UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Paralelización Automática Basada en Memoria Transaccional

Miguel Ángel González Mesa

Febrero de 2014

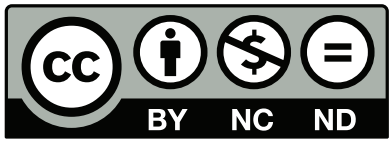
Dirigida por:
Óscar Plata,
Eladio Gutiérrez

UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica

AUTOR: Miguel Ángel González Mesa

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Dr. D. Óscar Plata González.
Catedrático del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. D. Eladio D. Gutiérrez Carrasco.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "Paralelización Automática Basada en Memoria Transaccional", ha sido realizada por D. Miguel Ángel González Mesa bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, a 15 de Enero de 2014

Dr. D. Óscar Plata González.
Codirector de la tesis.

Dr. D. Eladio D. Gutiérrez Carrasco.
Codirector de la tesis.

A mi familia

Agradecimientos

Esta tesis es el producto de la colaboración directa e indirecta de multitud de personas a las que quisiera dirigir unas breves palabras de agradecimiento con estas líneas.

En primer lugar, quiero dar las gracias a mis directores, Óscar Plata y Eladio Gutiérrez, por acogerme en su grupo de investigación y por su sabio consejo y su inestimable ayuda, sin la cual no habría sido posible la realización de esta tesis. Especialmente me gustaría resaltar el buen clima que desde el primer día se ha respirado tanto en las múltiples reuniones de trabajo, como en el trato diario. He de dar también las gracias al doctor Ezequiel Herruzo, de la Universidad de Córdoba, por su recomendación y consejo para que diera el paso de realizar mi tesis doctoral, y a Emilio Zapata que como director del departamento, allá por el año 2009, me acogió en este excelente equipo de trabajo.

Me gustaría extender este agradecimiento a todo el personal que integra el Departamento de Arquitectura de Computadores de la Universidad de Málaga, desde los técnicos (Juanjo, Paco y Mª Carmen) que me han sacado de más de un apuro, a la secretaria, Carmen, cuyo trabajo ha sido indispensable para que esto llegara a buen fin.

Debo agradecer la financiación recibida por parte del Ministerio de Ciencia e Innovación a cargo del proyecto de investigación TIN2010-16144; y de la Junta de Andalucía como parte de la Beca de Excelencia asociada al proyecto de investigación P08-TIC-4341 que he tenido el privilegio de disfrutar durante estos años.

Por último aunque no menos importante, quisiera agradecer encarecidamente el soporte anímico y afectivo que me ha brindado toda la gente de mi entorno. A mis compañeros y amigos de laboratorio que durante estos años han creado un ambiente de trabajo inigualable: Ricardo Quislant, Manolo Pedrero, Manolo R. Cervilla, Antonio Vilches, Óscar Torreño, Jose A. Arjona, Alejandro Villegas,

Sergio Muñoz, Alberto Sanz, Antonio J. Dios, Antonio Muñoz, Juan Lucena, Francisco Jaime, Pilar Poyato, Lidia Fernández, Iván Romero, Pau Corral, Rosa Castillo, Adrián Tineo, Miguel Ángel Sánchez, Juan Villalba, Juan Luna y a todos aquellos que haya podido olvidar. A mis amigos en Córdoba [aunque muchos de ellos ya estén dispersos por el mundo ;)] que han supuesto una válvula de escape esencial para superar el estrés a lo largo de estos años. A mi madre que me ha apoyado y soportado siempre, en especial en mis malos momentos (que no han sido pocos) y a toda mi familia.

¡Muchas gracias a todos!.

Resumen

Con la aparición de los chips multiprocesador, la industria del hardware ha pasado la responsabilidad sobre la eficiencia de las aplicaciones a la comunidad de desarrolladores de software. La gran mayoría de programadores aún continua desarrollando programas secuenciales, mientras que la mayor parte del desarrollo paralelo aun se realiza por grupos fuertemente especializados en programación paralela de alto rendimiento, debido a su complejidad y a la necesidad de disponer de una profunda comprensión de conceptos hardware, algoritmos y herramientas de programación paralelas.

La Memoria Transaccional (TM) emerge como una alternativa a la programación paralela convencional, facilitando el desarrollo de programas concurrentes. Estos sistemas proporcionan al programador las construcciones necesarias para definir bloques de instrucciones que ejecutan como unidades atómicas y aisladas (transacciones), siguiendo un modelo de concurrencia optimista. Estas transacciones se ejecutan en paralelo, a menos que se detecte un conflicto entre ellas en el acceso a un objeto común. La detección de conflictos se realiza de manera transparente por parte del sistema transaccional y supone un aspecto crítico para el rendimiento del sistema. Sin embargo, la memoria transaccional, como paradigma de programación, aun presenta algunas limitaciones a la hora de paralelizar determinados tipos de códigos, especialmente aquellos que presentan restricciones de precedencia, puesto que tienden a serializar en exceso su ejecución, desperdiciando así gran parte de la ganancia de rendimiento obtenida por el proceso de paralelización.

En esta tesis, proponemos un modelo de ejecución transaccional pensado para explotar la máxima concurrencia de programas secuenciales y paralelos que presenten restricciones de precedencia. Una sección de código del programa se divide en *chunks*, los cuales se asignan a diferentes hilos para ejecutar como transacciones. Estas transacciones ejecutan concurrentemente y de manera desordenada, intentando aprovechar el máximo paralelismo al tiempo que

se gestiona su finalización (*commit*) asegurando que las dependencias de datos mantengan una semántica secuencial basada en su orden de precedencia. Este orden de precedencia parcial, derivado de la sección de código del programa, es extraído por el compilador y/o definido por el programador para garantizar la corrección de la ejecución paralela. Cuando se detecte un conflicto, el sistema transaccional utilizará el orden de precedencia definido para aplicar la mejor política para resolverlo, preservando el orden y maximizando la explotación de la concurrencia. En este modelo se define un nuevo estado en el ciclo de vida de una transacción, denominado *executed-but-not-committed*, en el que esta ha finalizado su ejecución pero sus cambios aun no han sido confirmados en memoria. Este nuevo estado permite explotar la concurrencia en dos niveles: *intra-thread* and *inter-thread*. El primero es mejorado al permitir a los hilos lanzar nuevas transacciones manteniendo las anteriores activas, a la espera de poder realizar su *commit* cumpliendo con la semántica secuencial del programa original, evitando así que el hilo se bloquee. La explotación del segundo nivel se debe al desacoplo de las fases de ejecución y *commit*, que permiten relajar la condición de *commit* de las transacciones (condición necesaria para poder hacer visibles sus cambios en memoria), permitiendo desordenar los *commits* sin afectar a la semántica secuencial del programa original.

Además, debido a que las operaciones de reducción aglutinan una importante parte del tiempo de ejecución en muchas aplicaciones del mundo real, y a que su uso está fuertemente ligado a construcciones iterativas, hemos considerado los bucles de reducción como un escenario de especial interés en nuestro modelo de ejecución. En concreto, hemos analizado cómo el manejo de las direcciones de memoria pertenecientes a variables de reducción en entornos transaccionales, ayuda a facilitar y mejorar la explotación de concurrencia. Tras este análisis hemos propuesto un soporte como extensión a nuestro modelo de ejecución, con el fin de gestionar las operaciones de reducción como un caso especial de accesos conflictivos a memoria.

Todas nuestras aportaciones se han implemento sobre un conocido STM (TinySTM) y evaluadas utilizando un conjunto de *benchmarks* representativo, tanto para el ámbito transaccional como para la computación paralela de alto rendimiento.

Índice general

Índice de figuras

Lista de tablas

1 Introducción

1.1. Multiprocesadores

Tradicionalmente, los programas informáticos se han desarrollado pensando en una ejecución en serie. Para resolver un problema, se desarrolla un algoritmo y se implementa como un flujo en serie de instrucciones. Estas instrucciones son ejecutadas en un procesador una a una. Sin embargo, la exigencia de la computación de alto rendimiento y las aplicaciones científicas pronto mostraron las limitaciones de este modelo de ejecución. El desarrollo tecnológico permitió la integración de un número cada vez mayor de transistores dentro un chip, y por ende, un incremento en las prestaciones de los procesadores. Este mayor número de transistores trajo aparejadas mejoras en la arquitectura de los procesadores gracias a la inclusión de nuevas unidades funcionales, nuevos registros, así como más y mayores cachés. Además, estas mejoras vinieron acompañadas por un incremento en la frecuencia de reloj que se tradujo en una mayor velocidad de computación. Así pues, la complejidad y el rendimiento de los procesadores han tenido una evolución fuertemente dependiente de la escala de integración.

En 1965, Gordon Moore observó empíricamente que el número de transistores dentro del procesador se duplicaría cada 18-24 meses y que esta tendencia se prolongaría a lo largo de las siguientes décadas (*Ley de Moore* [68]). Hoy en día, esta observación aún es válida, llegando a cifras de en torno a los 1000 millones de transistores en los procesadores actuales. Estos dos aspectos, frecuencia de reloj y número de transistores, propiciaron la aparición de las primeras formas

de paralelismo en los procesadores con flujo de control secuencial. Concretamente, tres tipos de paralelismo interno [83, 74] (complementarios entre sí) fueron desarrollados:

- *Paralelismo a nivel de bit:* Al aumentar el tamaño de palabra, se redujo significativamente el número de instrucciones necesarias para realizar una operación determinada. De esta forma, hasta mitad de la década de los 80, el tamaño de palabra utilizada por los procesadores para sus operaciones fue incrementando gradualmente desde 4 a 32 bits. Esta tendencia se suavizó, acabando en la adopción de los 64 bits como estándar a principios de los 90.
- *Paralelismo por pipeline:* La idea detrás del paralelismo a nivel de instrucción o ILP (*instruction-level parallelism*) consiste en solapar la ejecución de múltiples instrucciones. Para ello, las instrucciones deben ser segmentadas en distintas fases (generalmente *búsqueda, decodificación, ejecución y escritura*) que son ejecutadas por diferentes unidades funcionales una tras otra. Haciendo estas fases de igual tamaño y sincronizándolas mediante ciclos de reloj, se consigue que múltiples instrucciones ejecuten en paralelo, manteniéndolas en etapas diferentes. Este tipo de paralelismo funciona bien en ausencia de dependencias.
- *Paralelismo por múltiples unidades funcionales:* Este paralelismo consiste en utilizar múltiples e independientes unidades funcionales, tales como ALUs (unidades aritmético lógicas), FPUs (unidades de punto flotante), unidades de carga/almacenamiento, o unidades de salto. Estas unidades crean varios cauces de ejecución que permiten ejecutar diferentes instrucciones independientes en paralelo. No obstante, el número de unidades funcionales que puede ser utilizado de manera eficiente está restringido por las dependencias de datos entre instrucciones.

Durante años, los factores que han mantenido el incremento en el rendimiento de los procesadores han sido el aumento de la frecuencia de reloj y el aprovechamiento del paralelismo interno obtenido por la explotación del *pipeline* y de las *múltiples unidades funcionales*. Sin embargo, desde hace algún tiempo su límite parece estar cada vez más cerca en los procesadores de ámbito general. Además, el incremento de la densidad de transistores en chip y la frecuencia de reloj hicieron aparecer nuevos problemas de consumo de potencia y disipación de calor (figura 1.1).

Por ello se optó por pasar de un enfoque basado en el incremento de la complejidad de la organización interna del chip procesador, a una nueva arquitectura

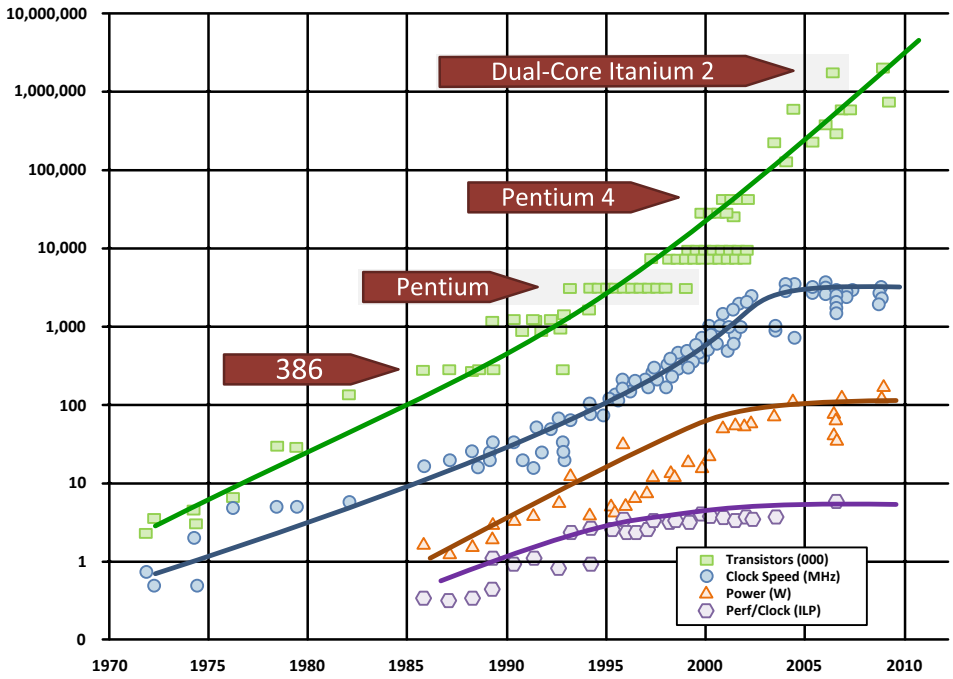


Figura 1.1: Evolución de los procesadores Intel desde 1970 hasta 2010 (Intel, K. Olukotun). Extraído de la presentación *Introduction to many-core architectures by Henk Corporaal (ASCI Wintherschool on Embedded Systems)*.

que integrase múltiples núcleos de procesamiento independientes, y relativamente simples, dentro de un mismo chip. Este nuevo enfoque permitía no solo resolver el problema en la disipación de calor, al reducir la densidad de transistores en cada núcleo y su frecuencia de reloj, sino también reducir el consumo de energía mediante la desactivación de aquellos núcleos de procesamiento inactivos durante los periodos de poca exigencia computacional. El resultado han sido los llamados *procesadores multinúcleo* (*multicore processors*), en los que cada núcleo dispone de un flujo de control separado. De este modo comienza a cobra mayor importancia la explotación eficiente del conocido como *Paralelismo a nivel de hilo/proceso*; lo cual tiene una consecuencia directa en la manera en que se escriben y codifican los programas.

1.2. Complejidad de la programación paralela

Los computadores paralelos han sido utilizado durante muchos años, a lo largo de los cuales se han propuesto muchas numerosas alternativas en cuanto a arquitectura. Una manera simple de clasificarlas es siguiendo la *taxonomía de Flynn* [28], la cual distingue cuatro categorías:

- *Single-Instruction, Single-Data (SISD):* Se corresponde con la arquitectura convencional de computadora secuencial de acuerdo al modelo *Von Neumann*, en la que hay un único elemento de procesamiento que accede a un único programa y almacenamiento de datos. En cada paso, se carga una instrucción y sus correspondientes datos, para seguidamente ejecutarla.

- *Multiple-Instruction, Single-Data (MISD):* Su arquitectura de compone de múltiples elementos de procesamiento, cada uno con su memoria de programa privada, pero un único acceso común a una única memoria compartida. En cada paso, todos los elemento de procesamiento obtienen un mismo dato, sobre el que cada uno aplica una instrucción diferente obtenida de su memoria privada de programa.

- *Single-Instruction, Multiple-Data (SIMD):* En esta categoría se agrupan aquellas arquitecturas con múltiples elementos de procesamiento, cada uno de los cuales tiene acceso a una memoria de datos (compartida o distribuida); pero solo dispone de una única memoria de programa, gestionada por un procesador de control. En cada paso, cada elemento de procesamiento obtiene una misma instrucción del procesador de control, la cual aplica a un conjunto diferente de datos.

- *Multiple-Instruction, Multiple-Data (MIMD):* Las arquitecturas en esta categoría disponen de múltiples elementos de procesamiento que trabajan de manera asíncrona e independiente, disponiendo de acceso a una memoria de datos y programa compartida o distribuida. Así en cada paso, cada elemento de procesamiento puede ejecutar diferentes instrucciones sobre diferentes datos. Los procesadores multinúcleo o los *clusters* son claros ejemplos de tipo de arquitecturas.

De entre estas categorías, aquellas que siguen un modelo MIMD, son las que han despertado un mayor interés en los últimos años, en especial para computación paralela de propósito general. Los MIMD pueden a su vez organizarse en diferentes grupos, en base a la organización física de la memoria (maquinas

de memoria distribuida (DMM) o compartida (SMM)) y a la percepción de esta desde el punto de vista del programador.

En esta tesis centraremos nuestro estudio en arquitecturas con una memoria física compartida (SMM), y más concretamente, en una variante que aglutina múltiples núcleos de ejecución independientes en un único chip, también conocida como *Chips Multiprocessors (CMP's)*. Estos sistemas disponen de una red de interconexión que conecta cada núcleo de procesamiento con una memoria compartida (memoria global). El modelo de programación natural en esta clase de arquitecturas consisten en utilizar la memoria compartida para intercambiar información entre los núcleos de procesamiento mediante el uso de variables compartidas. Pese a las ventajas que presenta, la existencia de una memoria compartida también presenta algunos inconvenientes. Los accesos concurrentes por parte de los núcleos de procesamiento a variables compartidas deben coordinarse puesto que pueden ocasionar *condiciones de carrera* con efectos impredecibles sobre la consistencia de memoria.

Para evitar esto, los programadores deben encargarse de sincronizar estos accesos, haciendo uso del modelo de *paralelismo a nivel de hilo*. La idea detrás de este tipo de paralelismo consiste en la ejecución de múltiples hilos (*threads*) que colaboran para resolver un problema de forma paralela. Un hilo consta de un flujo de control independiente, que comparte datos con otros hilos a través de un espacio de dirección global. Por lo general, el paralelismo a nivel de hilo no se puede explotar en su totalidad únicamente por hardware, puesto que se precisan varios pasos previos no triviales tales como descomponer el problema, asignar el trabajo a los hilos, gestionar y sincronizar estos hilos y mapearlos sobre la arquitectura.

Por ello, su explotación requiere de algún elemento software (compilador, librerías, *runtime*) y/o soporte por parte del programador. Una aproximación en la que el programador interviene mínimamente es la *paralelización automática* [65]. En ella el compilador, con la ayuda de librerías o soporte *runtime*, se encarga de analizar las dependencias en el programa y generar una versión paralela correcta de la aplicación. Este tipo de análisis suele ser muy complejo y conservador, especialmente con códigos que utilizan indirecciones, punteros o estructuras de datos dinámicas, por lo que solo es apropiado para un rango muy limitado de aplicaciones. Además, en muchos casos, el no determinismo ocasionado por la ejecución concurrente, así como el paralelismo oculto que presentan algunos códigos, impiden que se pueda realizar este análisis en tiempo de compilación como base la paralelización estática. Por ello, en una gran parte de estas aplicaciones debe ser el programador el que especifique explícitamente, a nivel de lenguaje, las partes del código a paralelizar.

Por otra parte, en base a cómo se realice la partición del trabajo asignado a cada hilo, podemos destacar las dos principales aproximaciones más ampliamente utilizadas: *Paralelismo a nivel de tarea* y *paralelismo a nivel de datos*. En el paralelismo a nivel de tarea, dividimos el problema en varias tareas y asignamos estas a los hilos de ejecución. De esta manera cada hilo efectúa su propia secuencia de operaciones. En el paralelismo de datos, se particiona el conjunto de datos utilizados para resolver el problema y se distribuyen entre los hilos de ejecución, de manera que cada uno de ellos realiza un conjunto de operaciones, mas o menos similar, sobre su parte de los datos. Este reparto de datos no necesariamente tiene que ser disjunto, ya que varios hilos pueden necesitar datos comunes para alcanzar una solución correcta.

Por ello, independientemente de cómo se desarrolle la versión paralela de la aplicación (automática o manual) y de cómo se divida el problema (tareas o datos), si existe riesgo de acceso concurrente a datos compartidos, es necesario tomar medidas para evitar inconsistencias en la ejecución paralela. El mecanismo tradicional para sincronizar los hilos en su acceso a datos compartidos siempre ha sido el uso de primitivas clásicas de sincronización (barreras, semáforos, monitores, etc.), siendo el uso de *locks* la más representativa de de todas ellas a bajo nivel. Un *lock* es una variable compartida que se modifica utilizando operaciones atómicas, de manera que solo un hilo pueda adquirirlo a la vez. Una vez adquirido el *lock*, el hilo accede a la sección crítica mientras que otros hilos esperan a que el *lock* se libere. Sin embargo, los *locks* presentan dos aspectos que degradan el rendimiento paralelo y limitan la escalabilidad: *overhead* y *contención*. El *overhead* hace referencia al tiempo necesario para inicializar el *lock*, el tiempo necesario para adquirirlo y liberarlo, y el espacio de memoria necesario para almacenarlo. La *contención* es el tiempo consumido por los hilos que están bloqueados a la espera de adquirir el mismo *lock*. Existe un compromiso entre ambos aspectos dependiendo de la granularidad del *lock*. *Locks* de grano fino pueden incrementar las oportunidades de paralelismo pero a cambio de elevar el *overhead*,y requerir un mayor esfuerzo de programación. *Locks* de grano grueso simplifican la programación paralela, pero añaden un alto grado de serialización.

Una alternativa a la exclusión mutua mediante *locks*, es el uso de algoritmos no bloqueantes (*lock-free, wait-free*) [37, 48]. Estos métodos utilizan primitivas especificas proporcionadas por el hardware, tales como CAS (*compare-and-swap*) o TS (*test-and-set*), que permiten leer y escribir una posición de memoria atómicamente, garantizando que ninguna otra instrucción puede intercalarse entre ellas. A diferencia de los *locks*, la programación no bloqueante permite a los hilos progresar de forma optimista, permitiendo ejecutar concurrentemente la sección crítica mientras no haya conflictos en memoria entre hilos diferentes. En el caso

de conflicto, los hilos reintentan la ejecución de la sección crítica hasta tener éxito. Sin embargo, el diseño de algoritmos no bloqueantes es complicado y depende del conjunto de primitivas atómicas disponible en el sistema para alcanzar un buen rendimiento.

1.3. La memoria transaccional

En 1977, David B. Lomet realizó la observación de que una abstracción similar al concepto de transacciones presente en bases de datos podrían constituir un buen mecanismo para asegurar la consistencia de datos compartidos en los lenguajes de programación [59]. Sin embargo, no sería hasta 1993 cuando se acuñaría el termino *memoria transaccional (TM)* de mano de la primera propuesta hardware basada en esta idea, realizada por Maurice Herlihy y J. Eliot B. Moss [48].

La memoria transaccional [54, 45] emergió, así, como una alternativa viable a las construcciones clásicas para mantener la consistencia (*locks*, construcciones *lock-free*, etc) en paradigmas de programación paralela. Se introduce el concepto de *transacción* como una secuencia de instrucciones que se ejecutan especulativamente en un procesador como una unidad ***atómica y aislada***. Un sistema de memoria transaccional ejecuta transacciones en paralelo, permitiendo finalizar (*commit*) a aquellas que no presenten conflictos en memoria. Durante su ejecución, el sistema TM registra los conjuntos de datos leídos (*read set*) y escritos (*write set*) por cada transacción. Cuando dos o más transacciones intentan acceder al mismo dato compartido y al menos una operación es una escritura, se dice que las transacciones están en conflicto. En este caso, el conflicto debe ser resuelto serializando la ejecución de las transacciones conflictivas a fin de preservar la atomicidad.

Los lenguajes de programación modernos proporcionan potentes mecanismos de abstracción, tales como la composición, que facilitan el desarrollo incremental y la reusabilidad de los componentes *software*. Sin embargo, las primitivas de bajo nivel para la sincronización explicita (*locks*) no son adecuadas para la construcción de estas abstracciones, ya que pueden causar situaciones de *deadlock*. La memoria transaccional permite resolver el problema de la composición, además reduce la serialización producida por los *locks* y otras construcciones clásicas gracias a un modelo de ejecución optimista que permite maximizar la cantidad de concurrencia explotada por la aplicación. Aunque la memoria transaccional resuelve muchos de los inconvenientes de las construcciones clásicas, también introduce nuevas situaciones problemáticas como *livelock* y otras

patologías [13] derivadas de las características propias de esta nueva abstracción, aunque todas ellas pueden ser resueltas por el sistema transaccional para asegurar el progreso del sistema.

Desde la propuesta de Lomet, muchas implementaciones de memoria transaccional han aparecido tanto en software (STM) [93, 49, 46, 38, 60, 22] como en hardware (HTM) [47, 80, 43, 81, 2, 69, 21], así como implementaciones híbridas (HiTM) [20, 53] y aceleradas por hardware [89]. Aunque los sistemas HTM se diseñan para presentar un rendimiento superior a los sistemas STM, estos últimos han demostrado ser mucho más versátiles y presentan menos limitaciones, lo que las convierte en una herramienta de investigación excelente. Aunque los fabricantes de hardware están empezando a incorporar los conceptos de TM a sus nuevos procesadores *multicore* [17, 19, 85], como es el caso de Intel con la inclusión de su soporte *Transactional Syncronizations Extensions (TSX)* en su arquitectura Haswell o AMD con su *Advanced Synchronization Facility (ASF)*, estos aun se presentan como una versión muy básica y limitada de memoria transaccional, lejos de las más novedosas y prometedoras propuestas.

1.4. Motivación y contribuciones

La memoria transaccional proporciona un modelo de ejecución desordenado, donde las transacciones se pueden ejecutar y realizar su *commit* en cualquier orden alcanzando un resultado correcto. En presencia de un conflicto entre dos transacciones, el sistema puede resolverlo abortando cualquiera de ellas. No obstante, esta aproximación solo es apropiada cuando se aplica a la paralelización de bloques de código sin relación semántica entre sí, más allá del acceso a objetos compartidos.

Sin embargo, son numerosas las construcciones de programación en las que los bloques de código presentan fuertes relaciones de dependencia entre sí que limitan o restringen el orden en que deben ser ejecutadas para garantizar la corrección de los resultados

En adelante, utilizaremos el término *códigos con restricción de precedencia* para hacer referencia a estos bloques de instrucciones, ya sean en forma de iteraciones individuales (o grupo de ellas) de un bucle, tareas o sencillamente conjunto de instrucciones de un programa secuencial, que precisan del resultado o la ejecución de otras previas para completar su trabajo produciendo un resultado correcto.. Es el caso de la paralelización de bucles con dependencias *loop-carried* o de conjuntos de tareas dependientes, en las que un conjunto de iteraciones

o tareas dependen directamente del resultado de la computación realizada en otras, con el fin de realizar su trabajo. Además, si estas dependencias vienen determinadas por los datos que se computan durante la ejecución de la aplicación, se hace muy complejo o incluso imposible su análisis y, como consecuencia, la forma habitual de asegurar la corrección es ejecutar estos bloques de código secuencialmente. Desafortunadamente, este tipo de construcciones no son infrecuentes, estando presentes en muchos dominios de aplicación, incluyendo la computación con matrices dispersas, dinámica molecular, biología molecular o procesamiento de imágenes.

Esta tesis está motivada por este tipo de aplicaciones en los que la explotación del paralelismo no es trivial, o directamente posible, sin incurrir en un alto grado de serialización. En esta tesis analizamos el paradigma de ejecución optimista y especulativa, propuesto por la memoria transaccional, como soporte para facilitar la paralelización de códigos con restricciones de precedencia. Aunque ya existen algunas propuestas que establecen algún tipo de restricción de orden entre las transacciones del sistema ([43, 4, 9, 27, 78, 82]), en general, este mecanismo de ordenación es utilizado para obtener un esquema de serialización de conflictos que evite posibles abortos de transacciones y no específicamente para asegurar el orden entre las dependencias de datos sea equivalente con el de la ejecución secuencial del programa.

Las aportaciones realizadas en esta tesis doctoral son las siguientes:

- Una evaluación de la idoneidad del paradigma de memoria transaccional aplicado a la paralelización de códigos secuenciales con restricciones de precedencia se presenta como una alternativa intuitiva y de baja complejidad respecto a otras propuestas clásicas.
- Se introduce y evalúa un modelo de ejecución transaccional para códigos con restricciones de precedencia que se basa en lo siguiente:
  - Definición explícita de las dependencias entre transacciones por parte del programador (o del compilador) de manera que el sistema transaccional subyacente pueda ejecutarlas en cualquier orden en tanto que las dependencias entre ellas sean mantenidas de acuerdo al orden impuesto.
  - Desacoplo de las fases de ejecución y *commit* de la transacción gracias a la introducción de un nuevo estado *executed-but-uncommited* que mantiene activo su contexto hasta que se realiza el *commit*, ayudando a simplificar y relajar las restricciones de precedencia entre transacciones.

  - Soporte para el lanzamiento continuo de nuevas transacciones en cada procesador, aunque las anteriores no hayan confirmado sus cambios a memoria, lo que permite mayor paralelismo y flexibilidad en el proceso de ejecución, ocultando a su vez la latencia derivada del mantenimiento del orden de precedencia.

- Desarrollo de un soporte transaccional especial para la paralelización de bucles de reducción que considera las características excepcionales de este tipo de construcciones para maximizar el paralelismo explotado, sin incurrir en los conflictos innecesarios que provocaría la aplicación de una propuesta transaccional clásica sobre estas estructuras que presentan un paralelismo total.

Fruto de este trabajo se publicado artículos en tres congresos de ámbito internacional [33, 35, 36]; en tres *workshops* también de ámbito internacional [34, 30, 31]; y así mismo, ha sido aceptada para su publicación una contribución derivada de este trabajo en la revista *ACM Transactions on Architectural and Code Optimization (TACO)* [32] indexada en el *ISI Journal Citation Reports (JCR)*.

A continuación se resumen las principales aportaciones incluidas en de cada una de ellas:

➲ M. Angel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Automatic loop parallelization using transactional memory support. In *16th Workshop on Compilers for Parallel Computers (CPC'12)*, Padova, Italy, January 2012.

➲ M. Angel Gonzalez-Mesa, Eladio Gutierrez, and Oscar Plata. Leveraging transactional memory to extract parallelism. In *Euro-TM Workshop on Tranactional Memory (WTM'12)* (co-located with EuroSys 2012), Bern, Switzerland, April 2012.

En estos artículos se presenta una primera propuesta acerca de la utilización de la memoria transaccional como vehículo para extraer paralelismo de códigos iterativos y que conforman la génesis del modelo propuesto en el capítulo 4. Básicamente, se analiza la idea de utilizar el orden lexicográfico definido por el código secuencial para filtrar los conflictos derivados de las falsas dependencias de datos, al tiempo que se intenta limitar la serialización del sistema ocasionada

por las verdaderas dependencias explorando alternativas como la detención de transacciones o el uso de adelantamiento de datos para reducir la tasa de abortos.

➲ Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez, and Oscar Plata. Parallelizing the sparse matrix transposition: Reducing the programmer effort using transactional memory. Procedia Computer Science, 18(0):501 – 510.*International Conference on Computational Science (ICCS'13)*, Barcelona, Spain, June 2013 .

Este artículo presenta un estudio detallado de la paralelización de un algoritmo para la transposición de matrices dispersas, como un caso de estudio ilustrativo en el que el uso del paradigma de memoria transaccional, unido a la definición de un orden de precedencia, ayuda a reducir de manera drástica la complejidad y cantidad de conocimiento necesaria para su paralelización en comparación con otras soluciones clásicas para este problema. Este estudio ha sido incluido al inicio del capítulo 5.

➲ M. Angel Gonzalez-Mesa, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Effective transactional memory execution management for improved concurrency. In *ACM Transactions on Architectural and Code Optimization (TACO)*, In Press.

En esta contribución, aceptada para su publicación, generalizamos y ampliamos las propuestas anteriores definiendo y validando el núcleo del modelo de ejecución transaccional con orden de precedencia propuesto en el capítulo 4. Además, de la descripción del modelo, también se presenta la implementación y evaluación del mismo sobre un conocido STM (segunda mitad del capítulo 5).

➲ M. Angel Gonzalez-Mesa, Eladio Gutierrez, and Oscar Plata. Parallelizing irregular reductions using transactions. In *17th Workshop on Compilers for Parallel Computers (CPC'13)*, Lyon, France, July 2013.

➲ M. Angel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Dealing with reduction operations using transactional memory support. In *25th Int'l. Symp. on Computer Architecture and High-Performance Computing (SBAC-PAD'13)*, Porto de Galinhas, Brazil, October 2013.

➲ Miguel A Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Exploring irregular reduction support in transactional memory. In *13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*, pages 257–266. Springer, Vietri sul Mare, Italy, December 2013.

Por último, estas aportaciones suponen el núcleo del soporte transaccional para reducciones propuesto en el capítulo 6. En ellas se realiza un estudio de las características de estas operaciones y se define un mecanismo de ejecución transaccional que evita los conflictos ocasionados por sus accesos concurrentes.

1.5. Estructura de la tesis

En esta tesis se recogen las propuestas, análisis, detalles de implementación y evaluación experimental acerca de las ideas introducidas en la sección anterior, organizándose de la siguiente manera:

En el capítulo 2 se presentan las principales decisiones de diseño a la hora de desarrollar un sistema de memoria transaccional. También se realiza una descripción de los sistemas TM precursores tanto, hardware como software, haciendo especial hincapié en aquellos que imponen restricciones en el orden de finalización entre transacciones.

A continuación, el capítulo 3 describe la metodología seguida para la evaluación de las propuestas realizadas en esta tesis. Esto incluye una descripción de la infraestructura de las simulaciones, así como de las aplicaciones de evaluación o *benchmarks* utilizados. Además, se presenta una pequeña descripción de las métricas.

Un estudio de las principales estructuras de programación que presentan restricciones de precedencia es presentado en el capítulo 4. Seguidamente, presentamos nuestro modelo teórico para la ejecución de transacciones bajo orden de precedencia, denominado TEPO (*Transaction Execution under Precedence Order*), que proporciona las bases para desarrollar un sistema que extraiga el máximo paralelismo en estos tipos de códigos.

En el capítulo 5 ponemos el foco de atención sobre una de las construcciones analizadas en el capitulo anterior, los códigos iterativos, a través del caso de estudio de la *transposición de matriz dispersa* y presentamos una implementación de nuestro modelo TEPO aplicada a este dominio. Sobre la base de un STM ya

existente, se describen los principales detalles de implementación y se realiza una evaluación utilizando un conjunto de aplicaciones representativo.

El capítulo 6 discute los inconvenientes del uso de la memoria transaccional en la paralelización de bucles reductivos y propone una extensión del modelo TEPO para tratar con las particularidades de este tipo de construcciones, así como una implementación software que hace uso de las ideas presentadas.

Finalmente, el capítulo 7 sintetiza las principales aportaciones realizadas en la tesis y discute las líneas de trabajo futuras derivadas de ellas.

2 Antecedentes y trabajos relacionados

Dos de las principales aproximaciones para la explotación optimista de la concurrencia son TM y TLS (*Thread-Level Speculation*) [95]. Mientras que TM se enfoca en la sincronización optimista de programas multihilo, TLS se centra en la paralelización especulativa de programas secuenciales. Una de las principales diferencias entre TM y TLS es que esta última asume que la semántica secuencial del programa original se mantiene en la ejecución paralela, mientras que TM presenta una naturaleza desordenada. Sin embargo, ambos conceptos están fuertemente relacionados, como se indica en [76], donde se analizan las implicaciones de usar TM para soportar *speculative multithreading*. En esta tesis se propone un modelo que hace uso de la memoria transaccional para soportar una ejecución especulativa de transacciones sujetas a un orden de precedencia definido por el usuario, el cual representa la semántica secuencial del programa.

Para ello, en este capítulo se introduce la memoria transaccional como paradigma de programación paralela (sección 2.1) y se presenta una descripción de las principales decisiones de diseño que afectan al desarrollo de sistemas TM (sección 2.2). Los sistemas precursores, así como las propuestas más destacadas, tanto hardware como software, se exponen en la sección 2.3, haciendo especial hincapié en aquellos sistemas que mantienen orden entre las transacciones y que están estrechamente relacionados con la motivación principal de esta tesis.

2.1. TM como paradigma de programación paralela

En el ámbito de la programación paralela, una transacción es una secuencia de instrucciones que aparecen como indivisibles e instantáneas para el programador. Una transacción debe cumplir con tres propiedades específicas: atomicidad (*atomicity*), consistencia (*consistency*) y aislamiento (*isolation*) - colectivamente conocidas como propiedades ACI[1] [50, 54]:

- ***Atomicidad***: Esta propiedad garantiza que, o bien todas las instrucciones que componen la transacción se ejecutan satisfactoriamente, o que ninguna lo haga. No es aceptable una situación en la que solo unas pocas instrucciones de la transacción ejecuten correctamente, ni que aquellas transacciones que hayan fallado en alguna instrucción dejen atrás evidencias de su ejecución en forma de cambios en memoria, registros, etc. Así, una transacción presenta solo dos posibilidades: completar satisfactoriamente (*commit*) o fallar y restaurar el estado del sistema al que había justo al inicio de la transacción (*abort*).
- ***Consistencia***: Esta propiedad es totalmente dependiente de la aplicación y hace referencia al conjunto de variables y estructuras de datos que esta maneja. Una transacción que modifica el estado del sistema, siempre debe partir de un estado consistente y dejar el sistema en un estado consistente a su finalización, con independencia que lo haga satisfactoriamente o no. Mantener la consistencia del sistema se consigue simplemente abortando las transacciones fallidas, lo cual deshace todos los cambios realizados y devuelve el sistema al estado consistente de partida.
- ***Aislamiento***: Requiere que una transacción no interfiera con ninguna otra mientras están ejecutando, independientemente de si están ejecutando en paralelo o no. Se entiende por interferencia entre transacciones el que accedan a objetos compartidos, pudiendo producir un posible estado de inconsistencia. Esta propiedad es precisamente la que convierte a la memoria transaccional en un modelo atractivo para la programación paralela.

La idea detrás de las propiedades ACI de una transacción es que proporcionen al programador un mecanismo para coordinar accesos concurrentes a objetos

[1] El conjunto de propiedades en el modelo transaccional en bases de datos (ACID) incluye una cuarta propiedad denominada durabilidad (durability), la cual requiere que una vez que la transacción ha realizado su *commit*, sus modificaciones deben ser permanentes y estar disponibles a todas las transacciones a partir de ese momento. Esta propiedad no es realmente importante en memoria transaccional puesto que los datos en memoria son normalmente volátiles.

compartidos (lecturas y escrituras) en sistemas multihilo o multiprocesador. Actualmente, esta coordinación es tarea de los programadores, que mediante el uso de mecanismos de sincronización de bajo nivel, tales como *locks, semáforos, operaciones atómicas, barreras, etc.*, previenen de estas interferencias. Sin embargo, estos presentan una serie de inconvenientes asociados principalmente al *overhead* y a la alta serialización que imponen sobre el sistema paralelo. Además requieren de un mayor conocimiento sobre la aplicación y las estructuras de bajo nivel, por parte del programador, para evitar problemas como *deadlocks* o degradaciones importantes del rendimiento.

La memoria transaccional, como paradigma de programación paralela, proporciona una abstracción en la que toda la coordinación necesaria para garantizar la correcta ejecución de los programas paralelos es realizada por el sistema transaccional de manera eficiente y transparente al programador.

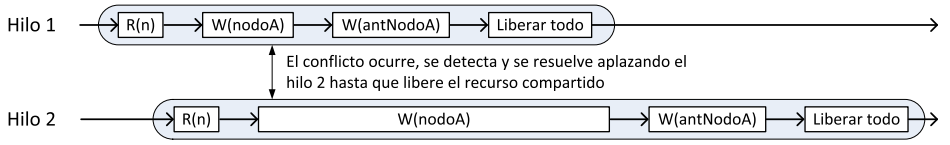
2.2. Decisiones de diseño en memoria transaccional

En esta sección se introducen las principales decisiones de diseño que caracterizan un sistema de memoria transaccional, cada una de las cuales representa un aspecto de la semántica de una transacción que puede influir significativamente en el desempeño global del sistema. Hay que tener en cuenta que la relación entre algunas de las decisiones de diseño es tal, que la elección de una de ellas puede llegar a condicionar otras. No obstante, conviene destaca que no existe una configuración que desempeñe universalmente bien, ya que el rendimiento del sistema transaccional esta, en cierta medida, ligado a las características de la aplicación sobre la que se aplique, por lo que cualquier decisión de diseño puede resultar favorable para un domino concreto de aplicación.

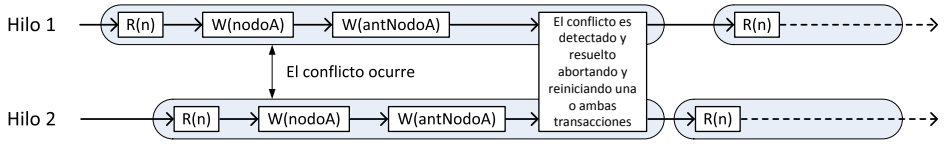
2.2.1. Control de concurrencia

Un sistema de memoria transaccional necesita mediar en los accesos concurrentes a objetos [2] compartidos, a fin de evitar los riesgos potenciales asociados a los conflictos. De ello se encarga el control de concurrencia, que se define por tres eventos que conforman el ciclo de vida de un conflicto:

[2] El término *objeto* es utilizado en esta sección para hacer referencia la unidad mínima a la cual el sistema transaccional detecta los conflictos, pudiendo usarse para identificar direcciones de memoria, estructuras de datos o instancias a objetos. Su significado estará sujeto al nivel de granularidad seleccionado para el sistema transaccional.



(a) Control de concurrencia pesimista



(b) Control de concurrencia optimista

Figura 2.1: Diferentes aproximaciones en control de concurrencia

- Un conflicto *ocurre* cuando dos transacciones realizan operaciones conflictivas sobre el mismo objeto - ambos escriben concurrentemente, o bien una transacción escribe y la otra lee.
- El conflicto es *detectado* cuando el sistema de memoria transaccional se percata de que ha ocurrido una situación que puede derivar en una inconsistencia.
- El conflicto es *resuelto* cuando el sistema transaccional toma las acciones pertinentes para resolver la situación de inconsistencia.

Estos eventos necesariamente tienen que suceder en este orden, sin embargo el instante en que ocurren a lo largo de la vida de la transacción define los dos tipos de control de concurrencia.

En el ***control de concurrencia pesimista*** todos los eventos (ocurrencia, detección, resolución) suceden en el mismo punto de la ejecución, cuando una segunda transacción intenta acceder a un objeto compartido ya accedido (Fig 2.1a). Este control de concurrencia garantiza el acceso exclusivo de una transacción a un objeto potencialmente conflictivo. Por otro lado, en el ***control de concurrencia optimista*** (Fig 2.1b) la detección y resolución del conflicto suceden después de que se haya producido el conflicto. Esto supone que múltiples transacciones puedan acceder concurrentemente a un mismo objeto y continuar su ejecución. En algún punto de la ejecución, antes de que la transacción efectúe el *commit*, el conflicto es detectado y resuelto.

Otra dimensión del control de concurrencia es la garantía de progreso de la transacción a fin de que pueda completar su ejecución en un número finito de pasos. Pueden destacarse dos propuestas:

- ***Sincronización bloqueante*** es aquella que utiliza construcciones tradicionales tales como *locks*, *monitores o semáforos*, no obstante esta aproximación no garantiza el progreso de los hilos, los cuales pueden incurrir en *deadlock* o ser detenidos por un periodo de tiempo indefinido.

- Los sistemas que utilizan ***sincronización no bloqueante*** garantizan que un hilo detenido no pueda bloquear a otros hilos indefinidamente, lo cual garantiza el progreso del sistema. Dependiendo del grado en que se cumpla esta garantía, podemos encontrar tres aproximaciones:

  - *Wait freedom* es la garantía no bloqueante más fuerte y asegura que todos los hilos que intentan acceder a un conjunto de objetos común, lo conseguirán en algún momento si continúan ejecutando. Como resultado, problemas tales como *deadlocks* y *starvarion* no pueden ocurrir. Aunque existen gran variedad de técnicas de propósito general para construir algoritmo *wait-free*, estas implementaciones son normalmente demasiado lentas.

  - *Lock freedom* asegura que al menos un hilo intentando acceder a un grupo de objetos común lo conseguirá si continua ejecutando. En el contexto de la memoria transaccional, esto significa que una transacción pueda abortar a otra, pero solo si eso asegura que alcanzará su *commit*. Así, si una transacción T_A aborta otra transacción T_B entonces T_B (o cualquier otra transacción) nunca podrá abortar a T_A.

  - *Obstruction freedom* es, posiblemente, la garantía más débil. Asegura que si todos los otros hilos intentando acceder al grupo de objetos común son suspendidos, entonces el hilo que queda podrá continuar y eventualmente completar su trabajo. En la práctica, esto implica que una transacción tenga la capacidad de abortar a otras transacciones que han adquirido un objeto que necesitan. Este esquema no está libre de *livelock*, ya que dos transacciones podrían indefinidamente abortarse la una a la otra por el acceso al mismo objeto.

Los STMs más recientes han demostrado que la sincronización no bloqueante puede llegar a obtener un rendimiento comparable a la bloqueante [61].

2.2.2. Gestión de versiones

Los sistemas de memoria transaccional necesitan de un mecanismo que les permita gestionar las escrituras especulativas realizadas por las transacciones durante su ejecución. A este mecanismo se conoce como ***gestión de versiones (VM)*** y admite básicamente dos políticas.

La primera política, se conoce como **gestión de versiones *eager*** o de *escritura directa* porque permite a las transacciones actualizar directamente los valores de los datos en memoria. Debido a que estas escrituras no son definitivas hasta que la transacción realice su *commit*, los valores antiguos de cada dato modificado deben ser almacenados en un *undo-log* a fin de poder restaurar el estado original de la memoria en caso de que la transacción aborte. Esta política hace rápido el proceso de *commit* debido a que los datos se encuentran escritos en memoria, sin embargo, penaliza los abortos que precisan de un proceso costoso para restaurar los valores originales. La gestión de versiones *eager* necesita de un control de concurrencia pesimista ya que las transacciones necesitan acceso exclusivo a los objetos en memoria si van a escribirlas directamente.

La segunda política es la **gestión de versiones *lazy*** o *escritura diferida*. En esta aproximación, las modificaciones en memoria son pospuestas hasta el *commit* de la transacción. Mientras tanto, las escrituras realizadas por la transacción son almacenadas en un *redo-log* (o *write buffer*) durante su ejecución, el cual también debe ser accedido por las lecturas (de la propia transacción) para recuperar los valores anteriormente escritos. Cuando la transacción alcanza su *commit*, las copias privadas almacenadas en su *redo-log* con actualizadas en memoria. Esto hace más costoso el *commit*, pero acelera el proceso de aborto, ya que solo requiere desechar los valores del *redo-log*.

2.2.3. Detección de conflictos

Una decisión de diseño clave en memoria transaccional es cómo detectar los conflictos. Cuando se utiliza un control de concurrencia optimista, la detección de conflictos viene impuesta porque el uso de *locks*, o algún otro tipo de mecanismo para obtener la exclusividad sobre un objeto, que generalmente solo permiten una única transacción propietaria a la vez. Así, si una transacción encuentra que el objeto compartido ya ha sido adquirido, puede determinar automáticamente que existe un conflicto. Por su parte, existe un amplio rango técnicas de detección de conflictos cuando se aplica un control de concurrencia optimista. En general, suele haber un proceso de *validación* en la cual, la transacción actual comprueba

si se han experimentado conflictos o no durante su ejecución.

La mayoría de aproximaciones para detección de conflictos pueden ser definidas en base a tres dimensiones ortogonales: granularidad, mecanismo y momento de la detección.

La primera dimensión, la ***granularidad***, determina la unidad mínima de almacenamiento sobre la cual el sistema transaccional es capaz de detectar conflictos. Muchas implementaciones de STM extienden un lenguaje orientado a objetos e implementan una granularidad a nivel de *objeto*. Esto permite detectar conflictos en el acceso a una misma instancia de una clase, aunque el acceso se realice a campos diferentes. Otros sistemas implementan una granularidad a nivel de *palabra* o *bloque* que permite detectar un conflicto en el acceso a una palabra o un grupo adyacente de palabras en memoria. La mayoría de propuestas HTM cuya detección de conflictos se realiza a través del protocolo de coherencia suelen detectar los conflictos a nivel de líneas de caché completas. A medida que aumenta la granularidad en la detección de conflictos se ve agravado el problema de los *falsos conflictos*[3] que suele estar ligado a perdida de concurrencia causada por abortos y esperas innecesarias.

La segunda dimensión comprende el ***mecanismo*** utilizado para detectar un conflicto en el acceso a un objeto compartido. Para ello es necesario marcar o anotar el objeto con el fin de conocer si ha sido accedido por una transacción. En la literatura se han propuesto varios mecanismo para tal propósito, siendo los más relevantes y utilizados:

- ***Lock***: Se utiliza un *lock* asociado a cada palabra en memoria, o un grupo de ellas (dependiendo de la granularidad), que es adquirido atómicamente por la transacción que accede al dato. Dado que los *locks* son estructuras que solo pueden ser adquiridas una vez antes de liberarse, una transacción puede determinar si existe un conflicto en el acceso a un objeto, si al intentar adquirir el *lock* falla porque ha ya sido tomado por otra transacción. Este es el mecanismo de detección de conflictos utilizado en la mayoría de implementaciones software.
- ***Bits en caché***: Es el mecanismo propuesto por la mayoría de HTMs que implementan una detección de conflictos utilizando el protocolo de cohe-

[3] Los falsos conflictos surgen cuando la unidad mínima a la que se detectan los conflictos es superior al de una palabra en memoria. Esto ocasiona que el elemento utilizado marcar los objetos como accedidos por una transacción sea común para un grupo de direcciones de memoria, lo que ocasiona que se detecten accesos como conflictivos, cuando en realidad no lo son. Este problema tiene por consecuencia la pérdida de rendimiento del sistema debido a abortos innecesarios de transacciones que podrían haber completado su ejecución satisfactoriamente.

rencia. Los bloques de caché se extienden para incluir dos nuevos bits *R* y *W* para marcarlos como accedidos transaccionalmente.

- ***Signatura o Filtro de Bloom***: Las direcciones de memoria son mapeadas a un array de locks por transacción, mediante el uso de varias funciones *hash*. Estos bits son activados cuando la transacción escribe la dirección y comprobados para determinar si existe un conflicto. Con los filtros de Bloom se puede asegurar la ausencia de conflictos, pero no su existencia, debido a que este sistema presenta un problema de falsos conflictos, ya que puede suceder que todos los bits mapeados por una dirección hayan sido activados por accesos a otras direcciones.

La tercera dimensión hace referencia al ***momento*** en el que se efectúa la detección del conflicto. Un conflicto puede ser detectado cuando una transacción realiza el primer intento de acceder al objeto. El término **detección de conflictos *eager*** es utilizado para referirse a esta política [69]. Una detección de conflictos *eager*, permite tomar medidas sobre una transacción nada más ocurrir el conflicto, lo que puede reducir significativamente la cantidad de computación perdida si la transacción finalmente aborta. Sin embargo, también es posible que propicie abortos en transacciones que podrían haber finalizado correctamente (figura 2.2). En este escenario, podemos ver como la transacción T_B causa que la transacción T_C aborte debido a una dependencia en el objeto X y finalmente termina abortando ella misma por un conflicto con la transacción T_A por el objeto Y. En este caso, la transacción T_C podría haber finalizado correctamente, ya que no presenta dependencias con la transacción T_A y la transacción T_B habría abortando igualmente.

Un conflicto también puede ser detectado durante la *validación* que es un proceso en el que se examina la colección de localizaciones de memoria leídas y escritas por la transacción para comprobar si otra transacción las ha modificado. La validación puede suceder en cualquier instante (incluso varias veces) a lo largo de la vida de la transacción.

Finalmente, un conflicto puede ser detectado en el instante de *commit*. Cuando la transacción finaliza su ejecución, se realiza una validación del conjunto completo de localizaciones para determinar si ha ocurrido un conflicto. El término **detección de conflictos *lazy*** se utiliza para definir esta política [69]. Una detección de conflictos en este punto de la ejecución puede resolver el problema ilustrado en la figura 2.2 gracias a que la detección del conflicto es postergada hasta que la transacción finaliza, en cuyo caso, la transacción T_C no encontraría que ha ocurrido un conflicto debido a que la transacción T_B ya habría sido abortada. Sin embargo, la cantidad de computación desperdiciada si la transacción

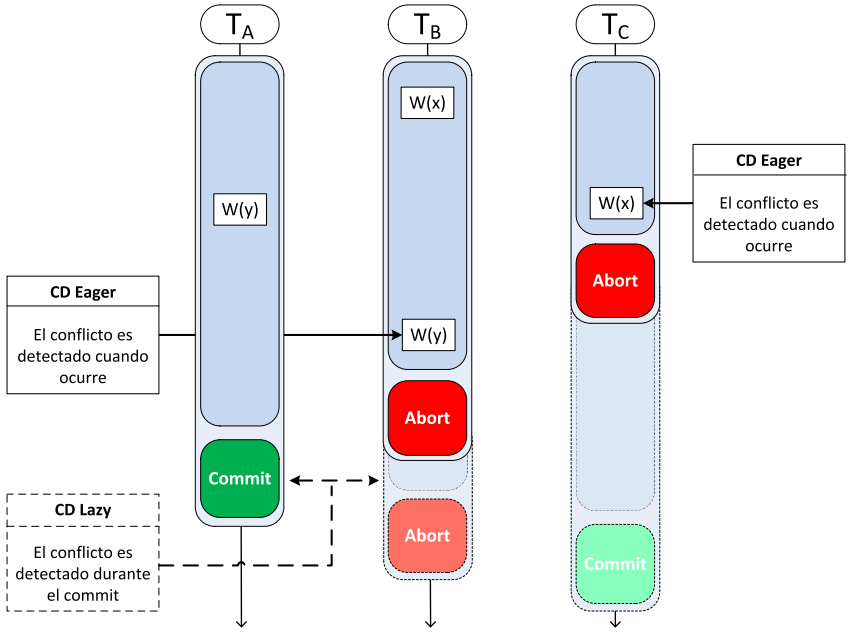


Figura 2.2: Problemática de la detección de conflictos.

finalmente aborta es máxima ya que la transacción completa su ejecución.

Un último aspecto relacionado con la detección de conflictos, aunque mucho más ligado al diseño de STMs, es el que determina la visibilidad de las lecturas en relación a otras transacciones. En general, las operaciones de lectura pueden ser *visibles* o *invisibles*. En el caso de las ***lecturas visibles***, las transacciones adquieren la propiedad sobre los objetos o localizaciones de memoria no solo durante la escritura, sino también durante la lectura. Esto permite a las transacciones escritoras detectar conflictos con transacciones lectoras, sin embargo, puede provocar un grave problema de contención cuando múltiples transacciones lectoras intentan acceder al mismo objeto. Por otro lado, con las ***lecturas invisibles***, las transacciones no anuncian sus lecturas al resto de transacciones, por lo que para asegurar su consistencia se debe recurrir a la validación del *read set* completo. En transacciones con grandes conjuntos de lectura, esto puede provocar una importante caída del rendimiento. Un estudio más profundo sobre las ventajas y desventajas de cada alternativa puede encontrarse en [90, 91].

2.2.4. Resolución de conflictos

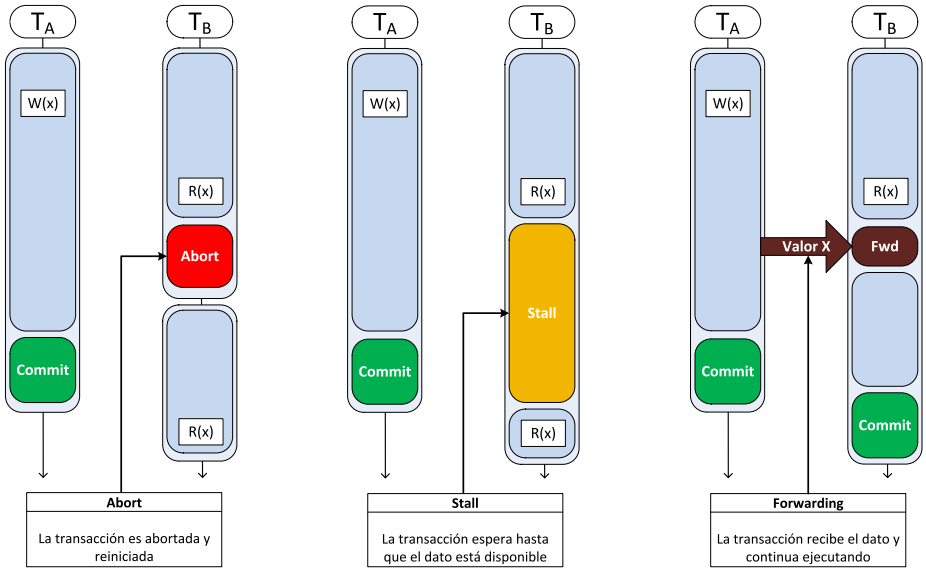
Una vez que el se ha detectado un conflicto en el acceso de dos transacciones a un mismo objeto, el sistema transaccional, a través de un *gestor de contención*, debe realizar las acciones pertinentes para resolverlo y evitar de un posible estado de inconsistencia. Tres acciones han sido consideradas históricamente en el ámbito de la memoria transaccional para resolver un conflicto:

- ***Aborto***: Es la aproximación por defecto, utilizada en la mayoría de HTM y STM existentes. Consiste en abortar todas aquellas transacciones presentes en el conflicto excepto una, la cual podrá progresar en su ejecución (figura 2.3a). El principal inconveniente de esta alternativa es que la cantidad de computación desperdiciada puede llegar a ser importante, sobretodo cuando el tamaño de la transacción es grande.

- ***Stall***: Consiste en permitir a una de las transacciones continuar ejecutando ante un conflicto, mientras el resto son detenidas a la espera de que la transacción ejecutando finalice satisfactoriamente y libere el aislamiento del objeto conflictivo. Esta aproximación permite reducir la cantidad de computación desechada, pero introduce un riesgo de *deadlock* debido a los ciclos, ya que dos transacciones podrían estar esperando a que la otra libere un objeto que necesitan (figura 2.3b).

- ***Forwarding***: Esta aproximación es utilizada en algunos sistemas [82] para resolver un conflicto de tipo *RAW* (*read after write*) permitiendo a una transacción lectora disponer de un valor escrito por otra transacción, antes de que esta realice el *commit*, para evitar que tenga que abortar o se vea detenida a la espera del objeto. Aunque el *forwarding* rompe con el aislamiento de las transacciones, puede propiciar un incremento de concurrencia significativo en un caso ideal, sin embargo, también crea dependencias entre transacciones, de manera que si una transacción que ha adelantado un valor aborta, todas aquellas que lo recibieron deben ser forzadas a abortar también (*abortos en cascada*), ya que el valor recibido ha quedado invalidado con el aborto de la transacción. Esta aproximación tampoco está libre de ciclos, por lo que será preciso gestionar el proceso de *forwarding* para evitar que se produzcan (figura 2.3c).

La decisión acerca de sobre qué transacción/es aplicar estas acciones (*abort, stall, forwarding*) es tomada en base a una o más *políticas de resolución de conflictos* y pueden afectar en gran medida al rendimiento del sistema. En la literatura se han



(a) Aborto (b) Stall (c) Forwarding

Figura 2.3: Diferentes métodos de resolución de conflictos.

propuesto numerosas y variadas políticas de resolución con diferente grado de dificultad y nivel de sofisticación. Entre las más destacadas podemos encontrar:

- ***Passive***: Es la política más simple, en la que la transacción que intenta adquirir el objeto es abortada.
- ***Polite***: La transacción que intenta adquirir el objeto efectúa un *stall* por un intervalo de tiempo. Tras cada intervalo, la transacción comprueba si la otra transacción ha liberado el objeto. Si no es así, el intervalo de espera se incrementa exponencialmente y se repite el proceso. Pasados un número de intentos sin éxito, la transacción aborta.
- ***Karma***: Esta política utiliza el número total de objetos en memoria accedidos por la transacción como una medida de la prioridad. Así, una transacción intentando adquirir un objeto compartido puede abortar inmediatamente cualquier otra transacción con prioridad inferior que lo posea. Sin embargo, si es la propia transacción la que tiene una prioridad menor, esta espera por un periodo de tiempo e intenta adquirirlo de nuevo, repitiendo este proceso N veces antes de abortar, donde N es la diferencia de

prioridades con la transacción que posee la propiedad sobre el objeto.

- ***Timestamp***: Esta política da prioridad a las transacciones más antiguas, abortando la transacción que ostenta la propiedad del objeto siempre que su instante de inicio sea posterior al de la transacción que intenta adquirirlo.

Otras muchas políticas (*eruption, greedy, kindergarten, polka ...*) han sido propuestas por diferentes sistemas [90, 91, 38, 39], sin embargo, ninguna ha demostrado ser la mejor en todas las posibles configuraciones, ya que su rendimiento dependerá de las características propias del sistema transaccional, de la carga de trabajo y del tipo de control de concurrencia utilizado.

2.2.5. Aislamiento

El concepto de aislamiento hace referencia a cómo interaccionan los accesos transaccionales a objetos entre sí y con los no transaccionales.

Se dice que existe un ***aislamiento débil (weak isolation)*** cuando solo se garantiza la propiedad de aislamiento entre secciones de código transaccional, pero no con secciones de código no transaccionales, las cuales pueden acceder a objetos escritos dentro de la transacción sin la protección del sistema transaccional. Los accesos efectuados fuera de una transacción pueden ocasionar conflictos con los objetos transaccionales, derivando en un comportamiento desconocido del sistema. Para evitar estos problemas, el programador debe proteger explícitamente estos accesos compartidos fuera de la transacción o utilizar barreras para separar y evitar que secciones de código transaccional y no transaccional ejecuten concurrentemente.

El ***aislamiento fuerte (strong isolation)*** garantiza el aislamiento de los objetos entre transacciones y con secciones de código no transaccional. Para ello, cada una de las operaciones fuera de un bloque transaccional se convierte en transacción individual que es gestionada por el sistema transaccional. Aunque el aislamiento fuerte previene de los comportamientos inesperados, no asegura la consistencia del resultado final, la cual quedará condicionada a la correcta delimitación de las fronteras de la transacción por parte del programador.

2.3. Sistemas de memoria transaccional

En esta sección se describen algunos de los sistemas transaccionales más relevantes de la literatura. Primero realizamos un repaso de los sistemas precursores y las principales propuestas tanto hardware como software. Seguidamente se presentan aquellos sistemas TM que imponen algún tipo de orden sobre la ejecución o finalización de sus transacciones.

2.3.1. Memoria transaccional hardware

La primera propuesta Hardware, así como el término *Memoria Transaccional*, fue presentada por Herlihy y Moss en 1993 [47]. Este HTM incluye tres características básicas:

- Un conjunto de instrucciones especial es puesto a disposición del programador para identificar aquellas localizaciones de memoria accedidas transaccionalmente (`load-transactional y store-transactional`). Estas instrucciones también ofrecen a los programadores la posibilidad de terminar explicitamente una transacción, haciendo los cambios visibles a otras transacciones (`commit`) o abortarla, desechando todos los cambios realizados (`abort`). Además, una instrucción `validate` comprueba si ha ocurrido un conflicto con el fin de abortar, ya que una transacción no aborta inmediatamente ante un conflicto y puede continuar ejecutando.
- Una caché transaccional dedicada, totalmente asociativa, en paralelo con la caché clásica que se encarga de almacenar todas las escrituras transaccionales sin propagar sus valores a otros procesadores o memoria principal hasta que la transacción realice el *commit* (gestión de versiones *lazy*). Esta caché dispone de un *tag* adicional que añade semántica a los estados clásicos de la caché. Entre los valores que puede tomar, podemos destacar: `empty` para reflejar que la línea se encuentra vacía; `normal` para una línea que contiene datos ya confirmados en memoria; `xcommit` y `xabort` para identificar líneas que deben ser descartadas ante un escenario de *commit* o *abort*, respectivamente.
- Una extensión al protocolo de coherencia caché convencional para detectar conflictos de datos con la nueva caché transaccional.

Además, se incluyen dos nuevos *flags* hardware para indicar si el procesador se encuentra dentro de una transacción activa (`TACTIVE`) y si ha recibido una

condición de aborto (`TSTATUS`). Esta propuesta no incluye ninguna instrucción para indicar explicitamente el inicio de una transacción, sino que será la primera instrucción transaccional ejecutada (mientras su *flag* `TACTIVE` se encuentra desactivado) la encargada de iniciar una transacción.

Virtual Transactional Memory (VTM) [81] extiende el primer trabajo de Herlihy para sobrepasar las limitaciones hardware. VTM mantiene la información de aquellas transacciones que exceden los recursos hardware en una tabla (*XADT*), mantenida en el espació de memoria virtual de la aplicación, así como todas las escrituras transaccionales realizadas por ella. Así, una transacción será movida a la tabla *XADT* si una de las líneas transaccionales es desalojada de la caché o si la transacción es suspendida y todas sus líneas de caché deben ser desalojadas. En cualquier caso, al existir líneas de caché desalojadas almacenadas externamente, se precisa de algún mecanismo capaz de comprobar conflictos con las entradas en dicha tabla. Para ello, VTM proporciona a cada procesador la capacidad de invocar micro códigos o rutinas, que se manera selectiva interrumpen las operaciones de lectura y escritura de manera transparente al usuario, y realizan las comprobaciones pertinentes. Para reducir el coste de la búsqueda de conflictos, VTM utiliza un filtro de conflictos software, llamado *XADT Filter* (*XF*), implementado como un filtro de Bloom [12]. Un fallo en el filtro garantiza que no existe conflicto, mientras que un acierto necesitará de un recorrido completo de la tabla para garantizar que es un conflicto real o solo un falso positivo.

Large Transactional Memory (LTM) [2] es otra propuesta que resuelve el problema de los desalojos de líneas transaccionales de la caché, sin abortar la transacción. Haciendo uso de un política de gestión de versiones *lazy* y una detección de conflictos *eager*, la propuesta de este sistema consiste en incluir un bit de *overflow* (*O* bit) que se activa cuando una línea de caché es movida fuera de esta. Para evitar perder la información y tener que abortar la transacción, las líneas desalojadas de la caché son movidas a una tabla *hash* almacenada en memoria local. Durante este proceso, todas las peticiones realizadas al procesador serán respondidas con una negativa (*NACK*). En el caso de que una petición entrante acierte en una línea de caché con el bit de *overflow* activado, el procesador deberá recorrer la tabla *hash* en busca del *tag* correcto. Si lo encuentra, el procesador intercambia la línea de caché con la entrada de la tabla *hash*. En caso contrario, el procesador lo trata como un fallo convencional.

LogTM [69] es un HTM desarrollado con la idea de hacer rápido el caso más común, es decir, el *commit*. Para ello se propone el uso de un *log* por cada hilo, que se encarga de almacenar los valores antiguos de los objetos modificados, mientras que los valores nuevos se almacenan en memoria (gestión de versiones *eager*). Esto favorece el caso más común, ya que un *commit* no requiere acciones

adicionales, más allá de descartar los valores del *log* y limpiar los bits de lectura y escritura de caché, usados para la detección de conflictos transaccionales. En caso de aborto, un manejador software se encarga de recorrer el *log* y restaurar los valores antiguos en memoria. Aunque este proceso pueda llegar a ser poco eficiente, se considera que los abortos son suficientemente inusuales como para que este coste perjudique el desempeño del sistema. Aunque, como ocurre en otras propuestas hardware, los conflictos son detectados gracias al protocolo de coherencia (en este caso, basado en directorio) y a los bits de lectura/escritura en caché, LogTM extiende su protocolo de coherencia añadiendo un nuevo estado (*sticky-M*) que permite detectar conflictos con bloques desalojados de caché, permitiendo así exceder las limitaciones hardware. Una evolución de esta propuesta es *LogTM Signature Edition (LogTM-SE)* [100] que desacopla la detección de conflictos de la caché, adoptando el uso de firmas (*signatures*) basadas en filtros de Bloom para trazar los accesos de las transacciones y detectar los conflictos de manera mucho más eficiente.

La propuesta de AMD para memoria transaccional *Advanced Synchronization Facility (ASF)* [19] que se engloba en el grupo de los sistemas TM implícitos. ASF proporciona al programador un par de instrucciones `speculate` y `commit` para marcar el inicio y final de la transacción respectivamente. Una vez delimitadas las fronteras de la transacción, se utiliza un prefijo `lock` para marcar los accesos a memoria que deben ser tratados transaccionalmente. En caso de aborto, ASF recupera mediante hardware el puntero a la pila, mientras que delega en un manejador software la recuperación del resto de los registros.

Intel en su *Transactional Synchronization Extensions (TSX)* [85] presenta dos interfaces para definir código transaccional: *Hardware Lock Elision (HLE)* y *Restricted Transactional Memory (RTM)*. La primera es compatible con el modelo convencional de programación con *locks*, y para ello presenta dos prefijos para instrucciones (`XACQUIRE` y `XRELEASE`). De esta manera, el código escrito con HLE puede utilizar el soporte tradicional (*locks*) o el soporte transaccional partiendo del mismo código. En caso de activar el soporte TSX, ambos prefijos definirán las fronteras de la transacción. En caso de aborto, las instrucciones vuelven a ejecutarse, pero esta vez de manera no transaccional. Por su parte, RTM permite una mayor flexibilidad en la definición de las transacciones que HLE. Esta interfaz añade tres nuevas instrucciones al ISA: `XBEGIN`, `XEND` y `XABORT`. Aunque Intel no ha proporcionado detalles acerca de su implementación, todo hace indicar que TSX sigue una aproximación *best effort*, que consiste en aborta una transacción cuando esta excede los recursos hardware o encuentra determinados eventos, tales como fallos de página, fallos de cachés o interrupciones.

2.3.2. Memoria transaccional software

Los STM son sistemas que implementan toda la semántica necesaria para manejar transacciones cumpliendo con las propiedades ACI para la gestión concurrente de objetos compartidos. Esta semántica incluye preservar tanto la atomicidad como el aislamiento, garantizar la consistencia de las lecturas transaccionales, así como realizar la detección y resolución de conflictos. Todo ello, unido a la expansión software de las operaciones de lecturas y escrituras transaccionales, que añaden decenas de instrucciones adicionales, suponen un *overhead* que clásicamente han limitado a estos sistemas.

No obstante, el rendimiento de los STMs más actuales, particularmente aquellos integrados con optimizaciones de compilador, ha alcanzado niveles que los hace, no solo un vehículo razonable para la experimentación y el prototipado, sino también una alternativa sencilla, real y al alcance de todos los programadores para desarrollar aplicaciones paralelas.

Los STMs ofrecen varias ventajas sobre los HTM:

- Son más flexibles que los sistemas hardware, lo que permite a los desarrolladores implementar una amplia variedad de sofisticados algoritmos para mejorar el rendimiento del sistema, así como para proporcionar mecanismos de resolución de conflictos más complejos que un mero aborto de transacción.
- Los sistemas software son más sencillos de modificar y hacer evolucionar que los sistemas hardware, debido a que cada cambio o mejora puede implementarse rápidamente y eludiendo los costes asociados a los procesos de fabricación, ensamblaje y/o sustitución.
- Para los usuario finales, las aproximaciones STM ofrecen un entorno para obtener versiones transaccionales a partir de sistemas software ya existentes de manera fácil, intuitiva e integrando estos beneficios con los lenguajes de programación actuales y sus principales abstracciones.
- Los sistemas software no tienen las limitaciones intrínsecas impuestas por el tamaño fijo de las estructuras hardware, tales como cachés, *buffers* o *logs*.

Muchos conceptos y principios de implementación utilizados en los STMs actuales ya fueron anticipados por Lomet en 1977 [59]. En su trabajo, Lomet estudió las desventajas y defectos de los mecanismos clásicos de sincronización utilizados hasta la fecha, tales como semáforos, secciones críticas o monitores.

Lomet notó que los programadores usaban dichas estructuras principalmente para mantener la consistencia al ejecutar bloques atómicos de código que accedían a variables compartidas y relacionó esta necesidad con el concepto de transacción presente en bases de datos, proporcionando una sintaxis para nuevo procedimiento atómico:

```
<identificador> : acción(<lista-parámetros>);
<lista-instrucciones>
fin;
```

Este procedimiento se ejecutaba en aislamiento respecto a otras ejecuciones concurrentes, anotando los objetos compartidos para coordinar su acceso. Lomet también introdujo una operación de *reset* que abortaba los procedimientos en ejecución y restauraba el estado del programa.

Pese a esta primera idea, no sería hasta 1995 cuando Shavit y Touitou [93] acuñarían por primera vez el término *Software Transactional Memory* (STM), en la que sería la primera implementación software de un sistema transaccional práctica y competitiva hasta la fecha. Esta propuesta, englobada dentro del conjunto de implementaciones no-bloqueantes, requería que el programador identificase, a priori, aquellos objetos de memoria que podrían ser accedidos dentro de una transacción, para permitir al sistema adquirir la propiedad sobre todas ellos. Si lo consigue, la transacción almacena los valores antiguos y ejecuta el cuerpo de la transacción, para finalmente actualizar la memoria con los nuevos valores y liberar la propiedad sobre los objetos.

Este sistema, aunque incurre en un *overhead* significativo, ofrece una fuerte garantía de progreso ya que una transacción que adquiere la propiedad sobre las localizaciones que accede, puede completar sin posibilidad de *rollback*. Si la transacción falla al intentar adquirir la propiedad de alguna de las localizaciones, el procedimiento de *helping* se pondrá en marcha e intentará ejecutar la transacción que posee la propiedad sobre estos objetos, bajo la asunción de que el hilo ejecutando dicha transacción ha sido desplanificado.

Otros STM han seguido la senda de Shavit y Touitou, explorando técnicas que implementan algoritmos no bloqueantes, tales como Herlihy's Dynamic STM (DSTM), Object-based STM (OSTM) o Word-based STM (WSTM). DSTM fue la primera implementación de STM que no requería que el programador especificase, de antemano, las localizaciones que se iban a acceder, además de introducir el uso de un módulo de gestión de contención explícito. Implementado en Java, esta aproximación requería que todos los objetos manipulados dentro de una transacción fueran instancias de una clase `TMObject` que propor-

cionaba la semántica transaccional. Así, los objetos transaccionales quedaban implementados con dos niveles de indirección.

El sistema utiliza una instancia a un `TMObject` para identificar un localizador único, asociado a cada objeto del programa, que a su vez posee apuntadores hacia: la última transacción que abrió el objeto para escritura, el antiguo valor del objeto y el nuevo valor del objeto. Además, dado que cada transacción dispone de un descriptor que indica el estado actual de la transacción (`ACTIVE, COMMITTED, ABORTED`), el estado lógico de un objeto puede ser determinado fácilmente de la siguiente manera:

- Si la última transacción escritora se encuentra en estado `COMMITTED`, el apuntador al nuevo valor corresponde con el estado lógico del objeto.
- Si la última transacción escritora se encuentra en estado `ACTIVE o ABORTED`, el estado lógico del objeto se corresponderá con el apuntador al valor antiguo.

Si bien DSTM proponía una política de detección de conflictos *eager*, OSTM ofrecía una propuesta muy similar, con una detección de conflictos *lazy*, donde la instancia de un `TMObject` enlaza con un descriptor de transacción, en lugar de con un localizador. Harris y Fraser presentaron un Word-based STM sin indirecciones en el que los datos son almacenados de manera normal en la pila, utilizando una tabla separada para mantener los metadatos y cuyas entradas están asociadas con la pila a través de una función *hash*. WSTM utiliza detección de conflictos *lazy* y lecturas invisibles.

Pese a que los sistemas no-bloqueantes presentan características interesantes, han sido las propuestas bloqueantes o *lock-based* STM los que han demostrado, sobretodo en los últimos años, obtener un mejor rendimiento. En estas propuestas, una estructura de exclusión mutua (*lock*) es asociada a cada dirección de memoria o rango de ellas (dependiendo de la granularidad) para permitir garantizar la atomicidad. El momento en que se adquieren dichos *locks* puede afectar no solo al rendimiento general del sistema, sino también limitar algunas de las decisiones de diseño expuestas en la Sección 2.2. En la literatura se presentan dos algoritmos para la adquisición de *locks*.

En el algoritmo *Encounter-time locking (ETL)*, los *locks* se adquieren cuando la transacción realiza la primera escritura en una dirección de memoria. ETL permite una gestión de versiones tanto *eager* como *lazy*, sin embargo, impone una restricción a la política de detección de conflictos, la cual solo podrá ser *eager*, ya que un acceso conflictivo se detectará en el momento en que una transacción

intente adquirir un *lock* ya tomado. Aunque la adquisición de *locks* de escritura resuelve conflictos entre escrituras concurrentes, no previenen de posibles conflictos de lectura-escritura que se puedan producir durante la ejecución de las transacciones (*invisible reads*)[4].

Para detectar conflictos con transacciones lectoras, un número de versión local debe ser incluido en los metadatos asociados al objeto de memoria a fin de verificar, en una fase de validación, si el dato leído por una transacción ha sido modificado por otra transacción. Ambos conceptos (*lock* y versión) pueden ser combinados dando lugar a lo que se conoce como *versioned lock*. Básicamente, si el *lock* está libre significa que ninguna transacción tiene escrituras pendientes sobre la dirección de memoria cubierta por el *lock*, en cuyo caso el *versioned lock* contiene la versión actual. Esta versión es incrementada cada vez que una transacción libera el *lock*, indicando que el objeto ha sido modificado. Así, una transacción al realizar una lectura, almacena en su *read set* la dirección de memoria y el número de versión del *lock* que la protege. Al realizar el *commit*, una fase de validación verifica que las versiones en el *read set* coinciden con las versiones actuales de los *locks*. Si no es así, significa que alguna otra transacción ha escrito la dirección después de haberla leído, por lo que el conflicto lectura-escritura es detectado y resuelto. McRT-STM y Bartok-STM son dos propuestas que implementan esta aproximación.

Por otro lado, el algoritmo *commit-time locking (CTL)* propone adquirir los *locks* en el instante de *commit* para realizar las actualizaciones en memoria de manera aislada y seguidamente liberarlos. Esta aproximación impone una gestión de conflictos *lazy* ya que una transacción no puede actualizar la memoria sin antes adquirir el *lock*. TL2 [22] hace uso de esta aproximación utilizando, al igual que los STMs descritos en el párrafo anterior, el concepto de *versioned locks* solo que la identificación de versiones se realiza ahora utilizando un reloj global que se incrementa cada vez que una transacción efectúa su *commit*. De este modo, cada transacción comienza leyendo el reloj global y almacena esa marca de tiempo como su *versión de lectura*. En cada lectura, la transacción comprueba si la versión del objeto es más actual que su *versión de lectura* (instante de inicio), abortando en caso positivo.

TinySTM [26, 25] propone combinar una aproximación similar a TL2 con una forma de bloqueo jerárquico. Un array de *locks*, cada uno de los cuales

[4]Existen varios métodos para tratar con lecturas invisibles, además de una fase de validación. Algunas configuraciones de McRT-STM y RSTM permiten dispensar lecturas invisibles, pero a cambio mantienen su *read set* visible a otras transacciones. Otros trabajos como [58, 57] introducen la idea de lecturas *semi-visibles* de manera que una transacción escritora pueda detectar que el dato esta siendo leído por otra transacción actualmente en ejecución, pero no le permite saber su identidad

cubre un conjunto de direcciones de memoria adyacentes y etiquetas de tiempo basadas en un reloj global, son utilizadas para dotar al sistema de opacidad. Este STM implementa soporte tanto para ETL como CTL, por lo que se puede configurar para funcionar de acuerdo a las principales políticas de gestión de versiones y detección de conflictos. Dado que este sistema es utilizado como base de implementación para los aportes de esta tesis, una descripción mas detallada será presentada en el Capítulo 3.

2.3.3. Sistemas con orden

Hasta ahora hemos analizado los principales sistemas presentes en la literatura, de acuerdo al concepto original de TM, el cual no considera orden entre transacciones. Sin embargo, el orden puede llegar a ser una poderosa herramienta para mejorar algunos aspectos relacionados con la ejecución de las transacciones.

Así pues, en esta sección revisaremos las principales implementaciones TM presentes en la literatura, tanto *software* como *hardware*, que consideran información acerca de la precedencia entre las transacciones presentes en el sistema, con el fin de ordenarlas de acuerdo a algún determinado de criterio. La Tabla 2.1 resume las principales características de los sistemas expuestos.

TCC

La propuesta de *Transactional Coherence and Consistency (TCC)* [43] define un nuevo modelo de coherencia y consistencia en el cual, las transacciones son la unidad básica de trabajo paralelo. Las escrituras transaccionales no requieren solicitudes de coherencia porque son almacenadas en un *write buffer* hasta que finalice la transacción (gestión de versiones *lazy*). Un bit de lectura y otro de escritura son activados por cada línea de caché local que contenga datos leídos o escritos transaccionalmente y que se utilizan para la detección de conflictos.

Al alcanzar su instante de commit, cada transacción difunde un paquete con todas las direcciones modificadas al resto de procesadores (*broadcast*) para que estos comprueben si se produjeron conflictos (detección de conflictos *lazy*). Si han leído algún dato modificado por la transacción que realiza el *commit*, entonces deben abortar.

Un orden entre transacciones puede ser especificado opcionalmente por los programadores gracias a los denominados *Números de Fase*. Transacciones con un mismo número de fase puede efectuar el *commit* en cualquier orden entre ellas.

Transacciones con número de fase menor deben realizar el *commit* antes que todas aquellas con número de fase mayor, las cuales son forzadas a detenerse y esperar. Ante esta situación, una mejora conocida como *Double Buffering* permite duplicar las estructuras y bits en la caché para permitir al procesador lanzar una segunda transacción cuando los números de fase impiden efectuar el *commit* de alguna transacción.

DATM

Dependence-Aware Transaction Memory (DATM) [82] explora el concepto de *serialización de conflictos* con el fin de evitar abortos innecesarios. La serialización de conflictos es un mecanismo que consiste básicamente en reordenar las etapas de *commit* de las transacciones conflictivas para evitar una inconsistencia en memoria. La idea detrás de DATM es aceptar cualquier orden de *commit* que evite un conflicto, por lo que la ordenación de las transacciones dependerá del comportamiento dinámico del sistema. Además, también ofrece la posibilidad de realizar *data forwarding* entre transacciones, para resolver determinados conflictos que no pueden evitarse únicamente con el orden.

Todas las dependencias se trazan a nivel de línea de caché cuando ocurren (detección de conflictos *eager*), mientras que las escrituras son mantenidas en los niveles de caché local hasta que se efectúe el *commit* (gestión de versiones *lazy*). En DATM, todos los conflictos restringen el orden de *commit* de las transacciones involucradas, así detectar una dependencia $T_A \rightarrow T_B$ implica que T_A deba realizar el *commit* antes que T_B para evitar el conflicto.

El *data forwarding* introduce dependencias adicionales, ya que si una transacción adelanta un dato a otra, la transacción receptora se vuelve dependiente de ella y por tanto el sistema debe retrasar su *commit* hasta que la transacción fuente finalice. Si se produjera un aborto de la transacción fuente, todas aquellas transacciones que hubieran recibido un dato adelantado por ella se verían obligados a abortar también, causando lo que se conoce como *abortos en cascada*. El sistema también previene los posibles ciclos debidos al *forwarding* abortando al menos una de las transacciones involucradas.

STMlite

STMlite [63] es un sistema de memoria transaccional software, que implementa una política *lazy-lazy* (detección de conflictos - gestión de versiones), pensado para facilitar la paralelización automática de bucles incurriendo en un bajo over-

head. La principal característica de STMlite es la asignación de un hilo dedicado a gestionar los *commits* de las transacciones presentes en el sistema. Este hilo dedicado, conocido como *Transaction Commit Manager (TCM)* actúa como coordinador central para el resto de hilos trabajadores cuando estos solicitan efectuar el *commit* de sus transacciones.

Además del TCM, dos estructuras de datos son necesarias para el funcionamiento del sistema. Un *Pre-Commit Log* donde se almacena la información de las transacciones en espera de realizar su *commit*, y un *Commit Log* que mantiene la información de aquellas transacciones que ya han finalizado.

La idea detrás de STMlite es que cada transacción se encargue de gestionar sus propios *read set* y *write set* durante su ejecución concurrente. Al realizar el *commit*, cada transacción copia estos conjuntos al *Pre-Commit Log* y bloquea su hilo a la espera de una notificación del TCM. A partir de este momento, es el TCM el encargado de recorrer las entradas del *Pre-Commit Log* y comprobar si los conjuntos de lectura de cada transacción candidata a efectuar el commit entra en conflicto con los conjuntos de escritura de alguna de las transacciones presentes en el *Commit Log*.

Dado que solo aquellas transacciones presentes *Commit Log* cuya ejecución se solapó en algún momento con la candidata a efectuar el commit son de interés, cada entrada incluye además un número de versión para indicar el instante de inicio S_{ver} (*Pre-Commit Log*) y el instante de commit C_{ver}(*Commit Log*) de cada transacción. De manera que si $S_{ver} < C_{ver}$ sabemos que la ejecución se solapó en algún instante. Si no se detectan colisiones de conjuntos, se determina que ninguna transacción concurrente con ella y ya finalizada, escribió en direcciones de memoria comunes, por lo tanto el TCM concede permiso al hilo para efectuar el commit y actualizar sus escrituras en memoria. En caso contrario, el TCM informa al hilo que debe abortar la transacción.

Usando este mecanismo STMlite permite la paralelización de bucles con dependencias entre iteraciones. El TCM se encarga de asegurar el orden correcto entre las transacciones que ejecutan bloques de iteraciones. Para ello se añade una nueva estructura denominada *Loop Chunck Commit Log (LCCL)* mantenida por el TCM y que contiene un identificador del último bucle ejecutado (*loop ID*) y un identificador del último bloque de dicho bucle ejecutado (*chunk ID*). Usando esta información, el TCM se encarga de permitir el commit solo a la transacción que contenga el siguiente bloque de iteraciones en orden secuencial de programa.

SONTM

SONTM [4] es un HTM en el que el orden no viene impuesto por el programador o por el flujo secuencial del programa, sino por los conflictos que ocurren durante la ejecución del mismo. Al igual que DATM, haciendo uso del concepto de serialización de conflictos, el sistema traza las dependencias de datos a medida que ocurren y reordena los *commits* de las transacciones para evitar que estos produzcan un aborto.

Para ello, se opta por una política de detección de conflictos *eager*, utilizando los mensajes de coherencia caché para registrar los accesos conflictivos y una gestión de versiones *lazy*, el cual proporciona aislamiento entre transacciones hasta el instante de *commit*. Para determinar el orden de las transacciones, se define un algoritmo de *Serializabilidad de Conflictos (CS)* más relajado que la aproximación 2PL. En él, se considera que dos esquemas de *commit* presentan una *Equivalencia de Conflictos* si contienen exactamente los mismos accesos y el orden de las acciones conflictivas es el mismo. Así pues, se dice que un esquema cumple con la *Serializabilidad de Conflictos* si presenta una equivalencia de conflictos con algún esquema secuencial.

Para llevar esto a la práctica, se utilizan unos *Serializability Order Numbers (SON)* que son asignados en el instante de *commit* cuando todos los conflictos de la transacción han sido trazados, y cuyo valor se determinan utilizando dos reglas básicas:

- **Forward-serialization:** Si una transacción T_A lee o intenta efectuar el commit de una dirección que ya ha sido comitada por otra transacción T_B, entonces el $SON(T_A) > SON(T_B)$.

- **Backward-serialization:** Si T_B intenta efectuar el commit de una dirección que ya ha sido leída por otra transacción T_A, entonces el $SON(T_A) < SON(T_B)$.

Si en algún momento, una transacción no puede ser situada en un esquema de conflictos equivalente a alguno secuencial, entonces la transacción aborta. Si la transacción completa su ejecución sin abortar, significa que se encuentra dentro de algún esquema de conflictos válido, por lo que tiene permitido efectuar el *commit* cuando llegue su turno.

SoC-TM

SoC-TM [27] es un soporte HW/SW integrado para memoria transaccional en MPSoCs empotrados (*embedded*). En él, se propone desacoplar el sistema de memoria transaccional del sistema de coherencia caché. Para ello, se utiliza un módulo hardware externo basado en firmas, denominado *Módulo Bloom*, que se encarga de monitorizar todos los accesos transaccionales, salvarlos en unas firmas asociadas a cada procesador y notificar a las CPUs en caso de conflicto de datos.

El *Módulo Bloom* proporciona una detección de conflictos *eager*, mientras que las modificaciones realizadas por la transacción se mantienen en la caché local hasta que se realice el *commit*, manteniendo así una política de gestión de versiones es *lazy*. Además de actuar en ambientes clásicos de memoria transaccional, el sistema ha sido extendido para permitir paralelización especulativa de bucles, forzando que las transacciones realicen su *commit* respetando la semántica secuencial del programa.

SoC-TM soporta dos tipos de planificación: dinámica y estática. En la planificación dinámica, el planificador asigna transacciones a los núcleos de procesamiento siguiendo el orden de programa original, el cual establece la prioridad entre las transacciones.

Un buffer circular con N ranuras (cada una de las cuales representa un nivel de prioridad) es usado para mantener el orden, junto con dos punteros a la siguiente ranura disponible y a la siguiente ranura que debe realizar el commit. Así, cada vez que un procesador solicite una nueva transacción al *Módulo Bloom*, este inserta su ID en la siguiente ranura disponible del buffer. Solo el procesador cuyo ID esté almacenado en la ranura apuntada para efectuar el *commit*, tiene permiso para hacerlo. El resto, se ven forzados a efectuar un *stall* y esperar su turno. En la planificación estática, cada procesador calcula localmente el ID de la siguiente transacción a ejecutar (basándose en el número total de procesadores y su propio ID) y se lo comunica al *Módulo Bloom* que lo anota en un registro asociado al procesador. Otro registro se encarga de almacenar el ID de la última transacción que realizó el *commit*, de manera que el *Módulo Bloom* solo concederá permiso para finalizar a la siguiente transacción de acuerdo al orden secuencial.

BulkSMT

En [78] se presenta *BulkSMT* como el primer diseño SMT que soporta ejecución continua de transacciones, haciendo uso de sus múltiples contextos hard-

ware para aprovechar la proximidad de las estructuras hardware compartidas, tales como cachés y buffers. Gracias a ello, el hardware puede detectar rápida y eficientemente los conflictos para registrarlos y gestionarlos de la mejor manera posible para maximizar la concurrencia.

BulSMT propone dos modos de ejecución de transacciones: en un procesador SMT de *core* único o *multicore*. Para ambas propuestas se consideran unas políticas de detección de conflictos y gestión de versiones *eager-eager*. La detención de conflictos se efectúa utilizando bits de acceso en la caché. La principal diferencia entre ambos modos de ejecución se encuentra en el mecanismo de gestión de versiones. En procesadores SMT de núcleo único, las escrituras especulativas realizadas en caché son automáticamente visibles al resto de contextos hardware. Además, no es necesario almacenar los valores antiguos, ya que el sistema evita que las escrituras transaccionales pasen más allá de la caché L2, por lo que los valores antiguos de los datos siempre pueden leerse de memoria.

Diferente es el caso del modo de ejecución en procesadores SMT *multicore*, en el que se si actualiza la memoria con las escrituras transaccionales para hacer visible sus cambios. Por tanto, se requiere el uso de *Undo Logs* hardware para mantener los valores antiguos (uno para cada contexto hardware).

El orden en esta propuesta se incluye como parte de la política de resolución de conflictos. Además de permitir abortar o detener una transacción (hasta que el dato deseado esté disponible), BulkSTM permite registrar el tipo y dirección de las dependencias, de manera que forzando el commit de las transacciones en el mismo orden, se consigue evitar el conflicto y posterior aborto, siguiendo la misma idea de serialización de conflictos introducida en DATM o SONTM.

TLSTM

TLSTM [9] presenta un modelo unificado de memoria transaccional y *TLS (Thread-Level Speculation*), para extraer paralelismo y acelerar aplicaciones. El soporte transaccional para manejar las transacciones (definidas por el usuario) está a cargo de SwissTM [23], una conocida implementación software. Estas transacciones de usuario son divididas automáticamente (en tiempo de compilación o de ejecución) en tareas especulativas gestionadas por el soporte TLS.

Las tareas que conforman cada transacción de usuario pueden ejecutar fuera de orden de forma especulativa, sin embargo los accesos realizados en ellas deben comportarse como si hubieran ejecutado secuencialmente (escrituras de $Task_i$ deben ser vistas por cualquier tarea $Task_n$ siempre que $i < n$ en orden secuencial de programa, pero no al contrario). Para ello, a cada tarea se asigna un

número de serie único que representa su posición en orden de programa dentro de cada transacción de usuario. Solo cuando todas sus tareas se han completado en orden de programa, la transacción de usuario tiene permitido realizar su commit y hacer visibles todas sus escrituras (gestión de versiones *lazy*). Esto puede suceder en cualquier instante ya que el sistema no debe mantener orden entre transacciones de usuario, sino solo entre tareas especulativas dentro de cada transacción. La última tarea en orden de programa, conocida como *commit-task*, se encarga de verificar que todas las tareas precedentes han completado con éxito y de actualizar la memoria con las escrituras especulativas de todas ellas.

Este sistema requiere de una detección y resolución de conflictos en dos niveles. Los conflictos entre tareas dentro de una misma transacción de usuario pueden resolverse abortando únicamente la tarea posterior en orden de programa. Conflictos entre transacciones de usuario se resuelven abortando la más especulativa, lo que implica abortar todas sus tareas especulativas. De nuevo la última tarea en orden de programa es la encargada de liberar los *locks* de escritura y reiniciar todas las tareas precedentes.

Tabla 2.1: Sistemas TM que soportan transacciones ordenadas

	TCC	**DATM**	**STMLite**	**SONTM**	**SoC-TM**	**BulkSMT**	**TLSTM**
Aproximación:	Hardware	Hardware	Software	Hardware	Hardware	Hardware	Software
Detección de conflictos:	Lazy	Eager	Lazy	Eager	Eager	Eager	Eager
Gestión de versiones:	Lazy	Lazy	Lazy	Lazy	Lazy	Eager	Lazy
Escrituras transaccionales:	Caché local	Caché local	WriteBuffer	Caché local	Caché local	Caché compartida	WriteBuffer
Resolución de conflictos[†]**:**	A	A, F, O	A	A, O	A	A, S, O	A
Orden explícito[‡]**:**	Si	No	Si	No	Si	No	No
Orden de *commit:**	Estricto	Relajado	Estricto	Relajado	Estricto	Relajado	Relajado
#Transacciones activas[◦]**:** (por hilo)	Una (defecto) Dos (double buffer)	Una	Una	Una	Una	Una	Una
Arbitraje de *commit*:	-	-	Centralizado	Distribuido	Centralizado	Distribuido	-

[†]A=Abort, S=Stall, F=Forwarding, O=Forzar orden de commit.
[◦]Transacciones en ejecución o ejecutadas pero aún sin realizar el commit.
[‡]Si, en caso de que el orden de las transacciones pueda ser definido explícitamente; en caso contrario se utiliza como mecanismo para mantener la corrección.
*Estricto: el orden de precedencia es estrictamente mantenido en el momento de commit; Relajado: las transacciones pueden realizar su commit fuera de orden cuando los resultados son equivalentes a los derivados del orden de precedencia.

3 Metodología

En este capítulo se describe la metodología seguida para evaluar las propuestas presentadas en esta tesis. El sistema TM software utilizado como base para la implementación de las distintas aportaciones se describe en la sección 3.1. Seguidamente, presentamos un resumen del conjunto de aplicaciones de evaluación (sección 3.2) y describimos las principales métricas utilizadas para la evaluación de rendimiento.

3.1. Sistema base TinySTM

Todas las aproximaciones y técnicas propuestas en esta tesis han sido desarrolladas sobre un conocido sistema TM software (TinySTM [26, 25]), que presenta una implementación STM ligera con el soporte básico para la detección de conflictos y la gestión de versiones de datos. Este sistema transaccional software, como la mayoría de STMs con granularidad de palabra, utilizan un array de *locks* de grano fino para gestionar los accesos concurrentes a memoria, mientras que se utiliza una aproximación basada en marcas de tiempo (*time-based*) para garantizar la consistencia transaccional. Todo el sistema está programado en lenguaje ANSI C, mientras que para las operaciones atómicas y barreras de memoria se utiliza la librería *atomic_ops* de *Hans Boehm's*, la cual implementa una versión optimizada de estos mecanismos para arquitecturas tanto de 32 como 64 bits.

TinySTM presenta una gran variedad de configuraciones que cubren la mayoría de las decisiones de diseño para un sistema transaccional, entre las que

podemos destacar la elección del instante en que se adquieren los *locks*, el instante en que se realizan las actualizaciones en memoria, la política de contención o la visibilidad de las escrituras.

Las configuraciones CTL (*commit-time locking*) y ETL (*encounter-time locking*) permiten definir el tipo de control de concurrencia del sistema transaccional (optimista y pesimista respectivamente).

- *CTL:* Propone adquirir los *locks* asociados a las direcciones modificadas por la transacción durante la fase de *commit*. Esto impide que se detecten los conflictos hasta este punto de la ejecución transaccional. Así pues, esta aproximación asume una política de detección de conflictos *lazy*. Aunque la configuración CTL puede llegar a resolver conflictos en determinadas situaciones (ver sección 2.2.3), la detección tardía de los conflictos por lo general suele implicar una gran cantidad de computación desechada en aplicaciones con alta contención.

- *ETL:* Esta aproximación permite aplicar una política de detección de conflictos *eager*, gracias a que los *locks* sobre las direcciones modificadas se adquiere al realizar la operación de escritura. Este control de concurrencia pesimista posibilita detectar y resolver los conflictos tan pronto ocurren, minimizando así la cantidad de trabajo desechado en caso de conflicto, sin embargo, también puede provocar abortos innecesarios que podrían haberse resuelto por sí mismos llegado momento de *commit*. En esta configuración, TinySTM permite seleccionar entre dos diferentes estrategias para la actualización de datos en memoria: escritura directa (write-through) o diferida (write-back). La diferencia entre ambas, así como sus ventajas e inconvenientes han sido descritas en la sección 2.2, y su elección definirá la política de gestión de versiones del sistema transaccional (*eager* o *lazy* respectivamente).

En cuanto a política de contención, TinySTM ofrece varias estrategias:

- *SUICIDE:* Aborta inmediatamente la transacción que detecta el conflicto.

- *DELAY:* Similar a *SUICIDE* pero retrasando el reinicio de la transacción hasta que la dirección que provoca el conflicto sea liberada. La idea es evitar que la transacción reiniciada, vuelva a ser aborta por no haber sido liberado aún el dato compartido (típico con transacciones grandes y alta contención).

- *BACKOFF:* Similar a *SUICIDE* pero retrasando el reinicio de la transacción un periodo aleatorio elegido uniformemente en un rango que se incrementa exponencialmente con cada reinicio. Al igual que *DELAY*, su objetivo es minimizar los abortos múltiples causados por la adquisición de una dirección por parte de otra transacción, a costa de poder incurrir en una espera innecesaria.
- *AGGRESSIVE:* En caso de conflicto, se aborta la transacción propietaria actual en favor de la nueva transacción.
- *TIMESTAMP:* En todo conflicto, siempre se aborta la transacción más joven (más actual) en base a las etiquetas de tiempo asignadas a cada transacción a su inicio.

La visibilidad de las lecturas, es otra de posibles configuraciones que ofrece TinySTM.

- *Lecturas visibles:* Permite incluir las direcciones leídas por las transacciones en el mecanismo de detección de conflictos. Esto facilita la detección de conflictos por parte de transacciones escritoras que invalidan lecturas previas realizadas por otras transacciones, sin necesidad de que estas recurran a un proceso de validación.
- *Lecturas invisibles*: Es la configuración definida por defecto. En ella se mantienen las lecturas transaccionales al margen del proceso de detección de conflictos y se introduce un proceso de validación incremental sobre el conjunto de direcciones que componen el *read set* de la transacción.

Además de estas configuraciones, TinySTM también ofrece la posibilidad de generar estadísticas transaccionales de su ejecución, utilización de filtros de *Bloom* para la detección de conflictos o prevenir las entradas duplicadas en los sets de lectura o escritura. Por defecto todas estas opciones están desactivadas.

El sistema que tomaremos como base para la implementación de nuestras propuestas presentará una configuración ETL con política de escritura diferida (*write-back*), lecturas invisibles y una política de contención de partida basada en la estrategia *SUICIDE*.

Para implementar la aproximación ETL, TinySTM utiliza el ya comentado array de *locks*, de manera que el *bit* menos significativo (*LSB*) de cada *lock* asociado a una dirección de memoria se utiliza para indicar si este se encuentra adquirido. El resto de *bits* contiene, o bien la dirección de la transacción propietaria (LSB

activado), o bien la versión correspondiente al instante de tiempo del *commit* de la última transacción escritora. Por otra parte, la asignación de direcciones de memoria a *locks* se realiza mediante una función *hash* basada en desplazamiento de bits (obviando los menos significativos), lo cual permite que cada *lock* cubra una porción del espacio de direcciones en memoria (figura 3.1). Esto permite reducir el tamaño total del array de *locks*, y por tanto, el consumo de memoria necesario, sin embargo, también ocasiona un problema de *false sharing* ya que algunos accesos a direcciones de memoria diferentes pueden ser erróneamente identificados como conflictos.

Basándose en esta aproximación, las operaciones de memoria en TinySTM proceden de la siguiente manera para la configuración utilizada (detección de conflictos *eager* y gestión de versiones *lazy*):

- ***Operación de escritura***: Escribir una localización de memoria requiere adquirir el correspondiente *lock* durante la operación. Para ello, la transacción identifica el *lock* asociado a la dirección de memoria (aplicando la función *hash*) e intenta activar el *bit* menos significativo. Si la adquisición falla, por estar dicho *bit* activado, se comprueba si el propietario del *lock* es la propia transacción, utilizando para ello la dirección almacenada en los *bits* restantes. Si es así, el nuevo valor se escribe en el *write buffer* y el proceso finaliza. En caso contrario, se detecta un conflicto de datos e inmediatamente se aborta una de las transacciones (generalmente, la más actual). Si el *lock* no ha sido adquirido por nadie, entonces la transacción se vuelve su propietaria, adquiriendo el *lock* atómicamente mediante una operación *compare-and-swap (CAS)* y creando sendas entradas en el *write buffer* y el *write set* para la dirección.

- ***Operación de lectura***: La lectura de una localización de memoria no requiere de la adquisición del *lock*, sin embargo, se debe asegurar la consistencia de la lectura. Las versiones o marcas de tiempo son usadas para tal fin. El proceso consiste en realizar una lectura del *lock*, seguida de la lectura de la dirección de memoria y finalmente una nueva lectura del *lock*. Si entre ambas lecturas del *lock*, este no ha sido adquirido y mantiene el mismo número de versión, la lectura es válida. Una vez leída la dirección, el algoritmo LSA [86] determina si la lectura es consistente en el rango de actuación de la transacción (i.e., no haya sido modificada desde el inicio de la transacción, ya que en ese caso, su valor podría no ser consistente).

El procedimiento de *commit* en TinySTM variará también dependiendo de la configuración escogida. En nuestro caso, el proceso básicamente consiste en

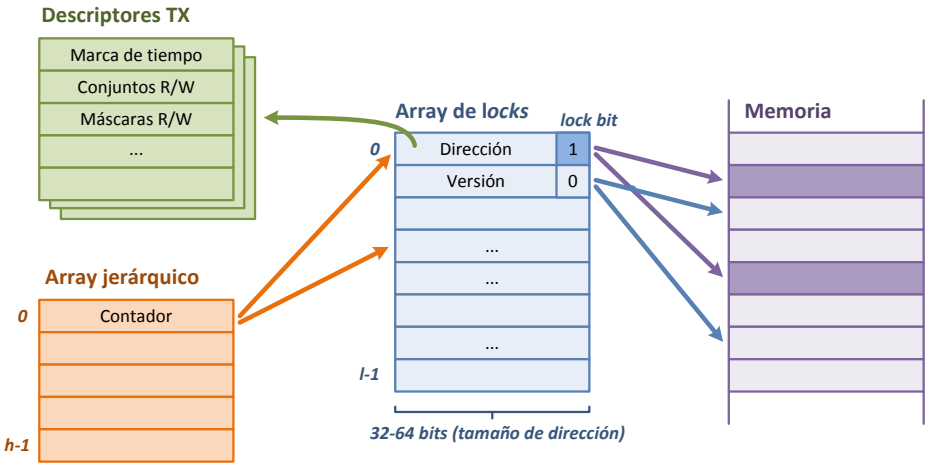


Figura 3.1: Estructuras de datos utilizadas en TinySTM

actualizar la memoria con los valores contenidos en el *write buffer*. Sin embargo, antes de eso, el conjunto de las direcciones de memoria leídas (*read set*) debe ser validado para asegurar su consistencia (i.e., que sus correspondientes *locks* no han sido adquiridos por otra transacción y aun mantienen la misma versión). Dependiendo del tamaño del *read set*, este proceso puede llegar a ser muy costoso. Una posible solución consiste en reducir el número de *locks* en el array, sin embargo, esto aumenta aún más el problema del *false sharing*.

TinySTM opta por una solución basada en *hierarchical locking* (figura 3.1), que consiste en mantener, además del array de l *locks* utilizado para la detección de conflictos, un array jerárquico de h contadores ($h << l$), el cual es mapeado utilizando una función *hash* tal que dos direcciones asignadas al mismo *lock* también sean asignadas al mismo contador [1]. Además, cada transacción dispone de unas máscaras de lectura y escritura de h *bits* para marcar los contadores accedidos durante su ejecución. De esta manera, durante una lectura o escritura, la transacción determina el contador compartido i que cubre la dirección, y comprueba el *i-ésimo* bit de la máscara de lectura. Si es cero, se pone a uno y se guarda localmente el valor del contador. En caso de escritura, también se comprueba la correspondiente máscara y si el bit es cero, se pone a uno y se incrementa atómicamente el valor del contador.

[1] Si se considera h múltiplo de l, por ejemplo, $l = 2^i$ y $h = 2^j$ para ($i > j$), los índices de los arrays se pueden calcular utilizando la misma función *hash* como $hash(addr)$ mód l (array *locks*) y $hash(addr)$ mód h (array contadores)

Así, durante la validación solo es preciso comprobar que los contadores asociados a los *bits* activos en las máscaras, aún mantienen el valor previamente leído (lectura) o el valor leído más uno (escritura). Esto supone muchas menos comprobaciones que verificar cada elemento del array de *locks*, ya que $h << l$.

3.2. Aplicaciones de evaluación

Con el objetivo de evaluar las aportaciones presentadas en esta tesis, un conjunto de códigos representativos han sido seleccionados para cubrir las distintos dominios de aplicación disponibles y medir diferentes aspectos del rendimiento. Estas aplicaciones de prueba pueden ser clasificadas en tres grupos principales:

- ***Aplicaciones transaccionales***: La *suite de benchmarks STAMP (Stanford Transactional Applications for Multi-Processing)*, especialmente desarrollada para investigación en memoria transaccional, ha sido utilizada para evaluar los aspectos puramente transaccionales de las aportaciones realizadas. Esta *suite* incluye ocho aplicaciones que cubren un amplio rango de configuraciones de interés (tamaño de transacciones, tiempo en transacción, contención, etc.).

- ***Aplicaciones de tipo Stencil***: Dos códigos de tipo Stencil que resuelven el mismo problema utilizando dos aproximaciones completamente opuestas (Jacobi es totalmente paralelo mientras que Seidel es completamente secuencial) han sido considerados para evaluar los *overheads* introducidos por las diferentes propuestas respecto del sistema transaccional software de base.

- ***Aplicaciones con reducciones***: Varios códigos con reducciones han sido considerados como aplicaciones de interés para evaluar las propuestas realizadas en el ámbito de los códigos iterativos, y más concretamente, en presencia de reducciones irregulares. *Fluidanimate*, *transformada de Legendre*, *Euler*, *Md2* y el cálculo del *histograma de una imagen* son los códigos incluidos en este grupo, cubriendo una amplia gama de aspectos de interés en el ámbito de las reducciones (número de bucles de reducción, reducciones por bucle, tamaños de array de reducción, carga computacional del cuerpo del bucle, etc.).

A excepción de las aplicaciones de STAMP y de Fluidanimate, cuyos códigos ya se proporcionan en una versión paralela, el resto de aplicaciones han

Tabla 3.1: Caracterización de las aplicaciones STAMP

Aplicación	Dominio	Longitud Tx	Tiempo Tx	Contención
Bayes	Aprendizaje automático	Larga	Alto	Alta
Genome	Bioinformática	Media	Alto	Baja
Intruder	Seguridad	Corta	Medio	Alta
Kmeans	Minería de datos	Corta	Bajo	Baja
Labyrinth	Ingeniería	Larga	Alto	Alta
Ssca2	Científico	Corta	Bajo	Baja
Vacation	Procesamiento de transacciones online	Media	Alto	Baja/Media
Yada	Científico	Larga	Alto	Medio

sido manualmente paralelizados utilizando OpenMP [64, 73]. A continuación se describen las características más destacables de cada uno de ellos.

3.2.1. Aplicaciones transaccionales (STAMP)

En esta sección se realiza una breve descripción de las aplicaciones que componen la *suite* STAMP [16, 67]. La tabla 3.1 presenta un resumen cualitativo de las principales características transaccionales para cada aplicación, entre las que se incluyen: Longitud de la transacción (número de instrucciones), tiempo invertido dentro de transacciones y cantidad de contención.

Bayes

Esta aplicación implementa un algoritmo para el aprendizaje de redes Bayesianas a partir de un conjunto de datos observados. En la representación de la red se utiliza un grafo dirigido sin ciclos, donde cada nodo representa una variable y cada arista representa una dependencia condicional entre las variables. Partiendo de un estado inicial sin dependencias entre variables, el algoritmo incrementalmente aprende las dependencias analizando el conjunto de variables observadas. En cada iteración, una variable es asignada a cada hilo para ser analizada y añadir nuevas dependencias a la red. En la figura 3.2 se muestra el pseudocódigo de la principal parte del algoritmo.

En esta aplicación las transacciones se utilizan para proteger el cálculo y la asignación de nuevas dependencias al grafo. Por tanto, *Bayes* pasa la mayor parte del tiempo de ejecución calculando las nuevas dependencias, lo que deriva en transacciones largas con grandes conjuntos de lectura y escritura (*read set, write*

```
1  global Queue dependencies
2  global Graph network
3
4  begin parallel
5    while true do
6      TxBegin
7        dependency ← dependencies.pop()
8      TxEnd
9
10     if dependency is null then
11       break
12     end if
13
14     TxBegin
15       network.apply(dependency)
16     TxEnd
17
18     TxBegin
19       fromVariable ← dependency.from()
20       subGraph ← network.subGraph(fromVariable)
21       newDependency ← network.getNewDependency(fromVariable, subGraph)
22     TxEnd
23
24     if newDependency is not null then
25       TxBegin
26         dependencies.push(newDependency)
27       TxEnd
28     end if
29   end while
30 end parallel
```

Figura 3.2: Pseudocódigo correspondiente a *Bayes*.

set). Además, este *benchmark* presenta una gran contención ya que el subgrafo cambia frecuentemente.

Genome

Este *benchmark* presentan una aplicación de bioinformática en la que se toman un conjunto de segmentos de ADN (relativamente grandes) y se comparan a fin de reconstruir el genoma original. El proceso consta de dos fases. Una primera fase utiliza un conjunto *hash* para eliminar posibles segmentos duplicados y crear un conjunto global con segmentos únicos. En la segunda fase, cada hilo extrae un segmento de dicho conjunto global y lo compara (utilizando el algoritmo

```
1 global Array segments
2 global Set uniqueSegments
3
4 begin parallel
5   for all segment in segments.myPartition() do
6     TxBegin
7       if segment not in uniqueSegments then
8         uniqueSegments.insert(segment)
9       end if
10    TxEnd
11  end for
12
13  Barrier
14
15  for all uniqueSegment in uniqueSegments.myPartition() do
16    TxBegin
17    matchedSegment ← uniqueSegments.findMatch(uniqueSegment)
18    if matchedSegment is not null then
19      uniqueSegments.remove(matchedSegment)
20      uniqueSegment.join(matchedSegment)
21    end if
22    TxEnd
23  end for
24 end parallel
```

Figura 3.3: Pseudocódigo correspondiente a *Genome*.

de comparación de cadenas *Rabin-Karp*) para finalmente añadirlo a su propio grupo de segmentos ya revisados. El pseudocódigo de la parte esencial de este *benchmark* se incluye en la figura 3.3.

Ambas fases son protegidas con transacciones para que puedan ser ejecutadas por múltiples hilos en paralelo sin incurrir en las posibles inconsistencias derivadas de añadir o extraer un mismo segmento desde el conjunto único durante su creación (primera fase) o durante el proceso de comparación de segmentos (segunda fase). Por ello, *Genome* pasa la mayor parte del tiempo ejecutando dentro de transacciones. Aunque los *data sets* presentan un tamaño moderado, la contención exhibida es realmente baja, debido principalmente a la gran cantidad de segmentos disponibles para analizar.

Intruder

Intruder simula un sistema de detección de intrusión en redes basado en firmas. Los paquetes que circulan por la red son escaneados en paralelo en busca de alguna firma de intrusión conocida. Este proceso se realiza a través de tres fases. Los paquetes son tomados de una simple cola FIFO en la fase inicial de captura y reensamblados seguidamente sobre un diccionario compartido, implementado como un árbol auto-balanceado, que contiene la lista de paquetes que pertenecen a la misma sesión. Finalmente, una fase de detección se encarga de determinar la ocurrencia de una intrusión. El acceso a las estructuras de datos compartidas, realizado en las dos primeras fases, debe ser protegido para evitar escrituras concurrentes. Concretamente, la versión transaccional del *benchmark* encierra cada una de estas fases en una transacción, siguiendo una aproximación similar a los *locks* de grano grueso. Aunque pueda parecer ineficiente, esta aplicación alcanza un buen rendimiento debido a la concurrencia optimista del sistema transaccional. El pseudocódigo correspondiente a este *benchmark* es presentado en la figura 3.4.

En general, aunque el tamaño de las transacciones es relativamente pequeño, el nivel de contención se mantiene moderadamente alto dependiendo de la frecuencia con que se efectúe el balanceo del árbol en la fase de reensamblaje. Debido a que dos de las tres fases transcurren dentro de transacciones, la cantidad de ejecución transaccional, en relación al total de la aplicación, es media/alta.

Kmeans

Kmeans es un algoritmo de clasificación en el que un conjunto de objetos en un espacio *N*-dimensional son agrupados en *K* diferentes *clusters*, que se utilizan para particionar datos en subconjuntos relacionados (pseudocódigo en figura 3.5). Cada hilo iterativamente se encarga de procesar una porción de los objetos, protegiendo en cada iteración la actualización de los centros del *cluster* mediante una transacción. La contención entre hilos dependerá del número de *clusters* definidos, ya que cuanto mayor sea este, menor será la probabilidad de que dos hilos operen sobre el mismo *cluster*. El tamaño de las transacciones, al igual que de los *read set* y *write set*, es relativamente pequeño, puesto que son proporcionales a la dimensionalidad del espacio. Además, la mayor parte del tiempo de ejecución es invertido en calcular nuevos centros para el *cluster*, para lo cual cada hilo solo precisa leer de su propia partición de objetos, fuera de la protección de las transacciones.

```
1  global Queue packets
2  global Decoder decoder
3
4  begin parallel
5    Detector detector while true do
6      TxBegin
7        packet ← packets.remove()
8      TxEnd
9      if packet is null then
10       break
11     end if
12
13     TxBegin
14       error ← decoder.reassemble(packet)
15     TxEnd
16     if error then
17       Invalid packet
18     end if
19     TxBegin
20       data ← decoder.getComplete(packet)
21     TxEnd
22
23     if data is not null then
24       error ← detector.check(data)
25       if error then
26         Intrusion detected
27       end if
28     end if
29
30   end while
31 end parallel
```

Figura 3.4: Pseudocódigo correspondiente a *Intruder*.

Labyrinth

Este *benchmark* implementa una variante al algoritmo de búsqueda de caminos en laberintos propuesto por *Lee* [55]. Básicamente, se considera una rejilla tridimensional compartida, sobre la cual, un conjunto de hilos concurrentes graban rutas desde un punto de inicio hasta otro final, conectados entre sí por puntos adyacentes de la rejilla. En la versión transaccional, se produce un conflicto cuando dos caminos que se cruzan en algún punto, son añadidos a la rejilla. Para evitar conflictos innecesarios, los nuevos caminos no se calculan sobre la rejilla compartida, sino sobre copias privadas actualizadas que cada hilo realiza al inicio del cálculo de cada nuevo camino. Por tanto, el conflicto es detectado al

```
1  global Array points
2  global Array memberships
3  global Array centers
4  global Integer delta
5
6  begin parallel
7    for all point in points.myPartition() do
8      k = FindNearestCenter(centers, point)
9      if k ≠ memberships[point] then
10       myDelta ← myDelta + 1
11       memberships[point] ← k
12     end if
13     TxBegin
14       UpdateCenter(centers[k], point)
15     TxEnd
16   end for
17   TxBegin
18     delta ← delta + myDelta
19   TxEnd
20 end parallel
```

Figura 3.5: Pseudocódigo correspondiente a *Kmeans*.

intentar escribir el camino previamente computado sobre la rejilla compartida. En caso de conflicto, la transacción aborta y reinicia el calculo de otro camino que una los mismos puntos, esta vez utilizando una nueva copia de la rejilla en su estado más actual.

Dado que la copia privada requiere leer todos los puntos de la rejilla y esto ocurre dentro de una transacción, estas lecturas igualmente serían añadidas al *read set* causando numerosos conflictos. Un sistema de *early-release* [49] ha sido utilizado para eliminar estos accesos transaccionales generados al crear la copia privada de la rejilla (figura 3.6).

Debido a que la mayor parte del tiempo es consumida por el cálculo del nuevo camino, lo cual ocurre dentro de una transacción, tanto el tamaño de los *data sets* como el de la transacción son especialmente grandes. Este gran número de accesos transaccionales a memoria, unido a hecho de que todo el código es ejecutado dentro de transacciones, hace que la contención exhibida por *Labyrinth* sea bastante alta.

```
1 global Queue tasks
2 global Grid maze
3
4 begin parallel
5   while true do
6
7     TxBegin
8       task ← tasks.remove()
9     TxEnd
10    if task is null then
11      break
12    end if
13    start ← task.start()
14    end ← task.end()
15
16    TxBegin
17      mazeCopy ← maze.copy()
18      TxEarlyRelease(maze)
19      success ← Expand(mazeCopy, start, end)
20      if success then
21        path ← Backtrack(mazeCopy, end, start)
22        valid ← maze.addPath(path)
23        if not valid then
24          TxRestart
25        end if
26      end if
27    TxEnd
28
29  end while
30 end parallel
```

Figura 3.6: Pseudocódigo correspondiente a *Labyrinth*.

Ssca2

Scalable Synthetic Compact Applications 2 son un grupo de cuatro *kernels* que operan sobre grafos. STAMP incluye uno de estos *Kernels* que implementa el proceso de creación de una estructura de datos tipo grafo, utilizando arrays de adyacencia y arrays auxiliares, haciendo uso de las primitivas TM para beneficiarse del paradigma de paralelización optimista. Cada hilo en encarga de añadir nuevos nodos al grafo de manera concurrente, utilizando el array de adyacencia, el cual debe ser accedido dentro de transacciones para evitar inconsistencias. Esta operación es realmente breve, por lo que la aplicación no invierte demasiado tiempo dentro de las transacciones. Por otra parte, el número de lecturas y escrituras transaccionales es realmente pequeño, lo que unido al gran número

```
1 global Array nodes
2 global Array edges
3
4 begin parallel
5   for all edge in edges.myPartition() do
6     fromIndex ← edge.from()
7     toIndex ← edge.to()
8     TxBegin
9       node[fromIndex].addChild(toIndex)
10    TxEnd
11  end for
12 end parallel
```

Figura 3.7: Pseudocódigo correspondiente a *Ssca2*.

de nodos presentes en el grafo, hace infrecuente que varios hilos modifiquen el mismo nodo concurrentemente (contención baja). La figura 3.7 presenta el pseudocódigo correspondiente a la parte más relevante de este *benchmark*.

Vacation

Esta aplicación simula un sistema de reserva de viajes distribuido de una empresa. Así, el sistema implementa estructuras tipo árbol que relaciona los clientes y sus reservas con los distintos *items* de viajes. Básicamente, el sistema permite tres tipos de acciones: realizar una nueva reserva, cancelar una reserva existente o actualizarla (figura 3.8). Para asegurar la validez de la base de datos, cada una de estas operaciones es protegida dentro de una transacción de grano grueso, lo cual se traduce en un gran tiempo dentro de transacción. Debido a las características de estas operaciones, el número de lecturas y escrituras transaccionales se mantiene moderado. En relación a la contención presentada por la aplicación, puede fluctuar entre baja y media dependiendo de la cantidad de operaciones que modifiquen grandes porciones de la base de datos.

Yada

Yet Another Delaunay Application implementa un algoritmo de refinado de mallas iterativo, en el que cada iteración efectúa los siguientes pasos: primero, cada hilo extrae un triángulo sin refinar de una cola de trabajo. Seguidamente, se procede a retriangular la malla, la cual está almacenada como un grafo que contiene los triángulos que la componen. Este proceso es el más costoso e implica que

```
1 global Manager manager
2
3 begin parallel
4   for i ∈ [1...N] do
5     action ← RandomAction()
6     items ← RandomItems()
7     TxBegin
8       if action is reserve then
9         manager.reserve(items)
10      else if action is cancel then
11        manager.cancel(items)
12      else if action is update then
13        manager.update(items)
14      end if
15    TxEnd
16  end for
17 end parallel
```

Figura 3.8: Pseudocódigo correspondiente a *Vacation*.

muchos triángulos de la malla compartida sean visitados y modificados. Puesto que esta operación debe protegerse mediante transacciones, tanto el tamaño de los conjuntos de lectura y escritura, como el tiempo dentro de la transacción son altos. El uso de las transacciones en *Yada* puede apreciarse en el pseudocódigo de la figura 3.9. Finalmente, cualquier nuevo triangulo no refinado resultante de la nueva triangulación es añadido a la cola de trabajo para ser procesado en una posterior iteración. En general, la contención que presenta la aplicación es moderada, debido al acceso compartido a la malla.

3.2.2. Aplicaciones de tipo *Stencil*

Las aplicaciones de tipo *Stencil* son muy comunes en el mundo de la computación y son aplicables a un amplio rango de campos, tales como la ingeniería o ciencias.

Básicamente, estas aplicaciones utilizan estructuras de mallas o *grids*, almacenados en forma de arrays multidimensionales, sobre los cuales se realizan un gran número de barridos en los que el valor correspondiente a cada elemento se modifica basándose en los valores de sus vecinos.

Algunas aplicaciones de este tipo son la resolución de ecuaciones diferenciales, métodos de refinamiento de mallas y todo tipo de filtros aplicados a

```
1  global Mesh mesh
2  global Set segments
3  global PriorityQueue work
4
5  begin parallel
6    while true do
7
8      TxBegin
9        element ← work.remove()
10     TxEnd
11
12     if element is null then
13       break
14     end if
15     point ← element.center()
16
17     TxBegin
18       if element is segment then
19         segments.split(element)
20       end if
21       region ← mesh.affectedRegion(point)
22       newRegion ← region.retriangulate(point)
23       mesh.replaceRegion(region, newRegion)
24     TxEnd
25
26     TxBegin
27       work.insert(newRegion.badTriangles())
28     TxEnd
29
30   end while
31 end parallel
```

Figura 3.9: Pseudocódigo correspondiente a *Yada*.

imágenes (detección de bordes, desenfoque, etc.). La gran variedad de códigos de tipo *Stencil* pueden ser caracterizadas en base a varios aspectos, tales como, la dimensión del array, la distancia máxima de los vecinos representativos o el número total de vecinos representativos para cada elemento.

Otros dos factores de gran impacto son el patrón de acceso a los datos y el orden a seguir. El primero de estos factores suele estar caracterizado por la regularidad del patrón de acceso para cada elemento, lo que proporciona una base estable sobre la que desarrollar diferentes mecanismos o estrategias para explotar la concurrencia. Por el contrario, el orden de recorrido del array puede llegar a convertirse en un factor que limite este aspecto. Basándonos en la operación de actualización sobre los datos, podemos clasificar los códigos *Stencil*

```
1 for (t = 0; t < _PB_TSTEPS; t++){
2   #pragma omp transfor schedule(static,S)
3   for (i = 1; i < _PB_N - 1; i++)
4     for (j = 1; j < _PB_N - 1; j++)
5       B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j]
6           + A[1+i][j] + A[i-1][j])/5;
7   #pragma omp transfor schedule(static,S)
8   for (i = 1; i < _PB_N-1; i++)
9     for (j = 1; j < _PB_N-1; j++)
10      A[i][j] = B[i][j];
11 }
```

Figura 3.10: Código correspondiente al bucle principal del *benchmark Jacobi*

en dos tipos.

Se dice que un *Stencil* es de tipo ***Jacobi*** [79] si en cada barrido las operaciones de lectura y escritura se realizan sobre diferentes arrays. La figura 3.10 muestra el bucle principal del *benchmark Jacobi* en su versión 2D obtenido de la *suite Polybench*. Al realizar las escrituras en un arrays auxiliar, separando las fases de cálculo y actualización del array global, el orden en que se procesen los elementos en cada bucle se vuelve irrelevante. Esto permite paralelizar ambos bucles totalmente sin necesidad de mayor sincronización que la mera separación entre fases.

Por otro lado, si tanto las lecturas como las escrituras se realizan sobre el mismo array (computación *in-place*), entonces el orden en que se realice el barrido es importante, ya que cada elemento modificado se utiliza como entrada para el cálculo de otro elemento vecino. En este caso decimos que el código es de tipo ***Gauss-Seidel*** [79] (figura 3.11). Como podemos apreciar, el orden para el cálculo de los elementos viene impuesto por el propio método, el cual debe ser mantenido para garantizar un resultado correcto. En este caso, una paralelización no es posible sin incurrir en una serialización total de los hilos.

Debido a que habitualmente el número de operaciones en punto flotante realizadas por elemento del array es constante, y generalmente bajo en comparación con el número de referencias a memoria (i.e., los *bytes* movidos), este tipo de códigos presentan una *intensidad operacional* [99] realmente baja, definiéndose esta como el cociente entre ambos valores. Esto supone un problema para el rendimiento de la aplicación, ya que, está limitado por el ancho de banda de memoria.

Las versiones paralelas transaccionales de ambos *benchmarks* se han obtenido

```
1 for (t = 0; t < _PB_TSTEPS; t++){
2   #pragma omp transfor schedule(static,S) ordered
3   for (i = 1; i<= _PB_N - 2; i++)
4     for (j = 1; j <= _PB_N - 2; j++)
5       A[i][j] = (A[i-1][j-1] + A[i-1][j]
6               + A[i-1][j+1] + A[i][j-1]
7               + A[i][j] + A[i][j+1]
8               + A[i+1][j-1] + A[i+1][j]
9               + A[i+1][j+1])/9;
10 }
```

Figura 3.11: Código correspondiente al bucle principal del *benchmark Seidel*

simplemente dividiendo el array por filas e instrumentando las operaciones de lectura y escritura con las variantes transaccionales propuestas. En base a ello, el tamaño de las transacciones y sus *data sets* varia de pequeño a medio, dependiendo de las dimensiones del array de datos. El tiempo en transacción de ambos *benchmarks* es del 100 %, ya que la única computación que existe se realiza dentro de estas. Respecto a la contención, *Jacobi* no presenta ninguna debido a que es un código totalmente paralelo. Sin embargo, *Seidel* presenta una contención muy alta, ya que cada elemento depende de todos sus vecinos, lo cual implica que dos hilos accederán necesariamente a los valores escritos por el otro, aunque esto solo ocurra en las fronteras.

3.2.3. Aplicaciones con reducciones

Fluidanimate

Fluidanimate es una aplicación que forma parte del *Intel RMS*, incluida en *suite* de *benchmark Polybench* [77], que implementa una extensión del método *Smoothed Particle Hydrodynamics (SPH)* para realizar la simulación física en el comportamiento de fluidos para su aplicación en videojuegos y otras formas de animación en tiempo real [72]. Para ello, esta aplicación necesita resolver las ecuaciones *Navier-Stokes*, las cuales son un conjunto de derivadas parciales no lineales que describen el movimiento de fluidos cuya viscosidad es independiente del movimiento del mismo (fluidos *newtonianos* [10]).

Antes de realizar los cálculos, el fluido es divido en celdas que actúan como contenedores de partículas, llegando a albergar hasta un total de dieciséis, cada una con sus correspondientes atributos (aceleración, densidad, posición,

velocidad y viscosidad).

Para cada instante de tiempo simulado por el algoritmo (*frame*), *Fluidanimate* ejecuta cinco *kernels* sobre el conjunto de partículas.

- **Reconstrucción de índices espaciales**: Puesto que las partículas solo interactuan con aquellas que se encuentren a una distancia igual o inferior a *h* (limitada por el *kernel* de suavizado), el sistema recoloca las partículas utilizando una estructura con indexación espacial para explotar la proximidad entre ellas y reducir el número de evaluaciones necesarias. De este proceso se encargan las funciones `ClearParticles` y `RebuildGrid`.
- **Cálculo de densidades**: Este procedimiento calcula la densidad del fluido para cada partícula en función de cómo de próximos se encuentren sus vecinos. Así, aquellas regiones cuyas partículas se encuentren más próximas, presentarán una mayor densidad. La función `ComputeDensities` se encarga de realizar estos cálculos.
- **Cálculo de fuerzas**: Una vez calculadas las densidades, estas son usadas para calcular las fuerzas. Para ello, los únicos factores externos tenidos en cuenta por el *kernel* son la presión, viscosidad y gravedad. Además de esto, este proceso también se encarga de gestionar las colisiones entre partículas. Esto se realiza en la función `ComputeForces`.
- **Manejo de colisiones con la geometría de la escena**: Además de las colisiones entre partículas del fluido, las colisiones con la geometría de la escena son calculadas en la función `ProcessCollisions`. En base a ellas, las fuerzas calculadas en el *kernel* anterior serán actualizadas.
- **Actualización de posiciones de las partículas**: El último paso considera los valores finales de fuerzas obtenidos del paso anterior para calcular los valores de aceleración de cada partícula y actualizar su posición de acuerdo a ella (función `AdvanceParticles`).

Puesto que en la versión paralela de este código, los conflictos pueden suceder cuando se opera sobre partículas, teniendo en cuenta la información de sus vecinas, se ha procedido a proteger mediante transacciones los dos principales *kernels* de la aplicación, correspondientes al calculo de *densidades* y *fuerzas* (alrededor del 96 % del tiempo de ejecución). En la figura 3.12 podemos ver el código asociado al procedimiento que se encarga del cálculo de fuerzas. Nótese que solo aquellas partículas que conforman las fronteras de cada celda tienen un riesgo implícito de conflicto. Hemos estimado que únicamente el 1.5 % de las escrituras

```
1 for (ck = 0; ck < nz; ++ck){
2   for (cj = 0; cj < ny; ++cj)
3     for (ci = 0; ci < nx; ++ci){
4       cindex++;
5       numPars = cnumPars[cindex];
6       numNeighCells = GetNeightborCells(ci, cj, ck, neighCells);
7       &cell = CELLS[cindex];
8       for (ipar = 0; ipar < numPars; ++ipar)
9         for (inc = 0; inc < numNeighCells; ++inc){
10          cindexNeigh = neighCells[inc];
11          &neigh = CELLS[cindexNeigh];
12          for (iparNeigh = 0; iparNeigh < numNeighPars; ++iparNeigh){
13            if(&neigh.p[iparNeigh] < &cell.p[ipar]){
14              Calculate dispSq = dispSq(ipar, iparNeigh)
15              if(distSq < HSQ){
16                Calculate acc = acc(cell, neigh)
17                cell.a[ipar] = cell.a[ipar] + acc;
18                neigh.a[iparNeigh] = neigh.a[iparNeigh] - acc;
19              }
20            }
21    ...
22 }
```

Figura 3.12: Código correspondiente al cómputo de fuerzas en *Fluidanimate*

en el array de reducción involucrarían conflictos potenciales, lo cual hace que la contención no sea especialmente alta si se asumen tamaños de transacción pequeños (pocas iteraciones). Por su parte, el tamaño de los *data sets* puede variar de moderado a alto dependiendo, de nuevo, del tamaño de transacción elegido.

Legendre

Este *benchmark* implementa un sistema de predicción del clima haciendo uso de las transformadas de *Legendre* [71]. Para ello utiliza una serie de arrays tridimensionales que representan distintos aspectos de la climatología, tales como, vorticidad, divergencia, humedad y temperatura. Definidas las condiciones iniciales, se inicial el bucle principal de la aplicación en el que cada iteración invoca repetidamente dos subrutinas, que calculan la transformada inversa y directa de Legendre, para cada una de las latitudes, precedidas de una pequeña fase de inicialización. Esta fase es necesaria debido a que los límites de algunos bucles, dentro de ambas rutinas, dependen de dos arrays que deben ser inicializados en tiempo de ejecución para cada iteración del bucle principal. En nuestro *benchmark* esta inicialización es realizada secuencialmente y consume alrededor de un 5 %

```
1 for (i = 1; i < JDT+1; i++){
2   im = M[i];
3   imb = MBEG[i];
4   ime = MEND[i];
5   for (is = imb; is<= ime; is=is+2){
6    pnms = PNM[is][irow];
7    pnms1 = PNM[is+1][irow];
8    if(is == ime){
9     for (ilev = 1; ilev <= JDLEV*2; ilev++){
10    FSV[ilev][1][im] = FSV[ilev][1][im] + SPAV[ilev][1][is] * PNMS;
11    FSD[ilev][1][im] = FSD[ilev][1][im] + SPAD[ilev][1][is] * PNMS;
12    FSQ[ilev][1][im] = FSQ[ilev][1][im] + SPAQ[ilev][1][is] * PNMS;
13    FST[ilev][1][im] = FST[ilev][1][im] + SPAT[ilev][1][is] * PNMS;
14    }
15   }else{
16    for (ilev = 1; ilev <= JDLEV*2; ilev++){
17    FSV[ilev][1][im] = FSV[ilev][1][im] + SPAV[ilev][1][is] * PNMS;
18    FAV[ilev][1][im] = FAV[ilev][1][im] + SPAD[ilev][1][is+1] * PNMS1;
19    ...
20    }
21   }
22 }
```

Figura 3.13: Código correspondiente a la subrutina que calcula la transformada inversa de *Legendre* para la fila *irow*.

del tiempo total de ejecución.

El 95 % restante se reparte a partes iguales entre las dos subrutinas principales, cada una de las cuales han sido a su vez paralelizadas y convenientemente instrumentada utilizando las llamadas transaccionales. En concreto, este *benchmark* presenta varios bucles anidados con múltiples operaciones de reducción en cada uno de ellos, accedidas mediante indirecciones. La figura 3.13 muestra uno de esos bucles.

Euler

El *benchmark Euler* es un código de hidrodinámica 3D multimaterial, extraído del conjunto de aplicaciones motivadoras propuestas por HPF-2 [29]. En él, se resuelven las ecuaciones de Euler para simulaciones físicas de que involucran flujo de fluidos y deformación de materiales. La estructura básica en esta aplicación es una malla tridimensional compartida sobre la que se realizan diferentes barridos en el dominio del tiempo para calcular distintas magnitudes físicas,

```
1 for (i = 1; i <= numEdges; i++){
2   n1 = edge(i,1);
3   n2 = edge(i,2);
4
5   a1 = foo(edgeData(i,1), edgeData(i,2), edgeData(i,3),
6       velocity(n1,1), velocity(n1,2), velocity(n1,3));
7   a2 = foo(edgeData(i,1), edgeData(i,2), edgeData(i,3),
8       velocity(n2,1), velocity(n2,2), velocity(n2,3));
9
10  r1  = a1*velocity(n1,1) + a2*velocity(n2,1) + edgeData(i,1);
11  r2  = a1*velocity(n1,2) + a2*velocity(n2,2) + edgeData(i,2);
12  r3  = a1*velocity(n1,3) + a2*velocity(n2,3) + edgeData(i,3);
13
14  vel_delta(n1,1) = vel_delta(n1,1) + r1;
15  vel_delta(n1,2) = vel_delta(n1,2) + r2;
16  vel_delta(n1,3) = vel_delta(n1,3) + r3;
17
18  vel_delta(n2,1) = vel_delta(n2,1) - r1;
19  vel_delta(n2,2) = vel_delta(n2,2) - r2;
20  vel_delta(n2,3) = vel_delta(n2,3) - r3;
21 }
```

Figura 3.14: Bucle de reducción que recorre los bordes de la malla durante el cálculo del cambio de velocidad en *Euler*.

tales como la fuerza o velocidad, sobre cada uno de los nodos que la componen. Esta estructura se implementa como un array de tres dimensiones (x,y,z) accedido mediante indirecciones para realizar las operaciones de reducción correspondientes al cómputo de fuerzas y velocidades. Gracias a que la estructura de la malla es estática (no se modifica la descripción de las aristas en cada paso del algoritmo), esté código presenta algunas características beneficiosas como es el hecho de utilizar un mismo patrón de indirección a lo largo de los diferentes bucles de reducción.

Básicamente, la mayor parte de la computación en está aplicación reside en tres métodos: *ComputeForces()*, *ComputeVelocityChange()* y *SmoothenVelocity()*. En la versión transaccional, los bucles de reducción incluidos en estos tres métodos han sido paralelizados e instrumentados con las correspondientes instrucciones transaccionales para proteger el acceso a las estructuras compartidas. En la figura 3.14 presentamos uno de dichos bucles de reducción.

MD2

MD2 [70] presenta una simulación de dinámica molecular (en dos dimensiones) con interacciones entre partículas de corto alcance. Esta simulación se realiza a lo largo de una serie de pasos temporales, al final de los cuales se obtienen las coordenadas y energías finales de cada molécula. Así, el movimiento de cada molécula en cada instante es determinado por dos aspectos: su velocidad al inicio del proceso y las fuerzas que ejercen sobre ella el resto de moléculas vecinas (problema N-cuerpos). La parte computacionalmente intensiva en este *benchmark* se corresponde con el bucle que realiza el cálculo de fuerzas, el cual se ejecuta en cada instante de tiempo. Este bucle irregular, itera sobre una lista que contiene todos los pares de moléculas que pueden interactuar entre si, para cada uno de los cuales se calculan las fuerzas ejercidas por cada molécula sobre la otra. Concretamente, esta lista almacena pares de números asociados a las moléculas y que se utilizan para indexar los arrays que contiene la información acerca de la velocidad, posición y fuerza de las moléculas. Dado que el cálculo de fuerzas es un proceso acumulativo en el que se tienen en cuenta todos los vecinos cuya distancia sea inferior a un determinado umbral, las operaciones aplicadas corresponden en su totalidad a reducciones. En la figura 3.15 podemos apreciar una sección del código correspondiente al bucle que recorre los pares de moléculas para el cómputo de fuerzas. Hay que tener en cuenta que la posición de las moléculas cambia debido al efecto de estas interacciones, por lo que la lista de interacción entre moléculas debe ser reconstruida cada cierto número de pasos, aunque esto realmente no supone un coste computacional excesivo en relación con el cálculo de fuerzas.

Para la versión transaccional, se ha paralelizado el bucle que realiza el cálculo de fuerzas, de manera que cada hilo se encargue de procesar un subconjunto de moléculas de la lista. Los conflictos sucederán cuando dos hilos extraigan de la lista sendos pares de moléculas, con al menos una de ellas en común. El tamaño de la transacción quedará definido por el número de pares de moléculas analizadas dentro de cada transacción e influirá de manera decisiva en la contención presentada por la versión transaccional.

Histogram

Se trata de un *benchmark* sintético que realiza el cálculo del histograma correspondiente a una imagen digital de entada en escala de grises. El histograma de una imagen (con niveles de gris en el rango [0, 255]) es una función discreta $p(r_k) = n_k/n$, donde r_k es el k-ésimo nivel de gris, n_k es el numero de píxeles

```
1 for (kk = 0; kk < nbls; kk=kk+2){
2   i = NBLaux->NBLpt[kk];
3   j = NBLaux->NBLpt[kk+1];
4   Calculate xper, yper
5
6   rx = RXpart[i] - RXpart[j] - xper;
7   ry = RYpart[i] - RYpart[j] - yper;
8   r = rx*rx + ry*ry;
9   if(r < THRESHOLD){
10      Calculate rho
11      FXpart[i] = FXpart[i] + rx * rho;
12      FXpart[j] = FXpart[j] + rx * rho;
13      FYpart[i] = FYpart[i] + ry * rho;
14      FYpart[j] = FYpart[j] + ry * rho;
15  }
16 }
```

Figura 3.15: Bucle de reducción que recorre todos los pares de moléculas para el cálculo de fuerzas en *MD2*.

de la imagen con ese nivel de gris y n el numero total de píxeles de la imagen ($k = 0, ..., 255$). De forma general, podemos decir que $p(r_k)$ da una idea acerca de la probabilidad de que aparezca el nivel de gris r_k. La representación gráfica de esta función, para todos los valores de k, proporciona una descripción global de la apariencia de la imagen.

Aunque el histograma no proporciona información específica sobre el contenido de la imagen, si que aporta información útil de la que parten una gran cantidad de técnicas modernas de procesado de imagen, entre las que podemos incluir mejoras de brillo y contraste o redistribución de tonos. En el campo de la visión artificial, el cálculo de los histogramas también se utiliza para la delimitación de umbrales, necesarios para el proceso de detección de bordes o segmentación de imagen.

Desde el punto de vista computacional, el cálculo del histograma de una imagen supone un ejemplo clásico de bucle reductivo, ya que la información sobre los niveles de gris presentes en la imagen son agrupados en una estructura de menor tamaño mediante el uso de una operación asociativa y conmutativa (+). En la figura 3.16 se presenta el código correspondiente al bucle principal del *benchmark Histograma*. En nuestro caso, la imagen de entrada se almacena en un array unidimensional (por simplicidad). Para la obtención de la versión transaccional paralela basta con dividir el espacio de iteraciones del bucle, que recorre los píxeles de la imagen, entre los diferentes hilos e instrumentar ade-

```
1 #pragma omp transfor schedule(static,S)
2 for (i=0; i<_N_; i++) {
3   g_hist[pixel[i]] = g_hist[pixel[i]] + 1;
4 }
```

Figura 3.16: Bucle principal del *benchmark Histograma*

cuadamente las operaciones de lectura y escritura. El tamaño de la transacción, y por consiguiente de los *data sets*, dependerá de tamaño de bloque (número de iteraciones consecutivas) asignado a cada transacción. Por su parte, la contención estará definida por el tamaño y distribución de los niveles de gris en la matriz de entrada, aunque en general, dado que solo hay 256 niveles sobre los que reducir la información.

3.3. Métricas

En esta sección se detallan todas las métricas utilizadas en esta tesis, siendo cada una de ellas necesaria para evaluar diferentes aspectos de las aportaciones realizadas.

Speedup

En programación paralela, el *speedup* se define como el cociente entre el tiempo de la versión secuencial y el de la versión paralela de una misma aplicación. Esta métrica representa un factor de aceleración que refleja cuánto más rápido ejecuta la versión paralela de la aplicación. Idealmente, el valor máximo dependerá del número de hilos de ejecución paralelos utilizados en la ejecución. En la práctica, esto máximos ideales difícilmente se alcanzan en la mayoría de aplicaciones, incluso aquellas sin accesos compartidos, debido a efectos tales como los expuestos por la *Ley de Amdahl* [1, 88] y otras limitaciones como la velocidad y capacidad de los buses de acceso a memoria. La figura 3.17a muestra la ecuación para el cálculo del *speedup*.

Además de este *speedup*, también conocido como *speedup absoluto*, existen otros tipos presentes en la literatura que ofrecen otras perspectivas para cuantificar la mejora de una aplicación paralela variando el tiempo de ejecución de referencia (i.e. ejecución secuencial). A continuación enumeramos dos variaciones de *speedup* utilizadas en esta tesis.

$$\frac{time_{SERIAL}}{time_{STM}(n\ threads)} \qquad \frac{time_{STM}(1\ thread)}{time_{STM}(n\ threads)} \qquad \frac{time_{STMA}(n\ threads)}{time_{STMB}(n\ threads)}$$

(a) Absoluto (b) Relativo (c) Comparativo

Figura 3.17: Expresiones correspondientes a las métricas de *speedup* en sus tres variantes. La variable $time_{SERIAL}$ representa el tiempo de ejecución de la versión secuencial, mientras que $time_{STM}$ representa el tiempo de ejecución la versión transaccional.

Speedup relativo

En esta versión de *speedup* se mide la mejora del tiempo de la ejecución paralela respecto a la misma versión paralela con un único hilo de ejecución. Hay que ser consciente de que un aplicación implementada para ser ejecutada en paralelo, generalmente realiza más trabajo que la versión secuencial, debido a que necesita crear los hilos, inicializarlos, particionar y distribuir el trabajo a cada hilo trabajador, entre otras cosas. Cuando medimos el rendimiento de un STM este problema se agrava significativamente debido a que la versión secuencial no comparte los *overheads* introducidos por cada operación de memoria (*read* y *write*), así como por la creación de la transacción (*begin*) y su finalización (*commit*).

El *speedup relativo* nos permite, en estos casos, paliar este desajuste de ejecución entre ambas versiones, ya que al utilizar la misma versión paralela con 1 hilo, los *overheads* asociados a la gestión transaccional por software de las operaciones de memoria, así como el de la creación de hilos, transacciones y el proceso de *commit* ahora estarán presentes en el tiempo de ejecución del sistema de referencia. Así, esta métrica nos proporcionará una verdadera medida de las aportaciones realizadas sin considerar los *overheads* propios del sistema transaccional.

En nuestro caso, dado que nuestras aportaciones han sido implementadas sobre la base de TinySTM, utilizaremos como referencia la versión paralela de los benchmark ejecutados sobre una versión sin modificar de TinySTM con un único hilo de ejecución. La ecuación para el cálculo del *speedup relativo* es mostrada en la figura 3.17b.

Speedup comparativo

Este *speedup* se utiliza para comparar dos implementaciones diferentes de una misma aplicación paralela y medir el factor de mejora de una respecto a otra. Para ello se calcula el cociente entre el tiempo de ejecución de ambas imple-

mentaciones para el mismo número de hilos (figura 3.17c). El *speedup comparativo* en el ámbito de TM, está fuertemente relacionado con el número de conflictos y el coste de los abortos. En la comparación de dos implementaciones de sistemas de memoria transaccional aplicados al mismo *benchmark*, estos costes son los que realmente marcan la diferencia entre ambos sistemas. Así, se espera que si el número de dependencias se incrementa, el sistema que consiga gestionarlas de tal manera que número de conflictos y abortos se reduzca en relación la otra, obtendrá un mayor valor en esta métrica. Hay que destacar que aunque en esta métrica parezca no tener una gran influencia el número de hilos (porque el cociente se calcula con el mismo número de hilos), en realidad tiene una gran influencia, ya que un mayor número de hilos, supone un mayor número de transacciones ejecutando concurrentemente lo cual incrementa significativamente la contención a la que se ve sometido el sistema transaccional.

Throughput

Esta métrica se utiliza generalmente para medir el volumen de trabajo o información que fluye a través de un sistema. En el ámbito de TM, el *throughput* mide el volumen de transacciones que realizan el *commit* por unidad de tiempo en el sistema (figura 3.18). Esta métrica es especialmente sensible al tamaño de las transacciones, ya que el *throughput* medido en una aplicación con transacciones cortas será, por lo general, mayor que otra aplicación con transacciones más largas. De esta métrica se espera una buena escalabilidad ante la variación del número de hilos, ya que un mayor número de hilos implican una mayor cantidad de transacciones que efectúan su *commit*, a la vez que reducen el tiempo de ejecución.

$$\frac{committedXacts_{STM}(n\ threads)}{time_{STM}(n\ threads)}$$

Figura 3.18: Expresión correspondiente a la métrica *throughput*. La variable $committedXacts_{STM}$ es el número de transacciones que han realizado el *commit*. La variable $time_{STM}$ representa en tiempo de ejecución de la versión transaccional del programa.

TCR

El *Transaction Commit Rate (TCR)* [3] se define como el porcentaje de transacciones que efectúan el *commit* respecto a todas las transacciones ejecutadas (figu-

ra 3.19). A diferencia de otras métricas como *thoughput*, el *TCR* proporciona una medida de la efectividad de la concurrencia explotada por el sistema transaccional. En general, se espera que el incremento del número de hilos afecte negativamente esta medida, ya que al incrementar la contención, la tendencia natural es que aumente el número de transacciones abortadas respecto al total de las ejecutadas, reduciendo este factor.

$$\frac{committedXacts_{STM}(n\ threads)}{executedXacts_{STM}(n\ threads)}$$

Figura 3.19: Expresión correspondiente a la métrica *transaction commit rate*. La variable $committedXacts_{STM}$ es el número de transacciones que han realizado el *commit*, mientras que La variable $executedXacts_{STM}$ es el número total de transacciones que ha iniciado el sistema transaccional.

Sobrecarga de memoria

La sobrecarga de memoria o *memory overhead* es una métrica utilizada para determinar los requerimientos de memoria necesarios para la ejecución transaccional de una aplicación. En el caso de sistemas basados en una política de gestión de versiones *lazy*, como es el caso de los sistemas presentados en esta tesis, esta medida se relaciona con el tamaño del *write buffer*, el cual es el encargado de almacenar las escrituras tentativas de la transacción hasta su instante de *commit*. En el caso de sistemas basados en una política de gestión de versiones *eager* estaría relacionada con el tamaño del *undo-log*.

Esta métrica se ha utilizado en el capítulo 6 para evaluar el ahorro de memoria proporcionado por el soporte transaccional de reducción. Por ello, considerando que el peor de los casos implica el acceso a la totalidad de las estructuras de datos compartidas por parte de la transacción (lo que equivaldría a una aproximación de privatización total de dichas estructuras), la sobrecarga de memoria se expresa como el porcentaje de este consumo máximo requerido por el soporte transaccional.

4 Modelo de ejecución transaccional con orden de precedencia (TEPO)

Este capítulo analiza el problema de la paralelización de códigos que presentan restricciones de precedencia que deben ser preservadas para garantizar una solución correcta en su ejecución concurrente. La sección 4.1 analiza las principales situaciones con las que el programador se puede encontrar, así como las dificultades que impiden una paralelización trivial.

Seguidamente, proponemos un modelo de ejecución basado en memoria transaccional que permite paralelizar esta clase de códigos manteniendo las restricciones de precedencia definidas (sección 4.2). La naturaleza optimista y especulativa de la memoria transaccional facilita la máxima explotación de la concurrencia para estos códigos, que de otro modo se verían afectados por una alta serialización a fin de garantizar la corrección del resultado.

4.1. Códigos con restricciones de precedencia

Los códigos que presentan restricciones de precedencia son aquellos en los que las dependencias de datos existentes obligan a que las instrucciones se ejecuten en un determinado order, ya que en caso de ser alterado, el resultado podría resultar incorrecto. De acuerdo a esta definición, y salvo en contadas excepciones, estos códigos forman parte del núcleo de la mayoría de aplicaciones

existentes. De hecho, son la base del modelo de programación secuencial, aunque en este caso la ejecución de las instrucciones en estricto orden de precedencia está garantizada. El problema aparece cuando estos códigos son paralelizados para intentar obtener una mejora de rendimiento basada en la explotación de la concurrencia, ya que paralelizar código implica alterar el orden de ejecución original de las instrucciones del programa. Este problema se da en cualquiera de las diferentes formas de computación paralela, sin embargo, aquí nos centraremos en dos aproximaciones de alto nivel: paralelismo a nivel de tareas y paralelismo a nivel de datos.

El ***paralelismo a nivel de tarea*** es un paradigma de programación concurrente que consiste en asignar tareas, que realizan cálculos diferentes sobre cualquier conjunto igual o diferente de datos, a cada uno de los procesadores de un sistema de cómputo. De esta manera, dos tareas son independientes si el conjunto de datos utilizado es disjunto; sin embargo, si no lo es, se crea una relación de precedencia entre ellas, que restringe su orden de ejecución. Esta restricción de orden entre tareas viene determinada por la necesidad de una de ellas de disponer de algún valor computado en la otra, o de esperar a que se aplique una determinada etapa previa sobre los datos antes de proceder con la siguiente. En su modo más general, el paralelismo de tareas es representado mediante un grafo dirigido, cuyas aristas indican las relaciones de precedencia. Este grafo es subdividido en subgrafos que posteriormente son asignados a diferentes procesadores para su ejecución concurrente. Dado que las tareas dependientes no tienen más opción que ejecutar de manera ordenada (asegurando esto mediante mecanismos de sincronización), la manera en que se divida el grafo será la que determine la eficiencia del paralelismo resultante.

El ***paralelismo a nivel de datos***, consiste en asignar un conjunto igual o diferente de datos a cada procesador para que realicen los mismos cálculos sobre ellos en paralelo. Este tipo de paralelismo es típico en programas con bucles, los cuales conducen a menudo a secuencias de operaciones similares - aunque no necesariamente idénticas - en forma de iteraciones aplicadas sobre elementos de una gran estructura de datos. Este tipo de paralelismo está presente en muchas de las aplicaciones científicas y de ingeniería, debido a que es especialmente adecuado para cálculos sobre vectores y matrices, porque permiten aplicar la misma operación sobre cada uno de sus elementos. Idealmente, la ejecución simultánea de iteraciones de un bucle resulta en una aceleración global de la computación, sin embargo, al igual que en el paralelismo de tareas, esto solo ocurre cuando los conjuntos de elementos sobre los que trabaja cada procesador son disjuntos. Cuando las iteraciones del bucle (asignadas a diferentes procesadores) realizan accesos a datos compartidos se generan relaciones de dependencia que res-

tringen el orden de su ejecución. Por ello, la detección de estas dependencias, especialmente entre iteraciones sucesivas del bucle, se vuelve algo fundamental para maximizar la explotación del paralelismo. A continuación clasificamos los principales tipos de dependencia que pueden ocurrir entre dos iteraciones i_1 y i_2 (ejecutadas en ese orden) en un bucle [8]:

- **Dependencia de flujo (RAW)**: Sucede cuando i_2 lee una variable escrita en i_1. Este patrón de acceso constituye una verdadera dependencia de datos ya que establece una relación entre los datos en el flujo de ejecución secuencial. La ejecución paralela en este caso podría dar lugar a que la lectura de i_2 se realizase antes que la escritura de i_1, produciéndose la lectura de una valor incorrecto de la variable.

```
1 for (i=1; i<N; i++) {
2     A(i) = f(A(i-1));
3 }
```

- **Anti-dependencia (WAR)**: Sucede cuando i_2 escribe una variable leída en i_1. En una ejecución del bucle secuencial, este caso no constituye una verdadera dependencia, ya que la operación de escritura sucederá siempre después de la lectura. Sin embargo, esto no está garantizado en la ejecución paralela, que podría alterar el orden de ejecución y dar lugar a un resultado incorrecto.

```
1 for (i=0; i<(N-1); i++) {
2     A(i) = f(A(i+1));
3 }
```

- **Dependencia de salida (WAW)**: Sucede cuando tanto i_1 como i_2 escriben la misma variable. En este caso, el orden de las escrituras deben mantenerse ya que el valor que debe perdurar en memoria es el escrito por i_2. Una ejecución paralela, podría desordenar esta ejecución y realizar la escritura de i_1 en último lugar, siendo la que perduraría en memoria.

```
1 for (i=0; i<N; i++) {
2     A(i%n) = f(A(i));
3 }
```

Si se desea paralelizar un bucle, antes es necesario identificar las dependencias existentes entre iteraciones a fin de aplicar las técnicas adecuadas para explotar el máximo paralelismo. En el caso de que ninguna de las anteriores dependencias estén presentes, el bucle puede ser ejecutado en paralelo sin ningún

tipo de problema, ya que eso significa que cada iteración accede, tanto para lectura como para escritura, a subconjuntos de datos disjuntos del resto. Por otro lado, si las únicas dependencias entre iteraciones presentes en el bucle son anti-dependencias o dependencias de salida, la paralelización puede resolverse sin recurrir a primitivas de sincronización gracias al uso de técnicas como la privatización [56, 96, 97]. Sin embargo, la presencia de dependencias de flujo, por lo general, si que obliga a mantener la restricción de precedencia entre las iteraciones con el fin de garantizar un resultado consistente con la ejecución secuencial.

Un caso muy especial es el que constituyen los bucles de reducción, ya que generalmente puede presentar los tres tipos de dependencias entre iteraciones sobre una misma variable, y en los que sin embargo, las características del patrón del acceso y la propia naturaleza de las operaciones de reducción hacen posible reordenar las iteraciones sin afectar al resultado, siempre y cuando se mantenga la atomicidad de la operación de reducción. Debido a sus peculiaridades, esta tesis les dedicará el capítulo 6 para su estudio en profundidad.

Además de las reducciones, también existen otros ejemplos de códigos en los que el orden de algunas de las dependencias puede ser alterado produciendo múltiples resultados, siendo todos ellos válidos en la semántica del problema aunque no coincidan necesariamente con el resultado secuencial (el cual es solo una de las posible soluciones válidas). Estos tipos de códigos dan lugar a un conjunto relaciones de precedencia mucho menos restrictivas que favorecen el paralelismo, siempre que no nos importe obtener resultados diferentes al de una ejecución secuencial.

Centrándonos en códigos secuenciales, una amplia gama de técnicas y mecanismos han sido propuesto en la literatura para lograr su paralelización, especialmente en el caso de bucles con dependencias entre iteraciones, sin incurrir en una sincronización excesiva. Sin embargo, todas ellas basan su efectividad en la correcta detección de las dependencias por parte del programador, o el compilador. Este proceso actualmente no supone ningún problema en códigos regulares, en los que las estructuras de datos son simples (e.g, arrays, tablas) y que son accedidas de una forma directa (e.g, índices).

La dificultad se presenta cuando tratamos con códigos irregulares en los que se emplean estructuras de datos complejas, tales como punteros, listas enlazadas o arboles; o donde el acceso a los datos se realiza de manera indirecta, a través de indirecciones o mediante un patrón de acceso que no puede ser determinado hasta el tiempo de ejecución por venir impuesto por los datos de entrada, por ser computado por una fase previa del propio programa o simplemente por ser

```
1   for (i=0; i<5; i++) {
2       idx1 = ind1(i);
3       idx2 = ind2(i);
4       A(idx1) = f(A(idx2));
5   }
```

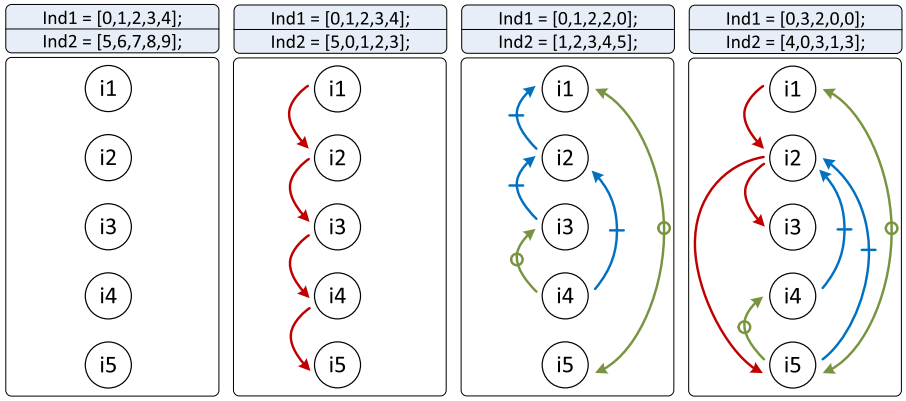


Figura 4.1: Problemática con los accesos irregulares. Las dependencias de flujo se representan mediante una flecha simple (en rojo), las anti-dependencias mediante una fecha marcada con una línea perpendicular (en azul) y las dependencias de salida con una fecha marcada con un círculo (en verde).

demasiado complejo para ser calculado en tiempo de compilación.

Para ilustrar esta situación, véase el caso simplificado presentado en la figura 4.1. El código muestra un bucle sencillo que opera sobre un array de datos, donde en cada iteración realiza la escritura en uno de sus elementos, indexado mediante un array de indirección `ind1`, de un valor computado a partir de otro elemento (indexado por el array de indirección `ind2`). Ambos arrays de indirección son tomados como entrada de la aplicación, por lo que son desconocidos hasta el tiempo de ejecución. Dependiendo del valor de los arrays de indirecciones introducidos al programa, el bucle puede ser totalmente paralelo (figura 4.1a), totalmente secuencial (figura 4.1b), presentar solo algunos tipos de dependencias entre iteraciones (figura 4.1c) o todas ellas (figura 4.1d).

Como hemos visto en este ejemplo, ninguna de las técnica estáticas clásicas puede ser aplicada en estas situaciones ya que se desconoce la naturaleza de las dependencias y, por tanto, la única manera segura de ejecutar el bucle es manteniendo la semántica del código secuencial original.

4.2. Nuestro modelo de ejecución transaccional

Para lidiar con las dificultades planteadas en la sección anterior, proponemos un modelo basado en TM, que denominamos *Transaction Execution under Precedence Order (TEPO)*, que ofrece un modelo de ejecución paralela definido para explotar la máxima concurrencia presente en la aplicación, al tiempo que oculta al programador las grandes complejidades derivadas del análisis de dependencias y la sincronización explícita de los hilos de ejecución.

El objetivo de nuestro modelo TEPO es explotar el paralelismo de secciones de código secuencial o aumentar la concurrencia de una sección de código paralela, recurriendo a un motor de memoria transaccional para tratar con las dependencias de datos y la sincronización de las diferentes partes del programa. TEPO consta de un gestor de ejecución transaccional que intenta maximizar la explotación del paralelismo mientras mantiene un orden de precedencia entre las transacciones; y un modelo de programación paralela definido para restar esfuerzos al programador.

El programador solo debe preocuparse de la definición de los segmentos o *chunks* de código (fragmentos del programa que serán ejecutados concurrentemente como una transacción) y su relación de orden de precedencia para establecer las dependencias de datos reales entre ellos. TEPO se encargará de planificar la ejecución de los *chunks* en forma de transacciones, manteniendo el orden de precedencia entre ellos al tiempo que explota la máxima concurrencia. TEPO garantiza el orden de precedencia realizando los cambios en memoria de aquellos *chunks* ya ejecutados siguiendo el orden establecido. El modelo de ejecución TEPO se ha construido sobre un motor de memoria transaccional que proporciona la funcionalidad básica para detectar y resolver los conflictos de memoria entre *chunks*.

4.2.1. *Chunks* de código y orden de precedencia

Tanto la definición de los *chunks* de código como el orden de precedencia tienen un importante impacto en la obtención de paralelismo. Si los *chunks* no son definidos apropiadamente, podrían exhibir una gran cantidad de conflictos de memoria, resultando en una ejecución optimista que podría ser drásticamente serializada. Por otro lado, el orden de precedencia entre *chunks* permite determinar el tipo de dependencia de datos en que se ven envueltos y que se corresponde con determinados conflictos transaccionales, lo cual a su vez afecta a la política de resolución de conflictos, la cual debe tomar la acciones necesarias

para degradar el rendimiento lo menos posible.

Podemos distinguir tres opciones diferentes para definir un orden de precedencia en la ejecución de transacciones, partiendo de una sección secuencial de programa:

1. ***Orden lexicográfico***, que se corresponde con el orden secuencial establecido por el programador cuando escribe el código fuente (las transacciones concurrentes obedecen este orden para asegurar que el programa paralelo es equivalente al secuencial).
2. ***Orden de flujo de datos***, que es un orden menos restrictivo que corresponde al orden impuesto por las dependencias de datos reales entre las operaciones de memoria (el tipo de dependencia de datos es derivado del orden lexicográfico).
3. ***Orden semántico***, que es un orden aún menos restrictivo asociado al orden impuesto por cualquier dependencia de datos existente en el problema, el cual puede ser alterado produciendo un resultado válido, aunque no coincida con el de la ejecución secuencial.

Como ejemplo, tómese la situación en la que un programador ha definido cada iteración individual de un bucle *DOALL* como *chunks* de código. El orden de precedencia lexicográfico es impuesto por la variable que indexa el bucle, mientras que, sin embargo, no existe ningún orden de precedencia de flujo de datos, ya que las iteraciones son independientes. Por otro lado, en el caso de un bucle de reducción escalar, los ordenes lexicográfico y de flujo de datos son el mismo, pero no existe ningún orden de precedencia semántico (el resultado final de la reducción es independiente del orden de ejecución de las iteraciones, debido a las propiedades asociativas y conmutativas del operador de reducción).

El programador debe decidir qué orden de precedencia debe ser tenido en cuenta. Si el objetivo es paralelizar una sección de programa secuencial, la opción más simple es aplicar un orden lexicográfico puesto que no requiere de ningún análisis de dependencia de datos. La información semántica sobre el problema también puede ser utilizada por el programador para reducir las restricciones de precedencia. Por ejemplo, puede declarar que el código secuencial (e.g, un bucle) incluye operaciones de reducción sobre algunas variables. Por otro lado, hay una clase de aplicaciones (algunos algoritmos de grafos, técnicas de segmentación de imagen, algoritmos de *clustering*, ...) donde el orden de las instrucciones puede ser alterado sin producir una versión paralela incorrecta, sino una solución diferente, también válida. En tales aplicaciones, el programador puede elegir diferentes posibilidades en cuanto al orden de precedencia.

Las siguientes definiciones formalizan los conceptos descritos anteriormente. Estas definiciones tienen por objeto definir el modelo de programación en TEPO.

Definición 4.2.1 Un *chunk* de código es una unidad de ejecución transaccionable del programa, en que se divide una sección de código del programa. El conjunto de *chunks* para una sección de código R lo representamos como $U = \{u_0, u_1, u_2, ...\}$. ■

El concepto de *chunk* es comúnmente usado en el contexto de TM [78] para referirse a bloques atómicos de instrucciones. Estos trabajos asumen que un núcleo de procesamiento ejecuta *chunks* continuamente. Sin embargo, nosotros consideramos que la ejecución de un *chunk* es definida solo para una o varias regiones de código de interés en el programa.

Una vez que el conjunto U es definido, se puede establecer el orden de precedencia entre *chunks* como puntualizan las siguientes dos definiciones.

Definición 4.2.2 La relación de *orden de precedencia inmediata* ($\prec_i$) es aquella definida por el programador en base a algún criterio, considerando que un *chunk* nunca se precede a sí mismo. ■

Definición 4.2.3 La relación de *orden de precedencia* ($\prec$) es el cierre transitivo de la precedencia inmediata:

$$u, v \in U, u \prec v \iff u \prec_i v, \text{ or } \exists a_0, a_1, ...a_n \in U \mid u \prec_i a_0, a_0 \prec_i a_1, ..., a_{n-1} \prec_i a_n, a_n \prec_i v.$$

■

Nótese que una relación de precedencia válida ($\prec$) debe ser, al menos, una relación estricta de orden parcial, es decir, debe cumplir las propiedades irreflexiva, asimétrica y transitiva. Esto implica que $\prec_i$ debe estar libre de ciclos.

4.2.2. Modelo de ejecución transaccional

Nuestra propuesta consiste en un modelo de planificación y ejecución de *chunks* que extiende a un sistema TM para cumplir con la relación de orden de precedencia definida por el programador o el compilador. Las dos principales características de este modelo son: (1) la ejecución de dos *chunks* sujetos al orden de precedencia mantiene una semántica secuencial equivalente al código original, es decir, el resultado es el mismo que ejecutar ambos *chunks* uno detrás del otro

siguiendo el orden de programa; y (2) los *chunks* son ejecutados atómicamente y de manera aislada, apoyándose en el sistema TM de base para la gestión de conflictos y versiones.

La relación de precedencia está dirigida a dos objetivos: En primer lugar, puede ayudar a paralelizar una región de código secuencial. Sin ningún conocimiento sobre el problema, el programador puede definir un orden de precedencia equivalente al orden lexicográfico, asegurando la corrección de la versión paralela. Sin embargo, se puede incrementar la concurrencia si el programador relaja las restricciones de precedencia usando información semántica y de flujo de datos del programa. En segundo lugar, puede ser usado para mejorar la concurrencia explotada de una región de código paralela. Nuestro modelo utiliza el orden de precedencia entre transacciones para hacer más eficiente la política de resolución de conflictos del sistema base. De esta manera, mediante el orden de precedencia, el programador puede interactuar con el mecanismo de resolución de conflictos.

Con el fin de proceder con al ejecución de una sección de código seleccionada bajo el modelo TEPO, se deben especificar los siguientes aspectos: (1) cómo se divide la sección de código en *chunks*; (2) el grafo de orden de precedencia entre los *chunks*; (3) el grupo de hilos a cargo de ejecutar los *chunks*; (4) la función encargada de asignar los *chunks* a los hilos.

Definición 4.2.4 La función de mapeo, o, asigna *chunks* a los hilos, tal que, $o : U \rightarrow T$, siendo T el conjunto de hilos, $T = \{0, 1...nThreads\}$ ($nThreads$ es la cardinalidad del conjunto de hilos). De aquí en adelante, la notación U_t será utilizada para describir aquellos *chunks* en un conjunto U asignados al hilo t, esto es, $U_t = \{u \in U \mid o(u) = t\}$. ■

Una vez se ha fijado todo lo anterior, TEPO ejecuta concurrentemente los *chunks* como transacciones normales, salvo por dos principales diferencias: (1) el gestor de conflictos toma ventaja del orden de precedencia para optimizar la resolución de conflictos; y (2) los *chunks* son planificados en hilos de acuerdo al grafo de orden de precedencia. Ambos aspectos serán discutidos en detalle en las secciones 4.2.3 y 4.2.4.

Cada hilo comienza ha ejecutar los *chunks* correspondientes, asignados de acuerdo al orden de precedencia. Inicialmente, se selecciona arbitrariamente uno de los *chunks minimal* [1] en su conjunto asignado (*chunk* con menor orden de precedencia que todavía no se ha ejecutado). Mientras el *chunk* se ejecuta

[1] Un *chunk* $a \in Y$ es *minimal* si $\nexists b \in Y / b \prec a$

transaccionalmente, el sistema TM subyacente registra las direcciones de memoria leídas y escritas (*read set (RS)* y *write set (WS)* respectivamente), necesarias para asegurar la atomicidad y el aislamiento. De esta manera, el sistema TM vigila la ocurrencia de conflictos de datos entre *chunks* concurrentes siendo ejecutados por el grupo de hilos. Tan pronto como un conflicto es detectado, el gestor de conflictos decide que acciones tomar para resolver la situación, lo cual normalmente implica abortar una de las transacciones.

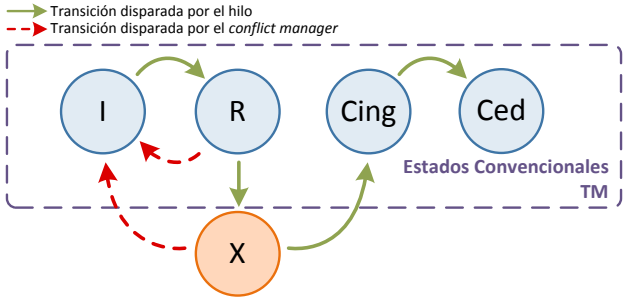
Además de trazar las direcciones de memoria accedidas, el sistema transaccional debe encargarse de las modificaciones en memoria (gestión de versiones). TEPO maneja las actualizaciones en memoria siguiendo una política *lazy*, manteniendo las escrituras transaccionales en un almacenamiento local (*write buffer (WB)*), mientras que se preservan los datos originales en memoria principal. Sin embargo, en un sistema TM con atomicidad fuerte, una gestión de versiones *eager* también sería posible.

Cuando un hilo finaliza la ejecución de una *chunk*, se comprueba si puede realizar el *commit* (ver sección 4.2.3 para condición de *commit*), asegurándose de mantener el orden de precedencia. Si el *chunk* realiza el *commit*, el hilo comienza inmediatamente la ejecución de un nuevo *chunk*, seleccionado de entre conjunto asignado y siguiendo con el orden de precedencia establecido. En caso contrario, se realiza un *checkpoint* del último *chunk* ejecutado (salvando *RS, WS* y *WB*) para mantenerlo en un estado *ejecutado-sin-commit*. Seguidamente, el hilo comienza la ejecución de un nuevo *chunk* (el siguiente orden de precedencia). Además, los *read set* y *write set* del *chunk* sobre el que se ha realizado el *checkpoint* se siguen manteniendo activos, ya que sus modificaciones sobre los datos aún no se han confirmado en memoria, y por tanto, aun pueden provocar conflictos con otras transacciones en ejecución (del mismo o de diferente hilo).

Bajo el modelo de ejecución TEPO, los *chunks* se ejecutan como transacciones y se pueden encontrar en uno de los estados descritos en la figura 4.2. A diferencia de los sistemas transaccionales convencionales, en TEPO, el final de la ejecución de un *chunk* y su *commit* son estados desacoplados.

Un *chunk* ya ejecutado puede realizar el *commit* inmediatamente o ser retirado (tomando un *checkpoint*) de manera que el hilo sea libre para iniciar nuevos *chunks*. El hilo queda a cargo de determinar periódicamente si los *chunks* pendientes pueden efectuar su *commit* o no, en algún momento posterior.

La figura 4.2 muestra el diagrama de estados por los que transita un *chunk* a lo largo de su vida y que describe el comportamiento de TEPO. Todos los *chunks* asignados a un hilo se encuentran inicialmenten en el estado `idle (I)`. En un momento dado, el hilo comienza la ejecución de uno de sus *chunks*, con lo que



(a) Diagrama de transición de estados

Estado	Descripción
I (Idle)	El *chunk* está esperando ser ejecutado
R (Running)	El *chunk* está en ejecución
X (eXecuted)	El *chunk* está ejecutado y listo para el *commit*
Cing (Committing)	El *chunk* está en proceso de *commit*
Ced (Committed)	El *chunk* ha finalizado el *commit*

(b) Descripción de los estados

Figura 4.2: Diagrama de estados del *chunk*

pasará al estado `running (R)`. Cuando el *chunk* finaliza su ejecución, transita al nuevo estado `executed (X)` propuesto como parte del modelo TEPO, para que el hilo compruebe la condición de *commit* asociada (ver siguiente sección). Si se satisface, el hilo comienza el proceso de *commit* del *chunk* (estado `committing (Cing)`) que una vez concluya la actualización en memoria principal de sus escrituras transaccionales, transitará al estado `committed (Ced)`. Si la condición de *commit* no se cumple, el *chunk* se mantiene en el estado `executed (X)` y el hilo inicia un nuevo *chunk*.

Nótese que este comportamiento es posible gracias al desacoplo entre ejecución y *commit* que ofrece el nuevo estado transaccional `executed (X)`. Cada vez que la ejecución de un *chunk* finaliza, el hilo comprueba de nuevo la condición de *commit* para todos los *chunks* en estado `X` (en su correspondiente orden de precedencia) y efectúa el *commit* de aquellos que lo cumplan. Hay que destacar que los hilos son los encargados de avanzar el estado de los *chunks* a lo largo de toda su vida (desde el `idle` hasta `committed`, estado por estado), mientras que el gestor de conflictos puede hacerles volver a un estado anterior si se detecta un conflicto. El diagrama de estados asume que un conflicto se resuelve abortando el *chunk* y devolviéndolo al estado `idle`.

4.2.3. Gestor de conflictos

La gestión de los conflictos está compuesta principalmente por dos operaciones: *detección* y *resolución*. La detección de conflictos se refiere a la identificación de accesos de datos concurrentes que ponen en peligro la consistencia del sistema, por parte de *chunks* en ejecución (estado `R`) y/o ejecutados (estado `X`) del mismo o diferentes hilos. La resolución de conflictos se refiere a las acciones tomadas para resolver dicha situación de inconsistencia (asegurando la atomicidad de los *chunks*). En TEPO, el sistema transaccional de base realiza ambas operaciones, detección y resolución, siguiendo una política *eager*, es decir, tan pronto como se producen los accesos conflictivos a la localización de memoria.

La acción común en estos casos para resolver el conflicto consisten en abortar uno de los *chunks* involucrados (haciéndolo volver a estado `I`) y reiniciando su ejecución. Normalmente, la acción de aborto tiene un importante impacto en el rendimiento, puesto que todo el trabajo realizado se pierde y debe ser rehecho. Para reducir este impacto negativo y mejorar la concurrencia entre *chunks*, TEPO toma ventaja de la relación de orden de precedencia definida para el conjunto de *chunks* para introducir dos nuevas acciones: detener la transacción (*stalling*) y adelantar datos (*data forwarding*).

Con el fin de describir la resolución de conflictos en TEPO, se deben definir los siguientes conjuntos:

Definición 4.2.5 Denotando como U_S el conjunto de todos los *chunks* en el estado S (siendo S uno de los estados `I, R, X, Cing` o `Ced`) en todos los hilos definimos, para un determinado *chunk*, los siguientes conjuntos de predecesores (*prev*), de sucesores (*next*) o elementos no relacionados (*norel*), de la siguiente manera,

$$\begin{aligned}
&\text{prev}_S(u) = \{v \in U_S \mid v \neq u, v \prec u\} \\
&\text{next}_S(u) = \{v \in U_S \mid v \neq u, u \prec v\} \\
&\text{norel}_S(u) = \{v \in U_S \mid v \neq u, u \not\prec v, v \not\prec u\} \\
&\text{prev}(u) = \bigcup_{U_S} \text{prev}_S(u);\ \text{next}(u) = \bigcup_{U_S} \text{next}_S(u);\ \text{norel}(u) = \bigcup_{U_S} \text{norel}_S(u)
\end{aligned}$$

Siendo consistente con al definición 4.2.4, la notación $\text{norel}_S(u)_t = \text{norel}_{S_t}(u)$, restringe el conjunto para aquellos *chunks* asignados al hilo t. Lo mismo es aplicable a las funciones prev() y next(). ■

La tabla 4.1 muestra las acciones que la política de resolución de conflictos en TEPO realiza cuando se detecta una dependencia de datos entre accesos concu-

Tabla 4.1: Resolución de conflictos en TEPO: Hilo t está ejecutando el *chunk* u que accede a la localización de memoria m en un momento dado; v es otro *chunk* en U, $v \neq u$, que accedió a m previamente ($RS(v)$: Read Set, $WS(v)$: Write Set)

	u lee m		u escribe m	
	$m \in RS(v)$	$m \in WS(v)$	$m \in RS(v)$	$m \in WS(v)$
$\forall v \in \text{norel}_R(u)$	Sin acción	(1) *Abort* u	(1) *Abort* u	(1) *Abort* u
$\forall v \in \text{prev}_R(u)$	Sin acción	(2) *Abort* u, *Stall* u, *Forward* $v \rightarrow u$	(3) Sin acción	(4) *Commit* v antes que u
$\forall v \in \text{next}_R(u)$	Sin acción	(5) Sin acción	(6) *Abort* v	(4) *Commit* u antes que v
$\forall v \in \text{norel}_X(u)$ $(o(v) \neq t)$	Sin acción	(1) *Abort* u	(1) *Abort* u	(1) *Abort* u
$\forall v \in \text{norel}_X(u)$ $(o(v) = t)$	Sin acción	(7) *Forward* $v \rightarrow u$	(7) Sin acción	(7) *Commit* v antes que u
$\forall v \in \text{prev}_X(u)$ $(o(v) \neq t)$	Sin acción	(2) *Abort* u, *Stall* u, *Forward* $v \rightarrow u$	(3) Sin acción	(4) *Commit* v antes que u
$\forall v \in \text{prev}_X(u)$ $(o(v) = t)$	Sin acción	(2) *Forward* $v \rightarrow u$	(3) Sin acción	(4) *Commit* v antes que u
$\forall v \in \text{next}_X(u)$ $(o(v) \neq t)$	Sin acción	(5) Sin acción	(6) *Abort* v	(4) *Commit* u antes que v
$\forall v \in \text{next}_X(u)$ $(o(v) = t)$	Imposible	Imposible	Imposible	Imposible
$\forall v \in \text{norel}_{Cing}(u)$ $(o(v) \neq t)$	Sin acción	*Abort* u, *Stall* u, *Forward* $v \rightarrow u$, Lee* m	*Abort* u, *Stall* u	*Abort* u, *Stall* u
$\forall v \in \text{norel}_{Cing}(u)$ $(o(v) = t)$	Imposible	Imposible	Imposible	Imposible
$\forall v \in \text{prev}_{Cing}(u)$ $(o(v) \neq t)$	Sin acción	(2) *Abort* u, *Stall* u, *Forward* $v \rightarrow u$, Lee* m	(3) Sin acción	(4) *Stall* u, *Commit* v antes que u
$\forall v \in \text{prev}_{Cing}(u)$ $(o(v) = t)$	Imposible	Imposible	Imposible	Imposible
$\forall v \in \text{next}_{Cing}(u)$	Imposible	Imposible	Imposible	Imposible

* Solo si m ya ha sido actualizada en memoria (*Cing* atómico dirección por dirección)

rrentes. En algunos casos, la política puede elegir entre varias posibles acciones. Estas acciones son definidas con el objetivo de maximizar la concurrencia en la ejecución de *chunks* asignados a hilos diferentes.

La política de resolución de conflictos depende de la condición de *commit* para los *chunks* sujetos a una relación de orden de dependencia. Definimos una condición de *commit* de la siguiente manera.

Definición 4.2.6 *Chunks* concurrentes sujetos a un orden de precedencia ($\prec$) cumplen con la condición de *commit* estricta si: Dado $u \in U$, u en el estado X, u transita al estado *Cing* si $\forall v \in U, v \prec u$, v está en el estado *Ced*. ■

Todas las acciones con la etiqueta (1) en la tabla 4.1 corresponden al comportamiento por defecto el sistema transaccional de base (i.e, *chunks* no ordenados siendo ejecutados por hilos diferentes). Por simplicidad, asumimos que la acción por defecto es el aborto (figura 4.3a). Acciones etiquetadas como (2) corresponden a conflictos de datos cuando un *chunk* u lee m, la cual fue previamente escrita por un *chunk* v tal que $v \prec u$ (dependencia de tipo *RAW*). En este caso, la condición de *commit* estricta asegura que u siempre realizará su *commit* después de v. De este modo el conflicto puede ser resuelto abortando/retrasando u o adelantando el dato desde v hasta u (figura 4.3b). Sin embargo, si ambos *chunks* son asignados al mismo hilo, la única acción posible es el adelantamiento (*forwarding*) de datos, ya que de lo contrario, el hilo entraría en *deadlock* (figura 4.3g). Por otro lado, si v se encuentra realizando su *commit*, es posible para u leer el dato en memoria si la actualización ya se ha efectuado (considerando un estado `Cing` no atómico).

Las acciones con la etiqueta (3) corresponden a una dependencia de tipo *WAR* (u escribe y v lee una misma dirección m, obligando a v a realizar su *commit* antes que u). La condición de *commit* estricta permite filtrar esta clase de conflictos (*no action*), ya que mientras se cumpla (y sabemos que lo hará) la ejecución concurrente será siempre correcta (figura 4.3c).

La etiqueta (4) marca las acciones correspondientes para las dependencias de tipo *WAW* (tanto u como v escriben en m y un *chunk* precede al otro). La condición de *commit* estricta asegura que la escritura que perdurará en memoria, será la correspondiente al último *chunk* según el orden de precedencia (figura 4.3d).

Las acciones etiquetadas como (5) también corresponden a las dependencias de tipo *WAR* porque el *chunk* que efectúa la lectura (u) debe realizar el *commit* primero (figura 4.3e). Así, estos conflictos se resuelven de manera similar al caso (3). Por su parte, las acciones con la etiqueta (6) corresponden a dependencias de tipo *RAW* en la que el *chunk* escritor (u) debe efectuar el *commit* en primer

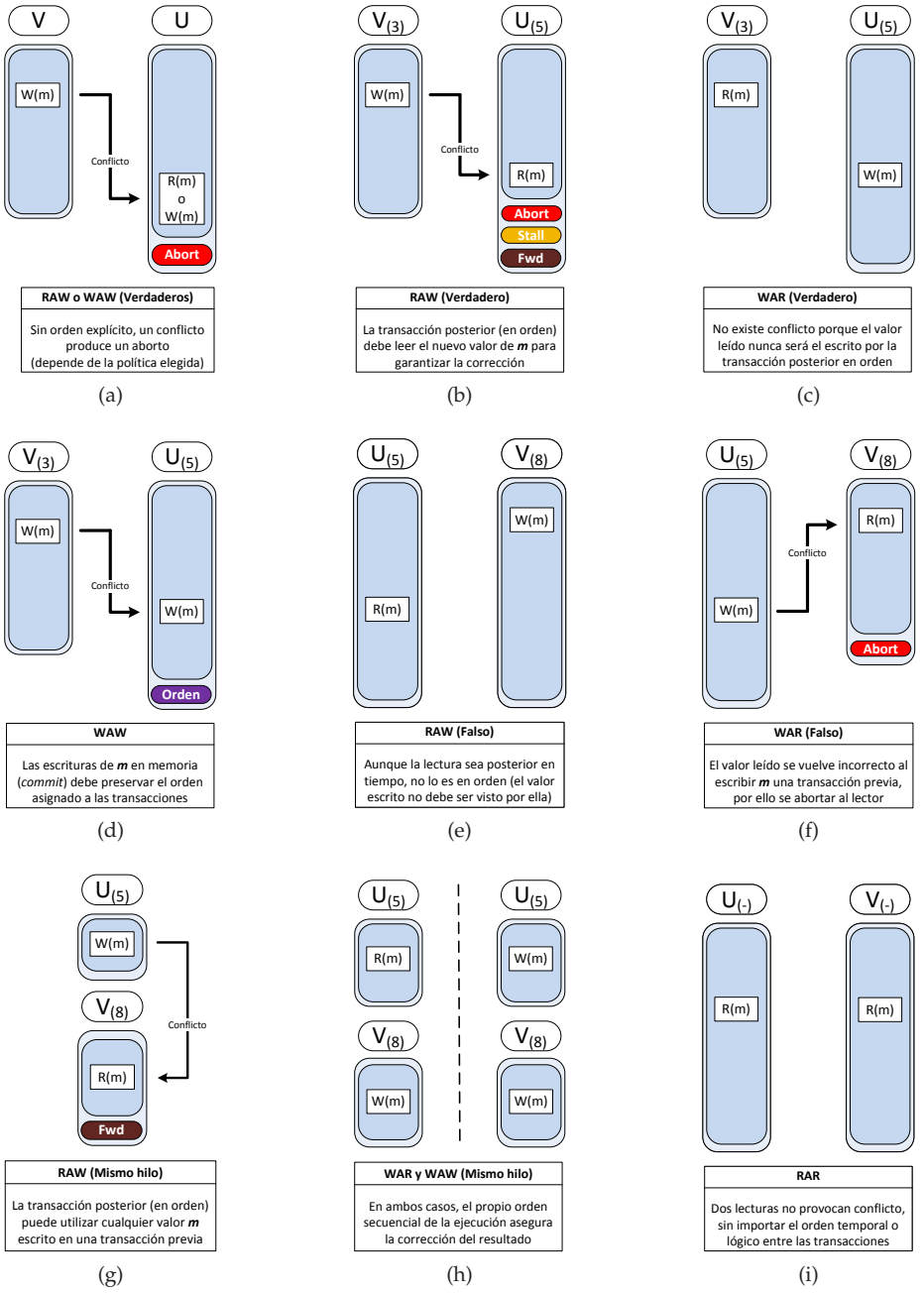


Figura 4.3: Escenarios destacados en la resolución de conflictos en TEPO (tabla 4.1). La precedencia entre las transacciones queda definida por el identificador numérico de cada transacción. Transacciones sin identificador númerico se consideran no relacionadas por un orden de precedencia

lugar (figura 4.3f). En estos casos, como la lectura en v fue realizada antes temporalmente que la escritura en u, la única acción posible es abortar v, ya que el valor que posee no se corresponde con el valor real que tendría en una ejecución secuencial (el escrito por u). Finalmente, las acciones la etiqueta (7) denota las acciones correspondientes al caso en que *chunks* no relacionados asignados al mismo hilo, accedan la misma dirección m. Por defecto, los chunks realizan el *commit* en el mismo orden en que fueron ejecutados (figura 4.3h).

De la anterior tabla se puede deducir que un *chunk* ejecutado v (en estado `X`) puede ser abortado únicamente por un *chunk* u precedente que esté en ejecución (u en estado `R`, $u \prec v$). Este hecho permite relajar la condición de *commit*.

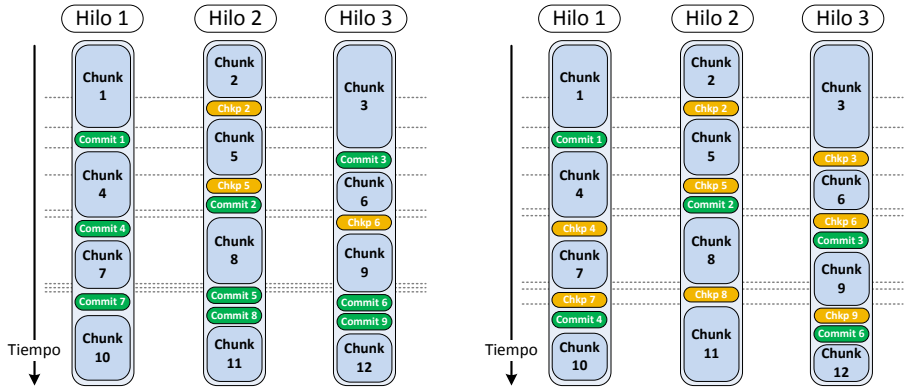
Definición 4.2.7 *Chunks* concurrentes sujetos a un orden de precedencia ($\prec$) satisfacen la condición de *commit* relajada si: Dado $u \in U$, u en el estado X, u transita al estado $Cing$ si $\forall v \in U$, $v \prec u$ y $WS(u) \cap WS(v) = \emptyset$, v se encuentra en el estado X, $Cing$ o Ced. ■

Esta condición relajada permite realizar *commits* fuera de orden (respecto al orden de precedencia definido por el programador) y, por tanto, aumentar la concurrencia entre *chunks* en algunas situaciones. La figura 4.4 ilustra un escenario simple de tres hilos ejecutando 12 *chunks* sujetos a una relación de orden de precedencia total [2] donde se aplican los dos tipos de condiciones de *commit*.

En la figura 4.4a se muestra cómo los *chunks* pueden finalizar fuera de orden, gracias a la condición de *commit* relajada. En ella puede apreciarse que el *chunk 3* realiza el *commit* antes que el *chunk 2*, lo cual es posible porque tanto el *chunk 1* (estado `Ced`) como el *chunk 2* (estado `X`) que le preceden han finalizado su ejecución.

En contraposición, la condición estricta (figura 4.4b) serializa los *commits* de los *chunks*, lo cual asegura la corrección pero reduce la concurrencia. Nótese que la condición de *commit* relajada no es aplicable a aquellos *chunks* sujetos a dependencias de tipo *WAW*, como se muestra en la última columna de la tabla 4.1, debido a que en este caso, el orden estricto entre escrituras debe ser preservado para la obtención de un resultado correcto.

[2] Esto significa una relación de orden, en la que se satisface totalmente que: $\forall a \neq b$, o bien $a \prec b$, o $b \prec a$.



(a) Condición de *commit* relajada (b) Condición de *commit* estricta

Figura 4.4: Escenario de comparación entre aplicar una condición de *commit* relajada y estricta

4.2.4. Planificación de transacciones

El planificador de TEPO está a cargo de planificar y ejecutar, como transacciones, los *chunks* asignados a cada hilo, además de iniciar el procedimiento de *commit* en aquellos *chunks* que cumplan con su condición de *commit* (estricta o relajada). La condición de *commit* se define de manera que el orden de precedencia se preserve, tal y como se indica en la siguiente definición.

Definición 4.2.8 La planificación de *chunks* limitados por un orden de precedencia se define como:

(1) *Planificación intra-thread*: Aquellos *chunks* asignados al mismo hilo comienzan su ejecución de acuerdo al orden de precedencia, de manera que si $u, v \in U_t$ y $u \prec v$ entonces el hilo t comienza la ejecución de u antes que v.

(2) *Planificación inter-thread*: Aquellos *chunks* asignados a diferentes hilos realizan su *commit* tan pronto como sea posible de acuerdo a su condición de *commit*, ya sea estricta o relajada. ■

La planificación de las transacciones en TEPO es una operación totalmente distribuida, diseñada para minimizar los costes asociados a la planificación y sincronización, además de para maximizar la concurrencia en la ejecución de *chunks*. Esta operación fue esbozada al final de la sección 4.2.2 y se describe en detalle en el algoritmo 1 junto con las tablas 4.2 y 4.3.

Algorithm 1 Planificación de transacciones para un determinado hilo

```
1  while chunks_avail(mytid) do ▷ Mientras existan chunks asignados al hilo (mytid) que
                                  no han efectuado el commit
2      u = get_chunk(I, mytid)
3      if u ≠ ∅ then              ▷ Si hay chunks pendientes en estado I
4          TX_begin()
5          Execute u              ▷ Tanto la transacción en ejecución como las previas
                                    en espera de commit del hilo pueden abortar aquí. En
                                    ese caso, la mínima transacción de las abortadas debe
                                    reiniciar la ejecución.
6          TX_checkpoint()        ▷ u transita al estado X
7      end if
8      while X_mytid ≠ ∅ do       ▷ Mientras el hilo tenga chunks en estado X
9          e = get_chunk(X, mytid) ▷ Recorrerlo en orden de precedencia
10         if TX_commit_check(e) then
11             TX_commit(e)
12         else
13             break              ▷ Sale del bucle while. Evita el bloqueo del hilo.
14         end if
15     end while
16 end while
```

Tabla 4.2: Planificación en TEPO: Primitivas transaccionales

Primitiva	Transición de estados	Acción
TX_begin()	$I \rightarrow R$	Reserva nuevos RS, WS, WB
TX_checkpoint()	$R \rightarrow X$	Salva los actuales RS, WS, WB
TX_commit_check()	-	Comprueba la condición de *commit*
TX_commit()	$X \rightarrow Cing \rightarrow Ced$	Realiza el *commit* del *chunk*, libera RS, WS, WB

RS: Read Set; WS: Write Set; WB: Write Buffer

Tabla 4.3: Planificación en TEPO: Funciones auxiliares

Función	Descripción
chunks_avail(t)	Devuelve *false* si todos los *chunks* asignados a t han realizado el *commit*; En caso contrario, devuelve *true*
get_chunk(A,t)	Devuelve el *chunk minimal** en estado A del hilo t; Si existen varios disponibles, se selecciona uno arbitrariamente; En otro caso, devuelve $\emptyset$

*Un *chunk* $a \in Y$ es *minimal* si $\nexists b \in Y / b \prec a$

5 Aplicación del modelo TEPO a códigos iterativos

Este capítulo está centrado en la aplicación de nuestro modelo TEPO a códigos iterativos, en los cuales se establece una relación de orden total entre los *chunks* del programa. El capítulo comienza presentando como aplicación motivadora y caso de estudio, un algoritmo para la transposición de una matriz dispersa (sección 5.1). Este código presenta un patrón de accesos irregular que lo hace especialmente conveniente para ser paralelizado utilizando las abstracciones TM, debido a que las dependencias de datos vienen determinadas por la matriz de entrada, lo que dificulta su análisis en tiempo de compilación. Este tipo de aplicaciones demandan un importante esfuerzo por parte del programador para desarrollar una versión paralela optimizada del código. En estos casos, la memoria transaccional ayuda a simplificar el trabajo de programación durante la paralelización del código, permitiendo obtener una versión paralela competitiva en términos de rendimiento.

En la siguiente sección, describimos las características del algoritmo de transposición de matriz dispersa. La sección 5.3 presenta la implementación de nuestro modelo TEPO como soporte transaccional para la paralelización de códigos iterativos con dependencias no resueltas estáticamente. Finalmente, la sección 5.4 presenta y compara los resultados obtenidos.

5.1. Caso de estudio: Transposición matriz dispersa

Como caso de estudio y aplicación motivadora se ha considerado el algoritmo de ***transposición de matriz dispersa*** propuesto por S. Pissanetzky [75], el cual es considerado uno de los algoritmos secuenciales más eficientes para la transposición dispersa. En él, se trabaja con matrices almacenadas en formato CRS [1] (*Compressed Row Storage*) [6], que es un esquema de representación de matrices dispersas ampliamente utilizado y eficiente en cuanto al espacio requerido para la compresión de la información [94]. La transposición de matriz dispersa presenta un patrón de acceso a memoria irregular que depende de la matriz de entrada, y por tanto, sus dependencias no pueden ser conocidas antes de su ejecución. Como puede observarse, este código representa exactamente ese gran bloque de construcciones iterativas con restricciones de orden descritas en el capítulo 4, susceptibles de ser paralelizadas aplicando nuestro modelo TEPO.

La figura 5.1 muestra el código secuencial de este algoritmo utilizando sintaxis Fortran. Después de una fase de inicialización, de complejidad $O(Nrows + Ncols)$, donde los contadores de fila de la matriz transpuesta son calculados, la transposición en sí misma tiene lugar (complejidad $O(NrowsxNcols)$). Observar que el código contiene patrones de indexación complejos con hasta tres niveles de indirección en el lado izquierdo de algunas sentencias de asignación. En el caso general, las dependencias entre iteraciones serán determinadas por la distribución de los elementos distintos de cero de la matriz a transponer. Por ejemplo, en el caso de una matriz puramente diagonal, no existiría ninguna dependencia presente.

Analizando las tres sentencias dentro del doble bucle (líneas 18-20) podemos observar varias fuentes de dependencias. Primero, una dependencia *loop-carried* es causada por el índice `ROWT(COLUMN(j) + 1)` en la actualización de los vectores `DATAT` y `COLUMNT`. Ambos arrays son indexados por `ROWT` cuyo valor puede haber sido calculado en una iteración diferente de la que lo está utilizando. Una segunda dependencia *loop-carried* es dada por la operación de reducción en `ROWT` dentro del bloque de transposición (línea 20), puesto que el elemento es leído y

[1] Una matriz dispersa de tamaño $Nrows \times Ncols$ con $Nnzeros$ elementos distintos de cero se representa en formato CRS utilizando tres vectores lineales: `COLUMN[Nnzeros]`, `DATA[Nnzeros]` y `ROW[Nrows+1]`. `DATA` contiene, fila por fila, los valores distintos de `null`, `COLUMN` contiene los índices correspondientes a las columnas, y cada elemento de *ROW* apunta al índice en `COLUMN` donde comienza la lista de elementos distintos de cero. Por ejemplo, $\begin{bmatrix} 0 & a & b \\ c & 0 & d \\ 0 & 0 & e \end{bmatrix}$ se almacenará como `ROW=[1, 3, 5, 6]`, `COLUMN=[2,3,1,3,3]`, `DATA=[a,b,c,d,e]`, considerando que todos los índices comienzan en 1. Nótese que `ROW[Nrows+1]=Nnzeros+1`.

```
1 c --- Initialization
2 do i=2, NCOLS+1
3   ROWT(i)=0
4 enddo
5
6 do i=1, ROW(NROWS+1)-1
7   ROWT(COLUMN(i)+2) = ROWT(COLUMN(i)+2)+1
8 enddo
9
10 ROWT(1)=1; ROWT(2)=1
11 do i=3, NCOLS+1
12   ROWT(i) = ROWT(i)+ROWT(i-1)
13 enddo
14
15 c --- Transposition
16 do i=1,NROWS
17   do j=ROW(i), ROW(i+1)-1
18     DATAT(ROWT(COLUMN(j)+1)) = DATA(j)
19     COLUMNT(ROWT(COLUMN(j)+1)) = i
20     ROWT(COLUMN(j)+1) = ROWT(COLUMN(j)+1)+1
21   enddo
22 enddo
```

Figura 5.1: Código secuencial del algoritmo de transposición de matriz dispersa de Pissanetzky en estilo FORTRAN.

escrito en la misma sentencia. Una posible tercera fuente de dependencia es la escritura en posiciones de `DATAT` y `COLUMNT`. Debido a que ambas están indexadas mediante un patrón de acceso irregular, dos iteraciones diferentes podrían potencialmente escribir en la misma localización de memoria. No obstante, con un conocimiento profundo acerca del algoritmo, el programador podría determinar que esta dependencia no es relevante, ya que cada posición de `DATAT` y `COLUMNT` solo se escribe una sola vez en memoria. Esto se debe a que cada elemento en la matriz de entrada aparece una única vez en la transpuesta. Sin embargo, una matriz mal formada en formato CRS podría romper esta hipótesis, generando una nueva fuente de conflictos. Por ello, una paralelización simple debería garantizar el acceso atómico a las localizaciones de memoria de los arrays dentro del bucle, así como mantener el orden entre iteraciones durante la escritura a una determinada posición. En la práctica, esto implicaría un alto grado de serialización.

Una aproximación bien conocida que resuelve la paralelización de este código consiste en aplicar una estrategia de particionamiento de datos [7]. El código de la figura 5.2 esboza esta posibilidad. Básicamente, la matriz dispersa de entra-

```
1 c --- Input matrix: A = {ROW, COLUMN, DATA}
2 c --- (Local) submatrix  for thread id:
3 c ---   Aloc_id = {ROWloc_id,
4 c ---   COLUMNloc_id, DATAloc_id}
5 c --- Transposed matrix: A^T = {ROW^T, COLUMN^T, DATA^T}
6 C$OMP PARALLEL
7   id = omp_get_thread_num()
8   Nthreads = omp_get_num_threads()
9
10  c --- Partition into submatrices
11  call sp_partition(A, Aloc_id, Nthreads)
12
13  c --- Local transposition
14  call transpose(Aloc_id, Aloc_id^T)
15
16 C$OMP BARRIER
17 C$OMP MASTER
18  c --- Gather transposed submatrices
19  call sp_join(Aloc_id^T, A^T, Nthreads)
20 C$OMP MASTER
21
22 C$OMP END PARALLEL
```

Figura 5.2: Pseudocódigo de la versión paralela de la transposición de matriz dispersa utilizando una estrategia de particionamiento de datos (*data parallel*) en estilo FORTRAN.

da necesita ser dividida en submatrices más pequeñas, también dispersas, que son distribuidas entre los hilos de ejecución. En el código presentado se asume que existen tantas submatrices como hilos disponibles. Esta división puede ser tan compleja como se desee (por bloques, cíclica, ...) e implica una traducción de los índices de acceso, desde la estructura CRS de la matriz de entrada original a las nuevas submatrices. Una división simple puede consistir en partir la matriz por filas, con todas las submatrices del mismo tamaño. En este caso, el procedimiento de división puede ser ejecutado totalmente en paralelo. Como asumimos una arquitectura de memoria compartida, el paralelismo ha sido expresado en el pseudocódigo utilizando notación OpenMP. Después de dividir la matriz de entrada, cada hilo invoca el procedimiento de transposición secuencial aplicada a la submatriz o submatrices de las que está a cargo. En un último paso, todas las submatrices transpuestas deben ser recolectadas para componer la transpuesta de la matriz de entrada. Dependiendo de la estrategia de división utilizada, este paso puede ser más o menos complejo, ya que diferentes bloques de datos, con resultados parciales, podrían necesitar ser ensamblados para reconstruir la

```
1 C$OMP TRANSDO SCHEDULE(static, blockSize) ORDERED
2 do i=1,NROWS
3   do j=ROW(i), ROW(i+1)-1
4     DATAT(ROWT(COLUMN(j)+1)) = DATA(j)
5     COLUMNT(ROWT(COLUMN(j)+1)) = i
6     ROWT(COLUMN(j)+1) = ROWT(COLUMN(j)+1)+1
7   enddo
8 enddo
9 C$OMP END TRANSDO
```

Figura 5.3: Pseudocódigo de la versión transaccional de la transposición de matriz dispersa utilizando notación OpenTM en estilo FORTRAN.

estructura de la matriz original. Por ejemplo, en el caso más simple de una división en bloques de filas contiguas de igual tamaño, las submatrices transpuestas necesitan ser concatenadas por columnas para obtener la transposición correcta. Por esta razón, y para ser conservador, este último paso ha sido marcado como secuencial en el código, aunque se podría explotar una cierta cantidad de paralelismo.

Como ya se ha discutido, la cantidad de información acerca del problema puede guiar al programador hacia una solución más eficiente cuando desarrolla versiones paralelas de códigos con patrones de memoria complejos, como el caso de la transposición de matriz dispersa. Las abstracciones de la memoria transaccional pueden facilitar el trabajo de programación en estos casos, permitiendo explotar la concurrencia de manera optimista sin necesidad de información adicional. La figura 5.3 muestra cómo el programador podría codificar una versión paralela para este algoritmo si dispusiera de un soporte transaccional que se encargase de preservar las dependencias de datos en el orden correcto (utilizando notación OpenTM). Nótese que los esfuerzos para desarrollar esta versión son significativamente menores que los requeridos para aplicar la estrategia de particionamiento de datos (figura 5.2), ya que no se precisa de ningún conocimiento en profundidad acerca de la aplicación.

5.2. Modelo TEPO en códigos iterativos

En esta sección se analiza cómo los códigos iterativos, i.e. bucles, pueden beneficiarse del modelo de ejecución transaccional TEPO. En los códigos ite-

```
1   #pragma omp transfor
        schedule (static,
        chunk_sz, xact_sz)
         [ordered]
2   for (i=0; i<N; i++) {
3         loop_body
4   }
```

Conjunto de *chunks*: $L = \{u_0, u_1, ...\}$
Definición de *chunk*:

u_k = {Iteraciones desde $chunkSz \times k$ hasta $chunkSz \times (k+1) - 1$}

Orden de precedencia: $u_k \prec_i u_{k+1}$
Mapeo: $o(u_k) = k \mod nThreads$

Figura 5.4: El bucle es particionado en *chunks*, con un orden de precedencia total.

rativos, el orden de precedencia es una relación de orden total [2]. El objetivo es maximizar la concurrencia cuando se implementan construcciones iterativas transaccionales, tales como el bloque `transfor` propuesto en OpenTM (véase anexo A), mostrado en la figura 5.4. De acuerdo a su definición, la construcción `transfor` puede ser complementada con la definición de tres parámetros: el esquema de particionamiento, el tamaño del *chunk* (iteraciones asignadas a cada hilo en cada decisión de planificación) y el tamaño de la transacción (número de iteraciones consecutivas del bucle que ejecutan como una única transacción). Por simplicidad, en adelante, el tamaño del *chunk* será considerado igual al tamaño de la transacción (*chunkSz*) para un esquema de particionamiento estático.

Con el fin de expresar un bloque `transfor` en términos del *framework* presentado en el capítulo 4, es necesario definir tres parámetros: Cómo particionar el código en *chunks*, la relación de precedencia entre dichos *chunks* y la función que mapea los *chunks* a los hilos de ejecución. Un particionamiento simple consiste en dividir el espacio de iteración en *chunks* de *chunkSz* iteraciones contiguas: $L = \{u_0, u_1, ...\}$. Si se consideran como una secuencia, los *chunks* están sujetos a la siguiente relación de precedencia inmediata: $u_0 \prec_i u_1 \prec_i u_2$.... El esquema de planificación `static` del bucle se corresponde con una asignación de *chunks* a hilos mediante una función de mapeo *block-cyclic*: $o(u_i) = i \mod nThreads$.

Un aspecto clave de TEPO es que el orden de precedencia definido se preserva a través de los *commits* de aquellas iteraciones con dependencias. La clausula `ordered` denota esta característica. Notar que incluso si el bucle fuera totalmente paralelo (bucle tipo DOALL), los *commits* tendrían lugar *chunk* a *chunk* de acuerdo al orden de precedencia secuencial. Sin embargo, cuando el orden de precedencia entre iteraciones no es una preocupación, e.g. cuando las iteraciones del bucle pueden ser reordenadas, nuestro modelo permite a los *chunks* realizar el *commit* tan pronto hayan finalizado satisfactoriamente su ejecución sin esperar a ningún

[2] Esto significa una relación de orden, en la que se satisface totalmente que: $\forall a \neq b$, o bien $a \prec b$, o $b \prec a$.

otro. Para ello, simplemente se omitiría la clausula que ordena la ejecución de las transacciones. No obstante, en estos casos, hemos considerado un orden artificial, como puede ser el instante de inicio de la transacción, con el objetivo de simplificar la implementación.

En códigos iterativos, nuestro modelo tiene por objetivo el incrementar la concurrencia explotada de dos maneras: *intra-thread* e *inter-thread*. La primera es incrementada debido a que se permite al hilo que comiencen la ejecución de nuevos *chunks* incluso cuando alguno previo no ha cumplido su condición de *commit* (para evitar bloquear el hilo), manteniendo estos *chunks* pendientes ejecutados pero aún sin realizar su *commit*. También la concurrencia *inter-thread* es mejorada, porque en caso de no existir conflictos, un *chunk* puede realizar su *commit* tan pronto como los *chunks* precedentes se encuentren, al menos, ejecutados (estado *ejecutado-sin-commit*). De esta manera, la restricción de orden de precedencia no necesariamente implica una serialización de aquellas iteraciones que son mutuamente paralelas.

5.3. Implementación del modelo TEPO

En esta sección se discuten los detalles acerca de la implementación del modelo TEPO sobre un sistema STM ligero. TinySTM (presentado en el capítulo 3) ha sido utilizado como sistema base, el cual ha sido apropiadamente modificado para incluir el modelo de ejecución TEPO. La configuración básica considera una aproximación basada en *encounter-time locking* (ETL), por la cual los conflictos son detectados tan pronto ocurren (*early*), con una política de escritura diferida, haciendo uso de un *write buffer* (o *redo log*) que almacena las escrituras transaccionales hasta que se produce la actualización de memoria durante la fase de *commit*. Esto requiere adaptar a nuestro modelo, las primitivas básicas de TM (*read*, *write*, *rollback* y *commit*).

Siguiendo una notación similar a la encontrada en [87], los algoritmos del 4 al 7 describen el núcleo de las primitivas transaccionales adaptadas. Alguna notación general, así como varias funciones de soporte para simplificar el código, son definidas en las tablas 5.1 y 5.2. En esta descripción, el término *transacción* es adoptado como equivalencia al concepto de *chunk* utilizado durante la descripción del modelo, en tanto que un *chunk* representa una porción del programa asignado, como unidad ejecutable, a una transacción.

En esta implementación se asumirá un único hilo de ejecución asociado a cada núcleo de procesamiento, por lo que la variable *P* referencia el identificador

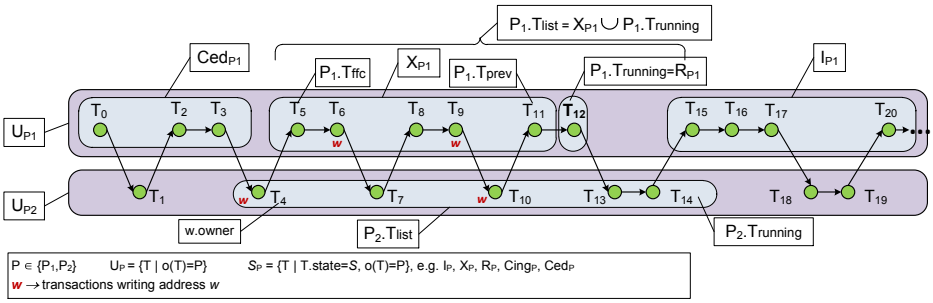


Figura 5.5: Implementación TEPO: un conjunto de transacciones son asignadas a dos hilos; las flechas representan el orden de precedencia inmediato en un estilo similar a los diagramas Hasse; Mientras no existan conflictos de datos, las transacciones pueden progresar; las transacciones en ejecución pueden entrar en conflicto bien con otras en ejecución o bien con aquellas ya ejecutadas a la espera de *commit* de otro hilo; para cada hilo las dos transacciones más destacables son la actual en ejecución y la primera esperando su turno de *commit* (i.e., el *minimal* de entre aquellas que se encuentran en estado `X`).

Tabla 5.1: Notación genérica

P	▹ Id del hilo (id del procesador)
T	▹ Transacción
w	▹ Estructura asociada a cada palabra en memoria
$\mathcal{S} = \{idle, running, executed, commiting, commited, killed\}$	▹ Posibles estados de la transacción
P_{list}	▹ Lista de todos los hilos en el sistema
$P.T_{list}$	▹ Lista de transacciones esperando *commit* en el hilo P, por orden de precedencia
$P.T_S$	▹ Primera transacción en P en orden de precedencia en estado S
$P.T_{ffc}$	▹ Primera transacción de P esperando *commit*, i.e., *minimal* de $\text{prev}_{X_P}(P.T_{running})$
$P.T_{prev}$	▹ La transacción precedente de $P.T_{running}$ en P, i.e., *maximal* de $\text{prev}_{X_P}(P.T_{running})$
$T.state$	▹ Estado de la transacción T
$T.rset$	▹ Read set (RS) de la transacción T
$T.wset$	▹ Write set (WS) de la transacción T
$T.wbuffer$	▹ Write buffer (WB) de la transacción T
$T.sco$	▹ Define el tipo de condición de commit para T: Estricta (true) or Relajada (false)
$T.P$	▹ Hilo en el que T ha sido asignada ($o(T) = P$)
$w.owner$	▹ Primera transacción activa (en orden de precedencia) que escribió la palabra w

Tabla 5.2: Funciones de utilidad general

getWord(m)	▹ Devuelve la estructura *palabra* para la dirección de memoria m
getFromWbuff(P, w)	▹ Devuelve el valor de la palabra w del *write buffer* de la última transacción escritora de w en P
addToWbuff(P, w, v)	▹ Inserta/Actualiza un valor v en la entrada w del *write buffer* de la transacción en ejecución
getFromMemory(w)	▹ Devuelve el valor de la palabra w desde memoria
copyToMemory(T)	▹ Vuelva el contenido del *write buffer* de una transacción a memoria
addToWset(P, w)	▹ Inserta/Actualiza una entrada en el *write set* de la transacción en ejecución
addToRset(P, w)	▹ Inserta/Actualiza una entrada en el *read set* de la transacción en ejecución
restorePC(T)	▹ Restaura el contexto del programa al momento de inicio de T
checkCommitCondition(T)	▹ Comprueba si T cumple su condición de *commit* de forma segura
existFreeSlot(P)	▹ *True* si hay algún *slot* libre para iniciar una nueva transacción
new(T)	▹ Obtiene los recursos para una nueva transacción T, almacenando un nuevo *slot* y sus *data sets* y *write buffer* asociados; el contexto del programa en el instante de inicio de la transacción necesita ser salvado
release(T)	▹ Libera los recursos asociados a T (data sets, write buffer, slot)

del hilo. Cada hilo mantiene en todo momento activo al menos un conjunto de datos (*read set*, *write set* y *write buffer*) correspondientes a la transacción actual en ejecución. Para mantener múltiples transacciones activas por hilo, se define una estructura adicional, un *pool* de *slots*, que será explicada durante la implementación del *commit* (sección 5.3.4). Al conjunto de estados por los que transita la transacción a lo largo de su vida (ver capítulo 4), se añade un nuevo estado transicional *killed*, para indicar que la transacción ha sido marcada como abortada y, en consecuencia, necesita abortar tan pronto sea posible. Dado que la implementación es aplicada a códigos iterativos, las transacciones tendrán asociado un orden de precedencia total, basado en el orden de las iteraciones que ejecuten. En la figura 5.5 se muestra este tipo de escenario para dos hilos y un conjunto de transacciones con un orden de precedencia total (iterativo). Las transacciones de interés han sido sido marcadas de acuerdo a la notación introducida en esta sección.

5.3.1. Read

El algoritmo 4 se corresponde con la primitiva de lectura transaccional (*read*). A diferencia de la implementación base, la lectura comienza comprobando si transacción actual en ejecución, o alguna las transacciones precedentes (de acuer-

Algorithm 2 Funciones de soporte

```
1 function checkState(P)                          ▷ Alguna transacción en el hilo P fue abortada?
2     for each T ∈ P.T_list do in order
3         if T.state ∈ {killed} then
4             Rollback(P, T)                      ▷ Reinicia la transacción
5         end if
6     end for
7 end function
1 function AbortSubsequentReader(P, T, w)         ▷ Aborta lectores de w posteriores a T en otros hilos
2     for each P_r ∈ P_list ∧ P_r ≠ P do           ▷ Para el resto de hilos
3         for each T_r ∈ P_r.T_list ⋂ next_{R,X}(T) do   ▷ Para toda transacción activa de orden posterior
                                                        (en ejecución o a espera de commit)
4             if w ∈ T_r.rset then                ▷ Si hay conflicto en w
5                 T_r.state ← killed              ▷ Marca como abortada
6             end if
7         end for
8     end for
9 end function
```

do a $\prec$) en el hilo, ya ejecutadas y pendientes de *commit*, han sido abortadas (estado `killed`) por alguna otra transacción predecesora. En ese caso, las transacciones abortadas deben ser reiniciadas. Nótese que este paso no es necesario en la implementación base ya que solo existe una transacción por hilo (la que se encuentra en ejecución).

El siguiente paso es comprobar si la dirección de memoria ha sido escrita y dónde. Mientras que la implementación base comprueba únicamente si la dirección se encuentra en el *write set* de la transacción en ejecución (porque no hay otra), nuestra implementación del modelo TEPO debe comprobar también si se encuentra en los *write sets* de alguna de las transacciones precedentes del hilo que aún no han efectuado el *commit*. Si ninguna de las transacciones precedentes del hilo ha escrito la dirección solicitada, el valor puede ser leído directamente de memoria. Si ha sido escrita por la transacción actual, el valor puede ser encontrado en su *write buffer*. En cualquier otro caso significa que alguna transacción precedente de otro hilo ha escrito la dirección y por lo tanto, el conflicto de datos es detectado, causando que la transacción lectora aborte[3]. Si la lectura tiene éxito, la dirección de memoria es finalmente añadida al *read set*.

[3]Como vimos en la descripción del modelo TEPO, este conflicto también puede ser resuelto efectuando un *stall* de la transacción lectora hasta que el dato esté disponible, o a través de un *forwarding* desde la transacción escritora a la lectora

Algorithm 3 Start transaccional en TEPO

```
1  procedure START(P)
2      NEW(T)                                  ▷ Se asume un slot disponible en este punto
3      T ← P.T_idle                            ▷ Primera transacción inactiva en orden de prec.
4      T.state ← running                       ▷ Transacción actual en ejecución
5      T.sco ← false                           ▷ Condición de commit relajada por defecto
6  end procedure
```

Algorithm 4 Read transaccional en TEPO para el hilo P

```
1  procedure READ(P, m)
2      checkState(P)                           ▷ Comprueba posibles abortos
3      w ← getWord(m)
4      T_w ← w.owner                           ▷ Propietario es el primer escritor activo de m
5      T ← P.T_running                         ▷ Transacción actual en ejecución
6      if T_w ≠ null ∧ T_w.state ∉ {killed} then
7          if T_w.P = P then                   ▷ Si el hilo actual es el propietario
8              v ← getFromWbuff(P, w)          ▷ Toma el valor de su write buffer
9          else
10             if T_w ∈ next_{R,X,Cing}(T) then
11                 v ← getFromMemory(w)        ▷ Si el escritor es posterior, lee el valor de memoria
12             else if T_w ∈ prev_{R,X,Cing}(T) then
13                 Rollback(P, T)              ▷ Conflicto RAW con una transacción precedente
14             else if T_w ∈ norel_{R,X,Cing}(T) then
15                 Rollback(P, T)              ▷ Otro conflicto RAW hace abortar a T
16             end if
17         end if
18     else
19         v ← getFromMemory(w)                ▷ Palabra no escrita por ninguna transacción activa
20     end if
21     AddToRset(P, w)                         ▷ Añade w al read set de la transacción en ejecución
22     return v
23 end procedure
```

5.3.2. Write

El algoritmo 5 muestra el pseudocódigo de la primitiva transaccional *write*. Al igual que en la operación de lectura, la escritura comienza comprobando el estado de todas las transacciones, tanto actual como previas, en busca de posibles abortos sobre las transacciones del hilo. Si el *write set* de la transacción en ejecución ya contiene la dirección, entonces el hilo fue el último en escribir la dirección y, por ello, la correspondiente entrada en el *write buffer* debe reescribirse con el nuevo valor. Si por el contrario, fue otro hilo en que escribió la dirección, se detecta un conflicto de tipo *WAW*. Como vimos en el capítulo 4, estos conflictos son resueltos considerando un criterio basado en el orden de precedencia,

Algorithm 5 Write transaccional en TEPO para el hilo P

```
1  procedure WRITE(P, m, v)
2      checkState(P)                                  ▷ Comprueba posibles abortos
3      w ← getWord(m)
4      T_w ← w.owner                                  ▷ Propietario es el primer escritor activo de m
5      T ← P.T_running                                ▷ Transacción actual en ejecución
6      if T_w ≠ null ∧ T_w.state ∉ {killed} then
7          if T_w.P ≠ P then                          ▷ Si el hilo actual no es el propietario
8              if T_w ∈ prev_{R,X,Cing}(T) then
9                  T.sco ← true                       ▷ Conflicto WAW: orden de commit estricto T_w → T
10                 AddToWset(P, w)
11             else if T_w ∈ next_{R,X}(T) then
12                 T_w.sco ← true                     ▷ Conflicto WAW: orden de commit estricto T → T_w
13                 w.owner ← T                        ▷ T nuevo propietario de w
14                 AddToWset(P, w)
15             else if T_w ∈ norel_{R,X,Cing}(T) then
16                 Rollback(P, T)                     ▷ Aborto por conflicto WAW no resuelto con orden
17             end if
18         end if
19     else                                           ▷ Palabra no escrita por ninguna transacción activa
20         w.owner ← T
21         AddToWset(P, w)
22     end if
23     addToWbuff(P, w, v)                            ▷ En cualquier caso inserta w en el write buffer
24     AbortSubsequentReader(P, T, w)                 ▷ La escritura es visible para lectores posteriores
25 end procedure
```

forzando el *commit* de las transacciones en conflicto en estricto orden secuencial. En este caso, el bit *SCO* (*Strict Commit Order*) se activa para el escritor con mayor orden de precedencia. Una transacción con el bit *SCO* activo, no podrá realizar su *commit* hasta que todas las transacciones precedentes a él lo hayan hecho. En cualquiera de los casos, tanto si la dirección fue escrita en otra transacción como si no fue escrita por ninguna, es necesario crear una nueva entrada en el *write buffer* local para almacenar el nuevo valor, además de insertar la dirección de memoria en el *write set*.

Finalmente, la operación de escritura transaccional fuerza un aborto para todas aquellas transacciones que la suceden en order de precedencia y que hayan leído dicha dirección. Este último paso es necesario para mantener la consistencia ya que la nueva escritura podría hacer que todos los valores leídos por transacciones con mayor orden de precedencia se vuelvan inconsistentes.

Algorithm 6 Rollback transaccional en TEPO para el hilo P

```
1 procedure ROLLBACK(P, T)                ▷ Aborta T (y todos sus sucesores) y reinicia su ejecución
2     repeat
3         RELEASE(P.T_running)            ▷ Libera todos sus recursos (Data sets, Write Buffer, Slot)
4         P.T_running.state ← idle        ▷ La transacción pasa a estar inactiva
5         P.T_running ← P.T_prev          ▷ Convierte la transacción previa en al actual
6     until P.T_running = T               ▷ Hasta que T sea la actual
7     RESTOREPC(T)                        ▷ Reinicia T como la transacción actual en ejecución
8 end procedure
```

5.3.3. Rollback

La primitiva *rollback*, mostrada en el algoritmo 6, permite al sistema recuperarse a sí mismo de un estado de inconsistencia, abortando una transacción conflictiva y restaurando el contexto del programa a uno consistente anterior. La operación de *rollback* es más rápida en sistemas con gestión de versiones *lazy* ya que los datos no tienen que ser restaurados en memoria. Así, el proceso consiste simplemente en vaciar el *write buffer* y reiniciar tanto el *read set* como el *write set*. Una vez hecho esto, la transacción puede volver al punto inicial y reiniciar su ejecución. La operación de *rollback* en el modelo TEPO es muy similar a la implementada por el sistema base. Solo es necesario soportar múltiples transacciones por hilo, además de la que está en ejecución. De esta manera, cuando una transacción aborta, todas las transacciones posteriores en el hilo deben ser abortadas también. De esta manera, comenzando por la transacción actual en ejecución, se procederá a repetir el proceso de *rollback* con todas las transacciones del hilo en orden inverso hasta que se alcance la transacción conflictiva y el contexto del programa pueda ser restaurado al inicio de esta.

5.3.4. Commit

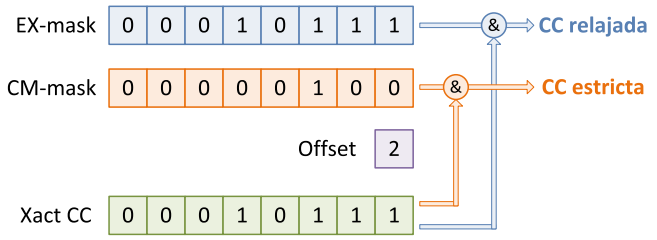
El algoritmo 7 muestra el pseudocódigo para el procedimiento de *commit*. Al igual que ocurre al inicio de las operaciones de lectura y escritura, el *commit* comienza con una comprobación para identificar posibles abortos de transacciones en el hilo. Si ninguna de ellas ha sido abortada, la fase de *commit* procede, empezando por la primera transacción pendiente, en orden de precedencia dentro del hilo. Con el fin de verificar la condición de *commit* y cumplir con las restricciones de precedencia entre transacciones, se necesita una estructura de datos compartida para mantener una traza de las transacciones que ya han efectuado el *commit*.

Algorithm 7 Commit transaccional en TEPO para el hilo P

```
1  procedure COMMIT(P)
2      repeat
3          checkState(P)                          ▷ Comprueba posibles abortos
4          T_c ← P.T_ffc                          ▷ Primera transacción esperando su commit
5          while checkCommitCondition(T_c) = true do
6              copyToMemory(T)                    ▷ Vuelca write buffer a memoria
7              release(T_c)                       ▷ Libera data sets, write buffer y slot
8              T_c.state ← committed              ▷ Esta transacción acaba de realizar el commit
9              T_c ← P.T_ffc                      ▷ Toma la siguiente transacción esperando por commit
10         end while
11     until existFreeSlot(P) = true              ▷ Sin slots libres, no puede iniciar nuevas transacciones
12 end procedure
1  function checkCommitCondition(T)               ▷ La transacción puede realizar su commit?
2      if T.sco = true then                       ▷ Orden de commit estricto
3          for each T_p ∈ prev(T) do              ▷ Comprueba estado de las transacciones precedentes
4              if T_p.state ∉ {committed} then
5                  return false
6              end if
7          end for
8          return true
9      else                                       ▷ Orden de commit relajado
10         for each T_p ∈ prev(T) do              ▷ Comprueba estado de las transacciones precedentes
11             if T_p.state ∉ {executed, committing, committed} then
12                 return false
13             end if
14         end for
15         return true
16     end if
17 end function
```

En nuestro caso, esta estructura compartida está compuesta por un par de arrays de bits que permiten llevar una traza del estado de las transacciones del sistema. Estos arrays, en los que cada *bit* identifica una única transacción, son utilizados en el proceso de comprobación de la condición de *commit* como máscaras que anotan las transacciones que han sido ejecutadas (*ex-mask*) y las que ya han realizado el *commit* (*cm-mask*). Con el objetivo de mantener la eficiencia del proceso de comprobación, el tamaño de estos arrays debe ser, significativamente menor al número total de transacciones dispensadas durante la ejecución del programa. Para evitar las limitaciones que esto supone, se ha implementado un sistema de ventana deslizante, en la que el contenido de los arrays es desplazado para descartar los *bits* correspondientes a aquellas transacciones con menor orden de precedencia que ya han efectuado el *commit* y que por tanto ya no son representativas (figura 5.6a). El desplazamiento actual de la ventana se mantiene como un contador de *offset*, asociado a ambos arrays en la estructura de datos

(a) Ventana deslizante



(b) Operación de comprobación

Figura 5.6: Estructura de array de bits para la evaluación de la condición de *commit*.

compartida.

De esta manera, considerando una transacción con un identificador de orden (*OID*) dado, el proceso puede ser descrito de la siguiente manera. Cuando la transacción finaliza su ejecución (transita al estado `executed`), el *bit* $2^{(oid-offset)}$, que identifica dicha transacción (teniendo en cuenta el desplazamiento de la ventana) en la máscara *ex-mask* es activado. La condición de *commit* (CC) asociada a la transacción, consiste básicamente en otro array de *bits* local que se utiliza como una máscara que identifica las transacciones previas con las que debe mantener el orden. Como ya de discutió en el capítulo 4, este valor puede ser asignado tanto por el programador como por el compilador. En el caso de códigos iterativos, en los que las transacciones mantienen un orden total, este valor es asignado como $2^{oid} - 1$, lo que genera una máscara de unos para todas las transacciones precedentes del sistema. El proceso de comprobación consiste simplemente en efectuar una operación *and* entre CC (eliminando los *offset bits* menos significativos) y las máscaras *ex-mask* o *cm-mask* para confirmar el cumplimiento de las condiciones de *commit* relajada o estricta, respectivamente (figura 5.6b). Si la operación *and* devuelve la propia CC, podemos garantizar que la transacción puede proceder con su *commit* de manera segura. Si es así, el *bit*

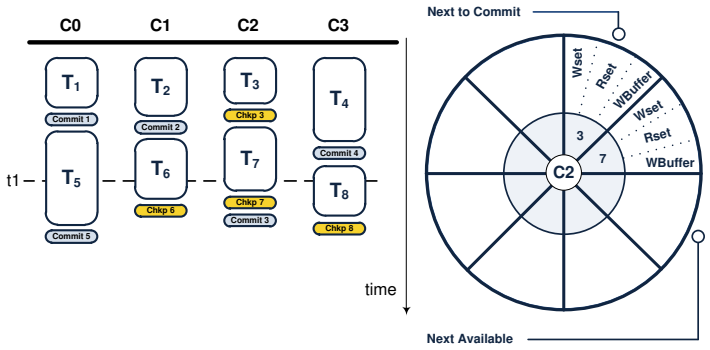


Figura 5.7: Un *pool* de *slots*, implementado como un *buffer* circular, permite mantener múltiples transacciones activas por hilo. El proceso de *checkpoint* está implícito al tomar un nuevo *slot*.

correspondiente a la transacción es activado, esta vez en el array *cm-mask*.

Esta estructura nos permite una gestión eficiente del orden de precedencia entre transacciones, con independencia de que presenten relaciones de orden total o parcial. Sin embargo, la presencia de un orden total (iterativo puro) admite soluciones, quizás más simples aunque también más restrictivas, tales como el uso de un conjunto de contadores compartidos para apuntar a la última transacción que realizó el *commit* en cada hilo [34].

Una vez garantizado el orden de precedencia, para cada transacción que realiza el *commit*, el contenido del *write buffer* necesita ser volcado a memoria, y los *read set* y *read set* reiniciados. Cuando una transacción *minimal* no satisface con su condición de *commit*, ninguna otra transacción el hilo podrá hacerlo. En este caso se toma un *checkpoint* de la transacción recientemente ejecutada, dejando el proceso de *commit* pendiente y se libera al hilo para iniciar una nueva transacción.

Debido a que podría darse el caso de un número ilimitado de transacciones por hilo sin poder efectuar su *commit*, tomando para cada una su respectivo *checkpoint*, los recursos necesarios para mantenerlas podrían crecer sin restricciones. Con el fin de limitar este aspecto, se ha definido un *pool* predefinido de *slots* [34]. Para un orden de precedencia total, este *pool* se ha implementado como un *buffer* circular (figura 5.7) con un número limitado de *slots* reutilizables, suficientemente largo como para permitir varias transacciones pendientes de *commit*, sin incrementar significativamente el riesgo de *overflow*. Una solución similar puede encontrarse en [27]. Por medio de dos punteros, el *slot* de la transacción que recientemente ha efectuado el *commit* puede ser liberado (puntero *next to com-*

mit en la figura 5.7), o un nuevo *slot* puede ser elegido (puntero *next available*). Cada *slot* está asociado al *read set*, *write set* y *write buffer* de la transacción correspondiente, además de con el contexto del programa en el punto de inicio de dicha transacción. Implementándolo de esta manera, no es necesario tomar un *checkpoint* explícito, como indica el algoritmo 1 (capítulo 4), ya que cada vez que una nueva transacción comienza, uno de los *slots* libres es asignado a ella. Si no quedan *slots* disponibles, el hilo no podrá iniciar ninguna nueva transacción. En esta situación, el hilo se mantiene a la espera de que una de las transacciones pendientes, ya ejecutadas, satisfaga su condición de *commit* y libere su *slots*.

En nuestra evaluación experimental, el número de transacciones por hilo se ha mantenido bien balanceado, de manera que el número medio de transacciones a la espera de *commit* por hilo ha sido 3. En nuestro caso se ha definido un máximo de 8 transacciones activas (sin realizar *commit*) por hilo, con el fin de permitir una buena explotación del paralelismo sin incurrir en *overheads* excesivos.

5.4. Evaluación

La implementación del modelo TEPO para códigos iterativos realizada sobre el sistema base TinySTM (versión 1.0.4), se ha evaluado utilizando un conjunto de *benchmarks* representativos. Esta sección discute los resultados de dicha evaluación. Todos los experimentos se han realizado utilizando un equipo con las características mostradas en la tabla 5.3. Todos los códigos han sido compilados utilizando la versión 4.3.4, utilizando las opciones de compilación `-O2` para el sistema transaccional (tal y como recomienda TinySTM) y `-O3` para los *benchmarks*.

Para esta evaluación se han propuesto tres conjuntos de experimentos. En primer lugar, se ha utilizado la *suite* transaccional STAMP [16], que constituye una referencia ampliamente utilizada en sistemas transaccionales. Con ello pretendemos comparar TEPO con la implementación base TinySTM. En segundo lugar, se ha evaluado la habilidad de TEPO para explotar la concurrencia con dos códigos iterativos e irregulares: el algoritmo de transposición de matriz dispersa propuesto por Pissanetzky (caso de estudio propuesto al inicio de este capítulo) y el *benchmark fluidanimate* de la *suite PARSEC* [11], el cual presenta bucles con reducciones de tipo histograma. Finalmente, con el objetivo de medir los *overheads* introducidos por el modelo TEPO con respecto al STM base, se han evaluado ambas implementaciones utilizando dos versiones diferentes de un código iterativo de tipo *Stencil* (*Jacobi2D* y *Seidel2D*) obtenidos de la *suite Polybench* [77]. En todos estos *benchmarks* (excepto STAMP), se han definido manualmente las

Tabla 5.3: Configuración de la plataforma de evaluación

Parámetro	**Descripción**
Procesador	Intel Xeon X7550
Nº núcleos	32 (4 *sockets* x 8 núcleos/*socket*)
Frecuencia	2GHz
Memoria	128GB
Sistema operativo	Ubuntu Server 12.04 LTS, Kernel 2.6.32 (64 *bits*)
Compilador	gcc v4.3.4

transacciones dividiendo el bucle original e introduciendo un nuevo bucle interior para crear bloques de iteraciones. Este bucle se encapsula en un *chunk* y se ejecuta como una transacción. En *Jacobi*, *Seidel* y *la transposición de matriz dispersa*, se utiliza el bucle que itera a través de las filas para crear los *chunks*. En *Fluidanimate*, se particiona el bucle que itera sobre las partículas vecinas. El capítulo 3 incluye una descripción más detallada acerca de cada uno de estos *benchmarks*.

Además del sistema de base TinySTM original, sin modificaciones, se ha utilizado una versión con orden de TinySTM, en la cual los *commits* de las transacciones suceden únicamente cuando todas las transacciones previas han efectuado su *commit* (condición estricta), deteniendo los hilos hasta que se satisfaga dicha condición (o abortando la transacción y ejecutándola de nuevo, dependiendo de la situación). La política de aborto para esta implementación asume invalidar siempre la transacción con mayor orden de precedencia en caso de conflicto de datos. En adelante nos referiremos a esta versión con el nombre de *TinySTM-ordered*, la cual también será utilizada como punto de referencia en la evaluación, principalmente en aquellos casos en los que el cumplimiento del orden de precedencia es requisito necesario para alcanzar un resultado correcto.

Para mitigar el efecto de las fluctuaciones aleatorias en los resultados, ocasionadas por el comportamiento transaccional, cada ejecución ha sido repetida al menos cincuenta veces, tomando como valor final la media de todos ellos. Las métricas utilizadas en esta sección incluyen el *speedup* tomando como referencia el tiempo de la versión secuencial del código [4], el *TCR (Transaction Commit Rate)* [3], como el porcentaje de transacciones que han realizado el *commit* respecto del total de transacciones ejecutadas, que mide la efectividad de la explotación de la concurrencia; y el *throughput*, como el número de transacciones que han realizado el *commit* por segundo. Ver capítulo 3 para más información sobre las métricas.

[4] En caso de que el código secuencial no esté disponible, se utilizará como referencia la implementación base de TinySTM ejecutada con único hilo.

Tabla 5.4: STAMP workloads

Benchmark	Argumentos de entrada	#TX	#TX Lecturas (media)	#TX Escrituras (media)	Tiempo en TX	Contención
Bayes	`-v32 -r4096 -n10 -p40 -i2 -e8 -s1`	2K	28.75	3.26	99 %	Alta
Genome	`-g16384 -s64 -n16777216`	2489K	36.23	0.89	98 %	Baja
Intruder	`-a10 -l128 -n65536 -s1`	5816K	22.40	1.60	77 %	Alta
Kmeans	`-m15 -n15 -t0.00001 -i rand-n65536-d32-c16`	4106K	599.99	599.99	1.5 %	Baja
Labyrinth	`-i random-x512-y512-z7-n512`	1K	180.52	177.02	100 %	Alta
Ssca2	`-s20 -i1.0 -u1.0 -l3 -p3`	22362K	1.00	2.00	28 %	Baja
Vacation	`-n2 -q90 -u98 -r1048576 -t4194304`	4194K	284.01	5.54	97 %	Baja/Media
Yada	`-a15 -i ttimeu1000000.2`	2415K	58.74	18.01	96 %	Media

5.4.1. STAMP benchmark suite

Los *benchmark* de STAMP se han configurado utilizando las cargas de trabajo presentadas en la tabla 5.4. Los siguientes experimentos se enfocan en comparar TEPO frente las versiones, original y ordenada, de TinySTM. Debido a que los códigos de STAMP ya se encuentran escritos transaccionalmente y se han ejecutado sin modificaciones adicionales respecto a la versión proporcionada, la implementación del modelo TEPO divide los bucles en *chunks* de una iteración por transacción.

Con el propósito de evaluar convenientemente tanto la implementación TEPO como TinySTM-ordered, se ha forzado un orden artificial entre transacciones para estas dos implementaciones, siguiendo como criterio de precedencia el instante de inicio de estas. Por su parte, la implementación TinySTM de base se ha evaluado sin considerar orden entre transacciones. Nótese que el resultado de la ejecución de los códigos de STAMP es correcta con independencia del orden entre las transacciones.

De esta manera, el orden artificial impuesto introduce un retardo entre la ejecución y el *commit* con respecto a la implementación base de TinySTM (transacciones desordenadas) como explica la figura 5.8. Mientras TinySTM base puede efectuar el *commit* de las transacciones tan pronto como su ejecución finalice, TinySTM-ordered debe esperar hasta que las transacciones precedentes hayan finalizado su *commit* satisfactoriamente. Esta situación deriva en un retardo del *commit* que afecta al rendimiento. Por otro lado, a la vez que preserva el orden de precedencia, TEPO permite a las transacciones comenzar la fase de *commit* cuando el conjunto de las transacciones que le preceden han sido, al menos, ejecutadas. Esta condición (*commit* relajado) puede llegar a reducir significativamente el retardo de *commit*, lo cual se traduce en una mejora en el tiempo de ejecución del *benchmark*. Además, TEPO puede ocultar esta latencia permitiendo al hilo iniciar nuevas transacciones cuando las anteriores no puedan realizar su

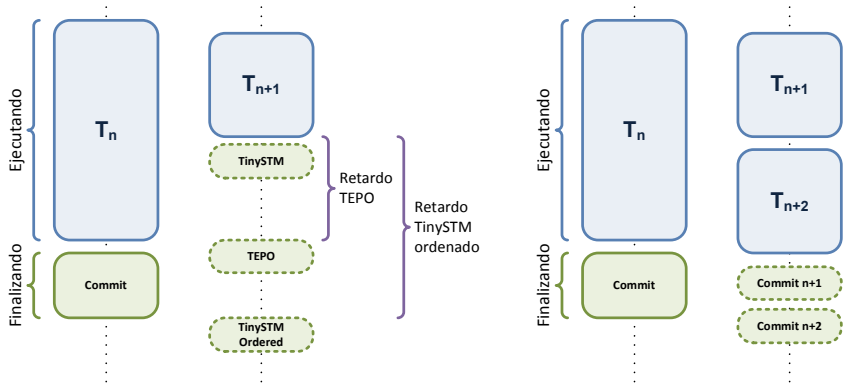


Figura 5.8: Escenarios de *commit*: En la implementación base de TinySTM una transacción puede realizar el *commit* justo después de su ejecución; TEPO realiza el *commit* cuando todas las transacciones precedentes han finalizado su ejecución (aunque no hayan efectuado el *commit*); en el caso de la versión ordenada de TinySTM, una transacción realiza el *commit* solo cuando todas las transacciones precedentes han efectuado su *commit*. Además, TEPO puede lanzar varias transacciones sin necesidad de que las anteriores realicen el *commit*, lo cual oculta el este retardo.

commit.

Los resultados en la medición del speedup se muestran en al figura 5.9. Como era lo esperado, en casi todas las situaciones, TEPO obtiene medidas de *speedup* intermedias entre las implementaciones de TinySTM base y su versión ordenada. A diferencia de TinySTM, la implementación de TEPO no toma ventaja de la ejecución desordenada de las transacciones, sin embargo es capaz de mejorar el tiempo de ejecución respecto a la versión ordenada (ver figura 5.8). El objetivo de estos experimentos no es enfrentar TEPO con la implementación TinySTM de base en términos de tiempo de ejecución para códigos sin orden de precedencia (ya que no puede mejorarlo en ningún caso), sino mostrar que después de introducir un orden de precedencia en el código, TEPO es capaz de rendir mejor que este mismo sistema, manteniendo el mismo orden entre transacciones (TinySTM-ordered). No obstante, incluso con la desventaja de tener que mantener un orden de precedencia, para algunos experimentos concretos, TEPO mantiene un rendimiento muy cercano al de la versión base de TinySTM.

Observando la evolución del *speedup* en la figura 5.9, podemos clasificar los códigos en dos grupos principales. En un primer grupo, la implementación

base muestra una escalabilidad relativamente buena, la cual está principalmente limitada por las características de la memoria en sistema hardware de pruebas. Este grupo incluye *Genome, Labyrinth* y *Vacation*. Aunque el *speedup* de TEPO es inferior al de TinySTM en estos casos, se aprecia una tendencia similar en cuanto a escalabilidad, manteniéndose siempre por encima de los resultados obtenidos por TinySTM-ordered.

En un segundo grupo, compuesto por *Intruder, Kmeans* y *Yada*, la escalabilidad de la versión base está limitada por las características de los códigos para un gran número de hilos. En estos casos, TEPO sigue la misma tendencia que la versión base, manteniéndose mucho más cerca de ella que la versión ordenada, a pesar del hecho de cumplir con el orden artificial impuesto.

Los códigos *Ssca2* y *Bayes* tienen un comportamiento singular y no pueden clasificarse en ninguno de estos grupos. En *Ssca2*, todas las versiones obtienen un rendimiento bastante pobre, debido al alto número de pequeñas transacciones, que aunque presentan una baja contención, disparan los costes asociados a la gestión transaccional, ocasionando que no se extraiga demasiado paralelismo. Sin embargo, TEPO continua mostrando una ventaja respecto a TinySTM-ordered en cuanto a tiempo de ejecución. Finalmente, *Bayes* presenta una alta variabilidad debido a que el tiempo de ejecución es especialmente sensible al orden en que se ejecutan las transacciones en la fase de aprendizaje del árbol (véase capítulo 3 para más detalles del *benchmark*).

El retardo de *commit* asociado con las implementaciones con orden tiene además un efecto en términos de TCR (figura 5.10). Este retardo hace que el tiempo de vida de las transacciones se alargue, incrementando la probabilidad de conflictos entre transacciones concurrentes. El menor retardo de *commit* proporcionado por TEPO se traduce en una reducción significativa de la tasa de abortos con respecto a TinySTM-ordered. Sin embargo, este no es el único aspecto importante. Aunque la implementación TinySTM de base no cumple con ningún orden, TEPO obtiene mejores o iguales medidas de TCR en la mayoría de experimentos. Esto se debe a que el orden de precedencia artificial introducido por TEPO ayuda a filtrar un alto número de falsas dependencias de datos, las cuales ocasionan numerosos abortos en la versión de base. Este efecto puede observarse en la evolución de la escalabilidad del TCR, donde TEPO escala mejor que las versiones de TinySTM, especialmente cuando un número alto de hilos hace crecer la contención del sistema. Sin embargo, esto no se traduce en una mejora del *speedup* debido al efecto negativo de la introducción del orden artificial.

En general, las medidas de TCR en TEPO se mantienen en un rango entre

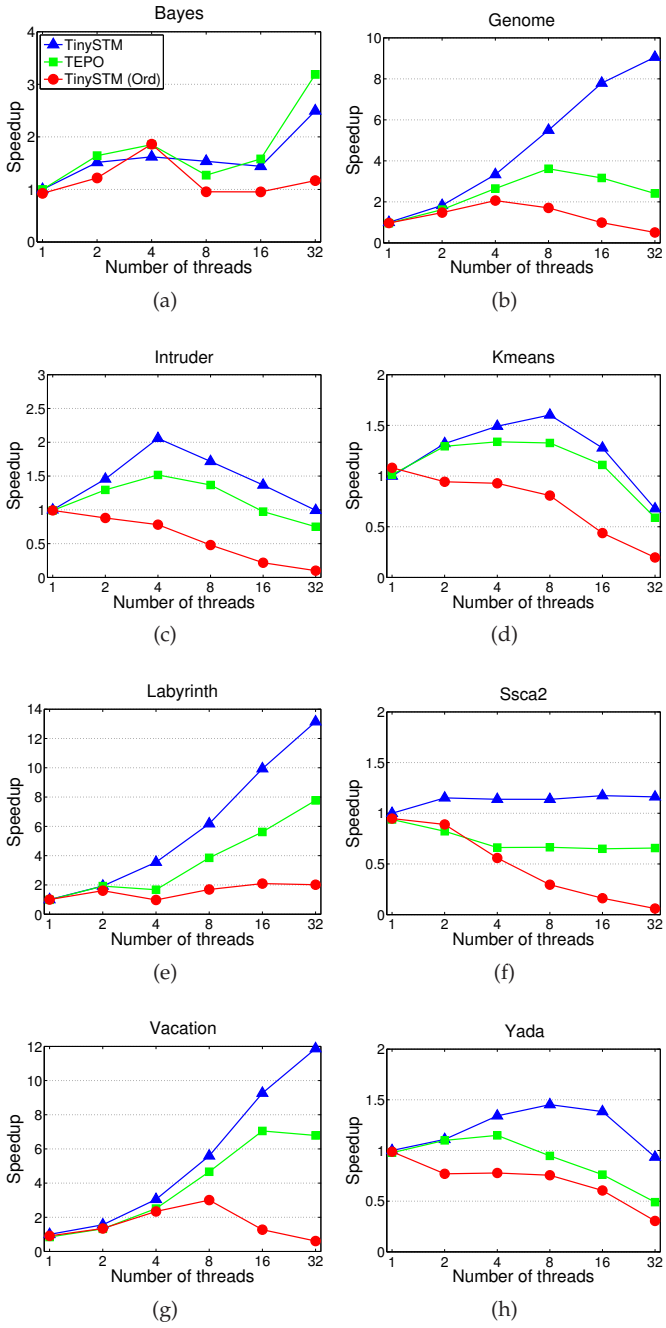


Figura 5.9: STAMP *benchmark*: *Speedup* utilizando la configuración de la tabla 5.4.

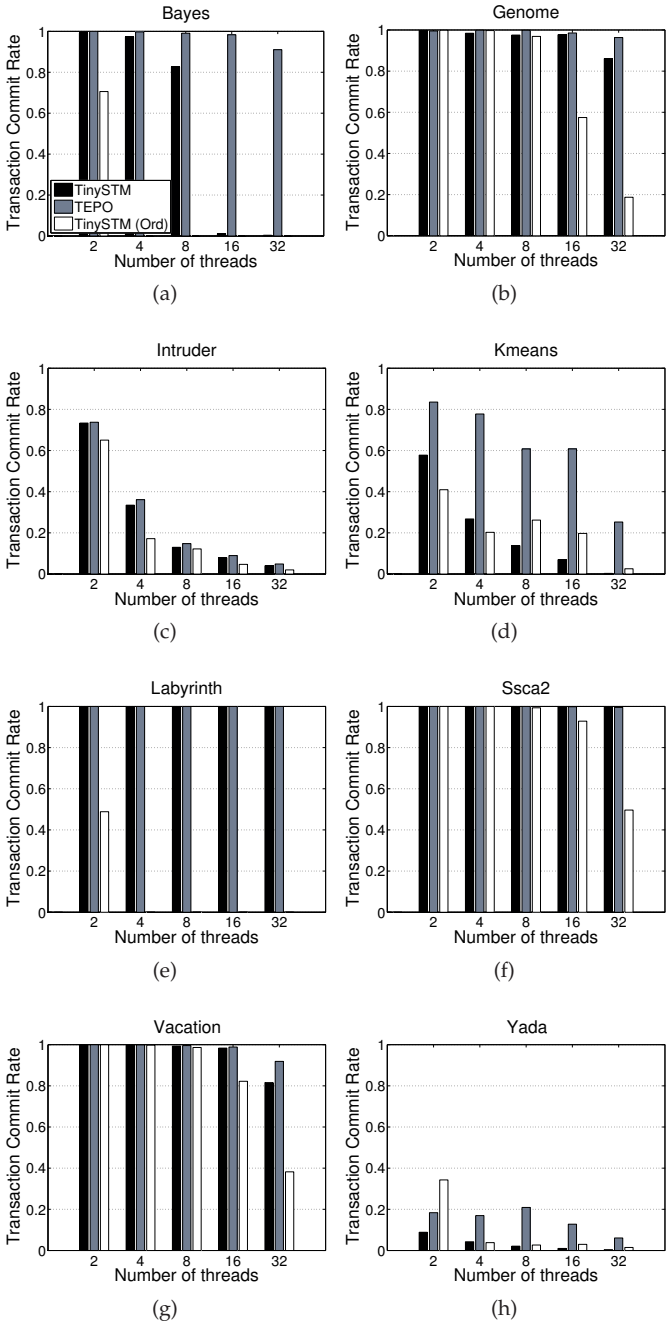


Figura 5.10: STAMP *benchmark*: TCR utilizando la configuración de la tabla 5.4.

Tabla 5.5: Características de las matrices dispersas

Matriz	Tamaño	#coef. no nulos por fila (med.) (NNR)	#coef. no nulos por fila / tamaño fila (NNRD, en %)
beacx	497x506	100	20.2 %
wm3	207x260	14	6.7 %
jpwh_991	991x991	6.1	0.62 %
1138_bus	1138x1138	3.6	0.32 %
bcsstk18	11948x11948	12	0.10 %
bcspwr10	5300x5300	4.1	0.07 %

el 70 % y el 100 % para todos los códigos, excepto aquellos que presentan un *speedup* limitado y falta de escalabilidad (*Intruder, Kmeans* y *Yada*). Para estos códigos, los sistemas TM incurren en una serialización de la ejecución como consecuencia de la alta tasa de abortos producida por los conflictos de datos. Como caso extremo, tenemos a *Yada* que muestra altas tasas de aborto debido a que el gran tamaño de sus transacciones y la cantidad de tiempo invertido en ellas (alrededor del 96 % del *benchmark*) disparan la contención, haciendo los conflictos de datos extremadamente comunes.

5.4.2. Códigos iterativos

Este segundo grupo de de experimentos incluye dos códigos iterativos irregulares: la transposición de una matriz dispersa y un código con reducciones de tipo histograma. Para el primero de estos códigos, dado que el orden de las iteraciones no puede ser cambiado, la implementación base de TinySTM no es aplicable, ya que no conduce a un resultado válido. Por ello, en estos experimentos se compara TEPO únicamente con la versión ordenada (TinySTM-ordered). Por su parte, el segundo código si que permite reordenamiento de las iteraciones, por lo que los resultados de las tres implementaciones será presentado.

Transposición de matriz dispersa

Las matrices de entrada y salida están representadas en formato CRS (*Compressed Row Storage*), quedando definidas por tres vectores unidimensionales: *row, column* y *data*. Como se analizó en la sección 5.1, el bucle principal del algoritmo de Pissanetzky presenta un patrón de acceso complejo con varios niveles de indirección, que ocasionan conflictos de datos dependiendo de la distribución de elementos distintos de cero en la matriz de entrada, lo cual es desconocido

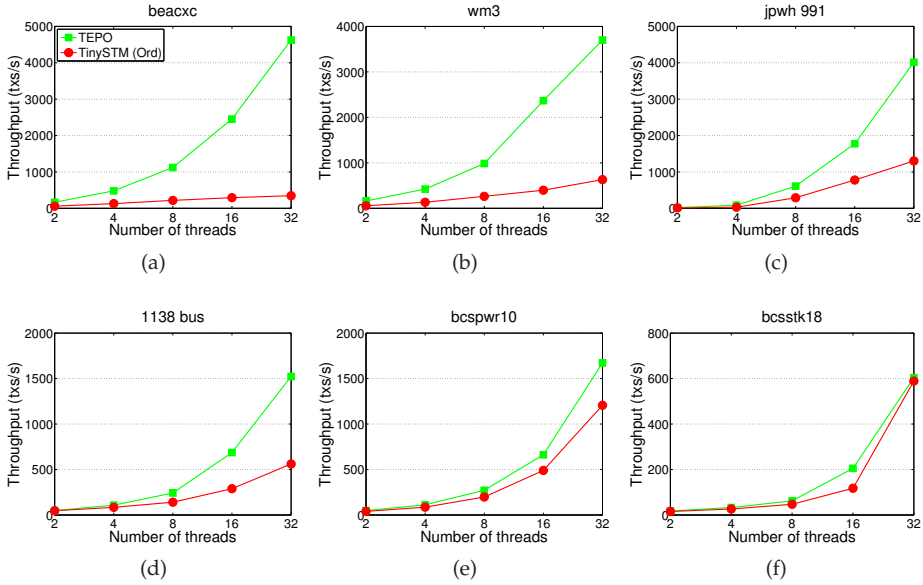


Figura 5.11: Resultados de *Throughput* para el código *sparse matrix transposition* para distintas matrices de entrada utilizando TEPO y TinySTM-ordered.

hasta el momento de ejecución. Debido a ello, el comportamiento del sistema puede ser modulado utilizando matrices con diferentes características. Para esta evaluación hemos seleccionado un conjunto de matrices del repositorio *Matrix Market* [14], cuyas características se resumen en la tabla 5.5.

El número de conflictos de datos, y por lo tanto la efectividad de la paralelización transaccional, estará relacionada con la densidad de las filas no nulas en la matriz CRS de entrada, más que con la densidad de la matriz completa, ya que estas filas son precisamente las que serán transpuestas para conformar la matriz CRS de salida. De acuerdo a la densidad media por filas no nulas (*non-null row density* o NNRD), las matrices evaluadas pueden ser clasificadas en tres categorías: *NNRD* alto (*beacxc*, *wm3*), *NNRD* medio (*jpwh_991*, *1138_bus*) y *NNRD* bajo (*bcsstk18*, *bcspwr10*). Las figuras 5.11 y 5.12 muestran el rendimiento obtenido para la ejecución de estas matrices. En términos de *throughput*, un *NNRD* mas alto hace que se incremente la mejora de rendimiento de TEPO frente a la versión ordenada de TinySTM. Esta tendencia relativa se mantiene también en las medidas del TCR. Hay que destacar que el TCR es menor para las matrices más pequeñas, aunque TEPO mejora los resultados de TinySTM-ordered, debido

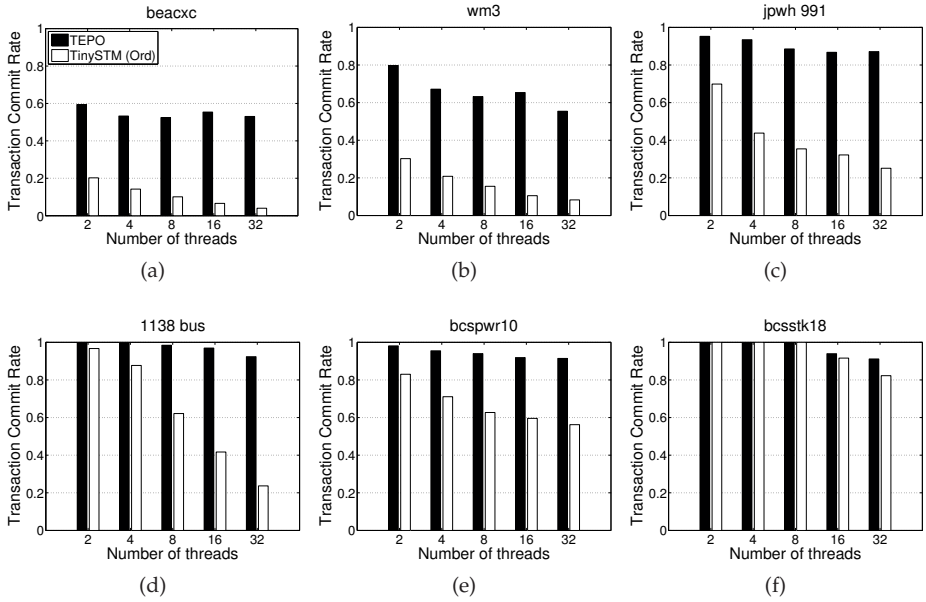


Figura 5.12: Resultados de *TCR* para el código *sparse matrix transposition* para distintas matrices de entrada utilizando TEPO y TinySTM-ordered.

a que estas matrices poseen la mayor densidad por fila no nula.

Debido a que la paralelización se ha aplicado al bucle más externo (véase figura5.3), las matrices más grandes tendrán mayor número de iteraciones en el bucle interior, resultando en una carga de trabajo absoluta mayor por cada iteración del bucle exterior, y como consecuencia, se ejecutarán un menor número de transacciones por unidad de tiempo. En todos los experimentos TEPO fue capaz de explotar una concurrencia mayor que la implementación base ordenada.

Nótese que la transposición de matriz dispersa es un código fuertemente afectado de *memory-bound*, ya que su principal operación es básicamente movimientos de datos. En estos casos en los que la aplicación presenta baja carga computacional, el ancho de banda de memoria fuera del chip es a menudo uno de los aspectos que más restringe el rendimiento del sistema. Con el fin de analizar cómo afecta este aspecto al rendimiento del sistema y su escalabilidad, el código ha sido evaluado añadiendo una carga computacional sintética, operando sobre cada coeficiente de la matriz original, antes de escribirlo en la matriz transpuesta de salida. Esta carga de trabajo sintética consiste en encontrar el último número

```
1 double lastPrime(double initVal){
2   double i, number, lastPrime;
3
4   assert(SEARCH_RANGE > 0);   \\ Define la carga de trabajo
5   number = initVal;
6
7   while (number <= (initVal + SEARCH_RANGE)) {
8       i = 2;
9       while (i <= number) {
10          if(number % i == 0) break;
11          i++;
12      }
13      if (i == number) lastPrime = number;
14      number++;
15  }
16  return lastPrime;
17 }
```

Figura 5.13: Función para el cálculo del último número primo

primero a partir del valor a transponer y en un rango fijo definido (figura 5.13).

Con el fin de normalizar la medida de esta computación añadida, utilizaremos la *intensidad operacional* [99] que mide la relación entre el número de operaciones en punto flotante (*FLOPS*) y el tráfico de memoria (Bytes movidos por las operaciones de lectura y escritura) de la aplicación. Para medir estos factores se ha utilizado la librería PAPI (*Performance Application Programming Interface*) [15] que proporciona un interfaz portable para acceder a los contadores hardware de los multiprocesadores modernos. Las medidas de intensidad operacional se han efectuado sobre una versión secuencial del *benchmark* introduciendo una carga de trabajo adicional de forma sintética.

Los resultados pueden verse en la figura 5.14, donde el *speedup* es resumido para varios niveles de carga computacional. Como era de esperar, a medida que crece la intensidad operacional, mayor es el paralelismo explotable en todos los casos, ya que el efecto de la memoria tiene menor peso. Por el contrario, y desde el punto de vista transaccional, una intensidad operacional alta implica mayor tiempo en transacción y como consecuencia, mayor coste en caso de aborto. La ventaja de TEPO sobre TinySTM-ordered se mantiene en casi todas las situaciones, especialmente para un gran número de hilos. Concretamente, para 8 hilos la mejora de *speedup* de TEPO es de alrededor del 10 % en la mayoría de los casos. Para 16 hilos, esta mejora se incrementa hasta el 30 %, mientras que para 32 hilos se consigue alcanzar una mejora de hasta el 60 % en algunas matrices.

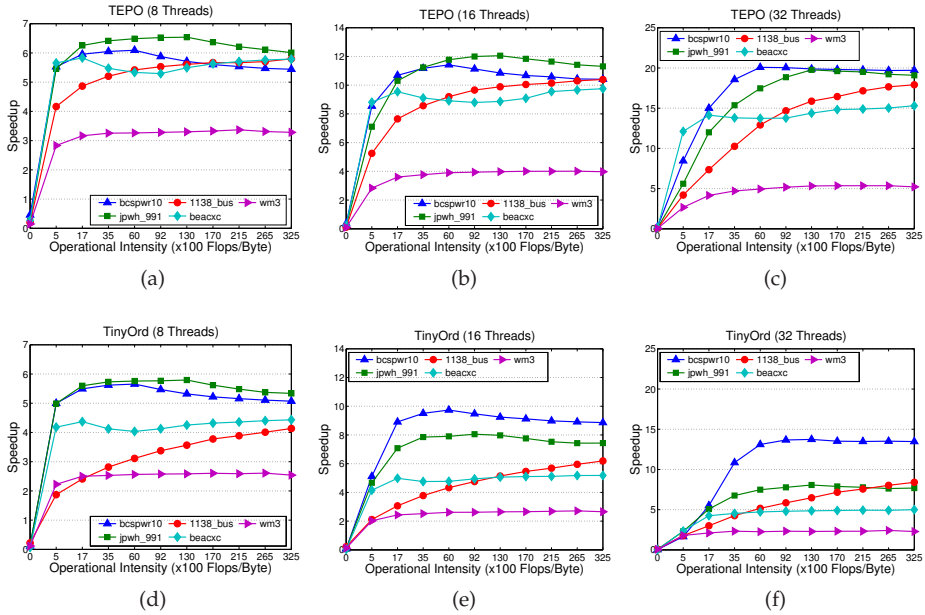


Figura 5.14: Resultados de *Speedup* para el código *sparse matrix transposition* con carga de trabajo extra (8, 16 y 32 hilos; eje x es presentado en escala cuadrática)

Como se aprecia en las figuras, los resultados son modulados por el patrón de la matriz de entrada, lo cual resulta en una tendencia convergente hacia un punto de máximo paralelismo, o saturación, alcanzable por TEPO. Las ejecuciones en la versión ordenada de TinySTM no muestran la misma tendencia, ya que el *speedup* observado queda por debajo de TEPO en todos los casos y difiere más de una matriz a otra. Singular es el caso de la matriz *wm3* cuyo pobre comportamiento está limitado por su pequeño tamaño.

Fluidanimate

Fluidanimate, del conjunto de *benchmark* PARSEC, es el segundo código iterativo testeado, en este caso con operaciones de tipo reducción. Los resultados de la evaluación se muestran en la figura 5.15 para la configuración *simlarge* que consta de una entrada de 300K partículas y 5 *frames*. Nótese que las propiedades asociativas y conmutativas de las operaciones de reducción hacen que la implementación base de TinySTM alcance resultados correctos aunque las itera-

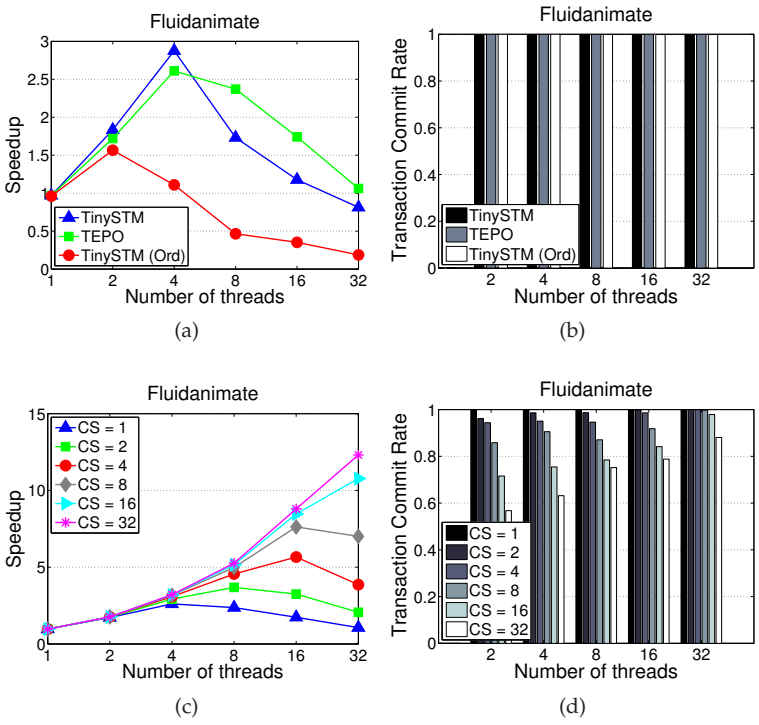
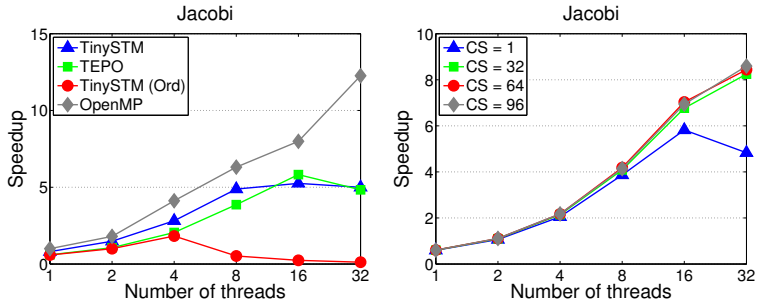


Figura 5.15: Evaluación de *speedup* y TCR para el *benchmark Fluidanimate*. (a,b) muestra la comparación de TEPO frente a las versiones base y ordenada de TinySTM. (c,d) muestra los resultados de TEPO para diferentes tamaños de *chunk* (CS) (iteraciones/*chunk*).

ciones puedan ser desordenadas [5]. En cualquier caso, TEPO obtiene los mejores resultados a pesar de todas las implementaciones alcanzan valores de TCR casi máximos.

En este *benchmark* cabe destacar que solo la computación asociada a aquellas partículas correspondientes a los límites, son encerradas en secciones críticas en la versión original del código, y por tanto, la probabilidad de conflictos no es muy alta (solo alrededor del 1.5 % de las escrituras en el array de reducción involucran conflictos potenciales). El tamaño del *chunk* tiene una influencia significativa en la escalabilidad de TEPO con el número de hilos, debido a que el cociente entre

[5] Salvo los errores acumulados de la aritmética en coma flotante



(a) (b)

Figura 5.16: Evaluación de *speedup* para *Jacobi*. (a) muestra la comparación de TEPO frente a TinySTM (tanto base como ordenada) y OpenMP. (b) muestra los resultados de TEPO para diferentes tamaños de *chunk* (CS) (iteraciones/*chunk*).

overhead y carga útil de trabajo es menos representativo con *chunks* grandes. Sin embargo, el TCR empeora en estos casos porque la probabilidad de conflictos incrementa en proporción con la vida útil de la transacción.

5.4.3. Evaluando overheads

Aunque en los últimos años se mejorado la eficiencia de las implementaciones STM hasta el punto de ser consideradas ligeras y competitivas, la realidad es que sus prestaciones aún quedan lejos de los sistemas HTM. Con el fin de evaluar el *overhead* introducido por nuestra propuesta en relación al sistema transaccional de base, se han utilizado dos implementaciones de un código de tipo *stencil* en su versión 2D, incluidas en el conjunto de *benchmark Polybench*. Nótese que este tipo de códigos [18] presentan una baja intensidad operacional, por lo que el acceso a memoria supone un importante cuello de botella para el rendimiento. El primer código, *Jacobi* puede ejecutar totalmente en paralelo debido a que las fases de computación y actualización de memoria están desacopladas por medio de un array auxiliar. Por otro lado, *Seidel* realiza el mismo cómputo *in-place*, y por tanto, su ejecución se serializa debido a las dependencias entre iteraciones. Una descripción más profunda de ambos códigos puede ser vista en la sección 3.2.2).

Las medidas de *speedup* para *Jacobi* se muestran en la figura 5.16. Al tratarse de un código totalmente paralelo, la medida del TCR se ha omitido por no ser representativa (no existen conflictos). Solo como referencia, se ha incluido la

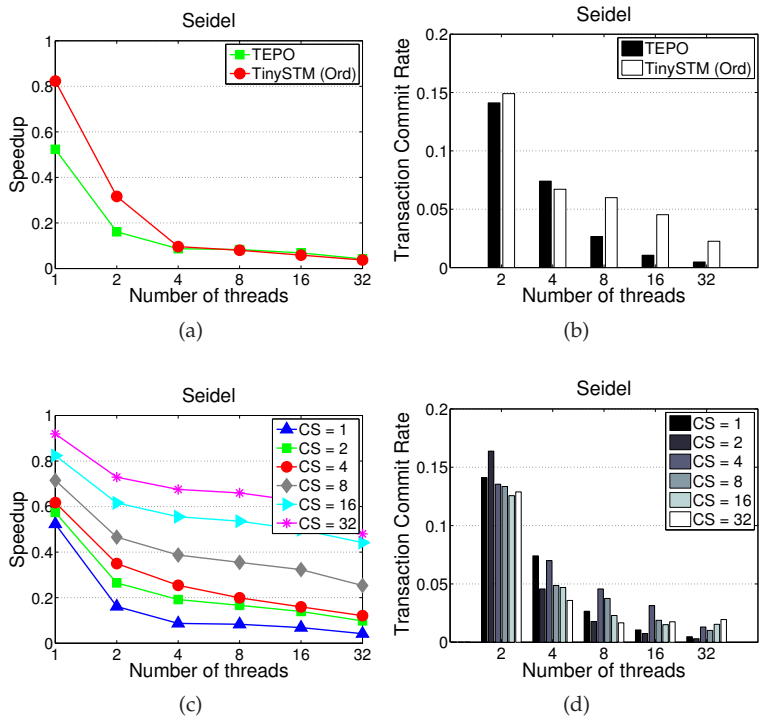


Figura 5.17: Evaluación de *speedup* y TCR para *Seidel*. (a,b) muestra la comparación de TEPO frente a la versión ordenada de TinySTM. (c,d) muestran los resultados de TEPO para diferentes tamaños de *chunk* (CS) (iteraciones/*chunk*).

medición de una versión paralela utilizando OpenMP (cada bucle se paraleliza se manera separada mediante la cláusula `pragma omp parallel for`). Puesto que reordenar las iteraciones en la versión paralela no afecta a la corrección de los resultados, la implementación base de TinySTM es también aplicable en este caso. El principal punto de interés en la medición del *speedup* es comprobar como la versión paralela ejecutada con TEPO no añade un *overhead* adicional significativo respecto al sistema transaccional de base. De hecho, se puede se puede observar una ligera ventaja en TEPO cuando el número de hilos crece. Al incrementar el tamaño del *chunk*, el *overhead* puede ser compensado con concurrencia, elevando el *speedup* hasta un nivel más alto, el cual alcanza su punto de saturación alrededor del 75 % del proporcionado por la versión OpenMP.

La naturaleza secuencial de *Seidel* implica un fuerte empeoramiento en el

speedup y el TCR como se muestra en la figura 5.17. Como las iteraciones no pueden ser reordenadas, la versión ordenada de TinySTM ha sido tomada como referencia. La baja carga computacional por iteración y el alto *overhead* asociado a las operaciones transaccionales (lecturas y escrituras), son los responsables de la caída de la eficiencia por debajo del secuencial, tornándose este efecto significativamente peor cuando un alto número de hilos concurrentes operan en el sistema. Incluso así, TEPO no rinde peor que TinySTM-ordered. Al igual que ocurría en *Jacobi*, esta degradación del rendimiento puede aliviarse incrementando el tamaño de *chunk* para minimizar los *overheads* transaccionales.

Nótese que *Seidel* es el único *benchmark* en el que la versión ordenada de TinySTM obtiene mejores valores de TCR que TEPO (aunque esta mejora sea mínima). Debido a que *Seidel* es *in-place*, las transacciones consecutivas provocarán conflictos entre sí. Puesto que TEPO permite un mayor número de transacciones activas por hilo que TinySTM-ordered, la tasa de abortos total será mayor, ya que mientras que esta última solo tiene que abortar una transacción por hilo (la actual en ejecución), TEPO debe abortar en cadena todas las transacciones iniciadas, cada vez que ocurra un conflicto.

6 Extensión del modelo TEPO para códigos con reducciones

En los capítulos anteriores hemos presentado un modelo de ejecución transaccional para códigos con restricciones de precedencia. Aunque este modelo también puede aplicarse a códigos con reducciones, las características que presentan estas construcciones, hace que gran parte del paralelismo que presentan se desperdicie. En este capítulo presentamos una extensión al modelo TEPO en forma de soporte para maximizar la explotación de paralelismo en códigos con reducciones.

En este capítulo describimos los fundamentos sobre los códigos de reducción (6.1), así como las técnicas clásicas utilizadas para la su paralelización (6.1.1). Seguidamente analizamos la problemática del uso de la memoria transaccional como sustituto de los *locks* en la paralelización de estas estructuras (6.2). Por último, presentamos y evaluamos nuestro soporte R-TM (6.3, 6.4 y 6.5).

6.1. Fundamentos sobre reducciones

Las operaciones de reducción forman parte esencial del núcleo de muchas aplicaciones de cálculo intensivo tales como computación con matrices dispersas o resolución de elementos finitos, y frecuentemente aparecen asociadas con patrones de acceso irregular.

```
1 int f1[fDim], ..., fn[fDim];
2 float A[ADim];
3 for (i=0; i<fDim; i++){
4   Calculate ξ1,ξ2,...,ξn;
5   A[f1[i]] = A[f1[i]] ⊕ ξ1;
6   ...
7   A[fn[i]] = A[fn[i]] ⊕ ξn;
8 }
9
```

Figura 6.1: Ejemplo de reducción histograma sobre un array de reducción

Se dice que una operación es una reducción cuando presenta un patrón de la forma $O = O \oplus \xi$, donde $\oplus$ es un operador asociativo y conmutativo aplicado al objeto O, y ξ es una expresión calculada con objetos diferentes a O. Algunos ejemplos de operadores de reducción son la suma, el producto y cálculos de máximos y mínimos. Dependiendo de la naturaleza del objeto O sobre el cual se aplica la reducción, podemos distinguir dos tipos: *reducción escalar* es aquella en la que la operación de reducción se aplica sobre una variable simple; *reducción histograma* es aquella en la que el objeto O es un array.

En muchos casos, las operaciones de reducción suelen estar ligadas a construcciones iterativas como bucles, sin embargo, contener una o más operaciones de reducción no es condición suficiente para constituir un bucle de reducción. Las condiciones de asociatividad y conmutatividad exigen que el orden en que se ejecuten las iteraciones del bucle no afecte al resultado final, para lo cual, se requiere que el bucle cumpla con una serie de restricciones adicionales. Se dice que un bucle es de reducción, o de reducción histograma, si incluye una o varias operaciones de reducción, sobre el mismo o distintos objetos, siempre que la operación aplicada sobre cada objeto no cambie. Además, no puede existir ninguna otra referencia a estos objetos en ninguna parte del bucle distinta a la propia sentencia de reducción. Los bucles con reducciones que cumplen con estas condiciones presentan una naturaleza totalmente paralela, en la que solo es necesario asegurar el acceso en exclusión mutua a los objetos reducidos para garantizar la corrección de la ejecución paralela (figura 6.1).

Existen situaciones en que la existencia de sentencias de reducción no es condición suficiente para que el bucle sea de reducción [84]. La figura 6.2 ilustra dos ejemplos de bucles con patrón de acceso irregular que contienen operaciones de reducción, pero que sin embargo, no conforman un bucle de reducción. El código de la figura 6.2(a) presenta un bucle con dos operaciones de reducción (suma y producto) sobre el mismo objeto. Aunque por separado sean operaciones de re-

```
1 for (i=0; i<N; i++) {
2   idx1 = ind1(i);
3   idx2 = ind2(i);
4   O(idx1) = O(idx1) + ξ;
5   O(idx2) = O(idx2) * ξ;
6 }
```

(a)

```
1 for (i=0; i<N; i++) {
2   idx1 = ind1(i);
3   O(idx1) = O(idx1) + ξ;
4   Z(i) = O(i);
5 }
6
```

(b)

Figura 6.2: Bucles con reducciones que no constituyen un bucle de reducción

```
1 for (i=0; i<N; i++){
2   A[K[i]] = ...;
3   ... = A[L[i]];
4   A[R[i]] = A[R[i]] ⊕ ξ;
5 }
```

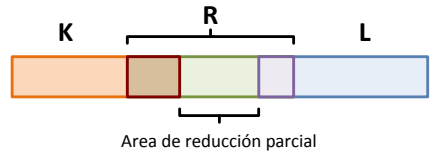


Figura 6.3: Ejemplo de reducción parcial

ducción, la operación realizada por el bucle no lo es, ya que en conjunto pueden dar lugar a una operación que no es ni asociativa ni conmutativa. En el bucle de la figura 6.2(b) se efectúa una lectura fuera de la operación de reducción, lo que hace que la operación no sea conmutativa ya que el valor $Z(i)$ está relacionado con el valor almacenado en $O(i)$, el cual depende en cada momento del orden en que se ejecuten las iteraciones.

Un caso particular de estas situaciones, son lo que se conocen como *reducciones parciales*, debido a que las condiciones de reducción se cumplen para algunos de los accesos a memoria del objeto de reducción, pero no para todos ellos (figura 6.3).

6.1.1. Revisión de técnicas de paralelización de reducciones

Encontramos abundante literatura en la que se presentan soluciones eficientes de paralelización de bucles de reducción, dada su gran relevancia computacional. Estas soluciones se pueden clasificar en tres grupos: basadas en exclusión mutua, basadas en privatización del array de reducción y basadas en la partición del array de reducción. La tabla 6.1 compendia algunas de las técnicas más importantes, resaltando sus ventajas e inconvenientes [41].

El método más sencillo e intuitivo para resolver la paralelización de un bucle

Tabla 6.1: Técnicas de paralelización de bucles de reducción

	Exclusión mutua	**Privatización del array de reducción**	**Particionamiento del array de reducción**
Implementaciones	Secciones críticas Locks de grano fino Operaciones atómicas	Privatización pura Expansión de array Privatización selectiva	LocalWrite (LW) SynchWrite (SW)
Ventajas	Bajo esfuerzo de programación	Bajo esfuerzo de programación Alta concurrencia	Explotación de localidad Bajo requerimiento de memoria
Desventajas	Serialización potencial Alto coste de sincronización Sin explotación de localidad	Alto requerimiento de memoria Sin explotación de localidad	Fase de inspección requerida Replicación de computación (LW) Carga de trabajo no balanceada (SW)

de reducción es bloqueando el acceso concurrente a los elementos del array de reducción mediante el uso de una ***sección crítica***. Utilizando primitivas y variables de exclusión mutua, tales como *locks*, se establece un bloque de acceso atómico que asegura que en un instante determinado, solo un procesador puede estar ejecutando la operación de reducción, lo cual garantiza la corrección del resultado final (Figura 6.4(a)). El principal inconveniente de este tipo de solución es que serializan la ejecución del bucle de reducción, además de incurrir en un *overhead* significativo (adquisición y liberación del *lock*). Esta serialización puede disminuirse mediante la utilización de ***locks de grano fino*** (Figura 6.4(c)), aunque esto requiere de un array de *locks* con tantas posiciones como elementos en el array de reducción, lo cual incrementa aún más el *overhead* de esta solución. El uso de ***operaciones atómicas*** (Figura 6.4(b)) es a menudo una alternativa mucho más eficiente, sin embargo, su disponibilidad está ligada a la arquitectura hardware y presentan serias limitaciones.

El segundo grupo de métodos para resolver el problema consiste en usar ***privatización***. La privatización consiste en proporcionar copias privadas del array de reducción a cada procesador, de manera que puedan operar concurrentemente, evitando la posibilidad de accesos conflictivos a un mismo elemento. La utilización de este método requiere de dos procesos adicionales, que deben ser ejecutados antes y después del bucle de reducción, con el fin de inicializar las copias privadas (*copy-in*) y acumular los valores parciales sobre el array original (*copy-out*). Varias técnicas han sido propuestas en la bibliografía para resolver la paralelización de bucles de reducción usando la idea de privatización.

La ***reducción privatizada*** [42], consiste en realizar tantas copias del array de reducción como procesadores, de manera que cada uno realice un porción de iteraciones del bucle sobre su copia privada del array (Figura 6.4(d)). Los principales inconvenientes de esta propuesta son la cantidad de memoria ne-

cesaria para las copias privadas del array, así como el coste computacional de su inicialización (*copy-in*) y sobre todo de la etapa final de agregación, la cual hay que realizar secuencialmente o bien en paralelo, garantizando la exclusión mutua de las escrituras sobre el array original mediante el uso de primitivas de sincronización.

La ***expansión del array de reducción*** [24] también emplea la idea de reducción privatizada, con la diferencia de que no se utilizan copias privadas del array de reducción en cada procesador, sino que el array global original se expande añadiendo una nueva dimensión, indexada por el identificador del procesador (Figura 6.4(e)). La principal ventaja de esta técnica respecto a la reducción privatizada, es que no requiere del uso de mecanismos de exclusión mutua en la etapa final de agregación (*copy-out*), ya que al utilizarse un mismo array global expandido, el total de elementos de dicho array pueden repartirse entre los procesadores disponibles, de manera que cada procesador pueda efectuar concurrentemente la acumulación de todos los valores parciales sobre los elementos asignados.

Puesto que el principal inconveniente de las técnicas basadas en privatización es que se requiere una copia del objeto privatizado por hilo, se han propuesto diferentes mejoras orientadas a disminuir el *overhead* de computación y, principalmente, de memoria que ello implica. Podemos mencionar entre ellas: *la expansión enlazada del array de reducción*, *la privatización selectiva* y *las tablas de reducción* [84].

La ***expansión enlazada del array de reducción*** es una evolución de los métodos de privatización cuyo objetivo es reducir la carga computacional asociada a la etapa de acumulación final. Para ello, se añade un campo adicional a cada elemento accedido del array para mantener una lista enlazada de los procesadores que han escrito cada elemento. En la etapa de acumulación final, solo se utilizarán los valores de las copias privadas de los procesadores presentes en esta lista, evitando así acumulaciones innecesarias con el elemento neutro (presente en procesadores que no han escrito el elemento). Este método presenta ventajas cuando los accesos al array de reducción son dispersos y solo representan un pequeño porcentaje del total de elementos.

Con el fin de reducir la memoria adicional requerida por la expansión enlazada se propone la ***privatización selectiva*** en la que el tamaño del array expandido no será el mismo que el del array de reducción original, sino que solo se replicarán aquellos elementos que sean accedidos por más de un procesador (aquellos cuya lista enlazada tenga más de dos elementos), utilizando una función de mapeo para transformar los índices originales al nuevo espacio de índices. El incon-

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3   Calculate ξ1,ξ2,...,ξn
4   #pragma omp critical
5   {
6     A[f1[i]]=A[f1[i]] ⊕ ξ1
7     A[f2[i]]=A[f2[i]] ⊕ ξ2
8     ...
9   }
10 }
```

(a) Lock grano grueso (sección crítica)

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3   Calculate ξ1,ξ2,...,ξn
4   #pragma omp atomic
5   A[f1[i]]=A[f1[i]] ⊕ ξ1
6   #pragma omp atomic
7   A[f2[i]]=A[f2[i]] ⊕ ξ2
8   ...
9 }
10
```

(b) Operaciones atómicas

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3   Calculate ξ1,ξ2,...,ξn
4
5   omp_set_lock(&my_lock[f1(i)]);
6     A[f1[i]]=A[f1[i]] ⊕ ξ1
7   omp_unset_lock(&my_lock[f1(i)]);
8
9   omp_set_lock(&my_lock[f2(i)]);
10    A[f2[i]]=A[f2[i]] ⊕ ξ2
11  omp_unset_lock(&my_lock[f2(i)]);
12  ...
13 }
14
```

(c) Locks grano fino

```
1 #pragma omp parallel private(Ap)
2 {
3   for (i=0; i<ADim; i++)
4   Ap[i] = 0;
5   #pragma omp for
6   for (i=0; i<fDim; i++){
7     Calculate ξ1,ξ2,...,ξn
8     Ap[f1[i]]=Ap[f1[i]] ⊕ ξ1
9     Ap[f2[i]]=Ap[f2[i]] ⊕ ξ2
10    ...
11  }
12  for (i=0; i<ADim; i++){
13    #pragma omp atomic
14    A[i] = A[i] ⊕ Ap[i];
15  }
16 }
```

(d) Privatización de Array

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3  Calculate ξ1,ξ2,...,ξn
4  A[tid][f1[i]]=A[tid][f1[i]] ⊕ ξ1
5  A[tid][f2[i]]=A[tid][f2[i]] ⊕ ξ2
6  ...
7 }
8 #pragma omp parallel for
9 for (i=0; i<fDim; i++){
10 for (tid=0; tid<nThreads; tid++)
11   A[i]=A[i] ⊕ A[tid][i];
12 }
13
```

(e) Expansión de Array (inicialización omitida)

```
1 for (d=0; d<nThreads; d++)
2  for (s=0; s<d+1; s++){
3   #pragma omp parallel for
4   for (m=s; m<nThreads-d; m+=d+a){
5    i=init(m,d);
6    cnt=count(m,d);
7    for (k=0; k<cnt; k++){
8     Calculate ξ1,ξ2,...,ξn
9     A[f1[i]]=A[f1[i]] ⊕ ξ1
10    A[f2[i]]=A[f2[i]] ⊕ ξ2
11    ...
12   }
13  }
14 }
```

(f) Fase de ejecución de SynchWrite

Figura 6.4: Alternativas para paralelizar un bucle de reducción en estilo OpenMP

veniente de este método es el *overhead* que supone determinar los elementos a expandir y el mapeo de los índices, por lo que para hacer provechosa esta aproximación, se requiere que el bucle sea ejecutado un alto número de veces con el mismo patrón de acceso (alta reusabilidad).

Cuando el tamaño del array de reducción es desproporcionado, entonces los métodos anteriores no son aplicables debido al gran gasto de memoria y al excesivo *overhead* en las fases de inicialización y acumulación. El método de las ***tablas de reducción*** [101] permite resolver estos inconvenientes gracias al uso de un par de tablas por procesador cuya extensión es del mismo orden, o menor, que el número de reducciones a realizar por el procesador. Estas tablas, accedidas aplicando una función *hash* al índice del array original, son utilizadas para almacenar los resultados de las reducciones y los índices de los elementos sobre los que se realiza. En una primera fase se realiza la primera ejecución del bucle y se rellenan las tablas por primera vez. La segunda fase se aplica a las ejecuciones posteriores del bucle donde se realiza una privatización selectiva de los elementos realmente accedidos.

Finalmente, existe un tercer grupo de técnicas que evitan la privatización realizando un particionamiento del array de reducción. Estas se basan en el uso del ***paradigma inspector/ejecutor***, en las que el patrón de acceso al array de reducción es analizado en *runtime* por una fase de inspección lo que permite determinar la computación que debe ser asignada a cada procesador. En este grupo podemos incluir métodos como LocalWrite [44] y SynchWrite [40] (ver Figura 6.4(f))

Si está garantizado que un bucle dado cumple las condiciones de reducción, porque ha sido reconocido por el programador o el compilador, las técnicas de paralelización descritas anteriormente proporcionan una solución eficiente para explotar la concurrencia en las arquitecturas *multicore* actuales. Sin embargo, no es fácil anticipar cual de estas técnicas es la mejor para un determinado problema [101, 51] debido a que las características del patrón de acceso (número de reducciones, indirecciones, referencia de localidad, contención de memoria, ...) pueden tener una influencia significativa en el rendimiento. Además, las limitaciones de la arquitectura tales como la memoria disponible o los costes de sincronización, necesitan ser evaluadas.

6.2. Reducciones en memoria transaccional

El uso de transacciones para la paralelización de reducciones como reemplazo directo de secciones críticas constituye una solución trivial. El modelo de ejecución propuesto por las transacciones es similar al uso de locks de grano fino protegiendo localizaciones de memoria individuales [92, 52]. La figura 6.5 muestra diferentes opciones para la paralelización de reducciones en un sistema TM, usando notación OpenTM. En la figura 6.5 (a) la transacción cubre solamente una operación de reducción dejando la carga de trabajo del bucle fuera. Esta opción minimiza el trabajo de requerido para reiniciar la transacción cuando ha abortado, pero a cambio del coste por el incremento del número de transacciones.

La unión de todas las operaciones de reducción en la misma transacción permite reducir el *overhead* total de la operación de *commit* pero a cambio de incrementar la probabilidad de aborto (figura 6.5 (b)). Si el *overhead* de iniciar/finalizar la transacción es muy alto, se pueden agrupar varias iteraciones del bucle en una misma transacción, como expresa la construcción `transfor` de OpenTM en la figura 6.5 (c). Sin embargo, el programador tiene que tener en mente que ahora la carga de trabajo está incluida en la transacción, de manera que el coste de reiniciar una transacción abortada supone además, la reejecución de esta carga.

Este uso de las transacciones para resolver la paralelización de bucles reductivos puede ser una solución tan efectiva como las secciones críticas basadas en *locks*, o incluso mejor. Sin embargo, cualquier conflicto detectado en memoria compartida ocasiona un aborto (o *stall*) de una transacción, degradando el rendimiento, lo cual ocurre frecuentemente en los bucles de reducción, si el acceso al objeto de reducción presenta una alta contención.

6.3. Soporte transaccional para reducciones

Las propiedades asociativas y conmutativas del operador de reducción pueden emplearse para filtrar conflictos en objetos de reducción, evitando abortos (o *stalls*) innecesarios de transacciones.

Lo que proponemos en esta tesis es incorporar un soporte específico para las operaciones de reducción en sistemas TM, al que nos referiremos, en lo siguiente, como *R-TM*. Este soporte se podría aplicar de entrada en la mayoría de sistemas TM tanto software como hardware. Esta propuesta requiere modificaciones livianas, con un bajo coste de implementación, en el mecanismo de gestión de

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3     Calculate ξ1,ξ2,...,ξn
4     #pragma omp transaction
5       A[f1[i]]=A[f1[i]] ⊕ ξ1
6     #pragma omp transaction
7       A[f2[i]]=A[f2[i]] ⊕ ξ2
8     ...
9 }
```

(a) Transacciones de grano fino

```
1 #pragma omp parallel for
2 for (i=0; i<fDim; i++){
3     Calculate ξ1,ξ2,...,ξn
4     #pragma omp transaction
5     {
6         A[f1[i]]=A[f1[i]] ⊕ ξ1
7         A[f2[i]]=A[f2[i]] ⊕ ξ2
8         ...
9     }
10 }
```

(b) Transacciones de grano grueso

```
1 chunk = fDim/nThreads;
2 #pragma omp transfor schedule(static, chunk, xactSize)
3 for (i=0; i<fDim; i++){
4     Calculate ξ1,ξ2,...,ξn
5     A[f1[i]]=A[f1[i]] ⊕ ξ1
6     A[f2[i]]=A[f2[i]] ⊕ ξ2
7     ...
8 }
```

(c) Transacciones de grano extra grueso

Figura 6.5: Utilización de la aproximación TM como reemplazo directo de secciones críticas.

versiones y la fase de *commit* del sistema transaccional subyacente. Por ello, debe ser considerado como una optimización al sistema transaccional más que como una propuesta software en sí misma.

El soporte R-TM que se plantea en este capítulo está restringido a un escenario simplificado en el que la totalidad de los conflictos de memoria proceden de las variables de reducción, y por tanto, el orden de ejecución de las iteraciones del bucle puede ser alterado manteniendo la corrección del resultado. Un caso de

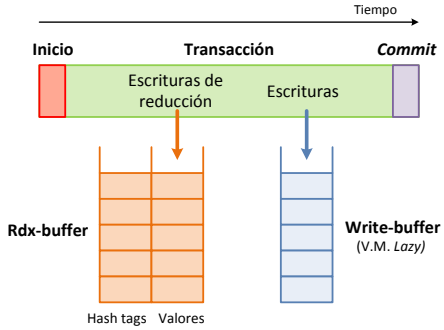


Figura 6.6: *Buffer* dedicado para escrituras de reducción en R-TM.

interés que involucre conflictos de memoria adicionales además de los conflictos reductivos (e.g, reducciones parciales), sí que requeriría que el orden de las iteraciones del bucle, o mas concretamente, el orden de las dependencias entre iteraciones, sea preservado. Este tipo de escenario puede ser manejado utilizando el soporte de reducciones propuesto sobre un sistema que garantice este orden, tal como nuestro modelo TEPO (Capítulo 4) o alguna otra propuesta ordenada. Sin embargo, en este capítulo y con el fin de simplificar la explicación del soporte de reducción, consideraremos únicamente este escenario simplificado , el cual puede ser considerado como un paso previo a la extensión de la propuesta a otras situaciones en las que sí puedan existir otro tipo de conflictos aparte de los de reducción.

La idea detrás de nuestro soporte es aprovechar el aislamiento que proporciona la memoria transaccional sobre sus escrituras con el fin de implementar una forma de privatización selectiva sobre las variables de reducción. Para ello asumiremos un sistema transaccional de base con una política de gestión de versiones *lazy* que simplifica el mecanismo adicional para implementar la privatización, debido a que las escrituras se mantienen invisibles a las otras transacciones hasta el *commit*. Por otro lado, la detección de conflictos puede implementar aproximaciones tanto *eager* como *lazy*. Esta decisión realmente no es crítica si el sistema únicamente paraleliza bucles reductivos, aunque sí deberá ser analizada teniendo en cuenta el resto de la aplicación. En la implementación propuesta se utilizará un sistema con detección de conflictos *eager*.

Considerando un sistema base con estas características, las modificaciones requeridas para implementar nuestro soporte para reducciones se describen a continuación:

Rdx-buffer: Un nuevo almacenamiento privado denominado *reduction buffer*

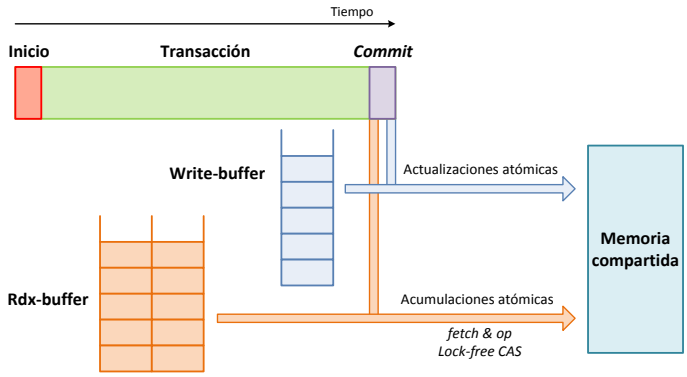


Figura 6.7: Mecanismo de *commit* para los diferentes *buffers* en el soporte de reducción R-TM.

(Rdx-buffer) es añadido a cada transacción (figura 6.6). Este *buffer* almacena todas las escrituras, sobre objetos de reducción, efectuadas por la transacción. Estas escrituras se mantienen separadas del resto de escrituras transaccionales (que utilizan el clásico *write buffer*) ya que deben ser tratadas de manera diferente durante la fase de *commit*, con el fin de acumularlos correctamente con los datos globales en memoria. El *Rdx-buffer* puede ser implementado como una tabla que almacena conjuntamente tres campos: la dirección de memoria perteneciente a la variable de reducción, el nuevo valor con el que hay que reducir y la operación de reducción que se debe aplicar. De manera que en cada escritura, el *Rdx-buffer* es accedido para determinar si la dirección de memoria fue anteriormente escrita. Si es así, el campo que contiene el valor del objeto de reducción es actualizado de acuerdo con la operación de reducción y el nuevo valor. En caso contrario, se genera una nueva entra entrada en la tabla para la dirección y el nuevo valor. Aunque este proceso podría acelerarse simplemente añadiendo una nueva entrada al *Rdx-buffer*, la reutilización nos permite reducir en gran medida la cantidad de memoria utilizada.

Lecturas y escrituras de reducción: Todas las lecturas y escrituras a variables de reducción en una transacción deben ser instrumentadas de manera adecuada. Los sistemas STM y los HTM explícitos ya proporcionan primitivas/instrucciones especiales para identificar las operaciones de memoria transaccionales. Por otro lado, en los HTM implícitos solo se especifican los límites de la transacción, y todos las accesos de memoria dentro de ellos son considerados transaccionales. En tales sistemas, se pueden definir instrucciones especiales ISA con el fin de distinguir entre lecturas y escrituras sobre reducciones, del resto de operacio-

Tabla 6.2: Primitivas de memoria necesarias para el soporte R-TM

Operación de memoria	Descripción
Lectura reductiva `tmrload`	Busca una dirección de memoria en el *Rdx-buffer* y devuelve el valor si la encuentra. En caso contrario se devuelve el valor por defecto (elemento identidad). La dirección de memoria no se añade al *read set*.
Escritura reductiva `tmrstore`	Busca una dirección de memoria en el *Rdx-buffer* y acumula el nuevo valor si la encuentra. La dirección de memoria no se añade al *write set*.

nes transaccionales. Un aproximación similar es seguida en [62] para proponer instrucciones adicionales utilizadas para optimizaciones de rendimiento.

La tabla 6.2 muestra una descripción de las primitivas propuestas para operaciones de reducción. Una lectura reductiva (`tmrload`) no lee de la memoria global. En su lugar, busca la dirección de memoria en el *Rdx-buffer* y toma el valor asociado a ella, si lo encuentra. Si la dirección de memoria no puede ser encontrada en el *Rdx-buffer*, la lectura devolverá el elemento neutro o identidad de la operación de reducción a que se ve sometida el objeto, como valor por defecto. Además, `tmrload` no añade la dirección al conjunto de lectura de la transacción en ejecución. Una operación de escritura transaccional (`tmrstore`) actúa de manera similar a la escritura transaccional clásica del sistema pero acumulando el nuevo valor (de acuerdo a la operación de reducción) con el almacenado en la entrada correspondiente del *Rdx-buffer*, en lugar de simplemente almacenar el nuevo valor en el *write buffer*. Además, `tmrstore` no añade la dirección de memoria al *write-set* de la transacción ni la incluye en el procedimiento utilizado para la detección de conflictos del sistema transaccional (adquirir *lock* asociado, insertar en filtros de *Bloom*, etc.).

Fase de Commit: El no incorporar las direcciones de memoria al sistema de detección de conflictos, permite evitar que los accesos concurrentes a objetos de reducción disparen el procedimiento de resolución de conflictos asociado. Este comportamiento es válido en nuestro escenario simplificado porque el objeto de reducción puede ser privatizado de manera segura. De hecho, el uso del *Rdx-buffer* descrito es equivalente a una privatización selectiva de los accesos reductivos dentro de la transacción. Consecuentemente, cuando una transacción alcanza su punto de *commit*, todos estos valores privatizados selectivamente deben ser acumulados apropiadamente en el objeto de reducción en memoria principal. Con el fin de conseguir esto, la fase de *commit* debe ser extendida para

```
1 chunk = fDim/nThreads;
2 #pragma omp transfor schedule(static, chunk, xactSize) reduction(+:A)
3 for (i=0; i<fDim; i++){
4     Calculate ξ1,ξ2,...,ξn
5     A[f1[i]]=A[f1[i]] ⊕ ξ1
6     A[f2[i]]=A[f2[i]] ⊕ ξ2
7     ...
8 }
```

Figura 6.8: Definición manual del objeto de reducción (notación OpenTM)

incluir un procedimiento que actualice atómicamente la memoria compartida con el contenido del *Rdx-buffer* (figura 6.7). El acceso mutuamente exclusivo a los datos compartidos puede ser asegurado utilizando los tradicionales *locks*. Sin embargo, existen otras opciones, tales como las operaciones atómicas, que reducen el elevado coste asociado a estas estructuras en caso de que sean soportadas por la arquitectura del procesador. Estas operaciones son normalmente implementadas bloqueando el bus de memoria o la caché.

Una tercera alternativa es el uso de algoritmos *lock-free*, basados en operaciones atómicas, tales como *compare-and-swap (CAS)* presentes en todas las arquitecturas actuales. Esta última opción es usual cuando se reducen variables de punto flotante, ya que disponer de un soporte nativo para operaciones atómicas de este tipo no es común.

6.4. Modelo de programación con R-TM

El soporte R-TM propuesto asume que el bucle de reducción se divide en transacciones del mismo tamaño (`xactSize` como se muestra en la figura 6.5 (c)). Estas transacciones son ejecutadas en paralelo, y cada una computa modificaciones parciales sobre la variable de reducción en su *Rdx-buffer* privado. Estos valores parciales son acumulados con la variable original cuando la transacción realiza el *commit*, liberando a su vez, el espacio utilizado en el *Rdx-buffer*.

Utilizando la notación OpenTM, los programadores pueden hacer uso de la clausula `reduction` en la construcción `transfor` para especificar sintácticamente, a nivel de programa, cuales son los objetos de reducción (figura 6.8). En un sistema que implemente nuestro soporte para reducciones, el compilador instrumentará todas los accesos a estos objetos de reducción dentro de una transacción, utilizando las primitivas descritas en la tabla 6.2. En este caso, se extiende la li-

Tabla 6.3: Paralelización de reducciones: R-TM vs. OpenMP

	R-TM		OMP (Reduction)		OMP (Critical)	OMP (Atomic)	
Tipo de variable de reduction	**Int**	**Float**	**Int**	**Float**	**Int/Float**	**Int**	**Float**
Privatización	Si		Si		No	No	
Sincronización hilos	Op. atómica	*Lock Free*	*Lock* / Op. atómica	*Lock* / *Lock Free*	*Lock*	Op. atómica	*Lock Free*
***Overhead* Privatización**	< O(#Hilos×tamaTx)		O(#Hilos×N)		0	0	
***Overhead* Sincronización**	O(#Transacciones)		O(#Hilos)		O(#Iterations)	O(#Iteraciones)	

mitación heredada de OpenMP para esta clausula, por la cual, solo las variables escalares pueden ser de reducción.

El modelo de ejecución que proporciona nuestro soporte R-TM resulta en una privatización selectiva y dinámica de los objetos de reducción, i.e, el *Rdx-buffer* almacena una nueva entrada solo cuando es requerido, y libera el espacio utilizado cada vez que la transacción realiza el *commit*. De esta manera el *overhead* extra de memoria debido a la privatización está limitado por el tamaño de la transacción. Suponiendo el peor caso posible, i.e, que cada transacción escriba en todas las posiciones del array de reducción, el consumo de memoria sería igual al de una privatización clásica del array completo. En la práctica, el tamaño de la transacción hará que el consumo de memoria sea significativamente menor.

Por otro lado, las acumulaciones en memoria principal, realizadas durante los *commits*, son protegidas por *locks* u operaciones atómicas, las cuales introducen algún *overhead* de sincronización entre hilos. El número total de sincronizaciones de hilos necesarias es igual al número de transacciones finalizadas. Observar que esta situación se corresponde con un caso intermedio entre paralelización de reducciones utilizando locks (u operaciones atómicas) y privatización total (*full privatization*) (figura 6.4).

El compromiso entre los *overhead* de la privatización y la sincronización dependerá del tamaño de la transacción, el cual es un parámetro que puede ser seleccionado por el programador/compilador. La tabla 6.3 resume la discusión anterior en comparación con los mecanismos proporcionados por OpenMP para la resolución de reducciones.

6.5. Evaluación

Las modificaciones propuestas, introducidas por nuestro soporte transaccional de reducciones (R-TM), se ha implementado de nuevo sobre TinySTM como sistema TM base. La evaluación experimental de esta implementación se ha realizado sobre la misma plataforma de evaluación utilizada en el capítulo anterior y cuyas principales características se resumen en la tabla 5.3. Tanto los códigos probados como el *runtime* STM fueron escritos en lenguaje *C*. En todas las ejecuciones, un único hilo ha sido asignado a cada núcleo de procesamiento (no considera *hyperthreading*).

Los resultados experimentales obtenidos corresponden a la media de un centenar de ejecuciones, con el fin de filtrar efectos aleatorios asociados con la ejecución transaccional.

6.5.1. *Benchmark* sintético: Histograma de imagen

Inicialmente, hemos evaluado el soporte R-TM utilizando un *benchmark* sintético que calcula el histograma de niveles de gris de una imagen. Este programa (descrito en la sección 3.2.3) está formado por un bucle con una única instrucción de reducción irregular (acumulación en el array histograma del número de píxel para cada nivel de gris) sin carga adicional de computación.

Los experimentos se han realizado para cuatro imágenes de entrada diferentes, creadas para evaluar distintas áreas de contención en el array histograma. Estas imágenes fueron generadas como un conjunto aleatorio de 4M píxeles con 256 diferentes niveles de gris. Las áreas de contención para el histograma han sido obtenidas seleccionando diferentes distribuciones *Gausianas* truncadas de los valores de gris.

La figura 6.9 muestra la función de densidad de probabilidad para las imágenes evaluadas. Todas las distribuciones han sido definidas en el rango $[0, 255]$ con la media localizada en 0, excepto para la primera imagen cuya media esta situada en 127. Las imágenes difieren en la desviación estándar (σ), clasificadas desde 0 (imagen uniforme con distribución de gris focalizada en una localización del array histograma de máxima contención) hasta infinito (imagen uniforme con distribución de gris aleatoria, sin áreas de contención). También se han seleccionado dos valores de desviación estándar, 16 y 70, los cuales introducen áreas de contención estrechas y amplias, respectivamente.

Los resultados experimentales del soporte R-TM para el programa de pruebas

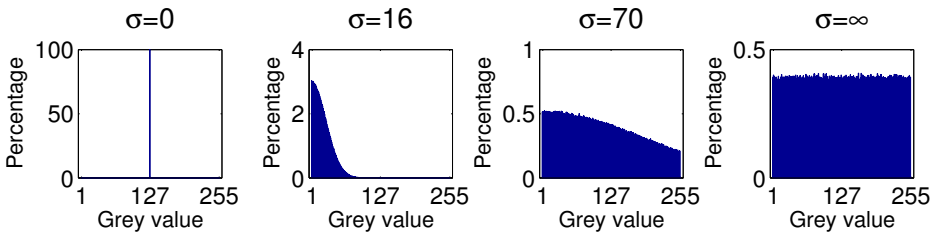


Figura 6.9: Densidad de probabilidad para las imágenes de prueba utilizadas en el *benchmark* Histograma

descrito se muestran en la figura 6.10, en términos de aceleración con respecto al sistema base TinySTM (*speedup* relativo), *Transaction Commit Rate* (TCR) y *overhead* de memoria incurrido por el *Rdx-buffer* durante la ejecución transaccional, expresado como el porcentaje de memoria utilizado con respecto a una privatización total del array de reducción (ver sección 3.3 para más información sobre las métricas).

El *speedup* de R-TM respecto a la implementación base de TinySTM (figuras 6.10(a, d, g, j)), para diferentes matrices de entrada) es particularmente interesante. Los altos valores para el *speedup* relativo son causados básicamente por las grandes disparidades exhibidas por las tasas de aborto. Mientras que R-TM muestra unas tasas de aborto nulas (ya que todos los conflictos son originados por la operación de reducción, y por tanto son ignorados), TinySTM experimenta altas tasas de abortos, debido principalmente a la contención sobre el pequeño array de reducción (256 elementos), la cual incrementa con el número de hilos.

De hecho, en el caso de máxima contención ($\sigma = 0$), R-TM es casi 90 veces más rápido que TinySTM para 32 hilos y transacciones de 100 (reducciones) iteraciones del bucle. Este *speedup* relativo decrece a medida que σ incrementa, ya que el área de contención sobre el array histograma se extiende. Como resultado, la tasa de abortos en TinySTM es menor (el patrón de conflictos es distribuido a través de las transacciones) y el *overhead* asociado al *commit* para R-TM crece (más valores en el *Rdx-buffer* para acumular en memoria). Nótese que, en general, R-TM rinde mejor para tamaños de transacciones entre 10 y 100 iteraciones del bucle de reducción, debido a este aspecto.

Las medidas de TCR para este *benchmark* se presentan en las figuras 6.10 (b, e, h, k) para las diferentes imágenes de entrada. Dado que R-TM lleva a cabo una privatización selectiva implícita, en la práctica, esto elimina todos los abortos causados por conflictos sobre el array de reducción. Este hecho no ocurre en

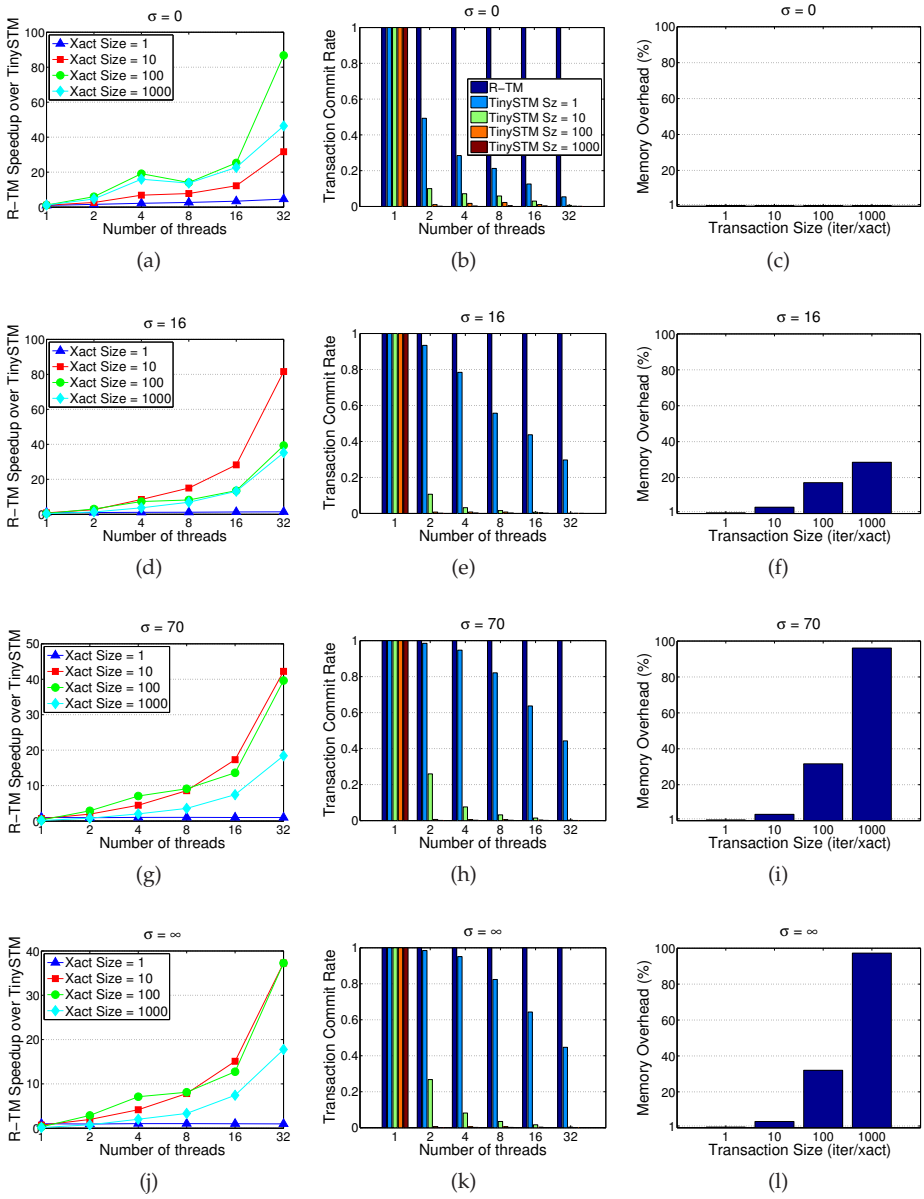


Figura 6.10: Resultados experimentales para el *benchmark* Histograma.

Tabla 6.4: Cargas de trabajo para las aplicaciones reales

	Legendre	MD2	Euler			Fluidanimate	
# Bucles de reducción	1	1	3			2	
# Reducciones por bucle	8	4	2	3	4	2	6
# Elementos del array de reducción	95K	160K	3K	18K	2K	135K	135K
# Iteraciones del bucle de reducción	30M	24M	9M			80M	

el sistema TM base, resultando en una importante pérdida de concurrencia. Además, la concurrencia explotada por TinySTM también decrece con el tamaño de la transacción, ya que las transacciones más grandes abortarán más veces antes de alcanzar con éxito su *commit*. Es importante observar que el sistema base no presenta una buena escalabilidad, ni con respecto al número de hilos ni al tamaño de la transacción. Mientras que para 32 hilos el TCR cae un 32 % de media, un tamaño de transacción de 10 iteraciones hace que la concurrencia de desplome alrededor de un 70 % de media, ya que ambos parámetros incrementan el número de conflictos de datos.

Finalmente, las figuras 6.10 (c, f, i, l) muestran el porcentaje de espacio en memoria (por hilo) requerido por el *Rdx-buffer* para las diferentes matrices de entrada, respecto al incurrido por una solución basada en privatización total (*overhead* de memoria del 100 %). Como se esperaba, la peor situación se da para transacciones grandes. En este *benchmark*, el *overhead* de memoria para el tamaño de transacción más grande evaluado (1000 iteraciones) es básicamente equivalente al método de privatización total. Sin embargo, para tamaños de transacción más pequeños, R-TM permite ahorrar una gran cantidad de espacio en memoria ya que el *Rdx-buffer* crece dinámicamente actuando como una efectiva privatización selectiva. En el caso de los tamaños de transacciones con un mejor *speedup* relativo (entre 10 y 100 iteraciones por transacción), el *overhead* de memoria debido al *Rdx-buffer* es de entre el 5 % y 35 % del requerido por una estrategia de privatización total.

6.5.2. Aplicaciones reales

Se han evaluado, así mismo, otros cuatro códigos representativos de aplicaciones reales que invierten una parte significativa de su tiempo de ejecución en bucles de reducción irregulares: *Euler* [29], transformada de *Legendre* [71], *2D Molecular dynamics (MD2)* [70] y *Fluidanimate* [11]. La tabla 6.4 recoge aquellos aspectos que definen la carga computacional de los experimentos, como la dimensión de los arrays de reducción, su número, y el número de iteraciones.

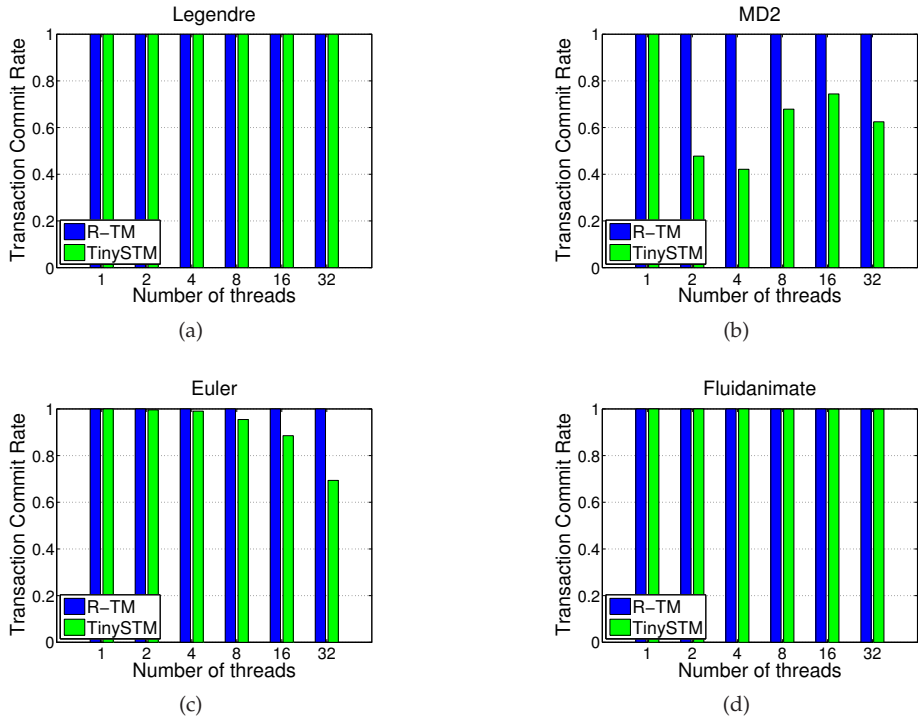


Figura 6.11: *Transaction commit rate* (TCR) de R-TM y TinySTM para distinto número de hilos.

La figura 6.11 muestra las medidas de TCR para los cuatro *benchmarks* utilizando un tamaño de transacción fijo (10 iteraciones). Podemos apreciar como la mejora de concurrencia obtenida por R-TM es menor para *Fluidanimate* y *Legendre*. Aunque nuestra propuesta siempre obtiene una concurrencia óptima, el bajo número de conflictos de datos en estos códigos hacen que TinySTM se mantenga también cerca de esta máxima concurrencia. Por otro lado, tanto *Euler* como *MD2* presentan un mayor número de dependencias de datos (moderado para el primero y elevado para el segundo).

La figura 6.12, por su parte, muestra los *speedup* relativos y comparativos de R-TM sobre la implementación base TinySTM (véase sección 3.3). *Legendre* presenta un muy bajo número de conflictos (niveles de TCR cercanos al 99 % en TinySTM) además de un coste por aborto transaccional muy reducido (baja carga computacional en transacción), lo cual explica el moderado comportamiento del

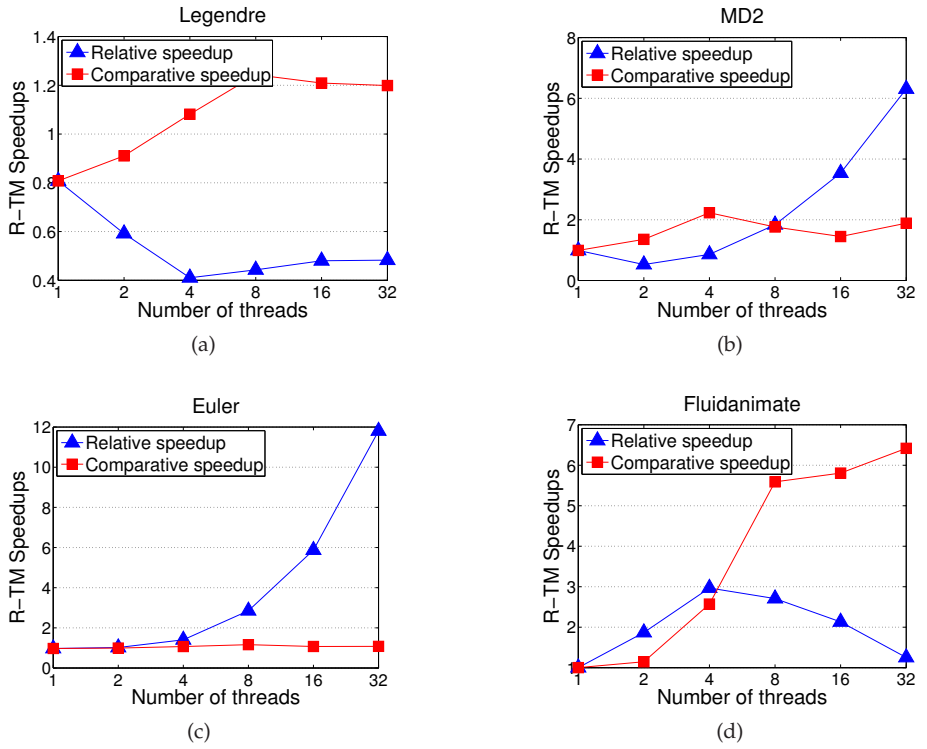


Figura 6.12: *Speedup* relativo y comparativo de R-TM frente a TinySTM. El *speedup* relativo se calcula de acuerdo a la expresión $\frac{time_{TinySTM}(1\ thread)}{time_{R-TM}(n\ threads)}$, y el *speedup* comparativo se calcula de acuerdo a $\frac{time_{TinySTM}(n\ threads)}{time_{R-TM}(n\ threads)}$.

speedup comparativo en este *benchmark* (código afectado de *memory bound*). *Fluidanimate* también muestra un comportamiento similar en cuanto al número de conflictos, sin embargo, el coste de los abortos es significativamente mayor. Esto se refleja en un mejor *speedup* comparativo por parte de R-TM. Por su parte, *Euler* también presenta mayor carga computacional en transacción que *Legendre*, aunque no tan grande como *Fluidanimate*. Esto, unido a que el número de conflictos es relativamente bajo (valores de TCR alrededor de 0.7 para TinySTM) explican porque el *speedup* comparativo para R-TM no alcanza valores superiores. Por último, *MD2* aunque muestra un gran número de conflictos, el coste asociado a estos no es muy alto (carga computacional entre baja y moderada). En este *benchmark* se puede observar como el máximo valor para el *speedup* comparativo

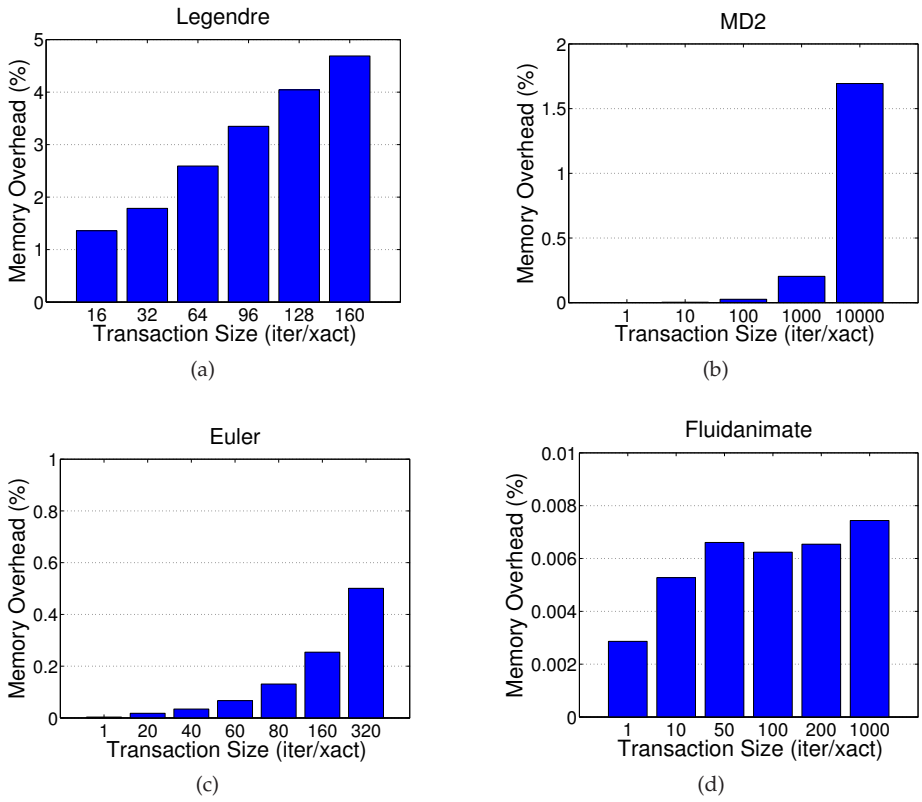


Figura 6.13: *Overhead* de memoria medio por hilo para R-TM en relación a una privatización total, utilizando distintos tamaños de transacción (iteraciones por transacción).

(2.2x para 4 hilos) se alcanza para el mínimo valor en la curva del TCR (0.45 para TinySTM). De la anterior discusión se puede deducir que el coste asociado a los abortos transaccionales tiene más impacto sobre el rendimiento que el número de conflictos, no tanto por el tiempo invertido en restaurar el estado previo de la transacción, sino por la cantidad de computación desechada que necesita volver a ser ejecutada.

En relación a *speedup* relativo, se pueden observar dos tendencias. Una incluye aquellas aplicaciones con carga computacional moderada en transacciones (coste del aborto moderado) y número de transacciones bajo/moderado (ver tabla 6.4), tales como *MD2* y *Euler*. Estos *benchmarks* muestran un buen y escalable *speedup*

relativo, ya que R-TM reduce a cero el coste de los abortos (porque no los hay) y el código presenta un buen equilibrio entre carga de trabajo transaccional y número de transacciones. *Legendre*, por otro lado, tiene un alto número de transacciones con una muy baja carga computacional. Como resultado, el *speedup* relativo es pobre debido al impacto del *overhead* transaccional. *Fluidanimate* tiene el mayor número de transacciones y además de una carga computacional bastante alta. Esto hace que el *speedup* relativo se incremente, casi linealmente, para un número pequeño de hilos, sin embargo no es escalable, debido al *overhead* asociado a tal elevado número de transacciones.

Finalmente, la figura 6.13 muestra la memoria requerida por nuestra propuesta R-TM en comparación a un método de privatización total. En todos los casos, la memoria requerida por R-TM es mucho menor que el máximo (replicación total del array de reducción). El incremento del tamaño de transacción tiene un efecto dañino sobre el *overhead* de memoria, sin embargo hay que tener en cuenta que aunque el uso de transacciones pequeñas tiene menores requerimientos de memoria, esto también incrementará la cantidad de *commits*. Nótese que *Fluidanimate* necesita de una cantidad extremadamente pequeña de espacio extra, debido a que nuestra versión transaccional está basada en un código que inicialmente separaba las partículas que generan conflicto de las que no lo hacen. Así, solo las partículas que tienen conflicto con otras (asociadas a las fronteras) están incluidas dentro de transacciones, mientras que el resto (la mayoría de partículas) operan directamente sobre el array de reducción global.

7 Conclusiones y trabajos futuros

Desde la popularización de los sistemas *multicore*, la programación paralela se ha convertido en un paradigma imprescindible para conseguir mayor eficiencia en los tiempos de ejecución de las aplicaciones a través del aprovechamiento de la concurrencia. Especialmente, la normalización de estas arquitecturas como entornos de desarrollo y explotación de uso común, ha elevado el número de programadores que trabajan sobre ellas. Sin embargo, el desarrollo de versiones paralelas eficientes, y por tanto la explotación de la concurrencia, no es trivial en la mayoría de situaciones, lo cual exige a los programadores una alta preparación y un alto nivel de esfuerzo para comprender la naturaleza del problema y aplicar los métodos necesarios para alcanzar dicho objetivo.

En esta tesis hemos abordado la temática de la paralelización de aplicaciones utilizando estrategias basadas en la memoria transaccional. Este paradigma simplifica el trabajo de programación, ayudando a desarrollar versiones paralelas de las aplicaciones sin necesidad de disponer de un conocimiento profundo acerca de la aplicación, de una manera total o parcialmente automática. En este ámbito de trabajo, nos hemos centrado en aquellos códigos que presentan restricciones de precedencia que determinan dependencias de datos cuyo orden debe satisfacerse para la obtención de un resultado correcto.

A continuación resumimos las conclusiones de cada uno de los apartados de esta memoria, finalizando con una exposición de las futuras líneas de trabajo derivadas de esta tesis.

El primer paso realizado ha sido analizar y comentar los diferentes tipos de construcciones que presentan restricciones de precedencia y que pueden ser

encontrados por los programadores cuando desarrollan versiones paralelas de sus aplicaciones. Tras esta revisión se ha concluido que los principales aspectos que dificultan y/o limitan la paralelización utilizando mecanismos clásicos son:

- **Los patrones de acceso**: Uno de los principales problemas se encuentra en los patrones de acceso a las estructuras de datos compartidas cuando estos son irregulares (basados en indirecciones, punteros, etc.), son extremadamente complejos como para ser calculados en tiempo de compilación, o bien, dependen de los datos de entrada a la aplicación. En todos estos casos se hace imposible resolverlos, y por tanto, determinar las dependencias de datos antes de la ejecución.

- **Las restricciones de orden**: En la mayoría de aplicaciones, la necesidad de aplicar sobre la información determinados procesos u operaciones siguiendo un orden específico restringe o dificulta la paralelización puesto que la corrección del resultado está condicionada al cumplimiento de dicho orden.

Ambos aspectos condicionan conjuntamente el mecanismo utilizado para explotar concurrencia, ya que no solo se requiere un sistema dinámico capaz de adaptarse y gestionar los conflictos de datos potenciales en tiempo de ejecución, sino que además se precisa mantener un orden que no viole las relaciones de precedencia entre operaciones de memoria conflictivas para alcanzar un resultado correcto.

Para hacerle frente a ambos aspectos hemos desarrollado y validado un modelo de ejecución transaccional TEPO (*Transaction Execution under Precedence Order*), que mantiene las relaciones de precedencia garantizando la semántica secuencial del programa original. Para establecer una relación de orden, el modelo admite la definición de un identificador de orden por transacción (aunque no es obligatorio) y dos condiciones de *commit* (una estricta y otra más relajada). Además, se ha ampliado el gestor de conflictos transaccional para hacerlo sensible al orden de precedencia transaccional.

Todo ello permite la paralelización de código secuencial sin intervención del programador siempre que el compilador se encargue de la detección de las construcciones potencialmente paralelas. El programador también puede utilizar este soporte para facilitar el desarrollo de aplicaciones paralelas marcando los bucles o las fronteras de las secciones presumiblemente conflictivas, dejando de lado los aspectos de sincronización y dependencias de datos que serán responsabilidad del sistema transaccional y del modelo de ejecución propuesto.

Para mantener dicho orden sin incurrir en una espera excesiva, se ha propuesto desacoplar las fases de ejecución y *commit* de la transacción, introduciendo un nuevo estado transaccional para mantener las transacciones ejecutadas que no han realizado el *commit*, de manera que el sistema pueda aprovechar este tiempo para iniciar y ejecutar nuevas transacciones.

Utilizando una implementación realizada sobre un conocido STM, enfocada especialmente a bucles, hemos validado nuestro modelo comprobando que se garantiza la corrección del resultado, manteniendo el orden definido, al tiempo que se incrementa significativamente la concurrencia explotada en relación a la versión que únicamente preserva el orden. Hemos determinado que este incremento de la concurrencia se debe principalmente a dos factores.

- Muchas dependencias de datos presentes, concretamente, todas las antidependencias (*WAR*) en el contexto secuencial, son filtradas totalmente por el orden impuesto. Esto evita que el gestor de conflictos dispare el aborto de transacciones sometidas a este tipo de dependencias de datos, cuya ejecución en el nuevo contexto ordenado, es completamente segura. Por otra parte, el aplicar una condición de *commit* estricta para aquellas transacciones que presenten dependencias de salida (*WAW*) consigue garantizar la consistencia de la ejecución, sin necesidad de abortar ninguna de ellas, lo que reduce su impacto negativo sobre el rendimiento.

- La ejecución continua de transacciones, propiciada por el desacoplo de las fases de ejecución y *commit*, consigue ocultar parte de la latencia entre ambas fases causada por el orden, al tiempo que incrementa la concurrencia. Este mayor número de transacciones en ejecución hace que se incrementen los recursos necesarios para mantener activos los contextos de las transacciones en espera de *commit*, lo cual podría llegar a afectar al rendimiento, limitando en parte esta mejora. Sin embargo, durante la evaluación se ha comprobado que, en la práctica, para bucles iterativos que por lo general mantienen un buen balanceo de carga (computacional), el número de transacciones activas por hilo es, entre bajo y moderado dependiendo de la aplicación.

En los experimentos llevados a cabo, el *overhead* observado apenas supone un sobre coste del 7 % respecto de la implementación de base del STM utilizado, considerando además, que este último no mantiene el orden.

Adicionalmente, se ha desarrollado un soporte especial para bucles de reducción, en aquellos casos en los que todas las dependencias provienen de los

objetos de reducción, que complementa el modelo TEPO y que permite lidiar con las especiales características de estas construcciones. Así, tras un análisis en profundidad de los aspectos claves de las reducciones y de revisar los principales métodos clásicos para su paralelización, hemos desarrollado un soporte transaccional efectivo.

Nuestra propuesta, denominada R-TM, queda definida por dos aspectos claves. Un *buffer* de reducción (*Rdx-buffer*) se usa para mantener las escrituras transaccionales realizadas sobre las variables de reducción. La inclusión de esta nueva estructura, separada del *write buffer* convencional, facilita y organiza el proceso de *commit*, ya que las escrituras de reducción no sustituyen a los valores en memoria, sino que se acumulan con ella. Por otro lado, los accesos realizados por la reducción no se tienen en cuenta en la traza realizada por el gestor de conflictos, por lo que su acceso concurrente no provoca abortos entre las transacciones. Estos dos aspectos unidos permiten utilizar el entorno transaccional para aplicar una privatización selectiva y dinámica sobre las variables de reducción.

La primera consecuencia directa de esta aproximación es una reducción significativa de la memoria necesaria en la paralelización de bucles de reducción respecto a una solución basada en privatización total, la cual no llega al 5 % de la requerida por esta última para ninguna de las aplicaciones reales evaluadas. La segunda consecuencia está relacionada con el incremento de concurrencia ya que, gracias a la privatización, los accesos en los bucles de reducción no presentan conflictos. Esto se traduce en una explotación del paralelismo máxima (TCR igual a 100 %) para todos los *benchmarks* utilizados, como se ha demostrado en la evaluación realizada. Finalmente, en la comparación directa entre las implementaciones con y sin soporte para reducciones, se ha demostrado que esta ganancia de concurrencia se traduce en una mejora del tiempo de ejecución de las aplicaciones, aunque ésta se encuentra fuertemente influenciada no solo por la cantidad de abortos ahorrados respecto al sistema de base, sino también por la cantidad de trabajo que las transacciones realizan.

El punto negativo viene determinado por los *overheads* asociados a las implementaciones STM. A diferencia de las implementaciones basadas en HTM, los STMs incurren en un *overhead* asociado principalmente a los accesos a memoria y estructuras software (*buffers*, *write set*, *read set*, etc), así como a los mecanismos para mantener la consistencia (detección de conflictos implementada con *locks*, restauración de contextos, etc) que limitan en gran medida el factor de mejora de las propuestas realizadas en cuanto a tiempos de ejecución se refiere.

Pese a todo, podemos concluir que nuestro modelo presenta una alternativa simple y eficaz que no requiere de ningún (si se delega sobre el compilador la

instrumentación de bucles y secciones potencialmente paralelas) o muy poco esfuerzo y conocimiento por parte del programador, sobre el problema para obtener una versión paralela correcta y eficiente, de todo tipo de códigos con restricciones de precedencia (especialmente construcciones iterativas).

Por último, durante el desarrollo de esta tesis han surgido nuevas ideas y mejoras que abren el camino a posibles líneas de trabajo que abordar en el futuro:

- Realizar una implementación hardware el modelo TEPO reduciría significativamente los *overheads* transaccionales respecto a la implementación software, lo que provocaría una mejora de la eficiencia. Sin embargo, para ello habrá que resolver aspectos claves tales como la limitación del número de transacciones en espera de *commit*, ya que en este caso, su control estaría asociado al uso de estructuras hardware, las cuales están mucho más limitadas.

- Como se comentó durante la definición del modelo, aunque nosotros hemos basado la implementación particular realizada en una configuración *eager-lazy* (detección de conflictos - gestión de versiones), una configuración *eager-eager* también es posible si el sistema transaccional presenta una atomicidad fuerte. Podría ser interesante proponer una implementación del modelo TEPO basada en esta configuración y realizar un estudio comparativo acerca de cómo influye la política de gestión de versiones en el rendimiento del sistema, así como de las causas más importantes que definen esta variación.

- Generalizar el soporte de reducciones para permitir detectar conflictos cruzados entre las escrituras transaccionales clásicas y las de reducción. Esta mejora permitiría atajar de manera mucho más eficiente el caso de las reducciones parciales, permitiendo que el soporte de reducción se encargue de manejar determinadas direcciones de memoria hasta que se detecte un acceso (fuera de la operación de reducción) que rompa con su condición de reducción, momento a partir del cual, pasaría a ser tratada como una dirección de memoria clásica por el sistema transaccional.

Apéndice A
La interfaz OpenTM

OpenTM [5] es un interfaz de programación de aplicaciones (API) que extiende a OpenMP [64] con nuevas construcciones relacionadas con memoria transaccional. Su objetivo es integrar TM en un entorno de alto nivel para programación paralela ampliamente conocido, a fin de facilitar y hacer más práctico su uso. La interfaz OpenTM es totalmente independiente implementación particular de TM, por lo que puede ser utilizada tanto con STMs como HTMs. Dado que está diseñado como una extensión de OpenMP, OpenTM hereda todo el modelo de ejecución de este, así como las semánticas de memoria, sintaxis del lenguaje y la mayoría de construcciones *runtime*.

A.1. Modelo transaccional

Su modelo transaccional se basa en tres conceptos básicos:

- **Transacciones implícitas**: El programador es el encargado de definir las fronteras del bloque transaccional (inicio y final) sin ningún tipo de indicación acerca de los accesos a datos compartidos. El compilador será el encargado de transformar todas las operaciones de memoria realizadas dentro del ámbito de la transacción en operaciones transaccionales para garantizar su atomicidad y aislamiento. Esto permite minimizar las molestias del programador, que solo tiene que delimitar un bloque que desee ejecutar en paralelo, sin embargo, requiere de soporte de compilación para instrumentar las operaciones de memoria como transaccionales.

- **Aislamiento fuerte**: en OpenTM las transacciones son atómicas y aisladas respecto a otras transacciones y código no transaccional. Además se impone una separación consistente entre las transacciones que han finalizado y los accesos en código no transaccional del resto de la aplicación. Todos los sistemas hardware y algunos híbridos garantizan el aislamiento fuerte. En STM el compilador debe insertar barreras de lectura y escritura adicionales fuera de la transacción para implementar esta propiedad.

- **Virtualización de transacciones**: Aunque infrecuente, los programadores esperan que el sistema TM subyacente garantice una correcta ejecución incluso aunque las transacciones excedan la capacidad de recursos tales como la caché o la memoria física. Por ello se requiere que este sistema soporte transacciones virtualizadas para resolver esta limitación.

A.2. Construcciones básicas

A continuación se describen las construcciones básicas que OpenTM proporciona para la programación paralela con transacciones.

A.2.1. Bloque transaccional

La definición de un bloque transaccional se realiza en OpenTM mediante el uso de la directiva **`transaction`**, la cual delimita las instrucciones sobre las que se aplicará el control transaccional. Su sintaxis es:

```
1 #pragma omp transaction [clausula [[,] clausula]...]
2   bloque-de-instrucciones
```

Las posibles clausulas a definir podemos encontrar:

- **`ordered`**: Se utiliza para forzar un orden secuencial de *commit* entre diferentes ejecuciones de esta transacción por parte de diferentes hilos y es especialmente útil en paralelización especulativa de códigos secuenciales. Si se activa esta clausula, el sistema transaccional al detectar un conflicto entre transacciones, abortará y reiniciará las transacciones necesarias para mantener la atomicidad y el aislamiento, siguiendo el esquema de orden y la política de contención establecidos. Si no se especifica, OpenTM generará transacciones desordenadas.

- **`nesting(open|close)`**: Especifica el comportamiento de las transacciones anidadas. Por defecto, el sistema se toma el tipo **`close`**, el cual define un control de anidamiento tal que solo permite a una transacción *padre* ver las modificaciones realizadas por las transacciones *hijas* cuando realizan su *commit*. Por su parte, si el anidamiento se establece como **`open`**, el *commit* en una transacción *hija* hará visibles sus modificaciones no solo a la transacción *padre*, sino también al resto de transacciones del sistema.

A.2.2. Bucle transaccional

OpenTM permite definir un bucle como transaccional gracias a la construcción **`transfor`**, que divide el espacio de iteración del bucle en bloques y los ejecuta en paralelo como transacciones atómicas. Su sintaxis es la siguiente:

```
1 #pragma omp transfor [clausula [[,] clausula]...]
2   bucle-for
```

Las clausulas permitidas por esta construcción son:

- **`private`**: Esta clausula, heredada de OpenMP, permite definir una variable como privada al bucle transaccional. Esto creará una copia de dicha variable, sin inicializar, en cada hilo de ejecución que comparte el mismo nombre y el mismo tipo que la original.
- **`reduction`**: Marca una variable compartida como reductiva para permitir a hilos concurrentes realizar acumulaciones sobre ella sin necesidad de realizar operaciones atómicas o secciones críticas, aunque si es preciso indicar para cada variable, la operación de reducción a la que se va a someter. Dado que hereda de OpenMP, acusa las mismas limitaciones, siendo la principal que solo es aplicable a variables de reducción escalares.
- **`ordered`**: Permite forzar que las transacciones que encapsulan las iteraciones realicen el *commit*, cumpliendo con el orden secuencial de las iteraciones que componen el bucle.
- **`schedule`**: Se utiliza para indicar el esquema de particionamiento del espacio de iteraciones del bucle y su agrupamiento en transacciones. Para ello de proporcionan tres parámetros:
  - *asignación de trabajo*: El primer parámetro indica cómo se asignarán las iteraciones a los hilos trabajadores: `static`, `dynamic`, `guided` y `runtime`.

- *tamaño de bloque*: El segundo parámetro define el número de iteraciones asociadas a cada hilo en cada asignación de trabajo.
- *tamaño de transacción*: El tercer parámetro indica cuántas iteraciones del bucle se ejecutan en cada transacción de cada hilo. Si no se especifica ninguna, cada iteración será ejecutada en una única transacción.

A.2.3. Sección transaccional

La construcción **`transsection`** se utiliza para definir secciones de código que ejecutan como tareas paralelas no iterativas, encapsuladas en transacciones, con dependencias potenciales. Su sintaxis es la siguiente:

```
1 #pragma omp transsections [clausula [[,] clausula]...]
2   [#pragma omp transsection]
3     bloque-de-instrucciones-1
4   [#pragma omp transsection]
5     bloque-de-instrucciones-2
6   ...
```

Las clausulas permitidas por esta construcción son:

- **`ordered`**: Fuerza un orden secuencial de *commit* entre las tareas definidas.

A.3. Construcciones avanzadas

Las construcciones básicas de OpenTM son suficientes para expresar el paralelismo en la mayoría de aplicaciones. Sin embargo, también se introducen algunas construcciones adicionales para soportar las técnicas de programación transaccional más recientes.

A.3.1. Ejecución alternativa

OpenTM proporciona la construcción **`orelse`** que permite definir un bloque de código que se ejecutará en caso de que la transacción aborte. Esto permite al programador especificar una serie de acciones adicionales, independientes del propio proceso de recuperación de aborto definido en la política de resolución de conflicto. De manera que si la transacción realiza el *commit* satisfactoriamente, el bloque alternativo no llegará a ejecutarse nunca. La sintaxis es la siguiente:

```
1 #pragma omp transaction
2   bloque-de-instrucciones-1
3 #pragma omp orelse
4   bloque-de-instrucciones-2
5 ...
```

A.3.2. Manejadores de transacción

La construcción **handler** permite definir un manejador software que se ejecutará cuando la transacción sufra un cambio de estado. Este manejador puede definirse para cualquiera de las tres construcciones básicas y la construcción avanzada **orelse**. La sintaxis es la siguiente:

```
1 #pragma omp transaction [clausula [[,] clausula]...]
2   bloque-de-instrucciones-1
3 #pragma omp handler clausula
4   bloque-de-instrucciones-2
5 ...
```

La clausula de la construcción **handler** permite indicar que evento o transición que disparará el manejador. Las posibles opciones son: **commit**, **abort** y **violation**. La principal diferencia entre estas dos últimas es que **violation** se refiere a la interrupción de la transacción causada por una dependencia, mientras que **abort** requiere que la transacción aborte explicitamente.

A.4. Rutinas adicionales en OpenTM

OpenTM extiende el conjunto de rutinas en *runtime* de OpenMP con el fin de soportar algunas características de interés para la ejecución transaccional. Una de estas características es permitir una sincronización condicional de transacciones. Para ello se facilitan dos rutinas: **omp_retry()** y **omp_watch()**. **omp_retry()** indica a una transacción que debe bloquearse a la espera de que se cumpla cierta condición. Esta condición puede ser especificada mediante la rutina **omp_watch()**, la cual notifica al *runtime* que debe monitorizar un conjunto de direcciones de memoria a la espera de que sean actualizadas y continuar las transacciones bloqueadas cuando esto ocurra. A continuación podemos ver un ejemplo de su uso:

```
1 #pragma omp transaction{
2   if (queue.status == EMPTY) {
3     omp_watch(addr);
4     omp_retry();
5   } else
6     t = dequeue(queue);
7 }
```

OpenTM también presentan dos rutinas que permite recuperar el esquema de gestión de contención actual (**omp_watch()**) o establecer uno nuevo (**omp_set_cm()**). Los parámetros exactos de esta última dependerán fuertemente de las políticas de contención disponibles en el sistema TM. Finalmente, también se incluyen algunas rutinas que permiten al programador determinar si el flujo de ejecución se encuentra dentro de una transacción (**omp_in_transaction()**), obtener el nivel de anidamiento de la transacción actual (**omp_get_nestinglevel()**) o abortar explicitamente una transacción (**omp_abort()**).

A.5. Limitaciones

Además de aquellas limitaciones heredadas de OpenMP, existen ciertos aspectos acerca de OpenTM que deben ser considerados:

- *Sincronización*: OpenTM no permite el uso de las construcciones de sincronización clásicas de OpenMP (e.g., **critical, atomic, mutex** y **barrier**) dentro de una transacción, ya que tienen una semántica bloqueante y su uso conjunto puede derivar en *deadlocks* o violaciones de la propiedad de aislamiento.

- *E/S y llamadas al sistema*: OpenTM requiere que todas las librerías llamadas desde el interior de una transacción sean *transaction-safe* para minimizar el comportamiento inesperado como consecuencia de efectos colaterales tales como instrucciones de *E/S* y llamadas al sistema. Así cada implementación de OpenTM deberá ser responsable de especificar qué librerías pueden ser invocadas de manera segura junto con transacciones.

- *Detección de conflictos relajada*: Aunque algunas de las nuevas propuestas de TM hacen hincapié en los beneficios de las políticas de detección de conflictos relajadas y proponen algunas directivas a implementar en las APIs de programación transaccional que permiten excluir ciertas variables

de la detección de conflictos (e.g., **`race`** [98] o **`exclude`** [66]), OpenTM no proporciona ninguna directiva para tal efecto.

- *Compilación*: En STM todas las instrucciones llamadas dentro de una transacción deben estar correctamente instrumentadas para que puedan hacer uso del soporte transaccional. Aunque esto es labor del compilador, existen ciertas situaciones en las que no es posible sin ayuda del programador. Es el caso de los punteros a función, los cuales podrían ser difíciles de resolver estáticamente por el compilador. Para evitar estas posibles situaciones, OpenTM requiere que los programadores usen la directiva **`tm_function`** para identificar todas aquellas funciones que pueden ser invocadas dentro de las transacciones, para que el compilar pueda clonar dichas funciones introduciendo la instrumentación adecuada.

Bibliografía

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. (Citado en página 67)

[2] C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson, and Sean Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316–327. IEEE, 2005. (Citado en páginas 8 y 28)

[3] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Int'l. Conf. on Parallel Processing (EuroPar'08)*, pages 719–728, 2008. (Citado en páginas 69 y 106)

[4] Utku Aydonat and Tarek S Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 15–26. IEEE, 2010. (Citado en páginas 9 y 37)

[5] W. Baek, C.C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 376–387. IEEE, 2007. (Citado en página 149)

[6] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. Van Der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*, volume 11. Society for Industrial Mathematics, 1987. (Citado en página 90)

[7] G. Bandera and E.L. Zapata. Data locality exploitation in algorithms including sparse communications. In *Parallel and Distributed Processing Sym-*

posium., Proceedings 15th International, pages 6–pp. IEEE, 2001. (Citado en página 91)

[8] Utpal Banerjee. *Loop transformations for restructuring compilers: the foundations*, volume 1. Springer, 1993. (Citado en página 73)

[9] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware 2012*, pages 187–207. Springer, 2012. (Citado en páginas 9 y 39)

[10] George Keith Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 2000. (Citado en página 60)

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 72–81, 2008. (Citado en páginas 105 y 138)

[12] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (Citado en página 28)

[13] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 81–91. ACM, 2007. (Citado en página 8)

[14] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement*, pages 125–137, London, UK, UK, 1997. Chapman & Hall, Ltd. (Citado en página 113)

[15] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000. (Citado en página 115)

[16] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, September 2008. (Citado en páginas 49 y 105)

[17] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *Micro, IEEE*, 29(2):6–16, 2009. (Citado en página 8)

[18] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science-Research and Development*, 26(3-4):205–210, 2011. (Citado en página 118)

[19] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, et al. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010. (Citado en páginas 8 y 29)

[20] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*, volume 41, pages 336–346. ACM, 2006. (Citado en página 8)

[21] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. 2009. (Citado en página 8)

[22] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006. (Citado en páginas 8 y 33)

[23] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *ACM Sigplan Notices*, volume 44, pages 155–165. ACM, 2009. (Citado en página 39)

[24] P. Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, pages 429–441. ACM, 1988. (Citado en página 125)

[25] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 21(12):1793–1807, 2010. (Citado en páginas 33 y 43)

[26] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 237–246, 2008. (Citado en páginas 33 y 43)

[27] Cesare Ferri, Andrea Marongiu, Benjamin Lipton, R Bahar, Tali Moreshet, Luca Benini, and Maurice Herlihy. SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, pages 39–48. ACM, 2011. (Citado en páginas 9, 38 y 104)

[28] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. (Citado en página 4)

[29] Ian Foster, Rob Schreiber, and Paul Havlak. HPF-2, scope of activities and motivating applications. Technical report, Technical Report CRPC-TR94492, Rice University, 1994. (Citado en páginas 63 y 138)

[30] M. Angel Gonzalez-Mesa, Eladio Gutierrez, and Oscar Plata. Leveraging transactional memory to extract parallelism. In *Euro-TM Workshop on Transactional Memory (WTM 2012) (co-located with EuroSys 2012)*, Bern, Switzerland, April 2012. (Citado en página 10)

[31] M. Angel Gonzalez-Mesa, Eladio Gutierrez, and Oscar Plata. Parallelizing irregular reductions using transactions. In *17th Workshop on Compilers for Parallel Computers (CPC'13)*, Lyon, France, July 2013. (Citado en página 10)

[32] M. Angel Gonzalez-Mesa, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Effective transactional memory execution management for improved concurrency. In *ACM Transactions on Architectural and Code Optimization (TACO)*, In Press. (Citado en página 10)

[33] M. Angel Gonzalez-Mesa, Eladio D. Gutierrez, and Oscar Plata. Parallelizing the sparse matrix transposition: Reducing the programmer effort using transactional memory. *Procedia Computer Science*, 18(0):501 – 510, 2013. 2013 International Conference on Computational Science. (Citado en página 10)

[34] M. Angel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Automatic loop parallelization using transactional memory support. In *16th Workshop on Compilers for Parallel Computers (CPC'12)*, Padova, Italy, january 2012. (Citado en páginas 10 y 104)

[35] M. Angel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Dealing with reduction operations using transactional memory support. In *25th Int'l. Symp. on Computer Architecture and High-Performance Computing (SBAC-PAD'13)*, Porto de Galinhas, Brazil, October 2013. (Citado en página 10)

[36] M. Angel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Exploring irregular reduction support in transactional memory. In *Algorithms and Architectures for Parallel Processing*, pages 257–266. Springer, 2013. (Citado en página 10)

[37] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):164–189, april 1983. (Citado en página 6)

[38] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *Distributed Computing*, pages 303–323. Springer, 2005. (Citado en páginas 8 y 26)

[39] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2005. (Citado en página 26)

[40] E. Gutiérrez, O. Plata, and E.L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Proceedings of the 14th international conference on Supercomputing*, pages 78–87. ACM, 2000. (Citado en página 127)

[41] E. Gutiérrez, O. Plata, and E.L. Zapata. An analytical model of locality-based parallel irregular reductions. *Parallel Computing*, 34(3):133–157, 2008. (Citado en página 123)

[42] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996. (Citado en página 124)

[43] 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 *ACM SIGARCH Computer Architecture News*, volume 32, page 102. IEEE Computer Society, 2004. (Citado en páginas 8, 9 y 34)

[44] H. Han and C.W. Tseng. Exploiting locality for irregular scientific codes. *Parallel and Distributed Systems, IEEE Transactions on*, 17(7):606–618, 2006. (Citado en página 127)

[45] T. Harris, J.R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers, 2010. (Citado en página 7)

[46] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003. (Citado en página 8)

[47] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, pages 289–300, 1993. (Citado en páginas 8 y 27)

[48] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, january 1991. (Citado en páginas 6 y 7)

[49] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003. (Citado en páginas 8 y 54)

[50] Gray Jim and Reuter Andreus. Transaction processing: concepts and techniques. *Transactions Processing: Concepts and Techniques*, pages 724–732, 1993. (Citado en página 16)

[51] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *Knowledge and Data Engineering, IEEE Transactions on*, 17(1):71–89, 2005. (Citado en página 127)

[52] Seonggun Kim, Hwansoo Han, and Kwang-Moo Choe. Region-based parallelization of irregular reductions on explicitly managed memory hierarchies. *The Journal of Supercomputing*, 56(1):25–55, 2011. (Citado en página 128)

[53] Sanjeev Kumar, Michael Chu, Christopher J Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220. ACM, 2006. (Citado en página 8)

[54] J.R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007. (Citado en páginas 7 y 16)

[55] Chin Yang Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, (3):346–365, 1961. (Citado en página 53)

[56] Shun-Tak Leung and John Zahorjan. *Improving the performance of runtime parallelization*, volume 28. ACM, 1993. (Citado en página 74)

[57] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. (Citado en página 33)

[58] Yossi Lev and Mark Moir. Fast read sharing mechanism for software transactional memory. In *23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004) poster presentation*, 2004. (Citado en página 33)

[59] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137. ACM, 1977. (Citado en páginas 7 y 30)

[60] Virendra J Marathe, Michael F Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N Scherer III, and Michael L Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006. (Citado en página 8)

[61] Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 227–236, New York, NY, USA, 2008. ACM. (Citado en página 19)

[62] Austen McDonald, JaeWoong Chung, Brian D Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *33rd Int'l. Symp. on Computer Architecture (ISCA'06)*, pages 53–65, 2006. (Citado en página 132)

[63] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*, pages 166–176, 2009. (Citado en página 35)

[64] Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000. (Citado en páginas 49 y 149)

[65] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers, USA, 2012. (Citado en página 5)

[66] Miloš Milovanović, Roger Ferrer, Osman S Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Transactional memory and openmp. In *A Practical Programming Model for the Multi-Core Era*, pages 37–53. Springer, 2008. (Citado en página 155)

[67] Chi Cao Minh. *Designing an effective hybrid transactional memory system*. ProQuest, 2008. (Citado en página 49)

[68] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. (Citado en página 1)

[69] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. Logtm: log-based transactional memory. In *HPCA*, volume 6, pages 254–265, 2006. (Citado en páginas 8, 22 y 28)

[70] J.J. Morales and S. Toxvaerd. The Cell-Neighbour table method in molecular dynamics simulations. *Computer Physics Communications*, 71:71–76, 1992. (Citado en páginas 65 y 138)

[71] N Mukherjee and JR Gurd. A comparative analysis of four parallelisation schemes. In *13th Int'l. Conf. on Supercomputing (ICS'99)*, pages 278–285, 1999. (Citado en páginas 62 y 138)

[72] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003. (Citado en página 60)

[73] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1, 2011. (Citado en página 49)

[74] Peter Pacheco. *An introduction to parallel programming*. Access Online via Elsevier, 2011. (Citado en página 2)

[75] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press, 1984. (Citado en página 90)

[76] Leo Porter, Bumyong Choi, and Dean M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *18th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 313–324, 2009. (Citado en página 15)

[77] Louis-Noel Pouchet. PolyBench: The polyhedral benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2011. (Citado en páginas 60 y 105)

[78] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. BulkSMT: Designing SMT processors for atomic-block execution. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012. (Citado en páginas 9, 38 y 78)

[79] Shah M Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, page 30. ACM, 2011. (Citado en página 59)

[80] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001. (Citado en página 8)

[81] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 494–505. IEEE, 2005. (Citado en páginas 8 y 28)

[82] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *41st Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'08)*, pages 246–257, 2008. (Citado en páginas 9, 24 y 35)

[83] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer, 2010. (Citado en página 2)

[84] Lawrence Rauchwerger and David A Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans, on Parallel and Distributed Systems*, 10(2):160–180, 1999. (Citado en páginas 122 y 125)

[85] James Reinders. Transactional synchronization in haswell (february 2012). *URL http://software. intel. com/en-us/blogs/2012/02/07/-transactional-synchronization-in-haswell*. (Citado en páginas 8 y 29)

[86] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Distributed Computing*, pages 284–298. Springer, 2006. (Citado en página 46)

[87] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pages 221–228, 2007. (Citado en página 95)

[88] David P Rodgers. Improvements in multiprocessor system design. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 225–231. IEEE Computer Society Press, 1985. (Citado en página 67)

[89] Bratin Saha, A-R Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 185–196. IEEE, 2006. (Citado en página 8)

[90] William N Scherer III and Michael L Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004. (Citado en páginas 23 y 26)

[91] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005. (Citado en páginas 23 y 26)

[92] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini. Towards transactional memory support for gcc. In *GCC Research Opportunities Workshop*, 2009. (Citado en página 128)

[93] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213. ACM, 1995. (Citado en páginas 8 y 31)

[94] F. Smailbegovic, G.N. Gaydadjiev, and S. Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pages 445–448, 2005. (Citado en página 90)

[95] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *22nd Annual ACM/IEEE International Symposium on Computer Architecture (ISCA'95)*, pages 414–425, 1995. (Citado en página 15)

[96] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. *ACM SIGPLAN Notices*, 28(1):64–67, 1993. (Citado en página 74)

[97] Peng Tu and David Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computing*, pages 500–521. Springer, 1994. (Citado en página 74)

[98] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89. ACM, 2007. (Citado en página 155)

[99] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. (Citado en páginas 59 y 115)

[100] Luke Yen, Jayaram Bobba, Michael R 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 *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272. IEEE, 2007. (Citado en página 29)

[101] H. Yu and L. Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 17(10):1084–1096, 2006. (Citado en página 127)