nosotros esbozamos una primera aproximación hacia un sistema híbrido que soporta diferentes políticas de gestión de versiones y detección de conflictos a nivel hardware, y pone a disposición del software mecanismos para adaptarse dinámicamente a la cargas de trabajo que se ejecutan.

*Palabras clave:* Multiprocesador en un chip (CMP), Sincronización, Memoria Transaccional (TM), Conflictos

# 1. Introducción y Motivación

Han transcurrido más de cuatro décadas desde que se enunciase la famosa *ley de Moore*, y su pronóstico todavía sigue vigente hoy día. Basándose en su observación de una tendencia de la microelectrónica, el a la postre co-fundador de Intel predijo que la cantidad de transistores incluidos en los circuitos integrados se duplicaría cada año y medio, aproximadamente. Este avance incesante de la microelectrónica ha hecho posible un crecimiento exponencial en el rendimiento de los microprocesadores durante estos cuarenta años. En la actualidad, cualquier ordenador personal supera con creces la potencia de cómputo de los grandes centros de cálculo de los años sesenta. En cualquier caso, un logro de esta magnitud no sólo ha estado sustentado por el escalado de la tecnología de integración, sino también por los avances en las técnicas de compilación y, en gran medida, por las innovaciones en la arquitectura de computadores.

## 1.1. Los Tres Grandes Muros

No obstante, conforme aumenta el número de transistores disponibles en el chip se hace cada vez más difícil traducir este mayor potencial en capacidad de cómputo real para arquitecturas monoprocesador. Los tres principales obstáculos al rendimiento serie se conocen comúnmente como *muro del ILP*, *muro del consumo* y *muro procesador-memoria*.

- *ILP*. A pesar de que todavía existen en las aplicaciones cantidades significativas de ILP por explotar, la complejidad de la lógica necesaria para extraerlo hace que ya no sea una decisión de diseño efectiva en términos de coste/rendimiento y consumo/rendimiento: Explotar más ILP requiere un aumento super-lineal en la complejidad de las unidades de ejecución y en su consumo de energía, sin conseguir con ello un *speedup* lineal. Además, las técnicas muy agresivas incrementan la energía consumida por cálculo útil, debido tanto al trabajo que se ha de descartar cuando falla la especulación como al mayor tamaño de los controladores.
- *Consumo*. La disipación de potencia en dispositivos digitales es proporcional a la frecuencia del reloj, lo cual impone límites naturales sobre la frecuencia. Mientras que las CPUs han incrementado su velocidad de reloj en un factor de 4000 en los últimos 10 años, la habilidad de los fabricantes para disipar calor ha alcanzado un límite físico. Junto a ello, la potencia disipada por la corriente de fuga se hace mayor conforme el tamaño de puerta disminuye, y cobra una importancia crucial con procesos de fabricación inferiores a 65 nm. Por todo ello, no es posible incrementar la frecuencia de forma considerable sin recurrir a mecanismos de refrigeración extremadamente sofisticados y costosos. El ejemplo más significativo es el del gigante Intel, que tan fuerte apostó por el escalado de la frecuencia con su Pentium 4, y que en 2004 se vio forzado a abandonar sus planes de llegar a los 4 GHz.
- *Memoria*. La creciente disparidad entre la velocidad de la memoria y del procesador hace que la latencia de acceso a memoria principal se haya convertido en un abrumador cuello de botella para el rendimiento de los computadores. De 1986 a 2000, la frecuencia de CPU creció una media anual del 55 %, mientras que las memorias sólo mejoraron a un ritmo del 10 % cada año. Esto ha acrecentado la distancia entre procesador y memoria a varios cientos de ciclos de CPU, y hace que los fallos de caché tengan un gran impacto en el rendimiento global del sistema. Invertir una gran parte de los transistores del chip en jerarquías de memoria caché de tamaño cada vez mayor es una forma de intentar aliviar este problema, pero incluso aquí se ha alcanzado el punto de rendimiento decreciente.

## 1.2. El Giro hacia Arquitecturas Multinúcleo

Por estas razones, estamos asistiendo a un masivo cambio de paradigma hacia arquitecturas multinúcleo que integran varios procesadores en un solo chip y permiten a los programadores explotar el paralelismo a nivel de hilo o proceso (TLP). Así pues, la explotación del TLP 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. Hoy en día, estas arquitecturas multinúcleo -también llamadas *chip-multiprocessors* o CMPs- se han convertido en omnipresentes y podemos encontrarlas en todo tipo de plataformas: desde servidores y estaciones de trabajo hasta ordenadores de escritorio y portátiles. Los fabricantes de microprocesadores han optado por los CMPs como una forma practicable y realista de transformar en rendimiento escalable el creciente número de transistores disponibles. Algunos ejemplos que ya se comercializan con éxito son el Core 2 Duo de

Intel, la serie X2 de AMD o el IBM Power5. La tendencia en un futuro cercano es hacia arquitecturas CMP con progresivamente más núcleos de ejecución, de forma que en los próximos años el número de *cores* se duplicará cada 18 meses [7]. Por ahora, valga comentar que Intel ha lanzado recientemente el Core 2 Quad y AMD presentará en breve su nuevo Opteron (*Barcelona*), ambos CMPs de cuatro núcleos.

En la práctica, el éxito de los sistemas basados en CMPs está limitado por la dificultad de la programación multi-hilo. Escribir programas concurrentes 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 en la aplicación 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.

La necesidad de un soporte hardware que facilite la programación paralela constituye uno de los principales obstáculos que la arquitectura multi-núcleo tiene en su camino, tal y como se recoge en el *HiPEAC roadmap* [7]. Es probable que en algún punto los métodos de sincronización actuales (basados en software) dejen de ser factibles y sean reemplazados por nuevos métodos basados en hardware. De hecho, uno de los retos más importantes es entender qué abstracción hardware/software puede mejorar la productividad del desarrollo de software paralelo, y a partir de ahí encontrar aproximaciones a la implementación más apropiadas para llevarla a cabo.

## 1.3. Memoria Transaccional: Facilitando el Desarrollo de Software Paralelo

De entre los métodos propuestos para dar mayor soporte hardware a la programación multihilo, la Memoria Transaccional (TM, *Transactional Memory*) aparece como el primer candidato, si bien puede que sea sólo una aproximación inicial. El modelo de programación de TM puede reducir significativamente la dificultad de escribir programas concurrentes correctos, al dar respuesta a los problemas de rendimiento, escalabilidad y programabilidad que acarrea la programación con cerrojos del modelo tradicional de memoria compartida. TM ofrece una forma alternativa de sincronización basada en transacciones, más sencilla para el programador y a su vez más eficientemente implementada en hardware.

Bajo este modelo, los programadores expresan *qué* regiones del código deben ser ejecutadas sin interferencias por parte de otros hilos/procesos, 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 (sección crítica) en el código se delimita bien mediante un par de instrucciones tipo `begin_transaction` y `end_transaction`, bien mediante bloques `atomic`, etc. Por su parte, el sistema TM subyacente se encarga de ejecutar dichas transacciones concurrentemente de forma optimista, al tiempo que detecta y resuelve dinámicamente los conflictos que puedan surgir entre ellas. Así, el paradigma de TM trata de combinar una interfaz declarativa bastante intuitiva al programador, con una implementación eficiente en hardware. En general, escribir código paralelo con TM requiere un esfuerzo de programación similar al uso de cerrojos de grano grueso, al tiempo que tiene el potencial de conseguir la escalabilidad y rendimiento de los cerrojos de grano fino.

El resto del trabajo se organiza como sigue. En la sección 2 hacemos un repaso cronológico de los distintos sistemas hardware de memoria transaccional que han aparecido en trabajos previos. La sección 3 introduce los mecanismos básicos de gestión de versiones y detección y resolución de conflictos que todo sistema de memoria transaccional debe proporcionar. En la sección 4 se describen las diferentes políticas aplicables a dichos mecanismos, señalando la influencia de cada estrategia en el impacto que los conflictos causan en el rendimiento de las aplicaciones. La sección 5 discute otros aspectos importantes de la memoria transaccional como son la semántica, el anidamiento y la virtualización de las transacciones. En la sección 6 presentamos la primera caracterización de conflictos entre transacciones que utiliza una colección de *benchmarks* verdaderamente transaccionales, realizada sobre un entorno de evaluación que simula un popular sistema TM hardware. En la sección 7 esbozamos algunas ideas que persiguen un diseño de un sistema de memoria transaccional hardware más flexible. Concluimos con un resumen de las principales aportaciones de este trabajo, recogidas en la sección 8.

## 2. Antecedentes y Trabajo Relacionado

Herlihy y Moss introducen a principios de los noventa la Memoria Transaccional (TM) [11] como una alternativa hardware a la sincronización basada en cerrojos. Los autores la presentan como una generalización de las primitivas LL/SC: su idea principal es utilizar instrucciones especiales para poder realizar accesos atómicos, no a una, sino a múltiples posiciones de memoria independientes, de forma que las secciones críticas ya no necesiten estar protegidas por variables cerrojo.

Casi una década después, tras el florecimiento de los sistemas de memoria transaccional software, la transacciones a nivel hardware empiezan de nuevo a cobrar protagonismo. El trabajo de Rajwar y Goodman, *eliminación de cerrojos transaccional* (TLR) [17] es el primero en aplicar el concepto de transacción a la ejecución de secciones críticas protegidas mediante cerrojos. En realidad, TLR toma como base un trabajo que los mismos autores habían llevado a cabo poco antes, denominado *elisión especulativa de cerrojos* (SLE) [16]. TLR se apoya en SLE como mecanismo base de sincronización libre de cerrojos en ausencia de conflictos, mientras que aporta una nueva solución basada en transacciones y marcas de tiempo para resolver los conflictos, dando lugar a lo que ellos denominan ejecución transaccional de secciones críticas.

Más tarde, Hammond et al. retoman la idea original de Herlihy y Moss para presentar un novedoso modelo de coherencia y consistencia de memoria basado en transacciones (TCC, *Transactional Coherence and Consistency*) [10]. En este artículo, el grupo de Stanford esboza el hardware TCC, que si bien parte de la idea de Herlihy y Moss, se trata de un diseño radicalmente diferente a las arquitecturas paralelas habituales. La novedad de TCC se encuentra en la idea *todo transacciones, todo el tiempo*: las transacciones no sólo se utilizan para envolver las secciones críticas de un programa paralelo, sino que componen o *recubren* todo el código. Con este enfoque, la comunicación entre procesos sólo ocurre en puntos ocasionales definidos por el programador -cuando la transacción completa su ejecución y hace visibles sus resultados al resto del sistema- en lugar de en cada una de las cargas y almacenamientos, como ocurre con el modelo convencional de memoria compartida. Esto simplifica ostensiblemente el hardware, al eliminar la necesidad de implementar protocolos de coherencia de caché tradicionales (estilo MESI). Tras la publicación del artículo de TCC, la memoria transaccional se ha convertido en uno de los temas de investigación más candentes en el área de la arquitectura de computadores.

Posteriormente, aparecen trabajos que giran en torno a la virtualización de las transacciones, un tema que hasta ese momento no había sido abordado explícitamente en el contexto de los sistemas hardware de memoria transaccional. Ananian et al. proponen UTM [1], un diseño que permite soportar transacciones ilimitadas en tamaño y duración, pero que requiere cambios significativos tanto en la arquitectura del procesador como en el subsistema de memoria. Esta propuesta del MIT se acompaña de una variación simplificada, denominada LTM, que gana en sencillez de implementación al ser más restrictiva, pero que todavía arrastra la complejidad de utilizar una zona de memoria para manejar las transacciones desbordadas. Por su parte, Rajwar et al. adoptan en VTM [18] una solución similar, basada en mantener en memoria virtual estructuras software que contienen el estado de las transacciones. En ambas propuestas, la detección de conflictos entre transacciones se realiza a partir de los mensajes de un protocolo de coherencia de caché basado en invalidación.

Moore et al. introducen LogTM [14] como una aproximación a la memoria transaccional mucho más evolutiva desde la perspectiva del hardware que las propuestas aparecidas hasta entonces. Al contrario que TCC, que introduce un modelo basado en consistencia de transacciones, LogTM combina el soporte de transacciones con la implementación del modelo de memoria compartida convencional tradicional. Así, LogTM parte de una arquitectura multiprocesador convencional con cachés privadas que se mantienen coherentes mediante un protocolo de directorio, e incorpora dos mecanismos principales para soportar transacciones. La primera y más significativa modificación es la utilización de un registro o *log* transaccional en memoria virtual cacheable, en el que se mantiene la información necesaria para abortar la transacción actual en cualquier momento. Junto al *log*, el sistema propuesto modifica sutilmente el protocolo de directorio para manejar el desbordamiento de bloques transaccionales. Precisamente, LogTM destaca sobre las propuestas que le preceden por su simplicidad de implementación y elegancia en el manejo de transacciones que desbordan los recursos hardware (tamaño de la caché). En cambio, en esta primera versión de LogTM las transacciones no pueden sobrevivir a cambios de contexto, migración de hilos o paginación, capacidades que sí poseen VTM o UTM.

Desde su aparición, el sistema LogTM ha sido refinado progresivamente por diferentes miembros del grupo Multifacet de la Universidad de Wisconsin-Madison. Moravan et al. [15] extienden LogTM para incluir tanto anidamiento cerrado con abortos parciales como anidamiento abierto, consiguiendo así un mejor manejo de transacciones anidadas. Más recientemente, Yen et al. [21] refinan la propuesta con el fin de separar de las cachés el soporte de TM. El sistema resultante, que han llamado LogTM-SE (*Signature Edition*), toma el mecanismo de firmas *hash* propuesto por Ceze et al. [6]. LogTM también constituye la base de OneTM [2], una propuesta de Blundell et al. que utiliza una caché para reducir la frecuencia con la que las transacciones desbordan los recursos del chip, y trata de simplificar la forma en que se tratan las transacciones desbordadas.

Por último, cabe mencionar que la idea de la desambiguación por lotes mediante firmas basadas en técnicas de dispersión también ha sido utilizado en una reciente propuesta de sistema transaccional híbrido del grupo de Stanford, denominada SigTM [5]. Este sistema acelera la detección de conflictos entre hilos mediante la implementación de las firmas *hash* en hardware para llevar cuenta de los bloques leídos y escritos por las transacciones, mientras que deja el resto de funcionalidad transaccional a la capa software.

# 3. Mecanismos Básicos de un Sistema de Memoria Transaccional Hardware

Para garantizar que una sección crítica se ejecuta correctamente, un sistema TM ha de satisfacer en todo momento las propiedades de *atomicidad* y *aislamiento*: La primera de ellas requiere que una transacción se ejecute completamente o no se ejecute en absoluto, mientras que la segunda exige que el estado atravesado por una transacción parcialmente completada permanezca oculto al resto del código.

- Para cumplir la atomicidad, el sistema debe ser capaz de almacenar simultáneamente tanto el estado de la máquina inmediatamente anterior a la transacción, como el nuevo estado al que se llega tras los cambios realizados por la misma. De esa forma, se asegura que la máquina quedará en un estado consistente independientemente del éxito o fracaso de la transacción, es decir, tanto si ésta se ejecuta completamente como si no llega a hacerlo. El manejo simultáneo de ambos estados se consigue mediante un mecanismo denominado habitualmente *gestión de versiones* de datos.

- Para conseguir el aislamiento, el estado intermedio por el que atraviesa una transacción debe mantenerse oculto al resto del código -asumiendo una semántica de atomicidad fuerte-. Si un núcleo de procesamiento accede a un bloque de datos que ha sido modificado previamente por una transacción aún no completada, el sistema debe detectar y resolver dicha violación del aislamiento. Esta tarea común a todos los sistemas de memoria transaccional se llama *detección y resolución de conflictos*.

Garantizar estas dos propiedades básicas requiere de mecanismos que pueden implementarse exclusivamente en hardware, en software o mediante aproximaciones híbridas. Las implementaciones y políticas aplicadas a los mecanismos de gestión de versiones, detección y resolución de conflictos constituyen el espacio de diseño de los sistemas de memoria transaccional hardware (HTM). En los siguientes apartados describimos las principales estrategias junto con las ventajas e inconvenientes de cada método.

En este trabajo nos centramos en los HTMs, que se antojan hoy día una posible alternativa hacia la que puede que evolucionen las arquitecturas de alto rendimiento del futuro. En comparación con los HTM, los sistemas TM software experimentan una sobrecarga de partida que varía entre un 40 y un 200 % más lentos que las implementaciones hardware. Si el rendimiento es su principal ventaja, entre los inconvenientes de los HTMs están el coste de implementación o la complejidad de conseguir la virtualización. Así, parece el éxito de los sistemas transaccionales pasa por encontrar el soporte hardware mínimo para proporcionar un alto rendimiento en el caso común, de forma que sea el software el que disponga de mecanismos para llevar a cabo tareas complejas y poco comunes.

# 4. Influencia de las Políticas de Diseño en el Rendimiento

Los sistemas HTM cuentan con hardware específico destinado a soportar de forma eficiente las funciones de gestión de versiones de datos y de detección y resolución de conflictos, mediante las cuales se consiguen las propiedades básicas de atomicidad y aislamiento requeridas por la semántica transaccional. Como ocurre con cualquier funcionalidad implementada a nivel hardware, una de las principales restricciones es la complejidad de la lógica necesaria o la modificación de elementos críticos de la arquitectura (como por ejemplo, las cachés). Con el fin de minimizar los cambios introducidos en la arquitectura, las propuestas HTM actuales implementan en silicio una única política prefijada para cada mecanismo. Esta inflexibilidad hace que el rendimiento del sistema quede totalmente condicionado por el comportamiento de la aplicación -marcado por las características de sus transacciones- que a su vez depende en buena medida de la pericia del programador o de la calidad del compilador. En esta sección, mostramos cómo cada una de las políticas afecta de una forma u otra al rendimiento de las aplicaciones, y cómo las características del código ejecutado determinan las prestaciones obtenidas al utilizar unas u otras políticas.

## 4.1. 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 se completa, mientras que el viejo valor se mantiene en caso de que la transacción aborte. Sólo una las dos versiones puede guardarse en la dirección de memoria compartida del dato en cuestión, *in-situ*; la otra versión deberá ser almacenada en algún otro lugar, digamos, *a un lado*, para después sobreescribir el valor anterior en su posición de memoria compartida. 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. La Figura 1 muestra de forma gráfica la diferencia entre una y otra política.

**Gestión de versiones perezosa.** Los HTMs con política de versiones perezosa mantienen temporalmente el resultado de los almacenamientos en un nivel de caché privado al procesador, hasta que la transacción finaliza. Cuando se llega al *commit*, los nuevos valores sobreescriben a los antiguos en sus posiciones de memoria compartida correspondientes, en niveles inferiores de la jerarquía. Durante la ejecución de la transacción, la memoria compartida siempre contiene las versiones antiguas de los datos. Las nuevas versiones pueden ser rápidamente descartadas en caso de que la transacción fracase debido a un conflicto, de ahí que con esta política sea fácil quitarse de en medio una transacción abortada. Entre los sistemas que utilizan esta política encontramos la idea original de TM a cargo de Herlihy y Moss [11], el sistema TCC de Stanford [9], LTM [1], VTM [18] y Bulk [6].

La propuesta original de *Memoria Transaccional* utiliza una caché transaccional de primer nivel (además de la caché de datos L1 convencional) en la que se colocan ambas versiones del dato. Esta caché cuenta con una etiqueta adicional en cada entrada que sirve para distinguir una versión del dato de la otra. Los posibles valores de esta etiqueta transaccional son EMPTY (vacía), NORMAL (datos confirmados), XCOMMIT (versión antigua), XABORT (versión nueva). Por cada escritura transaccional se reservan sendas entradas para las versiones antigua y nueva, y se marcan con las etiquetas XCOMMIT y XABORT, respectivamente. Las escrituras transaccionales modifican exclusivamente las entradas XABORT (valor nuevo). El aislamiento se preserva gracias a que sólo las líneas etiquetadas como NORMAL pueden ser fisgoneadas por otros nodos, o bien escritas a niveles inferiores de la jerarquía, de forma que el estado intermedio de la transacción (entradas XABORT) no puede ser observado por otros nodos.

En términos de rendimiento, el uso de esta caché especial basada en etiquetas consigue que tanto *commits* como *aborts* sean operaciones locales al nodo que sólo necesitan un cambio en el valor de la etiqueta y el descarte de la entrada apropiada, dependiendo del éxito o fracaso de la transacción. Esto favorece la concurrencia de transacciones y permite que varios nodos puedan hacer *commit* simultáneamente. Supuesto que la caché transaccional es una estructura pequeña, las líneas XABORT pueden cambiar su etiqueta a NORMAL en un sólo ciclo, por lo que el nuevo estado de la máquina se

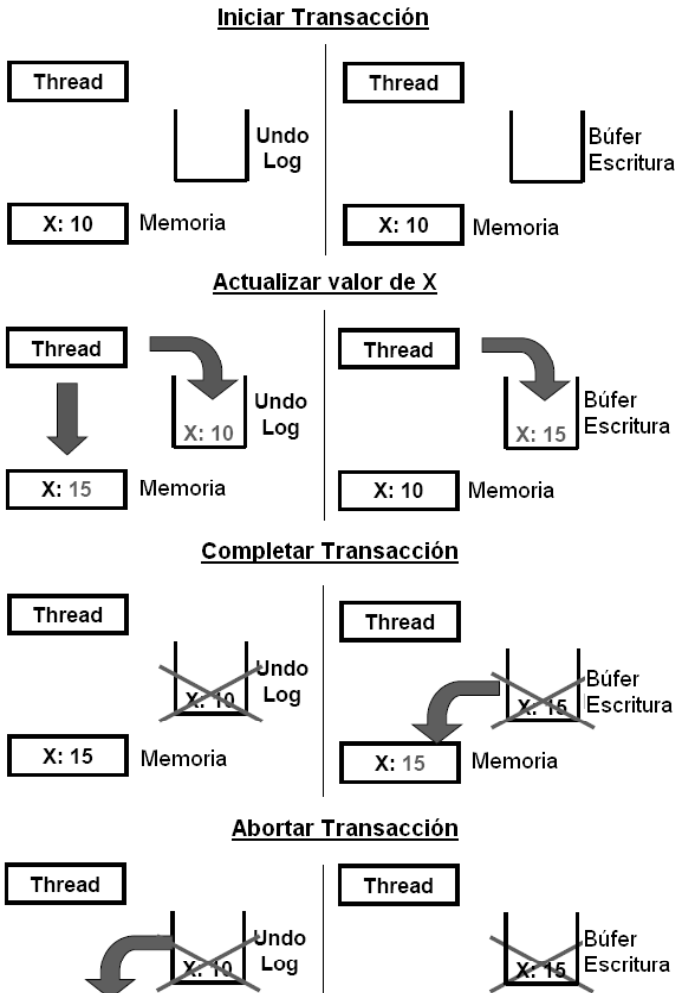


**Figura1.** Ilustración de las políticas de gestión de versiones de datos ansiosa (izquierda) y perezosa (derecha).

hace visible al resto de nodos de una vez. Por tanto, la latencia de *commit* es reducida al no implicar necesariamente la post-escritura de los bloques sucios en la caché transaccional. Las nuevas versiones de los datos son propagadas conforme otros nodos van solicitando los bloques, o bien cuando éstos son reemplazados de la caché transaccional y escritos a niveles inferiores de la jerarquía de memoria. Las entradas XCOMMIT (valor antiguo) se utilizan sólo por motivos de rendimiento, ya que permiten evitar continuas post-escrituras de bloques sucios cuando un procesador ejecuta repetidamente transacciones que acceden las mismas posiciones de memoria, en ausencia de espacio libre en la caché transaccional [11].

A pesar de que esta técnica obtiene niveles de rendimiento muy aceptables, la propuesta de Herlihy y Moss está inherentemente limitada a transacciones de tamaño pequeño. De hecho, el objetivo original que los autores persiguen con TM no es facilitar la programación paralela sino proporcionar sincronización libre de cerrojos, como forma de evitar problemas causados por las técnicas de cerrojos -inversión de prioridad, efecto convoy, interbloqueo, etc.-. Así, esta implementación pionera de la memoria transaccional se basa en la asunción de que las transacciones tienen una duración corta y acceden a un conjunto de escritura reducido, pues tan sólo reemplazan a las variables cerrojo en la forma en que protegen las secciones críticas, pero no cambian la naturaleza de las mismas. La gestión de versiones basada en caché transaccional de TM limita el tamaño de las transacciones a la capacidad de dicha caché, puesto que

una vez agotado el espacio para guardar las nuevas versiones de datos no es posible confinar el estado intermedio al completo y por tanto no se puede garantizar el aislamiento.

Propuestas más recientes de sistemas de memoria transaccional hardware tratan de superar ésta y otras limitaciones relacionadas con la virtualización de transacciones. El sistema TCC de Stanford [9], cuyo esquema hardware se muestra en la Figura 2, fue el primero en recuperar el concepto de memoria transaccional a nivel hardware. TCC introduce un novedoso modelo de consistencia y coherencia, pero sigue basando su gestión de versiones en la política perezosa usada por H&M, aunque soporta transacciones de tamaño mayor que la caché. Si el tamaño del conjunto de datos modificado por una transacción no excede los recursos hardware disponibles (capacidad de la caché), mantener el estado intermedio oculto al resto del sistema hasta el final de la transacción se consigue de forma inmediata, pues la caché es local y privada al núcleo de procesamiento. Si la transacción logra completarse con éxito, los resultados se difunden al resto del sistema mediante la creación y envío de un paquete de datos con las direcciones y nuevos valores de los datos modificados. Este método basado en *broadcasting* hace que los requisitos de ancho de banda de TCC sean elevados, factibles para las redes de interconexión on-chip de los CMPs actuales, pero presumiblemente inaceptables para los CMPs futuros (CMPs densos) con un gran número de cores y en los que pueden existir grupos de *cores* asignados a diferentes aplicaciones. Por otro lado, la arquitectura TCC se cimenta en un bus que todos los nodos pueden fisgonear y sobre el que se ordenan los accesos, mientras que la interconexión on-chip en CMPs suele llevarse a cabo con una red directa, no con un bus.

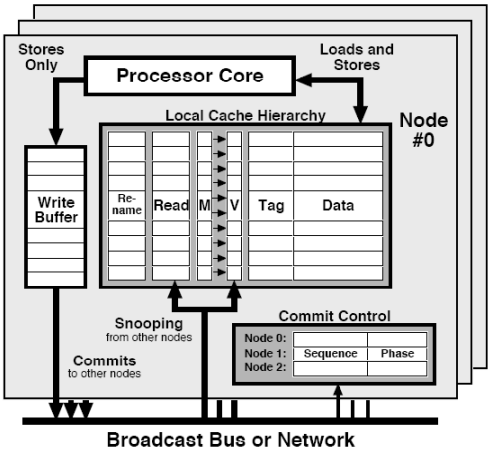


**Figura2.** Hardware del sistema TCC.[9]

Puesto que el modelo de coherencia TCC impone un cierto orden global al serializar los *commits*, en cada instante sólo puede haber un nodo confirmando sus resultados y propagándolos al resto del sistema. En términos de rendimiento, la fase de arbitraje (conseguir permiso de *commit*) junto con la creación y envío del paquete de datos modificados hacen que la latencia de compleción de las transacciones sea considerable. Los autores proponen una mejora basada en *buffering doble* para aliviar el impacto que la latencia de *commit* tiene en el rendimiento, con el coste de duplicar el buffer de escritura y los bits RW de cada línea de caché. De esta forma es posible permitir que la siguiente transacción pueda comenzar su ejecución mientras la anterior está en espera de permiso de *commit* o confirmando sus resultados.

Los sistemas de memoria transaccional con gestión de versiones perezosa deben afrontar el problema de qué hacer cuando los datos escritos por la transacción desbordan la capacidad o asociatividad del buffer especulativo o caché, con el fin de soportar transacciones de mayor tamaño que la capacidad de la caché y superar parte de las limitaciones de la propuesta TM original. TCC adopta una aproximación sencilla, que se basa en detener la transacción desbordada hasta obtener permiso de *commit*, y partir de ahí ejecutarla 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 soportan transacciones de tamaño ilimitado [1,18], utilizando para ello estructuras de datos en memoria virtual que mantienen el estado de la transacción. No obstante, estos diseños acarrean modificaciones significativas tanto en el procesador como en el subsistema de memoria, introduciendo bastante complejidad.

**Gestión de versiones ansiosa.** Los sistemas de memoria transaccional hardware con gestión de versiones ansiosa escriben directamente el nuevo valor del dato en su posición de memoria compartida. Las nuevas versiones no están confinadas a la caché sino que pueden ser reemplazadas y escritas en niveles inferiores (compartidos) de la jerarquía de memoria. En la sección posterior veremos cómo el protocolo es capaz de detectar conflictos con estos bloques, a pesar de que la información transaccional (bits R y W) se pierde cuando el bloque es expulsado de la caché. Para ser capaz de recuperarse ante transacciones conflictivas, esta política ansiosa registra los viejos valores junto con su dirección de memoria en un *undo-log*. Éste sólo se utiliza en caso de que la transacción deba abortarse, para deshacer los cambios y restaurar la memoria a su estado anterior al comienzo de dicha transacción. Ejemplos de HTM con política de versiones ansiosa son UTM [1], LogTM [14], y otras propuestas basadas en este último [21,2].

Al inicio de su ejecución, cada hilo asigna una parte de su espacio de direcciones para ser utilizado como *log*, de forma que cada núcleo de procesamiento cuenta con sendos punteros al inicio y final del *log*. En la Figura 3 podemos ver de forma esquemática las necesidades hardware del sistema LogTM (elementos no sombreados). Puesto que el *log* es privado y se encuentra en el espacio de direcciones virtuales, su contenido es cacheable y por ello las escrituras al *log* suelen ser aciertos de caché. Adicionalmente, se puede utilizar un pequeño buffer de *log* para reducir la contención en el acceso a la caché de primer nivel, puesto que cada almacenamiento genera dos escrituras, una al bloque que contiene el dato y otra al *log*. Puesto que el *log* no se necesita hasta el momento en el que la transacción ha de abortar, las escrituras al mismo se pueden retrasar de forma que o bien se descartan si la transacción se completa, o bien se efectúan cuando se llena el espacio del buffer de *log*.

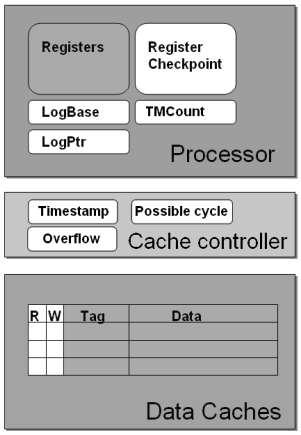


**Figura3.** Hardware del sistema LogTM.

Una gestión de versiones ansiosa hace que sea más rápido completar una transacción exitosa que abortar una transacción conflictiva, pues los nuevos valores ya están en su posición de memoria correspondiente y sólo es necesario descartar el *log*. 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. No obstante, algunos autores opinan que un sistema TM ideal debería utilizar gestión de versiones ansiosa [14], puesto que a su parecer es de esperar que las transacciones exitosas sean mucho más frecuentes que las abortadas. Esta hipótesis se ve confirmada por los resultados de su propio estudio, cuya evaluación utiliza aplicaciones paralelas de la suite SPLASH-2 [20]. Estos *benchmarks* han sido cuidadosamente optimizados a lo largo de los años, con el fin de evitar la sobrecarga por sincronización, y explotan efectivamente el paralelismo de grano fino presente en el código. La evaluación de Moore et. [14] utiliza versiones transaccionales de SPLASH, en la que las llamadas a `lock` y `unlock` han sido reemplazadas

por por sus correspondientes `begin_transaction` y `end_transaction`. El resultado son programas que pasan sólo una pequeña parte de su tiempo en transacciones breves y por tanto experimentan pocos conflictos y prácticamente ningún aborto.

En nuestra opinión, este comportamiento no es necesariamente representativo de una clase importante de futuras aplicaciones transaccionales, puesto que va en contra de la facilidad de programación perseguida por el modelo de TM. Si por algo el modelo de TM resulta atractivo a los programadores, es porque fomenta el uso de transacciones de grano grueso. Sin embargo, la única forma de conseguir minimizar el número de conflictos y con ello el número de transacciones abortadas es extraer el paralelismo a un grano más fino, de forma que el conjunto de datos accedido se reduzca y con ello las posibilidades de que dos transacciones concurrentes entren en conflicto. Así, excepto para aquellos programas paralelos con inherentemente poca comunicación entre hilos, el hecho de asumir que los abortos serán un fenómeno más bien raro implica una mayoría de transacciones de grano fino en el código. Obviamente, esto resulta en un mayor esfuerzo de desarrollo y tira por tierra las pretensiones de simplificar la programación paralela del modelo de TM. Igualmente, esta asunción no tiene en cuenta que los compiladores paralelos encuentran extremadamente complicado extraer el paralelismo de grano fino, y que es probable que la generación automática de código paralelo se base en envolver cada tarea paralela de grano grueso dentro de una transacción. En resumen, las decisiones que afectan al diseño de los sistemas de memoria transaccional hardware deberían tomarse basándose en unos *benchmarks* transaccionales realistas, que reflejen estas y otras prácticas de programación comunes. Estas aplicaciones, al contrario que la *suite* SPLASH, emplean la mayor parte de su tiempo de ejecución en transacciones de gran tamaño, en las que la frecuencia de los conflictos dependerá de factores como la experiencia y pericia del programador y su conocimiento del problema, los patrones de comunicación entre hilos, el paralelismo disponible en la aplicación, etc.

## 4.2. 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. Las políticas de detección de conflictos varían en función de *cuándo* se examina la información de los conjuntos R y W. Unos esquemas tratan de detectar los accesos conflictivos tan pronto como se producen -política ansiosa- mientras que otros optan por aplazar la detección hasta que una de las transacciones concurrentes llega a su final e intenta confirmar sus resultados para hacerlos visibles al resto de la máquina -política perezosa-. Estas aproximaciones a la detección de conflictos también se denominan, respectivamente, *pesimista* y *optimista*, ya que la primera prevé frecuentes conflictos e intenta detectarlos cuanto antes, mientras que la segunda espera que los conflictos no aparezcan a menudo, y no le importa detectarlos más adelante.

**Granularidad de la detección de conflictos.** Independientemente de la política seguida, cualquier HTM debe ser capaz de seguir la pista de los datos leídos y escritos por cada transacción para dar soporte a la detección de conflictos. Esto 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. Aunque el compilador puede prevenir en algunos casos la aparición de falsa compartición de datos entre transacciones, aquellos sistemas que detectan conflictos con una granularidad de bloque crean otra responsabilidad más para el programador. Evitar estas violaciones espúreas del aislamiento es una carga adicional que va claramente en contra de la simplificación de la programación paralela perseguida por TM. No obstante, numerosos sistemas HTM optan por aumentar la caché sólo dos bits R y W, para indicar si el bloque ha sido leído o escrito por la transacción actual. Otras propuestas utilizan un novedoso mecanismo basado en firmas para codificar mediante *hashing* las direcciones accedidas por un hilo, agrupando en sendos registros los conjuntos R y W. Con esta aproximación, no es necesario modificar una estructura tan crítica como es la caché, sino que basta con añadir dos registros en el procesador que contengan respectivamente las firmas de lectura y escritura. Sin embargo, la codificación *hash* hace que la firma resultante contenga

un superconjunto de las direcciones de memoria accedidas, y provoca la aparición de falsos conflictos debidos al denominado *aliasing* (dos direcciones de memoria diferentes con el mismo *hash*).

Los HTMs que aumentan la caché con bits R/W deben afrontar de nuevo el problema del desbordamiento de bloques transaccionales -pertenecientes a los conjuntos de lectura o escritura-, ya sea por falta de capacidad o de 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. Unos lo hacen de manera simplista [10]: cuando algún bloque con el bit R o W activo ha de ser reemplazado, el sistema transita a modo *overflow* tal como se comentó anteriormente, cuando los recursos para guardar el estado especulativo se agotan -ya sea por falta de capacidad o asociatividad de la caché-. Otras propuestas posteriores refinan un poco más la solución: LogTM extiende un protocolo de coherencia 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 -necesitan acceder a estructuras en memoria principal-.

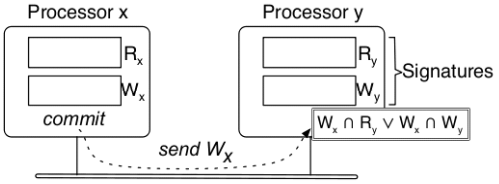


**Figura4.** Detección de conflictos en Bulk utilizando firmas *hash* [6].

En propuestas como Bulk [6] y LogTM-SE [21] no hay lugar a desbordamiento de la información necesaria para la detección de conflictos, ya que ésta está contenida en las firmas R y W. En la Figura 4 se ilustra de forma simplificada cómo se lleva a cabo la detección de conflictos basada en firmas en Bulk (política optimista). Estas firmas codifican un super-conjunto de las direcciones de memoria del conjunto de lectura y de escritura de la transacción, utilizando para ello registros del orden de 1Kbit de tamaño. Por contra, la codificación *hash* de la firma provoca en ocasiones la aparición de falsos positivos, es decir, puede que haya accesos que se detecten como conflictivos pero que en realidad se refieran a direcciones de memorias no pertenecientes a los conjuntos R&W de la transacción. En el caso de Bulk, no importa que existan bloques en el área de *overflow* en memoria (bloques escritos por la transacción), ya que no es necesario acceder a dicha zona para comprobar si existen conflictos.

**Detección de conflictos ansiosa o pesimista.** La mayoría de sistemas HTM propuestos se basan en el protocolo de coherencia de caché para detectar conflictos entre transacciones concurrentes [1,18,14,21,2], incluyendo la propuesta original de Memoria Transaccional ideada por Herlihy and Moss [11]. Estos sistemas monitorizan el tráfico de coherencia de caché que afecta a bloques transaccionales, para determinar si otro núcleo de procesamiento está realizando un acceso conflictivo. Esta política se ilustra en la Figura 5 b and c. Cuando un núcleo A pretende modificar un bloque que no tiene en exclusividad, debe enviar una petición exclusiva sobre el mismo, que eventualmente llega al propietario, si lo hay. Éste observa que la petición corresponde a un bloque leído o modificado en el transcurso de la transacción actual todavía no completada, con lo que detecta que se acaba de producir un acceso conflictivo y por tanto pasa a tratarlo. La política de resolución también puede variar, tal y como detallamos en el siguiente apartado.

Puesto que ninguna transacción puede observar el estado no confirmado, la política de detección ansiosa puede mejorar el rendimiento de un sistema transaccional al resolver algunos conflictos utilizando pausas en lugar de drásticos abortos. Este tipo de detección de conflictos puede reducir la cantidad de trabajo malgastado que ha de ser descartado si la transacción debe ser finalmente abortada para evitar interbloqueos. Sin embargo, una política pesimista puede experimentar una serie de patrones de ejecución

que afectan negativamente al rendimiento, como el fuego amistoso, duelo de actualizaciones, detención en vano e inanición del escritor. Estas patologías son descritas con mayor detalle por Bobba et al. en un reciente artículo [4].

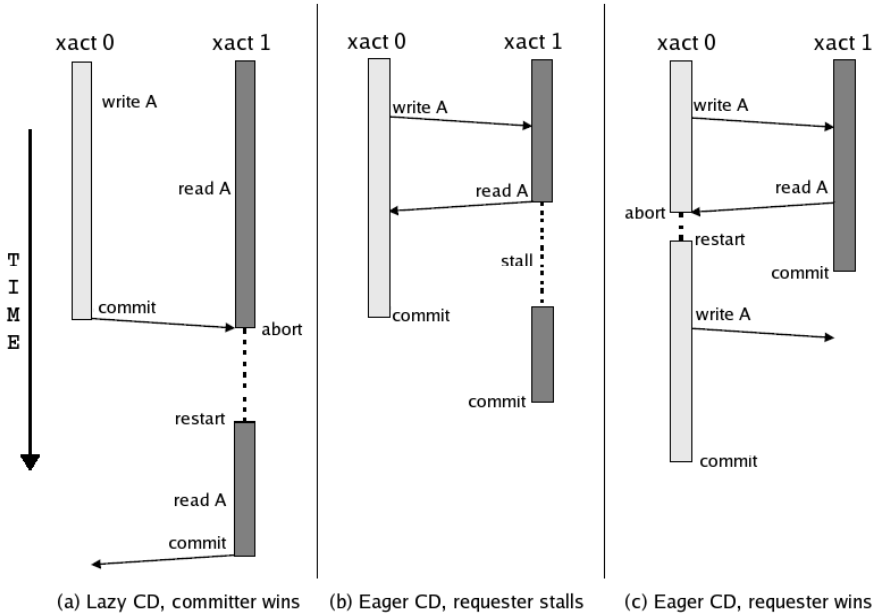


**Figura5.** Ilustración de las políticas de detección y resolución de conflictos.

**Detección de conflictos perezosa u optimista** Otros HTMs como TCC [10] y Bulk [6] se decantan por una política de detección de conflictos perezosa. En estas propuestas, la comprobación de accesos conflictivos no se lleva a cabo hasta que la transacción ha llegado a su fin. Una vez obtenido permiso para confirmar sus resultados y hacerlos visibles al resto del sistema, la transacción difunde su conjunto de escritura para que el resto de transacciones en curso pueda contrastar sus conjuntos R&W con los del *committer*. Los demás núcleos de procesamiento proceden a abortar su transacción actual en caso de que hayan leído datos modificados por la transacción completada, ante la existencia de una dependencia de datos real, como se muestra en la Figura 5 a. En el caso de TCC, el paquete de *commit* también contiene los datos modificados además de sus direcciones, de forma que las cachés con copia del bloque pueden actualizar los valores con la versión más reciente. En Bulk, se envía únicamente la firma de escritura para que el resto del sistema pueda detectar conflictos. Gracias a esta política, es posible implementar el chequeo de conflictos por lotes.

Una ventaja que conlleva esta política de detección es la garantía de que un programa siempre consigue progresar hacia adelante, pues siempre hay una transacción que consigue completarse y confirmar sus resultados. Una política perezosa también puede mitigar el impacto de algunos conflictos encadenados, como por ejemplo cuando una transacción T1 entra en conflicto con T2, y esta a su vez con T3, siendo T1 y T3 independientes. En un sistema con política perezosa, si T1 llega al *commit* primero, abortará a T2 pero dejará que continúe su ejecución. En cambio, bajo política ansiosa, es posible que T2 entre en conflicto con T3 y la aborte, antes de aparezca el conflicto T1-T2 y ésta última sea abortada por T1.

En general, esta política optimista conduce a cantidades de trabajo malgastado potencialmente mayores que las de una detección pesimista. Esto se debe a que puede pasar mucho tiempo desde que sucede un acceso conflictivo hasta que éste se detecta, lo cual lleva consigo una gran cantidad de operaciones realizadas y de energía consumida en vano. En cambio, dicho inconveniente puede tener un efecto colateral favorable, ya que es posible que la transacción reiniciada encuentre en caché muchos de sus datos -particularmente los privados al hilo-, puesto que la anterior ejecución abortada actúa como un mecanismo de prebúsqueda muy preciso. Otras patologías indeseables que pueden afectar a un sistema con política de detección optimista son la compleción serializada, el convoy de reinicios o la inanición del más viejo [4].

**Tabla 1.** Tabla. Resumen de sistemas de memoria transaccional hardware propuestos y sus políticas.

| | | **Gestión de Versiones** | |
|---|---|---|---|
| | | **Ansiosa** | **Perezosa** |
| **Detección de Conflictos** | **Ansiosa** | UTM (MIT)<br>LogTM (U.Wisc)<br>LogTM-SE (U.Wisc)<br>OneTM (U.Penn) | TM (H&M)<br>LTM (MIT)<br>VTM (Intel/Brown) |
| | **Perezosa** | - | TCC (Stanford)<br>Bulk (U.Illinois) |

## 4.3. 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. Los sistemas HTM con política de detección optimista, como TCC y Bulk, siempre resuelven el conflicto dejando que la transacción que se completa lo gane, abortando al resto. En cambio, para las propuestas HTM que utilizan políticas de detección ansiosas, las estrategias de resolución pueden variar. Se puede optar por detener o abortar al peticionario o bien abortar a los demás. Por ejemplo, LTM y VTM permiten que el hilo peticionario gane el conflicto, abortando a los demás. UTM y LogTM, por contra, bloquean la petición conflictiva para detener al peticionario P hasta que el nodo N que detecta el conflicto finaliza su transacción. Puesto que N puede solicitar posteriormente datos conflictivos a P, es posible que se produzcan interbloqueos. La solución que aporta LogTM es evitar interbloqueos de forma conservadora mediante el uso de marcas de tiempo.

Cuando es necesario abortar una o más transacciones para recuperarse de un conflicto, todos los sistemas HTM propuestos hasta ahora siguen una misma filosofía: se descarta todo el trabajo realizado por la transacción -utilizando la información del *log*, en su caso- y se restaura el estado de los registros de la máquina gracias al *checkpoint* que se guarda al comienzo de la transacción. Este enfoque está claramente orientado a mantener el hardware simple, y tiene un impacto mínimo en el rendimiento cuando las aplicaciones están compuestas por transacciones cortas, puesto que los abortos no ocurren muy a menudo y cuando lo hacen, reiniciar la transacción al completo sólo requiere la ejecución de un pequeño número de instrucciones. Por el contrario, cuando las transacciones tienen un tamaño grande, abortar todo el trabajo realizado debido a un simple conflicto parece una estrategia demasiado poco ambiciosa, especialmente si dicho conflicto tiene lugar más bien cerca del final de la transacción.

Un mecanismo de recuperación ideal no descartaría todo el trabajo realizado sino que trataría de mantener cuanto mayor número de operaciones y resultados parciales como fuese posible, y sólo reharía aquellos cómputos que necesitan usar nuevas versiones de datos obtenidas de la transacción que ganó el conflicto. Dado que el modelo de programación de TM fomenta el paradigma de transacciones de grano grueso, es de esperar que las cargas de trabajo transaccionales del futuro sean abundantes en transacciones largas y que lean/modifiquen un gran conjunto de datos. En un escenario transaccional de este tipo, los esquemas de recuperación actuales pueden degradar el rendimiento de forma considerable, no sólo por el trabajo útil y el consumo de energía que se desperdicia en cada transacción abortada, sino también debido a que la tasa de aborto suele ser mucho mayor en programas que cuentan con pocas pero grandes transacciones. En términos de consumo de energía, evitar repetir grandes cantidades de trabajo conlleva un ahorro de energía muy valioso, algo especialmente importante en los CMPs presentes y crítico en los chips del futuro. Ante tasas de aborto elevadas, la única opción que el programador tiene para mejorar el tiempo de ejecución de su aplicación es buscar transacciones de grano más fino, lo cual resulta en un esfuerzo similar a la utilización de cerrojos de grano fino, precisamente el inconveniente de la programación paralela que TM trata de evitar. Por esta razón, nuestra opinión es que sería muy beneficioso para los sistemas HTM contar con un mecanismo más avanzado, que permita recuperarse de forma más eficiente de una transacción abortada.

# 5. Otros Aspectos Importantes en el Diseño de un HTM

## 5.1. 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.

## 5.2. 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 [5,19] 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.

### 5.3. 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.

## 6. Evaluación y Caracterización de Conflictos en un Sistema HTM

En esta sección, presentamos una caracterización de los conflictos bajo un popular sistema de memoria transaccional hardware como es LogTM [14]. Esta caracterización muestra que la frecuencia de aborto puede ser muy alta en cierto tipo de cargas de trabajo transaccionales, y por tanto invalida la asunción de que las transacciones completadas son mucho más frecuentes que los abortos en los sistema de memoria transaccional hardware. Esta es la primera caracterización de abortos en un sistema HTM que usa *benchmarks* representativos, desarrollados desde cero bajo el paradigma de transacciones de grano grueso fomentado por TM.

### 6.1. Resumen de LogTM

LogTM es un sistema de memoria transaccional hardware propuesto por el grupo *Multifacet* de la Universidad de Wisconsin-Madison. LogTM implementa detección de conflictos pesimista y gestión de versiones ansiosa. Cada hilo utiliza un *log* privado en su espacio de direcciones de memoria virtual, que contiene las direcciones y los valores antiguos de los valores modificados por la transacción actual. Cada bloque de caché es aumentado con dos bits R y W para la detección de conflictos con granularidad a nivel de bloque. LogTM extiende un protocolo de directorio tipo MOESI, y es capaz de detectar rápidamente conflictos sobre bloques transaccionales expulsados de la caché gracias al uso de estados pegajosos (*sticky states*). El anidamiento de transacciones esta soportado mediante el alisamiento (*flattening*) de las transacciones interiores, que pasan a formar parte de la transacción de nivel más exterior.

### 6.2. Metodología y Entorno de Simulación

Para esta evaluación, usamos la simulación de un sistema completo dirigida por la ejecución (*full-system*, *execution driven*) que ofrece el conjunto de herramientas de simulación Wisconsin GEMS [13]. GEMS funciona en conjunción con Virtutech Simics [12], un simulador que proporciona corrección funcional para el ISA SPARC y es capaz de arrancar una versión no modificada de Solaris 10. En este entorno de evaluación, utilizamos la implementación del protocolo LogTM incluida en GEMS y el modelo de simulación temporal detallada del subsistema de memoria (*ruby*), junto con el modelo de procesador en orden ofrecido por Simics. El simulador *ruby* ha sido modificado para obtener estadísticas más específicas sobre los abortos de transacciones. Todas las estadísticas aquí presentadas son valores promedio, obtenidas a partir de múltiples simulaciones de cada configuración. La principal métrica de esta caracterización es la tasa de aborto, definida como:

$$tasa\_aborto = \frac{transacciones\_abortadas}{transacciones\_abortadas + transacciones\_completadas}$$

En este entorno de evaluación, hemos simulado un sistema CMP cuyos principales parámetros se detallan en la Tabla 2. En particular, usamos configuraciones de 8 y 16 núcleos de ejecución (*cores*) con cachés privadas, conectadas entre sí a través de un torus 2D y mantenidas coherentes mediante un protocolo MOESI basado en directorio. Cada *core* tiene 512KB de caché L2 privada en el sistema de 8 núcleos, y 256KB en el sistema de 16 núcleos. El controlador de directorio esta integrado en el chip junto con un primer nivel ideal de caché de directorio, de forma que siempre proporciona información de compartición precisa sin incurrir en una latencia adicional. Esta asunción de caché de directorio ideal no afecta a las conclusiones de nuestra caracterización, como pudimos comprobar simulando el mismo conjunto de configuraciones que aquí mostramos, con un directorio completamente *off-chip* cuya latencia es aquélla de la memoria principal (300 ciclos). Obtuvimos comportamientos similares en las aplicaciones, en lo que se refiere a la tasa de aborto así como al peso del código transaccional en el tiempo de ejecución global, lo cual pone de manifiesto la generalidad de nuestra caracterización.

**Tabla 2.** Parámetros del sistema.

| **CMP basado en directorio MOESI** | |
|---|---|
| Parámetros del Núcleo de Procesamiento | |
| Núcleos | 8/16, emisión simple, en orden, no mem IPC=1 |
| Parámetros de Memoria y Directorio | |
| L1 I&D caches | Privadas, 32KB, separadas, 2 vías, 1 ciclo latencia |
| L2 cache | Privada, 512/256KB, unificada, 4 vías, 12 ciclos latencia |
| Memoria | 4GB, 300 ciclos latencia |
| Directorio | Vector de bits, 30 ciclos latencia |
| Parámetros de la Red | |
| Topología | Torus 2D |
| Latencia enlace | 1 ciclo |
| Ancho banda enlace | 40 bytes/ciclo |

## 6.3. Benchmarks Transaccionales

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 componen la versión 0.9.2 de 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. Para producir los resultados de esta caracterización, hemos usado el mismo conjunto de entradas que Cao Minh. et al. en [5], como se muestra en la Tabla 3.

La suite de aplicaciones transaccionales STAMP se distribuye junto con una versión para Linux/x86 de TL2, un simulador de un sistema de memoria transaccional software [8]. En el momento del comienzo de nuestro trabajo, en la página web de STAMP (`stamp.stanford.edu`) se podía obtener una versión transaccional de las aplicaciones que funcionaba sin problemas en un entorno Linux/x86 junto al simulador TL2. Previamente a la realización del trabajo que ha conducido a esta tesis, la suite no había sido probada en un entorno Solaris/SPARC como el ofrecido por Simics, ni tampoco adaptada para funcionar sobre un sistema de memoria transaccional hardware como LogTM, simulado con la herramienta GEMS. Por ello, una parte importante de nuestro trabajo ha consistido en portar estos programas a Solaris/SPARC, y modificar lo necesario para adecuarlos a la interfaz del protocolo LogTM disponible en GEMS.

El proceso de trasladar la suite STAMP al simulador Simics no ha sido tan inmediato como pudiese parecer, sino que ha requerido el descubrimiento y corrección de algunos fallos en el código de los programas, unos *bugs* que no se habían manifestado hasta entonces en el entorno Linux/x86. Por otra parte, el código de las librerías ha tenido que ser adaptado para poder funcionar en una máquina SPARC de 64 bits -en particular la parte del gestor de memoria- ya que las aplicaciones experimentaban problemas de accesos a memoria no alineados (*Bus error*) al ejecutarse en la máquina UltraSPARC III+ simulada. Por último, tuvimos que modificar apropiadamente las macros de la interfaz transaccional, así como codificar algunas funciones de apoyo, para ser capaces de ejecutar las aplicaciones con el protocolo LogTM de GEMS, y así poder extraer estadísticas sobre la ejecución transaccional. En resumen, nuestro trabajo ha contribuido notablemente a la mejora de STAMP, y de nuestro esfuerzo para llevar STAMP a Simics/GEMS se ha beneficiado directamente el grupo de investigación *High-Performance Synchronization* de la Universidad de Rochester (Nueva York), a través de su miembro Arrvindh Shriraman.

**Tabla 3.** Benchmarks STAMP y tamaños de entrada.

| Benchmark | Entrada |
|---|---|
| DELAUNAY | Malla gen3.2, Ángulo min. 30 |
| GENOME | 16K segmentos, longitud gen 256, longitud segmento 16 |
| KMEANS-LOW | 40/40 grupos, umbral 0.05, 1000 puntos de 12 dimensiones |
| KMEANS-HIGH | 20/20 grupos, umbral 0.05, 1000 puntos de 12 dimensiones |
| VACATION-LOW | 64K entradas, 4K tareas, 4 consultas, 90 relaciones, 80 usuarios |
| VACATION-HIGH | 64K entradas, 8K tareas, 8 consultas, 10 relaciones, 80 usuarios |

## 6.4. Resultados

En esta sección, presentamos de manera cuantitativa cómo ciertas aplicaciones desarrolladas bajo el modelo de programación de TM son proclives a experimentar un número importante de abortos, utilizando un sistema HTM basado en *log* como entorno de evaluación. Comenzamos con una breve caracterización de los benchmarks y las transacciones que contienen, pues nos ayudará a interpretar los resultados de nuestras simulaciones posteriormente. Esta caracterización es necesaria puesto que cada carga de trabajo tiene sus características particulares que la distinguen en cómo es afectada por los conflictos, a pesar de estar desarrolladas bajo el mismo estilo de transacciones de grano grueso. Así, encontramos dos factores dependientes de la aplicación que determinan la frecuencia e influencia de los abortos en el rendimiento: el tamaño de la transacción y el peso del código transaccional en el tiempo de ejecución global.

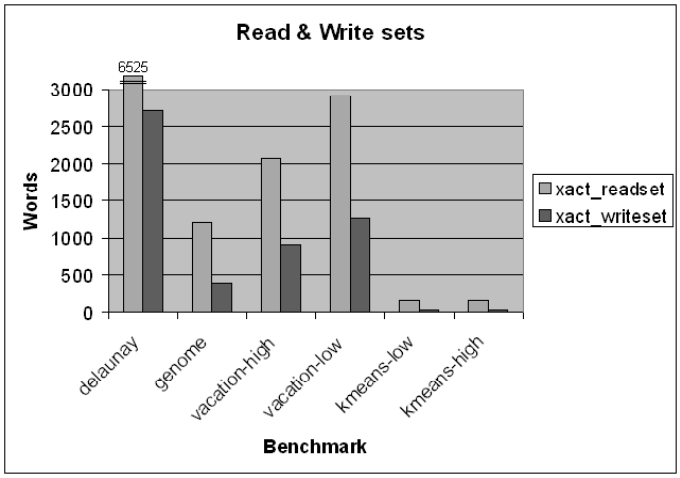


**Figura6.** Tamaño medio de los conjuntos de lectura y escritura en los benchmarks STAMP.

La cantidad de trabajo llevada a cabo por una transacción -cuya métrica cuantitativa es el tamaño de sus conjuntos de lectura y escritura- afecta directamente a la tasa de aborto, puesto que las posibilidades de un conflicto con una transacción concurrente aumentan con la cantidad de datos accedidos por la transacción.

La Figura 6 muestra el tamaño medio de dichos conjuntos, para cada uno de los benchmarks STAMP. Un análisis más detallado de cada transacción nos aporta información más valiosa, que resumimos en la Tabla 4. Estas estadísticas sólo dependen del tamaño del problema (entrada), y por tanto el número de núcleos es irrelevante. Podemos ver que tanto *delaunay* como *vacation* tiene dos transacciones principales que realizan el grueso del cómputo. En *vacation*, XID0 es de hecho la única transacción del benchmark. En *delaunay*, XID1 no sólo tiene los conjuntos de lectura y escritura más grandes, sino que también es la transacción ejecutada en más ocasiones; XID3, que no aparece en la Tabla 4 por motivos de espacio, tiene un tamaño pequeño y sólo se ejecuta una vez por cada hilo. Así, tanto para *delaunay* como para *vacation*, su transacción principal tiene conjuntos R&W enormes -de una a diez mil palabras- en comparación con *kmeans* y *genome* -de decenas a un pocos cientos-. Sólo la primera fase del algoritmo de *genome* utiliza una transacción que accede a un conjunto de datos mayor; las restantes cuatro transacciones (las dos últimas no aparecen en la tabla) se ejecutan muchas más veces que XID0 y acceden a pocos datos. En lo que se refiere a *kmeans*, sus tres transacciones acceden conjuntos de datos muy pequeños (incluso la principal, XID0) en comparación con *vacation* y *delaunay*. Esto resultados anticipan tasa de aborto más elevadas para *delaunay* y *vacation* que para *genome* y *kmeans*, si bien la tasa de fallo final depende del grado real de comunicación entre hilos.

**Tabla 4.** Caracterización de transacciones en los benchmarks STAMP.

| Benchmark | ID de Transacción/ Característica | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | XID0 | | | XID1 | | | XID2 | | |
| | Cuenta | Rset | Wset | Cuenta | Rset | Wset | Cuenta | Rset | Wset |
| DELAUNAY | 1900 | 20 | 12 | 3101 | 12975 | 5381 | 1209 | 104 | 54 |
| GENOME | 512 | 10927 | 3471 | 241 | 19 | 4 | 3615 | 110 | 54 |
| VACATION-LOW | 4096 | 2079 | 911 | - | - | - | - | - | - |
| VACATION-HIGH | 4096 | 2844 | 1237 | - | - | - | - | - | - |
| KMEANS-HIGH | 1000 | 212 | 26 | 333 | 2 | 2 | 8 | 2 | 1 |
| KMEANS-LOW | 1000 | 212 | 26 | 333 | 1 | 2 | 8 | 2 | 1 |

En la Figura 7 presentamos la duración media de la transacción principal de cada benchmark, junto con el ciclo relativo en el que dicha transacción suele abortar, en promedio. Como era de esperar, la duración de la transacción se incrementa conforme el número de cores aumenta, estando claramente correlacionada con el tamaño de sus conjuntos de lectura y escritura -aunque no completamente determinado por éste-. Vemos cómo la transacción XID0 de *genome* tarda más tiempo en completarse que la XID1 de *delaunay* en un entorno de 8 cores, a pesar de tener conjuntos de lectura y escritura más pequeños. Por motivos relacionados con la robustez del entorno de simulación, ninguno de los experimentos de 16 cores con *genome* se completó con éxito, de ahí que omitamos los resultados. *kmeans* en el único benchmark evaluado que no experimenta prácticamente ningún aborto para el tamaño de la entrada elegido, debido principalmente al pequeño tamaño y corta duración de sus transacciones. En *delaunay*, *vacation* y *genome* observamos que las transacciones terminan abortando cuando están más bien cerca (o incluso pasado) su tiempo medio de compleción. Esto indica que los abortos a menudo ocurren tras haber realizado una gran cantidad de trabajo así como generado bastante tráfico en la red, debido a las peticiones de coherencia de rechazo y reintento. Este resultado apunta en cierto modo la necesidad antes mencionada de mecanismos de recuperación más sofisticados, capaces de retener aquellas partes válidas de la transacción, las que no dependen de los datos que crearon el conflicto. Además, en un sistema con gestión de versiones como LogTM, abortar estas grandes transacciones no sólo significa un malgasto de tiempo y energía en operaciones que son descartadas posteriormente, sino también una sobrecarga de *rollback* considerable debido a la restauración de los viejos valores almacenados en el *log*.

Por último pero no por ello menos importante, el peso de la transacción en el tiempo de ejecución total determina la influencia de los conflictos en el rendimiento de la aplicación. A pesar de la habilidad del programador para extraer paralelismo de grano más fino, la naturaleza de la computación determina en última instancia cuánta sincronización es necesaria y por tanto cuánto tiempo se debe emplear en

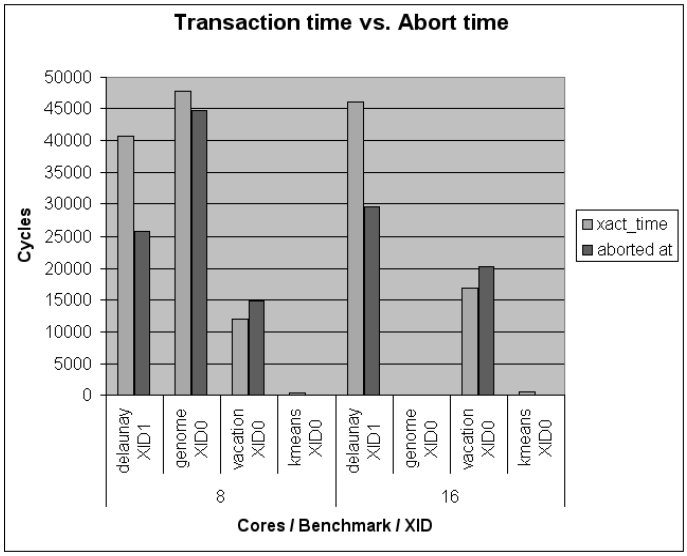


**Figura7.** Duración y tiempo de aborto promedio de la transacción principal en cada benchmark.

código transaccional. La Figura 8 muestra el tiempo de ejecución normalizado para cada benchmark en configuraciones de 8 y 16 núcleos, desglosado en tres partes: transacciones exitosas, transacciones abortadas, código no transaccional. Como podemos ver, *delaunay* es la aplicación que pasa más tiempo en transacciones que acaban abortando, alcanzando entre un 17 y 20 % del tiempo de ejecución total, para 8 y 16 cores, respectivamente. Esto se produce como resultado de unas elevadas tasas de aborto, de entre 0,33 y 0,46 para 8 y 16 núcleos, respectivamente. En otras palabras, una de cada dos transacciones iniciadas en una triangulación *delaunay* con 16 hilos no hace ningún trabajo útil. En el otro extremo se sitúa *kmeans*, que debido a sus pequeñas transacciones no experimenta prácticamente ningún aborto y por lo tanto los conflictos no tienen ninguna repercusión en el rendimiento de esta aplicación. Tanto *genome* como *vacation* presentan tasas de aborto moderadas, entre el 0,07 y el 0,10, respectivamete, y su tiempo empleado en transacciones abortadas no sobrepasa el 7 % en ambos casos. Las tasas de aborto obtenidas están en plena concordancia con el tamaño medio de la transacción y el peso del código transaccional, lo cual pone de manifiesto una clara correlación entre la granularidad de las transacciones en el código y el impacto de los abortos en el rendimiento global. En resumen, nuestros experimentos muestran que los sistemas de memoria transaccional hardware actuales basados en *log* son susceptibles de sufrir una degradación importante de rendimiento para aquellas cargas de trabajo transaccionales que pasan una gran mayoría de su tiempo de ejecución en transacciones grandes, de grano grueso.

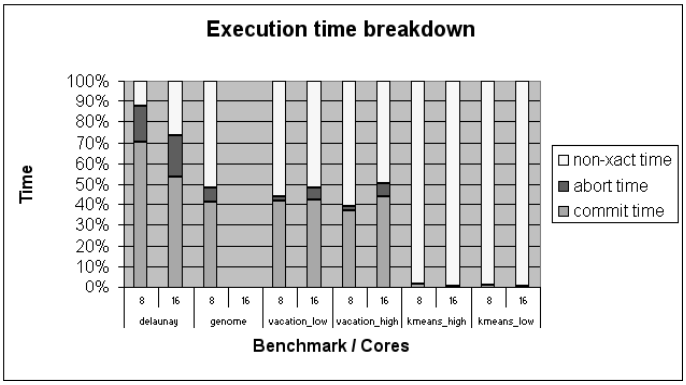


**Figura8.** Desglose del tiempo de ejecución normalizado para las aplicaciones transaccionales.

# 7. Hacia una implementación HTM flexible

A priori, un sistema de memoria transaccional desconoce la naturaleza del código transaccional que va a ejecutar. Por esa razón, sería deseable dotar a los HTMs de cierta flexibilidad de políticas, con el fin de evitar la aparición de interacciones indeseadas como las que Bobba et al. enumeran en su reciente estudio [4]. Cada política de gestión de versiones y detección y resolución de conflictos está expuesta a unas u otras *patologías* degradantes del rendimiento, y su aparición depende entre otros factores de las características específicas del código. Dada la potencial variabilidad del código transaccional, ninguna combinación de políticas parecer ser superior a las demás en todos los casos, y así lo muestra el estudio de Bobba et al. para las aplicaciones evaluadas.

A este respecto, en esta sección esbozamos 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. Nuestra meta es conseguir, de forma sencilla y con un mismo coste hardware similar al 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 la naturaleza de las transacciones. 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 reseteará periódicamente.

De esta forma, nosotros proponemos combinar las políticas ansiosas de gestión de versiones y detección de conflictos de LogTM-SE [21] 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, de LogTM-SE se heredan los registros base y puntero para soportar un *log* transaccional. Su novedad radica en que el *log* puede actuar no sólo como como *before-image* sino también como *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 buffer 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 *buffer especulativo*, 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 buffer.

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.

# 8. Conclusiones y trabajo presente y futuro

Con este trabajo hemos llevado a cabo dos aportaciones principales. Por una parte, hemos presentado las diferentes políticas de gestión de versiones y detección de conflictos que componen el espacio de diseño de un sistema de memoria transaccional hardware, señalando la influencia que la elección de cada política tiene en el grado de impacto de los conflictos sobre el rendimiento del sistema. Para cada decisión de diseño del sistema, nos hemos referido a propuestas concretas para comentar las bondades e inconvenientes de dicha aproximación, siguiendo un enfoque evolutivo -y en cierta forma cronológico- a la hora de presentar las diferentes soluciones en buena parte del trabajo. Creemos que como resultado, esta parte del trabajo es capaz de introducir los sistemas de memoria transaccional hardware a un lector no experimentado, paso a paso, de forma lógica y progresiva, hasta cubrir de forma breve el estado del arte en este área.

Nuestra segunda contribución se enmarca en un terreno más práctico y cuantitativo: Hemos llevado a cabo la primera caracterización de conflictos en un conocido sistema de memoria transaccional hardware (LogTM) que utiliza benchmarks verdaderamente transaccionales (STAMP). Así, la evaluación de estas aplicaciones transaccionales en LogTM ha requerido un esfuerzo a la hora de portar la suite de benchmarks al entorno de trabajo simulado, y nuestro trabajo ha contribuido a mejorar los recursos de los que dispone la comunidad investigadora en memoria transaccional. Con esta parte del trabajo tratamos de demostrar que las transacciones abortadas pueden llegar a ser bastante frecuentes en algunos tipos de cargas de trabajo transaccionales, y que este hecho ha de ser considerado a la hora de afrontar el diseño de los sistemas TM hardware. Nuestros resultados ponen de manifiesto cómo aquellas aplicaciones que pasan la mayor parte de su tiempo en unas pocas transacciones de gran tamaño son susceptibles de experimentar una degradación de rendimiento importante debido a la elevada tasa de aborto. Estos resultados respaldan nuestra postura acerca de las transacciones exitosas, que no son siempre el caso común, y nos llevan a apuntar que se necesitan esquemas más flexibles de gestión de versiones para que los sistemas TM hardware puedan llegar a tener éxito en entornos con una enorme variabilidad en la granularidad de las transacciones. En este sentido, hemos esbozado algunas líneas de un posible sistema flexible que soporte en hardware diferentes políticas de gestión de versiones y detección de conflictos. Además, la elevada tasa de aborto que observamos en nuestra evaluación también pone de relieve que la investigación en este área necesita centrarse en mecanismos de recuperación más avanzados, de forma que el sistema sea capaz de obtener buenas prestaciones a pesar de la presencia de frecuentes transacciones abortadas. De no conseguirlo, los sistemas TM hardware fallarán en su misión de liberar al programador del compromiso entre programabilidad y rendimiento que el modelo de memoria transaccional trata de evitar.

# 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. Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. High-performance embedded architecture and compilation roadmap. In *Transactions on HiPEAC I*, pages 5–29, Jun 2007.

8. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, Sept 2006.
9. 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.
10. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, Jun 2004.
11. 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.
12. Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.
13. Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, pages 92–99, September 2005.
14. 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.
15. Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, Oct 2006.
16. Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th International Symposium on Microarchitecture*, pages 294–305, December 2001.
17. Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, October 2002.
18. 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.
19. 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.
20. Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.
21. 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.