Tesis doctoral

Soporte de Reducciones y Orden en Sistemas de Memoria Transaccional

Manuel Pedrero Luque

Dirigida por Eladio D. Gutiérrez Carrasco y Óscar Plata González

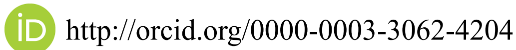
UNIVERSIDAD DE MÁLAGA

PROGRAMA DE DOCTORADO EN INGENIERÍA MECATRÓNICA
ESCUELA DE INGENIERÍAS INDUSTRIALES, NOVIEMBRE 2018

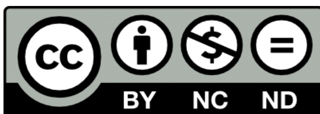
UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

AUTOR: Manuel Pedrero Luque


http://orcid.org/0000-0003-3062-4204

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:
http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode
Cualquier parte de esta obra se puede reproducir sin autorización
pero con el reconocimiento y atribución de los autores.
No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Soporte de Reducciones y Orden en Sistemas de Memoria Transaccional

Manuel Pedrero Luque

Noviembre de 2018

Dirigida por
Dr. Eladio Gutiérrez Carrasco
Dr. Óscar Plata González

UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

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

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

CERTIFICAN:

Que la memoria titulada "Soporte de Reducciones y Orden en Sistemas de Memoria Transaccional" ha sido realizada por D. Manuel Pedrero Luque 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, noviembre de 2018

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

Dr. D. Óscar Plata González.
Tutor y codirector de la tesis.

UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

A mi familia

UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Agradecimientos

Durante los últimos años he tenido el placer de convivir con multitud de personas que, directa o indirectamente, han sido partícipes de este trabajo. Sirvan estas líneas como reconocimiento a todas ellas.

En primer lugar quiero dar las gracias a mis directores Óscar Plata y Eladio Gutiérrez por su acogida en el grupo de investigación y por su confianza y apoyo durante estos años. Si esta tesis ha llegado a buen puerto es sin duda gracias a ellos, a su disponibilidad, su dedicación, su consejo y su ayuda. Quiero agradecer igualmente a Ezequiel Herruzo, de la Universidad de Córdoba, su recomendación para entrar a formar parte de este equipo y a Sergio Romero su ayuda y apoyo en varias de las contribuciones de esta tesis.

Reitero mi agradecimiento a Óscar por darme además la oportunidad de impartir docencia durante este periodo, así como a los profesores Julián, Francisco, Gerardo, Luis Felipe y Julio, con los que he tenido el privilegio de trabajar, sin olvidar al resto de profesores del departamento, siempre disponibles para echarme una mano. Gracias por vuestra confianza y vuestro consejo.

Del mismo modo, me gustaría extender mi agradecimiento al resto del departamento de Arquitectura de Computadores de la Universidad de Málaga; muy especialmente a los técnicos, Paco, Juanjo y Mª Carmen y a la secretaria, Carmen. Gracias por vuestra disponibilidad y vuestra ayuda.

Finalmente no puedo dejar de agradecer a mis compañeros de laboratorio y de fatigas su acogida, su apoyo y el gran ambiente de trabajo que me he encontrado desde el primer día: Mángel, Ricardo, Cervilla, Alberto, Pirlo, Sergio, Carlos, Villegas, Vilches, Upton, Arjona, Fran, Lázaro, Serrucho, Denisa, Andrés, Bernabé, José Carlos, Iván y aquellos a los que haya podido olvidar. Gracias a todos.

UNIVERSIDAD DE MÁLAGA

Debo mencionar además las fuentes de financiación que me han permitido llevar a cabo las investigaciones que han dado lugar a esta tesis: los proyectos TIN2013-42253-P y TIN2016-80920-R del Gobierno de España, el proyecto P12-TIC-1470 de la Junta de Andalucía y el plan propio de la Universidad de Málaga.

UNIVERSIDAD DE MÁLAGA

Resumen

La popularización de los chips multiprocesador ha supuesto un cambio profundo en la comunidad de desarrollo software. La programación secuencial tradicional no permite aprovechar al máximo los recursos hardware disponibles en los procesadores actuales, mientras que la programación paralela eficiente es una tarea compleja que requiere de conocimientos detallados sobre la arquitectura del procesador, así como de nuevos algoritmos y herramientas.

El paradigma de Memoria Transaccional (TM) ha surgido como una alternativa a los modelos basados en *locks* para la programación paralela convencional basada en exclusión mutua, con la promesa de facilitar el desarrollo de programas multihilo siguiendo un modelo de concurrencia optimista. Los sistemas TM se basan en el concepto de *transacción*; un bloque de instrucciones que se ejecuta en paralelo manteniendo la consistencia con el resto de código y propagando los cambios de forma atómica al resto de hilos concurrentes. El sistema TM es responsable de detectar y resolver posibles conflictos entre las transacciones para mantener estas propiedades de manera transparente al programador. Dado que la resolución de un conflicto implica normalmente el aborto de alguna de las transacciones implicadas, el volumen de estos conflictos es un aspecto crítico que condiciona en buena medida el rendimiento final de la aplicación. Los sistemas TM pueden implementarse mediante librerías software (STM) o ser soportarlos directamente en el hardware del procesador (HTM).

En esta tesis se proponen soluciones basadas en memoria transaccional para optimizar la detección y resolución de conflictos en códigos que presentan operaciones de reducción. Las reducciones aparecen frecuentemente en aplicaciones científicas en combinación con patrones de acceso irregulares que dificultan en gran medida su paralelización de forma eficiente. Nuestra propuesta contempla las reducciones como una operación transaccional adicional en el modelo TM. A diferencia de un sistema TM tradicional, el tratamiento explícito de las reduccio-

UNIVERSIDAD
DE MÁLAGA

nes permite filtrar los conflictos originados por dichas operaciones manteniendo la consistencia del sistema sin necesidad de abortar transacciones.

Este trabajo introduce además un orden total de precedencia entre transacciones que implica que éstas confirmen sus cambios a memoria en un orden específico, facilitando así la paralelización de aplicaciones secuenciales. El uso de este orden permite además detectar *reducciones parciales*; accesos a memoria que violan las condiciones de reducción. El sistema TM utiliza la información de precedencia para resolver otros conflictos entre transacciones sin producir abortos. Esta propuesta se evalúa comparándola con otros métodos utilizados tradicionalmente para paralelizar códigos con operaciones de reducción, incluyendo el uso de *locks* de grano grueso y de grano fino, privatización y el uso de un TM clásico con restricciones de orden.

Dado que un orden de precedencia estricto penaliza el rendimiento y puede ser demasiado restrictivo en muchos algoritmos, esta tesis aborda también el diseño de un orden de precedencia parcial, donde las restricciones a la hora de confirmar los cambios a memoria no se aplican entre transacciones individuales sino entre grupos de ellas. Este modelo permite aumentar el grado de paralelismo de la aplicación, eliminando restricciones a la hora de confirmar los cambios entre transacciones de un mismo grupo y mejorando así el rendimiento. Con el objetivo de proporcionar una interfaz familiar al programador, introducimos el concepto de *barrera transaccional*, una primitiva cuya semántica es similar a una barrera de sincronización tradicional, pero que aprovecha TM para permitir a los hilos seguir ejecutando código de forma especulativa. Nuestra propuesta utiliza estas barreras para establecer un orden parcial de precedencia, lanzando transacciones con restricciones de precedencia dinámicas que permiten evitar bloqueos debidos a la sincronización y mantener a la vez la corrección del programa. En este caso, los escenarios de interés son códigos que utilizan barreras para sincronizar diferentes hilos de ejecución.

Los experimentos de esta tesis se han llevado a cabo sobre sistemas reales. Los modelos STM con soporte de reducciones se han implementado como una librería accesible por las aplicaciones paralelas mediante una API similar a la utilizada en otros STM. Por su parte, los modelos HTM se han implementado sobre las extensiones transaccionales del procesador POWER8, aprovechando la posibilidad que brinda de ejecutar instrucciones no transaccionales en el contexto de una transacción. Para ello hemos instrumentado las llamadas a las instrucciones de inicio y fin de transacción aprovechando el soporte transaccional existente.

UNIVERSIDAD DE MÁLAGA

La evaluación se ha llevado a cabo utilizando una selección de *benchmarks* reales y sintéticos que incluye la suite STAMP, específicamente diseñada para evaluar sistemas TM; Eigenbench, que permite analizar ortogonalmente características de los sistemas TM; códigos seleccionados de las suites SPEC, PARSEC, y Polybench; y otros kernels de aplicaciones HPC de interés.

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

Índice general

Agradecimientos I

Resumen III

Índice general VII

Índice de figuras XIII

Índice de tablas XVII

1. Introducción **1**

1.1. La complejidad de la programación paralela 3

1.2. Memoria transaccional 5

1.2.1. Ventajas de la memoria transaccional 6

1.2.2. Técnicas de especulación con soporte transaccional 7

1.2.3. Operaciones de reducción y memoria transaccional 8

1.3. Motivación y contribuciones 8

1.4. Estructura de la tesis 11

2. Antecedentes **13**

2.1. Transacciones 13

2.2. La interfaz transaccional 15

UNIVERSIDAD DE MÁLAGA

2.3. El espacio de diseño en memoria transaccional 16
2.3.1. Control de concurrencia 16
2.3.2. Gestión de versiones 17
2.3.3. Detección de conflictos 18
2.4. Semántica transaccional 20
2.4.1. Atomicidad 20
2.4.2. Criterios de consistencia 22
Historias y ejecuciones 22
Criterios de consistencia 26
2.4.3. Transacciones anidadas 30
2.4.4. Restricciones de orden 31
2.5. Sistemas de memoria transaccional 32
2.5.1. Sistemas TM software 33
Transactional Mutex Locks 34
Transactional Locking II 35
TinySTM 36
NOrec 37
InvalSTM 39
2.5.2. Sistemas TM hardware 40
Herlihy HTM 41
Virtualized Transactional Memory 42
LogTM 43
Transactional Coherence and Consistency 43
Intel Transactional Syncronization Extensions 44
IBM Blue Gene/Q 45
IBM System Z 47
IBM POWER8 48
2.5.3. Sistemas TM híbridos 51

UNIVERSIDAD DE MÁLAGA

HyTM . . . 51
Phased Transactional Memory . . . 52
Hybrid NOrec . . . 53
Invyswell . . . 54

3. Soporte de reducciones y orden en memoria transaccional software 57
3.1. Aplicaciones irregulares . . . 57
3.2. Patrones de reducción . . . 59
Reducciones parciales . . . 60
3.2.1. Técnicas clásicas para la paralelización de reducciones . . 63
Técnicas basadas en exclusión mutua . . . 63
Técnicas basadas en privatización . . . 64
Técnicas basadas en el particionado del espacio de reducción 65
3.2.2. Paralelización de reducciones con soporte TM . . . 65
3.3. ReduxSTM . . . 68
3.3.1. Características . . . 68
3.3.2. Diseño . . . 74
ReduxSTM-TS . . . 77
ReduxSTM-CTI . . . 82
3.3.3. Implementación . . . 88
Mecanismos de sincronización . . . 88
Barreras de memoria . . . 89
Estructuras de datos para búferes de escritura y reducción 92
Estructuras de datos para timestamps (ReduxSTM-TS) . . 95
Estructuras para los conjuntos de datos (ReduxSTM-CTI) . 96
Criterios de corrección: Opacidad y VWC . . . 97
Optimizaciones adicionales . . . 104
3.4. Evaluación . . . 105

UNIVERSIDAD DE MÁLAGA

3.4.1. Análisis de sensibilidad . . . 108
RXasRW (Reductions as Read+Write) . . . 108
Eigenbench . . . 115
Wormbench . . . 117
3.4.2. Reducciones irregulares . . . 120
FluidAnimate . . . 121
MD2 . . . 124
Euler . . . 124
Legendre . . . 125
Requerimientos de memoria . . . 126
3.4.3. Reducciones Parciales . . . 129
TWolf . . . 129
3.4.4. Códigos generales . . . 130
STAMP . . . 131
3.4.5. Soporte para técnicas de especulación . . . 133
SPEC2006 . . . 135
3.5. Trabajos relacionados . . . 138
3.6. Conclusiones . . . 141

4. Soporte de orden parcial en memoria transaccional hardware 143
4.1. Introducción . . . 143
4.2. Limitaciones de los HTM comerciales . . . 146
Comunicación entre transacciones y suspended mode . . . 147
4.2.1. Garantías de progreso en HTM *best-effort* . . . 148
4.3. Barreras Transaccionales . . . 150
Reemplazando barreras con transacciones ordenadas . . . 152
TMbarrier . . . 152
4.3.1. Restricciones de orden . . . 154

UNIVERSIDAD DE MÁLAGA

4.3.2. Transacciones anidadas 155
4.3.3. Funcionamiento de TMbarrier 155
4.4. Diseño de TMbarrier 156
4.4.1. Nuevos estados en los hilos de ejecución 157
4.4.2. La primitiva `tmbarrier` 159
4.4.3. Iniciando transacciones 159
4.4.4. Finalizando transacciones 160
4.4.5. Gestionando las limitaciones del HTM de POWER8 162
Granularidad de la detección de conflictos 162
Interacciones en suspended mode 163
Limitaciones del anidamiento de transacciones 164
Política de resolución de conflictos 165
4.5. Evaluación 165
4.5.1. Microbenchmark sintético 167
4.5.2. Livermore Loop 6: Recurrencia Lineal 170
4.5.3. Algoritmo CFL 174
4.6. Trabajos relacionados 179
4.7. Conclusiones 180

5. Conclusiones y trabajos futuros 183
5.1. Conclusiones 184
5.2. Trabajos futuros 187

A. API transaccional y simulación de trazas 189
A.1. Una interfaz transaccional unificada 189
A.2. Simulación de trazas 191

Bibliografía 195

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Índice de figuras

1.1. Evolución de los procesadores desde 1970 2

2.1. Interfaz transaccional mínima 15
2.2. Control de concurrencia 16
2.3. Diferencias entre *locks* y transacciones 21
2.4. Diferencias entre aislamiento fuerte y débil 22
2.5. Ejemplo de historias y ejecuciones 25
2.6. Límites de *serializability* 27
2.7. Criterios de consistencia en un sistema transaccional 29
2.8. Ejemplo del uso de orden en transacciones 31
2.9. Mejoras de TinySTM respecto a TL-2 38

3.1. Ejemplos de bucles de reducción 60
3.2. *Kernels* reales con bucles de reducción 61
3.3. Ejemplos de reducciones parciales 62
3.4. Paralelización de bucles de reducción utilizando TM 67
3.5. Bucles de reducción irregulares 69
3.6. Funcionamiento de ReduxSTM-TS 78
3.7. Funcionamiento de ReduxSTM-CTI 84
3.8. Sincronización en ReduxSTM: *commits* ordenados 88

UNIVERSIDAD DE MÁLAGA

3.9. Sincronización en ReduxSTM-CTI 90
3.10. Sincronización en ReduxSTM: barreras de memoria 91
3.11. Sincronización en ReduxSTM-TS 93
3.12. ReduxSTM: estructuras de datos 95
3.13. ReduxSTM-CTI: filtros de Bloom 96
3.14. Opacity y VWC en ReduxSTM 99
3.15. Extracto del *benchmark RXasRW* 109
3.16. RXasRW: rendimiento y TCR 110
3.17. RXasRW: análisis de sensibilidad I 112
3.18. RXasRW: análisis de sensibilidad II 114
3.19. Extracto del *kernel* de Eigenbench 115
3.20. Eigenbench: análisis de rendimiento 117
3.21. Wormbench: sensibilidad al tamaño del mundo 118
3.22. Wormbench: *speedup* relativo 119
3.23. FluidAnimate: rendimiento y sensibilidad al tamaño de transacción 122
3.24. MD2: rendimiento y sensibilidad al tamaño de transacción 123
3.25. Euler: rendimiento y sensibilidad al tamaño de transacción 125
3.26. Legendre: rendimiento 126
3.27. Uso de memoria de las diferentes técnicas de paralelización . . . 127
3.28. 300.twolf: rendimiento y TCR 131
3.29. STAMP: comparativa de rendimiento 134
3.30. ReduxSTM-TS: influencia del criterio de consistencia 135
3.31. SPEC CPU2006: rendimiento y TCR 137

4.1. Dependencias en Livermore Loop 6 151
4.2. Reemplazando barreras mediante TM con restricciones de orden 153
4.3. Funcionamiento de TMbarrier 156
4.4. TMbarrier: grafo de estados 157

UNIVERSIDAD DE MÁLAGA

4.5. Kernel del microbenchmark sintético 167
4.6. Microbenchmark: eficiencia y especulación 168
4.7. Recurrence: *speedup* y TCR 171
4.8. Speedup algoritmo CFL 176
4.9. TCR algoritmo CFL 177

A.1. Ejemplo del uso de la API de STAMP 192
A.2. Ejemplo de traza de ejecución 193

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

Índice de tablas

2.1. Diferencias entre criterios de consistencia TM 29

2.2. Resumen de propuestas STM analizadas 34

2.3. Resumen de propuestas HTM analizadas 50

3.1. Técnicas de paralelización de reducciones 66

3.2. Conflictos entre transacciones en diseños STM 73

3.3. Notación utilizada en la descripción de los algoritmos de ReduxSTM 76

3.4. Plataforma de evaluación de ReduxSTM 105

3.5. Resumen de los *benchmarks* utilizados en la evaluación 107

3.6. Códigos con reducciones irregulares 120

3.7. STAMP: características de las aplicaciones de la suite 132

3.8. Selección de bucles de la suite SPEC CPU2006 136

4.1. Notación utilizada en la descripción de los algoritmos de TMbarrier 149

4.2. Operaciones sobre metadatos en TMbarrier 162

4.3. Plataforma de evaluación de TMbarrier 166

A.1. Primitivas básicas de la API transaccional de STAMP 190

A.2. Primitivas añadidas a la API transaccional de STAMP 191

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Capítulo 1

Introducción

Durante décadas, el rendimiento de los procesadores ha aumentado de forma sostenida. Dos observaciones empíricas han descrito de este fenómeno. Moore predijo en 1965 un crecimiento exponencial del número de transistores que podía alojar un chip manteniendo el coste [75]. En 1974, Dennard observó que la densidad de potencia de los chips se mantenía constante al reducir el tamaño de los transistores, debido a que también se reducían sus requerimientos de voltaje y corriente [25]. Sin embargo cabe preguntarse, ¿cómo se traducen estas observaciones en el aumento de rendimiento que han experimentado los procesadores durante este tiempo? ¿Por qué se suele relacionar este aumento con la llamada *ley de Moore*?

Los avances de la tecnología de integración han conseguido disminuir progresivamente el tamaño de los transistores, permitiendo alojar un número cada vez mayor en un mismo chip, y correlándose así con las predicciones de Moore. Por su parte un transistor más pequeño puede cambiar de estado más rápidamente, permitiendo aumentar su frecuencia de funcionamiento. De acuerdo con Dennard, debería consumir también menos energía, permitiendo que el procesador resultante pueda emplear un mayor número de transistores, a mayor frecuencia, todo ello manteniendo el consumo. Esto, unido a la capacidad de los arquitectos de computadores para aprovechar los nuevos transistores disponibles provocó que se relacionase naturalmente la ley de Moore con el rendimiento de los procesadores.

Desde mediados de la pasada década, sin embargo, la frecuencia de funcionamiento de los procesadores se estancó alrededor de los 4GHz, límite que, como

UNIVERSIDAD DE MÁLAGA

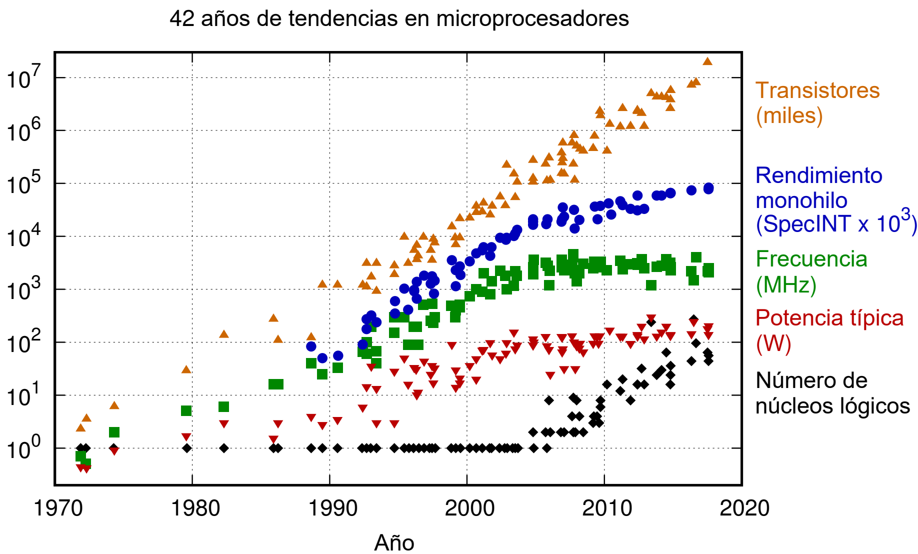


Figura 1.1: Evolución de los procesadores desde 1970. Gráfica adaptada de original de K. Rupp. Los datos hasta 2010 han sido extraídos por M. Horowitz, F. Labonte, O. Shacham, K. Olukotun y C. Batten. Los datos hasta 2017 han sido recopilados por K. Rupp [97].

se observa en la figura 1.1, permanece vigente en la actualidad. La razón no es tanto que la frecuencia no pueda seguir aumentando, sino la energía necesaria para ello. Las observaciones de Dennard dejaron de cumplirse hace años, ya que las corrientes de fuga y el voltaje mínimo de los transistores marcan un límite inferior a su consumo que no escala con el tamaño de los mismos. Conforme los transistores se hacen más pequeños, la densidad energética del procesador aumenta. A mediados de la década de los 2000, este aumento no se pudo seguir ignorando y se llegó al denominado *power wall* que, en la práctica y unido a otros problemas, limitó la frecuencia a la que podían funcionar los procesadores manteniendo un consumo asumible. En los procesadores actuales, de hecho, un porcentaje cada vez mayor del total de transistores debe permanecer inactivo para mantener el consumo del chip controlado. Esta limitación se conoce como *dark silicon* y es un problema cada vez mayor, ya que impide aprovechar al máximo los transistores disponibles en el procesador para mejorar su rendimiento. Además, el porcentaje de *dark silicon* aumenta conforme lo hace la densidad energética del procesador, lo que dificulta justificar la creación de nuevos nodos de fabricación desde un punto de vista comercial.

UNIVERSIDAD DE MÁLAGA

Consecuentemente al *power wall* se produjo un cambio en la industria de los procesadores de consumo, que pasó de aumentar la complejidad de los núcleos e incrementar su frecuencia de funcionamiento a disminuir dicha frecuencia e integrar varios núcleos independientes en un mismo chip. Este cambio obedece a un doble propósito: permite reducir la energía disipada por el procesador y, al mismo tiempo, maximiza el aprovechamiento de los transistores disponibles. La tendencia se ha mantenido hasta el día de hoy, donde la inmensa mayoría de procesadores de propósito general están compuestos de varios núcleos independientes encapsulados en un mismo chip que acceden a una memoria compartida. Esta configuración conforma una arquitectura MIMD[1] conocida como *chip multiprocessor* o CMP y es en la que nos centramos en esta tesis. Los CMP necesitan además de una red de interconexión que comunique los distintos núcleos de procesamiento con la memoria global, compartida entre todos ellos. Desde el punto de vista de la programación, esta arquitectura presenta la ventaja de poder intercambiar información entre distintos núcleos mediante el uso de un espacio de memoria compartido. Sin embargo los accesos a datos compartidos necesitan de una coordinación para mantener la consistencia del programa.

1.1. La complejidad de la programación paralela

A diferencia de las mejoras arquitecturales desarrolladas desde los años 70, la disponibilidad de varios núcleos de procesamiento no se traduce automáticamente en un mayor rendimiento. Mientras que avances como la segmentación, la ejecución fuera de orden o la memoria caché son transparentes al programador, el uso de procesadores multinúcleo supone el paso de la responsabilidad de explotar los recursos hardware de la industria al desarrollador. A su vez, interfiere en conceptos tradicionales para manejar la complejidad en el desarrollo de software tales como la abstracción o la composición.

Para ilustrar lo anterior pensemos en una clase `account` que permita controlar los depósitos en un banco. Dicha clase dispone de dos métodos para retirar o depositar cierta cantidad de dinero. En un modelo secuencial un programador puede implementar un nuevo método `transfer(account src, account tgt)` que realice una transferencia a partir de dos cuentas usando los métodos para retirar y depositar en los objetos correspondientes. No es necesario conocer los detalles de implementación de los métodos, por lo que la solución resulta trivial. En un entorno paralelo, sin embargo, el mismo problema requiere que el programador conozca los detalles de implementación de los métodos, si son

[1] *Multiple Instructions, Multiple Data*, según la taxonomía de Flynn [39].

UNIVERSIDAD DE MÁLAGA

atómicos, si usan alguna directiva de bloqueo y las interacciones que pueda tener dicha directiva con el resto de la aplicación. Aun si los métodos son atómicos, esto no garantiza que se puedan combinar de forma atómica. La solución en este caso dista de ser sencilla, pudiéndose producir condiciones de carrera o bloqueos debidos a *deadlocks* si no se consideran las posibles interacciones entre hilos. Finalmente, una implementación válida desde el punto de vista de la corrección de los resultados no tiene por qué ser más eficiente que la solución secuencial.

Estos problemas se deben a que el uso del paralelismo en la programación lleva implícito una falta de determinismo en la ejecución de los diferentes hilos que supone una dificultad añadida respecto a la programación secuencial. La ejecución de un hilo puede verse retrasada por interrupciones, fallos de caché o por el propio planificador del sistema operativo. La duración de estos retrasos puede variar en un rango de varios órdenes de magnitud. El programador debe considerar por tanto la totalidad de eventos que puedan ocurrir de forma concurrente y cómo afectan dichos eventos a la corrección del problema a resolver.

Además de una mayor dificultad a la hora de programar, los posibles errores suelen ser más complicados de detectar y de corregir. Las herramientas de análisis, depuración, modelos de programación y entornos disponibles para la programación paralela no están tan desarrollados como los disponibles para la programación secuencial.

Para explotar el paralelismo se suelen considerar dos modelos. El *paralelismo de datos* efectúa una serie de operaciones sobre un conjunto de datos de forma simultánea. Suele encontrarse en aplicaciones numéricas que operan sobre vectores o matrices, y puede ser explotado implícitamente en algunos lenguajes o incluso directamente por el compilador, pero no está presente un gran porcentaje de problemas. Por su parte el *paralelismo de tareas* hace referencia a un modelo más general donde la computación se reparte entre varios hilos que se sincronizan entre sí explícitamente por medio de primitivas como *locks*, *semáforos* o *barreras*. Este modelo no establece restricciones al código que ejecuta cada hilo ni a cómo se comunican los hilos entre sí, por lo que es suficientemente general para englobar a toda forma de paralelismo. Es sin embargo más complicado de programar correctamente, ya que su grado de abstracción puede bajar directamente a nivel ISA (*Instruction Set Architecture*).

UNIVERSIDAD DE MÁLAGA

1.2. Memoria transaccional

A pesar de las dificultades asociadas a la programación paralela, los sistemas de bases de datos llevan décadas aprovechando los recursos de los sistemas multiprocesador, ejecutando consultas y operaciones en paralelo sin que los programadores hayan tenido que lidiar explícitamente con la complejidad asociada al paralelismo descrita anteriormente.

Estos sistemas se basan en el concepto de *transacción*. En bases de datos una transacción garantiza que un conjunto de operaciones sobre la base de datos se realizan, desde el punto de vista del programador, como si el proceso que las ejecuta tuviese acceso exclusivo a la misma. A su vez, el sistema gestor de bases de datos puede permitir la ejecución simultánea de otras operaciones, pero controla y limita las interacciones entre ellas para mantener la consistencia del sistema, garantizando que el resultado final sea indistinguible del que se produciría si todas las operaciones se ejecutasen de forma secuencial. De este modo el programador sólo tiene que lidiar con las interacciones que pueden darse entre estados *finales* de transacciones, en lugar de tener que razonar sobre las interacciones a bajo nivel de su código con el resto del sistema. Adicionalmente el uso de transacciones permite componer varias transacciones individuales simplemente encapsulándolas en una transacción mayor. El motor de la base de datos es el encargado de mantener la consistencia del sistema y permite abstraer al programador de buena parte de la complejidad de la programación paralela a través de una interfaz sencilla y comprensible.

En 1977 Lomet propone utilizar una abstracción similar como mecanismo para garantizar la consistencia de datos compartidos entre diferentes procesos [71]. Introduce el concepto de *atomic actions* como una interfaz para especificar que una serie de instrucciones deben ser atómicas, dejando la responsabilidad del control de la concurrencia al sistema que implemente esta interfaz en lugar de al programador. En 1993 Herlihy introduce el término *memoria transaccional* (TM) con un propósito similar, y propone una arquitectura multiprocesador que hereda el concepto de transacción del modelo de programación utilizado en bases de datos [56]. Además incluye una implementación hardware para dar soporte a esta arquitectura.

La popularización de los CMP ha renovado el interés en la investigación sobre TM. Desde principios de los 2000 han aparecido numerosas propuestas de sistemas TM, tanto hardware como software. Éstas últimas han sido especialmente atractivas desde el punto de vista de la investigación, ya que han permitido desarrollar algoritmos más complejos que se alejan de las limitaciones inherentes

UNIVERSIDAD DE MÁLAGA

al hardware. Sin embargo añaden una instrumentación al código para mantener la consistencia del sistema que va en detrimento del rendimiento, especialmente en códigos en los que éste viene limitado de por sí por los accesos a memoria (*memory-bounded*).

En los últimos años, varios fabricantes de procesadores como Intel, IBM o AMD han desarrollado propuestas destinadas a soportar TM en sus arquitecturas. Intel añadió un conjunto de instrucciones a su ISA para ofrecer soporte HTM básico en su línea de procesadores de consumo a partir de la familia Haswell [112]. IBM ha añadido también este soporte en su línea de procesadores POWER [66] a partir de POWER8 y en sus series de *mainframes* Blue Gene [110] y System Z [59]. AMD por su parte propone extensiones similares, aunque aún no las ha incluido en sus arquitecturas [17]. En todos estos casos el procesador se encarga de la detección y gestión de conflictos entre transacciones pero no garantiza su eventual finalización, dejando en manos del desarrollador la implementación de una ruta software alternativa para estas transacciones. Estos diseños se conocen como *best-effort*, haciendo referencia a que el procesador intentará ejecutar la transacción sin proporcionar garantías de que pueda finalizar. Aun con ésta y otras limitaciones, representan un hito al ser las primeras implementaciones reales de TM en procesadores de consumo.

1.2.1. Ventajas de la memoria transaccional

En el contexto de TM, definimos una *transacción* como un conjunto de instrucciones máquina ejecutadas por un mismo proceso de forma atómica y serializable. Con atómica nos referimos a que los cambios que realice la transacción durante su ejecución son invisibles al resto de procesos concurrentes hasta que ésta finalice (*commit*), momento en el cual todos los cambios se propagarán al resto de hilos sin que éstos puedan observar un estado intermedio de dichos cambios. En caso de que el sistema TM detecte algún conflicto durante la ejecución de la transacción, ésta abortará sin propagar ningún cambio al resto del sistema. Con serializable nos referimos a que la historia de cambios que realice un conjunto de transacciones debe corresponder a la que se obtendría ejecutando secuencialmente *en algún orden* dichas transacciones. Un sistema TM, por tanto, ejecutará transacciones de forma concurrente garantizando estas propiedades. Para ello detectará y resolverá posibles conflictos entre transacciones que puedan violar dichas propiedades, generalmente abortando alguna de las transacciones implicadas.

La atomicidad y la serializabilidad, al igual que la ausencia de restricciones de orden, son características comunes entre las transacciones utilizadas en bases

UNIVERSIDAD DE MÁLAGA

de datos y las utilizadas en TM. Sin embargo, su propósito es diferente. De este modo, en un sistema TM, las transacciones están pensadas para tener una vida corta, acceder a un número limitado de posiciones de memoria y residir en una memoria volátil del sistema, generalmente a nivel de caché. Idealmente la duración de una transacción no debería exceder de un *scheduling quantum* del sistema operativo [56]. Por su parte, las transacciones utilizadas en bases de datos no tienen una duración limitada, pueden acceder a un número arbitrario de posiciones de memoria y son no volátiles, lo que implica que sus cambios se almacenan en memoria no volátil del sistema. De aquí en adelante cuando hablemos de transacciones nos referiremos siempre a un contexto de TM.

Además de simplificar la programación paralela, TM permite aplicar los conceptos de abstracción y composición. Por una parte no es necesario conocer los detalles de implementación de una función para utilizarla. Por otra parte es posible construir métodos complejos que mantengan la atomicidad combinando varias transacciones en una transacción mayor. Esta idea se conoce como *anidamiento de transacciones* [55], y puede soportarse de diferentes maneras. En esta tesis se hará uso de este concepto en el capítulo 4.

Para ilustrar lo anterior, consideremos de nuevo el ejemplo de la clase `account` visto en la sección 1.1. Con soporte TM, los métodos para retirar o ingresar dinero estarían ahora contenidos en sendas transacciones. Si dos hilos intentan actualizar a la vez una cuenta, el sistema TM detectaría un conflicto, que resolvería abortando una de las transacciones y reejecutándola posteriormente, evitando así problemas como *deadlocks* o *convoying*. Además sería posible utilizar la composición de los dos métodos para implementar una función de transferencia: bastaría con encapsular el código necesario en una transacción, sin necesidad de conocer la implementación de los métodos de ingresar o retirar efectivo.

1.2.2. Técnicas de especulación con soporte transaccional

Por su modelo de concurrencia optimista, algunos autores han propuesto utilizar TM como plataforma para soportar modelos de especulación a nivel de hilo (*Thread-Level Speculation*, TLS) [87, 80, 109]. La idea aquí es aprovechar el sistema TM para aumentar el rendimiento de aplicaciones secuenciales ejecutando especulativamente y en paralelo código cuyas dependencias son difíciles de analizar estáticamente. Un ejemplo son bucles cuyas iteraciones realizan accesos a posiciones de memoria irregulares. Aprovechar aquí TM generalmente pasa por ejecutar el bucle en paralelo encapsulando sus iteraciones en distintas transacciones y manteniendo al menos un hilo *safe*, que no ejecutará código especulativo. En un

UNIVERSIDAD DE MÁLAGA

contexto TLS es necesario añadir restricciones de orden a las transacciones de modo que cada transacción haga visibles sus cambios en el orden natural de las iteraciones del bucle. De este modo, si el bucle no tenía dependencias, o éstas eran escasas, es posible extraer paralelismo en códigos secuenciales. Aunque este modelo de paralelismo mediante especulación no requiere el uso de TM, ésta proporciona características deseables como la detección de conflictos o la gestión de versiones que facilitan su implementación.

1.2.3. Operaciones de reducción y memoria transaccional

Las reducciones son un conjunto de operaciones de la forma $O = O \oplus \xi$, que implican una lectura y posterior escritura sobre una misma variable (O) a través de un operador asociativo y conmutativo denominado *operador de reducción* ($\oplus$). Estas operaciones aparecen con frecuencia en aplicaciones científicas, por lo que su optimización ha sido objeto de estudio, existiendo diversas técnicas para su paralelización eficiente [50, 31, 86, 41]. Una limitación de estas técnicas, sin embargo, es la necesidad de garantías de que la variable objeto de la reducción O no sea accedida fuera de la operación de reducción y de que el valor a reducir ξ no dependa de O. Esto requiere un alto grado de conocimiento acerca del algoritmo a implementar, o bien un análisis de dependencias que garantice que los accesos de reducción mantengan estas propiedades durante la ejecución del programa. Dicho análisis no siempre es posible, especialmente en problemas con patrones de acceso irregulares, cuyas dependencias a menudo no son conocidas hasta el momento de la ejecución.

El uso de TM en este tipo de códigos irregulares puede facilitar su paralelización, pero no es una solución óptima debido a los conflictos que puede implicar la presencia de reducciones. Al consistir éstas en una lectura seguida de una escritura en una estructura compartida, el sistema TM puede detectar accesos de reducción realizados por transacciones diferentes como conflictos de datos, perjudicando el rendimiento. Existen propuestas de sistemas TM que evitan detectar estos accesos como conflictos, pero requieren de las mismas garantías que el resto de técnicas mencionadas [43].

1.3. Motivación y contribuciones

El uso de TM como herramienta para la implementación de secciones críticas puede facilitar la programación de códigos paralelos. Sin embargo, el rendimiento

UNIVERSIDAD DE MÁLAGA

obtenible utilizando TM está condicionado en gran medida por la tasa de abortos producidos durante la ejecución. Dichos abortos dependen de los conflictos generados por las transacciones y de la política de detección y resolución de los mismos que implemente el sistema TM.

En el caso de las operaciones de reducción se da la paradoja de que *semánticamente* pueden intercalarse reducciones ejecutadas por hilos diferentes sin alterar el resultado final siempre que cada operación individual sea atómica y que las condiciones de la reducción se mantengan a lo largo de la ejecución del programa. Sin embargo, dado que el sistema TM no es capaz de distinguir este tipo de accesos, producirá abortos por conflictos de datos que pueden ser innecesarios. Dichos abortos serían evitables si se tratasen adecuadamente por parte del sistema TM. Esto sería además especialmente útil en problemas con patrones de acceso irregulares, que dificultan en gran medida un análisis estático del conjunto de datos de reducción así como de la validez de las condiciones de reducción a lo largo de la ejecución del programa. Ante esta situación, la primera pregunta que motiva esta tesis es:

> *¿Tiene sentido tratar las reducciones como una operación independiente en un modelo de memoria transaccional?*

Por otra parte, un denominador común en el uso de soporte TM para implementar modelos de especulación son las restricciones totales de orden entre transacciones. Hay varias propuestas que utilizan un sistema TM para detectar posibles dependencias entre hilos especulativos y no especulativos. Dichas propuestas se suelen basar en el establecimiento de una ordenación secuencial entre transacciones que permita mantener uno de los hilos ejecutando código no especulativo. Sin embargo, tras comprobar experimentalmente que la penalización en rendimiento que implica mantener una restricción estricta de orden entre transacciones a menudo impide extraer rendimiento en aplicaciones, proponemos relajar dichas restricciones a un modelo de orden parcial, que afecte a conjuntos de transacciones y permita finalizar simultáneamente transacciones de un mismo conjunto sin necesidad de serializar la fase de *commit* de las mismas. Por tanto, la segunda pregunta motivadora de esta tesis es:

> *¿Tiene sentido introducir restricciones de orden totales o parciales en un modelo de memoria transaccional?*

Los capítulos 3 y 4 de este trabajo tratan de dar respuesta a ambas preguntas. El primero explora el soporte de operaciones de reducción en los sistemas TM y sus

UNIVERSIDAD DE MÁLAGA

implicaciones en el diseño y la implementación de estos sistemas. El segundo se centra en el soporte de restricciones parciales de orden entre transacciones para mejorar los modelos de especulación basados en un orden estricto de precedencia entre transacciones.

Las principales contribuciones de esta tesis son las siguientes:

- Ampliación del modelo TM para soportar operaciones de reducción transaccionales. Este modelo permite la concurrencia de las operaciones de reducción de distintas transacciones sin producir abortos siempre que sea posible, a la vez que detecta posibles violaciones de las propiedades de reducción en tiempo de ejecución.

- Inclusión de restricciones de orden total para dar soporte a códigos con reducciones parciales, donde las condiciones de reducción sólo se cumplen en un subconjunto del total de direcciones que presentan accesos de reducción. Esto permite que el programador pueda indicar patrones *potenciales* de reducción en el código sin necesidad de garantizar las propiedades de dichas reducciones a lo largo de la ejecución del programa.

- Adaptación del modelo anterior a sistemas TM reales. Concretamente se proponen dos implementaciones software que usan como base algoritmos de sistemas STM del estado del arte.

- Introducción de un modelo de *orden parcial* en TM, que ordene entre sí conjuntos de transacciones en lugar de transacciones individuales. De este modo, las transacciones pertenecientes a un mismo conjunto pueden finalizar sin restricciones, pero se garantiza que todas las transacciones de un conjunto anterior han terminado antes de que cualquiera de las transacciones de un conjunto posterior puedan hacer visibles sus cambios.

- Introdución del concepto de *barrera transaccional* como primitiva para soportar el modelo de *orden parcial*. Se trata de una directiva dentro de un entorno TM con la misma semántica que una barrera de sincronización tradicional, pero que permite que los hilos continúen la ejecución de forma especulativa sin bloquearse. Esta directiva proporciona al sistema TM información sobre el estado de ejecución de los diferentes hilos que permiten establecer un orden de precedencia parcial en tiempo de ejecución.

- Implementación del modelo de *orden parcial* con barreras transaccionales en un sistema de TM hardware real.

UNIVERSIDAD DE MÁLAGA

Estas contribuciones están orientadas a dotar a los sistemas TM de soporte a restricciones de orden y a operaciones de reducción, y a estudiar las posibilidades que habilita dicho soporte de cara a la paralelización de aplicaciones.

1.4. Estructura de la tesis

El resto de esta tesis está estructurado como sigue.

El capítulo 2 ofrece una perspectiva de la memoria transaccional desde diferentes ejes. En él se detallan los principales aspectos a tener en cuenta desde un punto de vista del diseño de sistemas TM, y se incluye un estudio de las diferentes visiones de la semántica transaccional que se pueden encontrar en la literatura. El capítulo concluye con un breve análisis de una selección de sistemas TM software, hardware e híbridos relevantes para esta tesis.

El capítulo 3 presenta nuestro trabajo para el soporte de reducciones y restricciones de orden total en TM. Incluye un estudio de diferentes aproximaciones para abordar la paralelización de códigos que incluyen reducciones, el diseño e implementación de los sistemas TM desarrollados y su evaluación en un conjunto de escenarios de interés. Este capítulo comprende las tres primeras contribuciones antes descritas.

El capítulo 4 presenta nuestro trabajo para el soporte de restricciones de orden parcial en sistemas HTM. Incluye un estudio de las limitaciones y características relevantes de los sistemas HTM de consumo a los que va orientada nuestra propuesta, el diseño e implementación de la misma en forma de barreras transaccionales, y su correspondiente evaluación en una serie de aplicaciones de interés. El capítulo comprende las tres últimas contribuciones de esta tesis.

El capítulo 5 concluye la tesis, sintetiza las contribuciones publicadas y ofrece posibles líneas de trabajo para extender las propuestas presentadas.

Adicionalmente, el apéndice A detalla la API[2] utilizada a lo largo de este trabajo, basada en trabajos anteriores sobre TM, y las ampliaciones realizadas para soportar las propuestas de los capítulos 3 y 4.

UNIVERSIDAD DE MÁLAGA

[2]Acrónimo de *Application Programming Interface* (interfaz de programación de aplicaciones).

UNIVERSIDAD
DE MÁLAGA

Capítulo 2

Antecedentes

Este capítulo ofrece una perspectiva de la memoria transaccional (TM) desde diferentes ejes. La sección 2.1 introduce las transacciones y sus propiedades básicas. La sección 2.2 describe una interfaz mínima de TM que se utilizará en el transcurso de la tesis. La sección 2.3 ofrece una visión de las diferentes decisiones de diseño a considerar a la hora de implementar sistemas TM. La sección 2.4 incluye consideraciones adicionales respecto a la semántica de las transacciones. En ella se detallan diferentes criterios a la hora de definir propiedades de las transacciones como la atomicidad, la consistencia o el anidamiento. Finalmente, la sección 2.5 analiza varias implementaciones de sistemas TM, incluyendo diseños software, hardware e híbridos. Esta sección incluye una revisión de las extensiones desarrolladas para el soporte de memoria transaccional en los procesadores más recientes.

2.1. Transacciones

Una transacción es una secuencia de acciones que se muestran como indivisibles e instantáneas a un observador externo [55]. Tradicionalmente se han establecido cuatro garantías que deben cumplir las transacciones, conocidas como propiedades ACID[3] [63]:

- *Atomicidad*: una transacción requiere que sus cambios sean visibles de forma completa e instantánea, o que no sean visibles en absoluto. Esto implica

[3]Acrónimo de *Atomicity*, *Consistency*, *Isolation*, *Durability*.

UNIVERSIDAD DE MÁLAGA

que toda transacción tiene dos finales posibles: finalizar con éxito, haciendo visibles sus cambios de forma instantánea al resto del sistema; o deshacer sus cambios hasta el instante del inicio de la transacción sin que el sistema detecte evidencia alguna de su ejecución. En el primer caso hablamos del *commit* de la transacción, mientras que en el segundo caso decimos que la transacción ha *abortado* su ejecución. La sección 2.4.1 ofrece un análisis más detallado de esta propiedad.

- *Consistencia*: dado que las transacciones modifican el estado de un sistema, esta propiedad implica que sus cambios deben partir de un estado consistente del mismo, y deben dejar otro estado también consistente al finalizar. Qué implica un estado consistente depende de la aplicación particular que se esté ejecutando, y hace referencia a los valores posibles que pueden tomar sus estructuras de datos en cada momento. La sección 2.4.2 analiza diferentes criterios de consistencia para TM propuestos en la literatura.

- *Aislamiento*: esta propiedad hace referencia a la necesidad de que una transacción no pueda interferir con el resto de transacciones concurrentes durante su ejecución. Garantizar el aislamiento requiere que el sistema TM detecte y resuelva las posibles interacciones que puedan producirse entre transacciones. Esta garantía es una de las diferencias más notables entre las transacciones y otras primitivas para la programación paralela y uno de los motivos del atractivo de TM como modelo de programación. La sección 2.3 ahonda en los mecanismos que permiten estas garantías.

- *Durabilidad*: implica que los cambios que realice una transacción exitosa sean permanentes en el sistema —se almacenen en memoria no volátil, e.g. un disco duro—, y estén disponibles para cualquier transacción posterior.

Estas cuatro propiedades provienen del modelo de transacciones utilizado en bases de datos. El paradigma de TM, sin embargo, no considera la durabilidad, ya que trabaja con datos inherentemente volátiles —memoria principal y caché—; ni contempla un criterio único de consistencia, al ser ésta dependiente de la aplicación concreta. Podemos decir por tanto que las propiedades básicas a garantizar por un sistema TM son la *atomicidad* y el *aislamiento* de las transacciones. La combinación de ambas propiedades produce la *serializabilidad* de las transacciones [83], que es el criterio de consistencia base de un sistema transaccional[4].

[4]La serializabilidad no es el único criterio de consistencia en un modelo transaccional, pero es el nexo que une el modelo transaccional de bases de datos y la memoria transaccional, siendo a la vez el criterio más garantista del primero y el más laxo del segundo.

UNIVERSIDAD DE MÁLAGA

```
1 // API TM básica
2 TmBegin();
3 bool TmEnd();
4 void TmAbort();
5
6 // Funciones adicionales para TM explícitos
7 T TmRead(T* addr);
8 TmWrite(T* addr, T val);
```

Figura 2.1: Interfaz transaccional mínima.

2.2. La interfaz transaccional

A lo largo de esta tesis utilizaremos una interfaz mínima para referirnos a TM en cualquiera de sus formas similar a la mostrada en la figura 2.1. Distinguimos tres operaciones básicas, similares al modelo original de Herlihy [56] y las complementamos con dos operaciones adicionales que permiten anotar los accesos transaccionales de lectura y escritura en sistemas TM que lo necesiten. Las funciones `TmBegin` y `TmEnd` delimitan el bloque de instrucciones que constituyen el cuerpo de la transacción. Cualquier acceso a memoria entre estas dos instrucciones será vigilado por el sistema TM para detectar posibles conflictos. La instrucción `TmEnd` intentará consolidar en memoria los cambios tentativos que haya realizado la transacción hasta el momento y devolverá si ha tenido éxito o no. Por su parte, la función `TmAbort` descarta los cambios realizados por la transacción desde el inicio de la misma hasta la llamada a `TmAbort`. Una transacción fallida restaurará el estado de la ejecución al momento inmediatamente posterior a la llamada a `TmBegin`.

Como se describirá en la sección 2.5.1, en sistemas TM explícitos es necesario especificar qué datos queremos que sean transaccionales. En estos sistemas son necesarias dos funciones adicionales que permitan realizar un acceso transaccional de lectura o escritura sobre una posición de memoria. `TmRead` lee el dato de la posición de memoria `addr` que se pasa como parámetro y devuelve su valor. El dato pasará a formar parte del conjunto de datos transaccional de lectura. `TmWrite` escribe transaccionalmente el valor `val` en la posición de memoria `addr` que se pasa como parámetro. Dicho dato pasará a formar parte del conjunto de datos transaccional de escritura.

En el apéndice A se ofrecen detalles adicionales de la interfaz transaccional utilizada a lo largo de este trabajo, incluyendo las primitivas propuestas en las contribuciones de los capítulos 3 y 4, *callbacks* adicionales para compatibilizar la interfaz con diferentes implementaciones TM y ejemplos de uso.

UNIVERSIDAD DE MÁLAGA

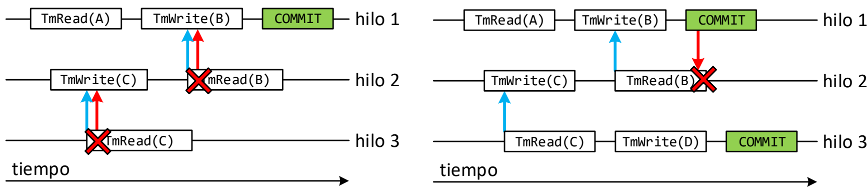


Figura 2.2: Control de concurrencia pesimista (izquierda) y optimista (derecha). Las flechas azules indican cuándo se produce un conflicto, las flechas rojas indican cuando se detecta por el sistema TM, las cruces indican cuándo se resuelve.

2.3. El espacio de diseño en memoria transaccional

A la hora de soportar la interfaz que hemos descrito en la sección 2.2 es necesario considerar una serie de decisiones de diseño que cubren diferentes aspectos de un sistema TM. Estos aspectos no son ortogonales, por lo que algunos condicionarán otras decisiones o limitarán el espacio de diseño en mayor o menor medida. En esta sección se analizan las principales cuestiones a tratar en el diseño de un sistema TM.

2.3.1. Control de concurrencia

En el plano más general, la función de un sistema TM es arbitrar el acceso concurrente de distintos hilos de ejecución a datos compartidos, lo que se traduce en detectar y resolver adecuadamente posibles conflictos en el acceso a estos datos. En general distinguiremos los accesos de lectura y de escritura a un objeto compartido y consideraremos que se ha producido un conflicto cuando dos transacciones diferentes accedan concurrentemente a un objeto compartido y al menos uno de estos accesos sea de escritura. El sistema transaccional resolverá el conflicto abortando o retrasando alguna de las transacciones implicadas.

Podemos distinguir tres eventos ante un conflicto: cuándo se *produce*, cuándo se *detecta* y cuándo se *resuelve*. Dependiendo del diseño del sistema TM estos eventos pueden producirse de forma simultánea o en instantes diferentes, siempre que lo hagan en el orden anterior. En el primer caso hablamos de control de concurrencia *pesimista*: cuando se produce un acceso a un objeto compartido, el sistema TM detecta posibles conflictos y los resuelve, abortando o retrasando alguna de las

UNIVERSIDAD DE MÁLAGA

transacciones implicadas. En el segundo caso la detección y la resolución pueden producirse en un instante posterior, por ejemplo justo antes de la finalización (*commit*) de alguna de las transacciones. En este caso nos referimos a un control de concurrencia *optimista*, ya que permite que varias transacciones accedan a un mismo objeto compartido sin abortar incluso si se han producido conflictos, que deberán resolverse antes de que los cambios de estas transacciones se hagan visibles al resto del sistema. Un sistema TM con concurrencia optimista tiene mayor libertad a la hora de resolver conflictos y puede implementar políticas complejas que minimicen el número de transacciones a abortar.

En la figura 2.2 podemos ver un ejemplo de ambos tipos de concurrencia. En ella, tres hilos ejecutan sendas transacciones que acceden a varios objetos compartidos. Las flechas azules indican cuándo se *produce* el conflicto, las rojas indican cuándo se *detecta* y las cruces cuándo se *resuelve*, en este caso abortando una de las transacciones. En un control de concurrencia pesimista (izquierda), el hilo 3 detecta un conflicto con el hilo 2 al intentar leer C, y lo resuelve abortando la transacción. Poco después el hilo 2 detecta otro conflicto con el hilo 1 al intentar leer B y aborta también su transacción. Por su parte la transacción activa del hilo 1 termina correctamente. Un control de concurrencia optimista (derecha) puede evitar el aborto del hilo 3 en este escenario. En este caso los conflictos se detectan y resuelven antes de la fase de *commit* de las transacciones. El hilo 1 es ahora quien detecta el conflicto con el hilo 2, que aborta igual que en el escenario anterior. El hilo 3 produce un conflicto con el hilo 2 al leer C, pero retrasa su detección y resolución hasta la fase de *commit*. En ese momento el conflicto no requiere ninguna acción porque el hilo 2 ha abortado, así que el hilo 3 puede finalizar su transacción. En este caso el modelo pesimista ha abortado una transacción extra por la necesidad de resolver el conflicto en el acto.

A la hora de su implementación los modelos de concurrencia pesimistas deben evitar la aparición de *deadlocks*, que pueden producirse si la política de resolución de conflictos permite retrasar o bloquear una transacción en espera de que se libere el acceso a algún dato compartido que esté en uso por parte de otra transacción. Por su parte, algunos modelos de concurrencia optimista pueden producir *livelocks* si una transacción abortada causa a su vez el aborto de otra al reiniciar su ejecución [10].

2.3.2. Gestión de versiones

Para mantener la atomicidad es necesario que el sistema TM pueda restaurar el estado anterior a una transacción en caso de aborto. Esto requiere un registro de los

UNIVERSIDAD DE MÁLAGA

cambios tentativos —escrituras— que realice cada transacción sobre los objetos compartidos a los que acceda o, dicho de otra forma, gestionar las diferentes *versiones* de dichos objetos en el sistema. Dos enfoques son posibles aquí: en una gestión de versiones *eager* las escrituras que realice una transacción se actualizan en la memoria compartida del sistema y se requiere un búfer que almacene los valores anteriores de esas posiciones para poder restaurarlos en caso de aborto. Este búfer se suele denominar *undo-log* en la literatura, ya que su utilidad es deshacer los cambios de la transacción en caso de aborto. Otra posibilidad es que ninguna escritura actualice la memoria compartida hasta que la transacción finalice. En este caso hablamos de gestión de versiones *lazy*, y se requiere un búfer que almacene los nuevos valores que la transacción vaya a modificar. Este búfer conoce como *redo-log*, y servirá para actualizar los datos durante la fase de *commit* de la transacción.

Intuitivamente podemos observar que la gestión de versiones *eager* sigue una filosofía *optimista*, ya que acelera el caso exitoso (esto es, que la transacción finalice sin abortar) al no tener que actualizar los valores en la fase de *commit*; pero penaliza la recuperación en caso de conflicto, ya que requiere deshacer todos los cambios tentativos que ha realizado la transacción. La gestión de versiones *lazy* sigue la filosofía opuesta: la fase de *commit* de cada transacción requiere actualizar la memoria con los nuevos valores, pero no es necesaria ninguna recuperación en caso de aborto, ya que los cambios tentativos aún no han actualizado la memoria.

2.3.3. Detección de conflictos

En un control de concurrencia pesimista la detección de conflictos viene predeterminada; cada transacción bloquea progresivamente los datos que vaya a modificar y los conflictos se detectan cuando una transacción intenta acceder a un dato y lo encuentra bloqueado por otra. En un control de concurrencia optimista, sin embargo, existen varias posibilidades que dependen de tres factores: cuándo se detecta el conflicto, con qué granularidad se detecta el conflicto y qué interacciones se consideran conflictos.

La mayoría de diseños TM consideran tres instantes a la hora de detectar conflictos. Una posibilidad es detectarlos cuando la transacción intenta acceder a un dato. Esta política se conoce como *eager conflict detection* y es la más similar a la utilizada en un sistema con control de concurrencia pesimista. Otra opción es efectuar la detección de conflictos en la fase de *commit* de la transacción. En este caso, denominado *lazy conflict detection*, la transacción validará los datos accedidos antes de finalizar. Una tercera opción es efectuar una validación parcial de los

UNIVERSIDAD DE MÁLAGA

datos accedidos en algún instante anterior a la fase de *commit*. Esta validación puede afectar a un subconjunto de dichos datos y efectuarse varias veces a lo largo de la vida de la transacción.

Independientemente de la opción elegida, la detección de conflictos requiere mantener un registro de las posiciones accedidas por cada transacción. Normalmente cada transacción define dos conjuntos: el *read-set*, que almacena las direcciones de los objetos leídos por la transacción; y el *write-set* que almacena las direcciones de los objetos escritos por la transacción. La información contenida en estos conjuntos sirve de base para la detección de conflictos de los sistemas TM. La sección 2.5 incluye una revisión de diferentes sistemas TM donde se pueden observar distintas estrategias a la hora de implementar estos conjuntos y su incidencia en los algoritmos de detección.

La *granularidad* hace referencia a la unidad mínima de memoria sobre la que un sistema TM detecta conflictos. Los sistemas TM software suelen tener una granularidad de *palabra*, donde se detecta un conflicto si dos transacciones acceden a una misma palabra de memoria; o de *objeto*, donde se detecta un conflicto si dos transacciones acceden a un mismo objeto independientemente de si lo hacen o no a atributos distintos. La mayoría de sistemas hardware, sin embargo, detectan los conflictos con una granularidad mayor, normalmente relacionada con el tamaño de bloque caché. Esto se debe a que suelen utilizar el protocolo de coherencia de caché para implementar la detección de conflictos. Cuanto mayor sea la granularidad a la hora de detectar conflictos, mayor es la posibilidad de que se produzcan abortos debidos a *falsos positivos* (*false sharing*), que se producen cuando el sistema TM detecta un conflicto inexistente entre dos transacciones que acceden a datos diferentes, pero que el sistema no es capaz de distinguir por encontrarse éstos en la misma unidad de detección (por ejemplo, en el mismo bloque de caché).

El último factor determina si el sistema TM detectará como conflictos dos accesos problemáticos entre transacciones activas, es decir, que no han sido aún confirmados en memoria; o si sólo considerará conflictos cuando una de las transacciones haya terminado, confirmando sus cambios en memoria. En general los sistemas TM con detección de conflictos y gestión de versiones *eager* suelen usar la primera opción, y sistemas con detección de conflictos *lazy* suelen emplear la segunda. Sin embargo esto no es una norma estricta; más adelante se analizarán varios sistemas TM para dar una perspectiva de distintas estrategias propuestas en la literatura.

UNIVERSIDAD
DE MÁLAGA

2.4. Semántica transaccional

En la sección 2.1 se introducían las propiedades ACID como una serie de garantías a cumplir por las transacciones. Paradójicamente no existe una definición única de dichas propiedades; distintos sistemas TM las interpretan de manera diferente, lo que puede dar lugar a incompatibilidades al ejecutar un mismo código transaccional en sistemas TM diferentes. En esta sección hemos tratado de establecer un marco común que permita categorizar los distintos sistemas TM en cuanto a las garantías que realmente ofrecen de cara al programador.

2.4.1. Atomicidad

Es habitual plantear TM como alternativa al uso de *locks* para implementar secciones críticas evitando los inconvenientes derivados de su bajo nivel de abstracción. Sin embargo, *locks* y transacciones no son equivalentes desde un punto de vista semántico [9]. Esto puede originar problemas a la hora de utilizar transacciones, especialmente si partimos de un código paralelo que utilice *locks*. La diferencia la encontramos en la atomicidad, concretamente en el ámbito de la misma —atómico respecto a qué—. Al usar *locks*, la semántica es clara: el código de una sección crítica protegida con un *lock* es atómico respecto a otro código protegido con *el mismo lock*. Cualquier código no protegido, o protegido con *otro lock* puede solaparse con la sección crítica. Si sustituimos ese *lock* y protegemos la sección crítica con una transacción, dicha sección será atómica *respecto a cualquier otra transacción*. El uso de transacciones amplía el alcance de la atomicidad, e impide algunas de las interacciones que podían producirse utilizando distintos *locks*. Si el código de la sección crítica dependía de esas interacciones para su corrección, usar transacciones puede dar lugar a *deadlocks*.

Un ejemplo de este fenónemo lo encontramos en la figura 2.3, donde se utilizan dos *locks* para proteger sendas secciones críticas en los hilos 1 y 2. En este caso, el uso de *locks* distintos garantiza la atomicidad de las secciones críticas respecto a otras secciones que hagan uso de los mismos *locks*, pero permite que ambos hilos ejecuten sus secciones en paralelo. En el ejemplo, el código de las líneas 4 y 5 de ambos hilos actúa a modo de barrera, garantizando que ambos hilos llegan a la línea 6 sólo cuando ambos *flags* están activos. Al usar transacciones ampliamos la atomicidad de las secciones a *cualquier otra transacción*, lo que produce un *deadlock*: no es posible que ninguno de los hilos pueda observar ambos *flags* activos sin que el sistema TM detecte un conflicto y aborte una de las transacciones, dejando a la otra bloqueada en su correspondiente bucle `while`.

UNIVERSIDAD DE MÁLAGA

```
1 bool flagA = False;
2 bool flagB = False;
3 Lock lock1, lock2;
```

Hilo principal

```
1 Acquire(lock1);
2 TmBegin();
3 ...
4 while(!flagA);
5 flagB = True;
6 ...
7 TmEnd();
8 Release(lock1);
```

Hilo 1

```
1 Acquire(lock2);
2 TmBegin();
3 ...
4 flagA = True;
5 while(!flagB);
6 ...
7 TmEnd();
8 Release(lock2);
```

Hilo 2

Figura 2.3: Diferencias entre *locks* y transacciones. El código se ejecuta correctamente utilizando *locks* (líneas 1 y 8 de los hilos 1 y 2), pero produce *deadlock* al usar transacciones. Código adaptado de [9].

Además de las interacciones entre transacciones se debe considerar la atomicidad con el código no transaccional. En este caso no hay un único modelo y distinguimos entre *aislamiento débil*, donde una transacción sólo garantiza su atomicidad respecto al resto de transacciones; y *aislamiento fuerte*, que fuerza a la transacción a mantener su atomicidad tanto respecto a otras transacciones como al resto de código no transaccional. En este último modelo, el sistema TM trata implícitamente cada instrucción no transaccional como una transacción individual. Aunque desde el punto de vista del programador un modelo de aislamiento fuerte puede parecer más atractivo, los modelos de aislamiento débil son más sencillos de implementar de forma eficiente, especialmente en software. Generalmente encontraremos modelos de aislamiento fuerte en sistemas TM hardware y débil en sistemas software.

Del mismo modo que las transacciones impiden algunas interacciones permitidas en un modelo con *locks*, los modelos de aislamiento fuerte impiden algunas interacciones que permiten los modelos de aislamiento débil. Por esta razón, un código que se ejecute correctamente en un modelo débil puede bloquearse al ejecutarse bajo un modelo fuerte si las condiciones de progreso dependían de este tipo de interacciones. La figura 2.4 ilustra esta situación. En este caso, el uso de un sistema TM con aislamiento fuerte evita que la transacción del hilo 2 pueda finalizar sin ser abortada, ya que debe esperar a que el hilo 1 actualice `flagB` antes de poder finalizar, lo que produce un conflicto que no ocurre en sistemas con aislamiento débil.

UNIVERSIDAD DE MÁLAGA

```
1    bool flagA = False;
2    bool flagB = False;
3    Lock lock1, lock2;
```

Hilo principal

```
1
2    ...
3    while(!flagA);
4    flagB = True;
5    ...
6
```

Hilo 1

```
1    TmBegin();
2    ...
3    flagA = True;
4    while(!flagB);
5    ...
6    TmEnd();
```

Hilo 2

Figura 2.4: Diferencias entre aislamiento fuerte y débil. El código se ejecuta correctamente en un sistema TM con aislamiento débil, pero causará un *deadlock* en sistemas con aislamiento fuerte. Código adaptado de [9].

Por otra parte, programas codificados bajo la asunción de un modelo de aislamiento fuerte pueden causar inconsistencias si se ejecutan en un modelo de aislamiento débil. El programador deberá conocer por tanto bajo qué modelo está trabajando y adaptar su código al mismo, dado que ninguno de los modelos de atomicidad engloba al otro.

2.4.2. Criterios de consistencia

Hemos presentado TM como una interfaz que proporciona al programador una forma de ejecutar instrucciones de forma atómica y aislada, dejando a cargo del sistema transaccional la responsabilidad de garantizar estas propiedades mientras intenta extraer todo el paralelismo posible. Sin embargo no existe un criterio único respecto a las garantías de consistencia en sistemas TM, a diferencia del modelo transaccional utilizado en bases de datos. Esto se debe a que TM proporciona un entorno mucho más rico en cuanto a interacciones que el modelo relacional de bases de datos. Sintetizamos en esta sección los criterios más comunes que se encuentran en la literatura. Un análisis más detallado puede encontrarse en [30].

Historias y ejecuciones

Con el objetivo de establecer un marco común a la hora de analizar estos criterios, consideramos un entorno en el que varios hilos de ejecución $P_0, P_1, ..., P_n$ ejecutan

UNIVERSIDAD DE MÁLAGA

transacciones de forma concurrente. Dichas transacciones se componen de una serie de primitivas denominadas *eventos* que incluyen el inicio de la transacción (*start*), su finalización (*commit*), y las operaciones de lectura (*read*) y escritura (*write*). Cada evento comprende la invocación del mismo y una respuesta. Una transacción se considera *completa* si comienza con un evento *start*, seguido de cero o más eventos de lectura o escritura y termina bien con una respuesta positiva a un *commit* o bien con una respuesta *abort* a alguno de sus eventos. Consideramos t_{ij} a la *j-ésima* transacción ejecutada por el hilo P_i. Las invocaciones y respuestas de los eventos de t_{ij} deben suceder en P_i secuencialmente y sin solaparse de modo que a cada invocación le siga su respuesta. En una ejecución correcta, cada transacción t lleva asociado un *punto de serialización* $t*$ que indica un instante de la ejecución en el que la secuencia de invocaciones y respuestas de los eventos de t son consistentes con el estado del sistema.

Si una transacción no ha finalizado aún, pero su secuencia de eventos es un prefijo de una transacción completa, la denominamos *live*. Si una transacción completa termina con una respuesta exitosa a un *commit* la denominamos *committed*. Por último, si una transacción termina con una respuesta *abort* a cualquier evento la denominamos *aborted*. Asimismo denominamos *intervalo de ejecución* al espacio temporal entre la invocación del primer evento de t y la respuesta a último. Un hilo P_i puede ejecutar varias transacciones $t_{ij}, t_{ik}, ..., t_{in}$ siempre que éstas sean completas —salvo, opcionalmente, la última— y no se solapen entre sí. Otros hilos pueden ejecutar transacciones que cumplan estas reglas, pudiéndose solapar en el tiempo eventos y transacciones ejecutadas por distintos hilos.

Para analizar el estado de un sistema como el anterior en un momento concreto, hablaremos de *historia* H para referirnos a una secuencia finita de *eventos* producidos por una o más transacciones en uno o más hilos. Una transacción t_{ij} estará en H si H contiene eventos de t_{ij}. Del mismo modo, un hilo P_i estará en H si H contiene eventos de P_i. Denominamos $H|t_{ij}$ a la subsecuencia de eventos de t_{ij} en H y denominamos $H|P_i$ a la subsecuencia de eventos del hilo P_i en H. En general, para n eventos podemos considerar $n!$ posibles historias. En adelante consideraremos que las historias están *bien formadas*. Esto implica que para toda transacción completa $t \in H$ no existen eventos de $H|t$ anteriores a la invocación a t_{start} ni posteriores a una respuesta t_{abort} o a una respuesta a un t_{commit}. Además, ninguna transacción completa t' ejecutada en el mismo hilo que t puede solaparse con ésta, es decir, que o bien el primer evento de t es posterior al último evento de t', o bien el último evento de t es anterior al primer evento de t'. En el caso de una transacción *live* $t'' \in P$ sólo se admitirá una por hilo $P \in H$, y no debe existir ningún evento de una transacción distinta de t'' posterior a ella.

UNIVERSIDAD DE MÁLAGA

Podemos clasificar las distintas transacciones que forman una historia H en tres subconjuntos disjuntos: el subconjunto $Comm(H) \subseteq H$, que engloba a todas las transacciones *committed* de H; el subconjunto $Live(H) \subseteq H$, que engloba a las transacciones *live* de H; y el subconjunto $Ab(H) \subseteq H$, que engloba a las transacciones *aborted* de H.

Finalmente, distinguimos historia de *ejecución* (α), refiriéndonos con ésta última a la secuencia de cambios en los objetos del sistema producidos por los eventos de una historia, es decir, a los *efectos* en el sistema de las acciones reflejadas en H. Una ejecución es *legal* si dicha secuencia de cambios obedece al algoritmo que se pretende ejecutar.

La figura 2.5 muestra una ejecución legal α de tres hilos concurrentes, donde los puntos de serialización $t_{0,1}*$ y $t_{2,1}*$ marcan los cambios en los objetos A y B del sistema. De acuerdo a las definiciones anteriores, $Comm(H) = \{t_{0,1}, t_{2,1}\}$, $Ab(H) = \{t_{1,1}\}$ y $Live(H) = \{t_{0,2}, t_{1,2}\}$.

La ejecución $\alpha = \{t_{0,1}*, t_{1,2}*\}$ muestra la secuencia de cambios en los objetos A y B consecuencia de las acciones de H. Podemos establecer α a partir de los puntos de serialización $t_{0,1}*$ y $t_{2,1}*$ por parte de las transacciones de $Comm(H)$.

La historia de una transacción:

$$H|t_{1,1} = \{t_{1,1}.start, t_{1,1}.ok, t_{1,1}.read(A), t_{1,1}.abort\},$$

o la historia de un hilo:

$$H|P_1 = \{t_{1,1}.start, t_{1,1}.ok, t_{1,1}.read(A), t_{1,1}.abort, t_{1,2}.start, t_{1,2}.ok\},$$

son únicas. Sin embargo, si consideramos al sistema en su conjunto, existen multitud de posibles historias H. Por ejemplo:

$$H = \{t_{0,1}.start, t_{1,1}.start, t_{0,1}.ok, t_{2,1}.start, t_{1,1}.ok, ..., t_{0,2}.read(B)\},$$

o bien:

$$H' = \{t_{1,1}.start, t_{1,1}.ok, t_{0,1}.start, t_{2,1}.start, t_{0,1}.ok, t_{0,1}.read(A), ..., t_{1,2}.ok\},$$

o bien:

$$H'' = \{t_{0,1}.start, t_{1,1}.start, t_{1,1}.ok, t_{2,1}.start, t_{0,1}.read(A), ..., t_{0,2}.read(B)\}.$$

Tanto H como H' son historias válidas y *bien formadas* de la ejecución α.

La historia H'', sin embargo, no está bien formada: en este caso la invocación del evento $t_{0,1}.read(A)$ se produce antes que la respuesta al evento $t_{0,1}.start$, resultando en una historia no válida.

UNIVERSIDAD DE MÁLAGA

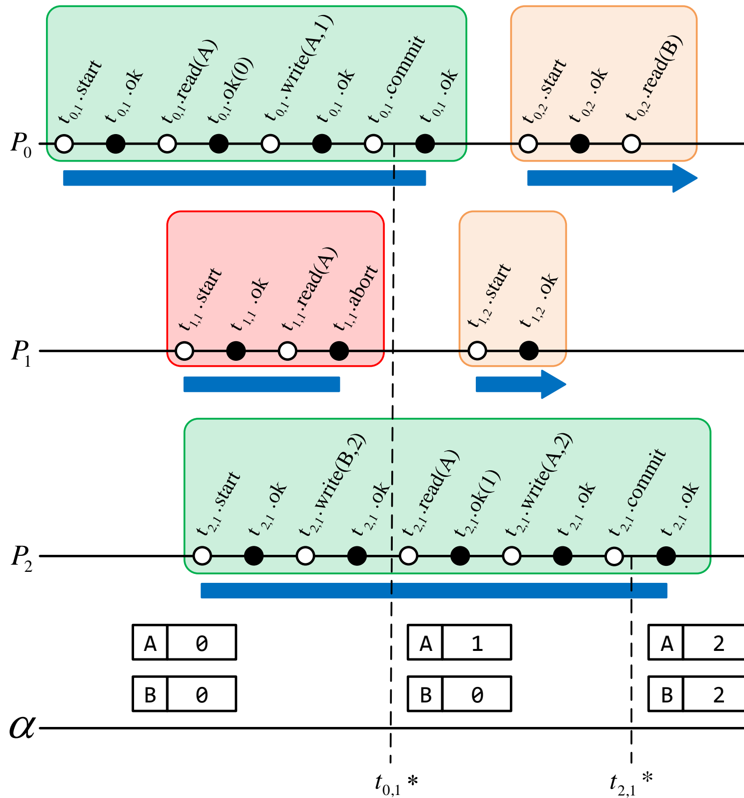


Figura 2.5: Representación de una ejecución legal de varias transacciones en tres hilos concurrentes. Los círculos blancos representan invocaciones a eventos y los círculos negros sus respuestas. α representa la secuencia de estados de la ejecución. Las transacciones marcadas en verde pertenecen al conjunto $Comm(H)$, las transacciones marcadas en rojo pertenecen al conjunto $Ab(H)$ y las transacciones marcadas en naranja pertenecen al conjunto $Live(H)$. Los rectángulos azules bajo cada transacción indican sus intervalos de ejecución. $t_{i,j}*$ representa el punto de serialización de la transacción $t_{i,j}$.

UNIVERSIDAD DE MÁLAGA

Criterios de consistencia

Un criterio de consistencia restringe el conjunto de todas las posibles historias *bien formadas* para una ejecución α en base a determinadas propiedades. El criterio base en TM es *serializability* [83]. El término, heredado de los criterios de corrección entre transacciones en bases de datos, implica que a toda $t \in Comm(H)$ se le puede asociar un *punto de serialización* $t*$ durante α. Con estos puntos se debe poder construir una ejecución secuencial σ equivalente a α, ejecutando cada $t \in Comm(H)$ secuencialmente en $t*$, de modo que cada transacción invoque los mismos eventos y reciba las mismas respuestas en α y en σ.

Strict serializability es un criterio más restringido que obliga a que los $t*$ de cada $t \in Comm(H)$ sólo puedan asociarse a t durante el intervalo de ejecución de la transacción. Ambos criterios implican que toda $t \in Comm(H)$ debe ser correcta en todo momento y debe observar una única vista secuencial del sistema, que sería σ.

La idea de observar una vista secuencial σ única y equivalente a α por parte de toda $t \in Comm(H)$ es un criterio básico en TM. Dado que *serializability* es el criterio más permisivo que garantiza esta propiedad, se suele considerar como la condición mínima de corrección en TM. En los modelos de bases de datos, sin embargo, se considera el criterio más garantista, existiendo otros más laxos como *causal consistency* [2] donde las transacciones $t \in Comm(H)$ pueden observar distintas ejecuciones α' que cumplan determinadas restricciones. Los detalles de estos criterios no se abordan en esta tesis, en la que consideramos *serializability* como garantía mínima de consistencia en TM.

Los criterios vistos hasta el momento sólo consideran $t \in Comm(H)$. En un entorno de interacciones controladas, como las bases de datos, las transacciones $t' \in Ab(H)$ y $t'' \in Live(H)$ no tienen por qué producir una ejecución incorrecta, ya que no han actualizado los objetos compartidos. Por esta razón es admisible que puedan leer datos inconsistentes durante su ejecución siempre que acaben abortando en algún momento. En un entorno de TM, sin embargo, podemos requerir mayores garantías, ya que no tenemos control sobre las operaciones que la transacción llevará a cabo con los valores tentativos que lea durante su intervalo de ejecución. La figura 2.6 ilustra este problema. Supongamos que los objetos compartidos A y B tienen inicialmente un valor de 1 y 0 respectivamente. La transacción $t_{1,1}$ lee ambos y los utiliza para calcular c. Por su parte, la transacción $t_{2,1}$ incrementa atómicamente ambos objetos. Si $t_{1,1}$ lee el valor inicial de A, pero lee el valor incrementado de B, el cálculo de c produciría una excepción que nunca debería haber ocurrido. Bajo *serializability* este escenario es admisible siempre que la transacción $t_{1,1}$ no llegue a finalizar con éxito. Accesos a vectores, operaciones

UNIVERSIDAD DE MÁLAGA

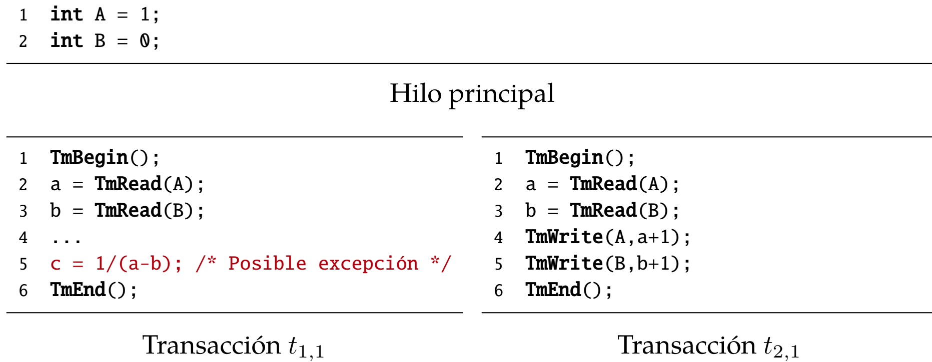


Figura 2.6: Límites de *serializability*.

de entrada y salida u operaciones con punteros son otros escenarios que pueden producir resultados inesperados para el programador si el sistema TM no ofrece mayores garantías. En general, *serializability* no es un criterio admisible en TM si pueden existir operaciones inseguras dentro de una transacción [105].

Para evitar este tipo de problemas Guerraroui y Kapałka introducen *opacity* como un criterio más restrictivo para la corrección en TM [48]. Intuitivamente este criterio es similar a *strict serializability* con el añadido de que las transacciones a considerar se amplían a cualquier $t \in H$. Esto implica que $t_{1,1}$ en el ejemplo anterior no podría leer el valor actualizado de B, ya que en ese caso no se podría establecer un punto de serialización $t*$ para $t_{1,1}$ que formase parte de una ejecución secuencial legal.

Garantizar *opacity* en el escenario de la figura 2.6 requiere que la invocación de $t_{1,1}.read(B)$ produzca una respuesta $t_{1,1}.abort$ que finalice $t_{1,1} \in Ab(H)$ si $t_{1,1}$ leyó un valor actualizado de A; o bien una respuesta $t_{1,1}.ok(0)$, lo que implicaría una posible ejecución $\sigma = \{t_{1,1}*, t_{2,1}*\}$. En este caso se podrían construir las historias válidas:

$$H = \{t_{1,1}.start, t_{1,1}.ok, t_{1,1}.read(A), t_{1,1}.ok(1), t_{1,1}.read(B), t_{1,1}.ok(0),$$
$$t_{1,1}.commit, t_{1,1}.ok, t_{2,1}.start, t_{2,1}.ok, ..., t_{2,1}.commit, t_{2,1}.ok\},$$

$$H' = \{t_{1,1}.start, t_{1,1}.ok, t_{2,1}.start, t_{2,1}.ok, ..., t_{2,1}.commit, t_{2,1}.ok,$$
$$t_{1,1}.read(A), t_{1,1}.ok(2), t_{1,1}.read(B), t_{1,1}.ok(1), t_{1,1}.commit, t_{1,1}.ok, \},$$

UNIVERSIDAD DE MÁLAGA

$$H'' = \{t_{1,1}.start, t_{1,1}.ok, t_{1,1}.read(A), t_{1,1}.ok(1), t_{2,1}.start, t_{2,1}.ok, ..., \\ t_{2,1}.commit, t_{2,1}.ok, t_{1,1}.read(B), t_{1,1}.abort\}.$$

Donde H implica que $t_{1,1}$ observó los valores A y B sin ser actualizados por $t_{2,1}$, que corresponde a la ejecución $\sigma = \{t_{1,1}*, t_{2,1}*\}$; H' observó los valores A y B actualizados, que corresponde a la ejecución $\sigma' = \{t_{2,1}*, t_{1,1}*\}$; y H'' observó el valor de A sin ser actualizado y abortó al encontrar el valor de B actualizado, que corresponde a la ejecución $\sigma'' = \{t_{2,1}*\}$. En el caso de H'' la transacción $t_{1,1}$ probablemente será reejecutada más adelante. Por otra parte, la historia S:

$$S = \{t_{1,1}.start, t_{1,1}.ok, t_{1,1}.read(A), t_{1,1}.ok(1), t_{2,1}.start, t_{2,1}.ok, ..., \\ t_{2,1}.commit, t_{2,1}.ok, t_{1,1}.read(B), t_{1,1}.ok(1), t_{1,1}.commit, t_{1,1}.abort\},$$

no cumpliría el criterio de *opacity*, al no existir $t_{1,1}*$, pero sí cumpliría el criterio de *serializability*, ya que $t_{1,1} \notin Comm(S)$.

Opacity requiere que toda $t \in H$ observe en todo momento la misma vista de la ejecución, la cual debe ser además serializable. Estas restricciones proporcionan una buena garantía frente a las acciones que pueda realizar una transacción. Sin embargo el coste de soportar *opacity* es elevado, ya que requiere validar el *read-set* parcial de cada transacción en cada operación de lectura.

Virtual World Consistency (*VWC*) [58] se introduce como un criterio intermedio entre *opacity* y *serializability*. En este caso, toda $t \in Comm(H)$ debe observar en todo momento una vista única y correcta de la ejecución α del sistema, pero se admite que transacciones $t' \in Live(H)$ y $t'' \in Abort(H)$ puedan tener otra vista α' del sistema *siempre que dicha vista sea también correcta y serializable*. En el escenario de la figura 2.6, se admitiría que la invocación de $t_{1,1}.read(B)$ produjese la respuesta $t_{1,1}.ok(0)$ y una tercera transacción $H|t_{3,1} = \{t_{3,1}.start, t_{3,1}.ok, t_{3,1}.read(A), t_{3,1}.ok(2), t_{3,1}.read(B), t_{3,1}.abort\}$. En este caso, $t_{1,1} \in Live(H)$ y $t_{3,1} \in Abort(H)$ están observando distintas ejecuciones, algo no admisible en *opacity*. La transacción $t_{3,1}$ debe abortar, no obstante, ya que toda $t \in Comm(H)$ sí debe observar la misma vista de la ejecución α. Aunque permita distintas vistas de la ejecución, la restricción de que todas deban ser siempre correctas y serializables permite evitar los problemas asociados a *serializability* en TM sin incurrir en el coste computacional de soportar *opacity*. En el capítulo 3 se detallan distintas estrategias para soportar estos criterios de consistencia y su impacto en el rendimiento del sistema transaccional.

De modo similar a *strict serializability*, *strict VWC* añade a *VWC* la restricción

UNIVERSIDAD DE MÁLAGA

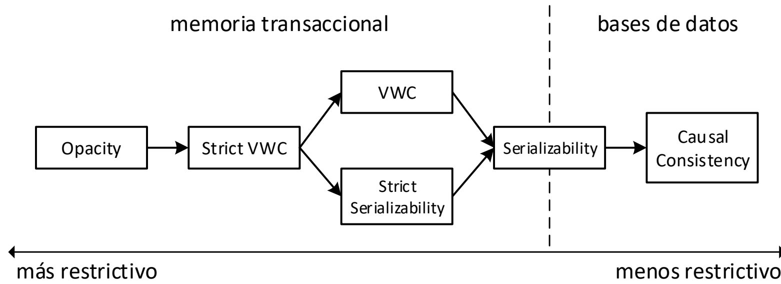


Figura 2.7: Diferentes criterios de consistencia en un sistema transaccional. Las flechas entre criterios indican implicación.

	Misma σ para $t \in Comm(H)$	$t*$ en el intervalo de ejecución de t	Existe $t*$ para $t \notin Comm(H)$	Misma σ para $t \notin Comm(H)$
Opacity	Sí	Sí	Sí	Sí
Strict VWC	Sí	Sí	Sí	No
VWC	Sí	No	Sí	No
Strict Serializability	Sí	Sí	No	No
Serializability	Sí	No	No	No
Causal Consistency	No	No	No	No

Tabla 2.1: Diferencias entre criterios de consistencia TM en base a las características que deben cumplir sus historias. σ indica una ejecución correcta para la historia H, y $t*$ indica el punto de serialización de la transacción t.

de que el punto de serialización $t*$ de cada $t \in H$ debe poder establecerse durante algún instante dentro de su intervalo de ejecución.

La figura 2.7 resume los criterios de consistencia vistos en esta sección en un grafo dirigido. La dirección de las flechas denota implicación, y se cumple la transitividad entre nodos. *Opacity*, por tanto, implica *strict serializability*. Si dos nodos no son alcanzables entre sí siguiendo las flechas, los criterios no son comparables, como pasa por ejemplo entre *VWC* y *strict serializability*. La tabla 2.1 resume las diferencias entre los distintos criterios de consistencia descritos en esta sección.

UNIVERSIDAD DE MÁLAGA

2.4.3. Transacciones anidadas

Denominamos transacción anidada a una transacción cuya ejecución está contenida en el cuerpo de otra. El soporte de transacciones anidadas requiere considerar las interacciones adicionales que pueden producirse entre una transacción anidada (*inner*) y la transacción que la anida (*outer*); y entre el resto de transacciones del sistema. Además es necesario especificar qué ocurre cuando una transacción anidada finaliza su *commit* o aborta. En los sistemas TM que soportan anidamiento se distinguen tres estrategias:

- *Flattened Nesting*: es la estrategia más sencilla; el aborto de una transacción *inner* aborta también su transacción *outer* correspondiente, mientras que el *commit* de una transacción *inner* no tiene efecto hasta que finalice la transacción *outer*. El término *flattened* (aplanadas) hace referencia a que este sistema es semánticamente idéntico a ignorar las transacciones *inner* e integrarlas en la transacción *outer* que las anida. Su mayor ventaja es la sencillez de implementación: únicamente se necesita un contador por hilo de ejecución que se incremente con cada nuevo nivel de anidamiento. De este modo los eventos de *start* y *commit* de las transacciones sólo se instrumentarán cuando el contador indique que se refieren a una transacción *outer*. Su mayor inconveniente es que el aborto de una transacción *inner* aborta también la transacción *outer* junto con el resto de transacciones anidadas que ésta pudiera contener. Esto no sólo penaliza el rendimiento de la aplicación transaccional, sino que puede afectar a la composición de varias transacciones si una transacción *inner* tiene una instrucción de aborto explícito [55].

- *Closed Nesting*: es un sistema más avanzado, en el que el aborto de una transacción *inner* no implica el aborto de la transacción *outer* que la contiene. Los cambios de las transacciones *inner* finalizadas siguen sin ser visibles al resto de hilos hasta que finalice la transacción *outer*, pero un aborto de una transacción *inner* simplemente descarta sus cambios sin afectar al resto de instrucciones de la transacción *outer*. Semánticamente el comportamiento de *closed nesting* es similar al de *flattened nesting*, siendo idéntico si todas las transacciones anidadas finalizan sin abortar.

- *Open Nesting*: es una alternativa a los dos esquemas anteriores que cambia la semántica del anidamiento. En este caso los cambios realizados por una transacción *inner* que finaliza correctamente se hacen visibles inmediatamente al resto del sistema, y el aborto en una transacción *outer* no deshace los cambios de las transacciones *inner* finalizadas. Este sistema de anida-

UNIVERSIDAD DE MÁLAGA

```
1
2 for(i=0;i<N;i++){
3   A[i] = A[i] + A[foo(i)];
4 }
5
6
```

Código secuencial

```
1 #pragma omp for
2 for(i=0;i<N;i++){
3   TmBegin(i);
4       A[i] = A[i] + A[foo(i)];
5   TmEnd;
6 }
```

Código transaccional

Figura 2.8: Ejemplo del uso de orden en transacciones. Suponemos un sistema TM implícito y ordenado, donde el parámetro de `TmBegin` indica el orden de la transacción.

miento puede violar las propiedades de atomicidad y aislamiento de las transacciones y requiere por tanto una serie de consideraciones adicionales que incluyen programar acciones compensatorias por parte de las transacciones *inner* para mantener la corrección del programa. A diferencia de los esquemas anteriores, el propósito de *open nesting* es relajar los criterios de consistencia de las transacciones para permitir ciertos conflictos que se sabe de antemano que no van a repercutir en la lógica del programa, lo que permite aumentar el rendimiento del sistema.

En esta tesis no se han considerado otros enfoques como *Parallel Nesting* [55] que parten de supuestos menos restrictivos. En los sistemas TM tanto software como hardware que se han utilizado en el transcurso de esta tesis el soporte para anidamiento de transacciones ha sido *Flattened Nesting*.

2.4.4. Restricciones de orden

La definición original de TM no considera ninguna restricción de orden; las transacciones son atómicas, pero pueden ejecutarse en cualquier orden siempre que sea serializable. Por tanto el orden de una transacción no está determinado si no se utilizan directivas de sincronización adicionales. Sin embargo, establecer ciertas restricciones de orden resulta conveniente para algunos escenarios, entre los que destaca la paralelización especulativa de códigos secuenciales [26, 98].

En la figura 2.8 podemos ver un ejemplo de este uso. A la hora de paralelizar el bucle `for` de la izquierda, no podemos suponer la ausencia de dependencias entre iteraciones (*loop-carried dependences*) debido a la indirección en el acceso al array `A` que introduce la función `foo` en la línea 3. En este caso encapsular cada iteración del bucle en una transacción no garantiza una ejecución equivalente al código secuencial.

UNIVERSIDAD DE MÁLAGA

Para paralelizar este tipo de bucles algunos autores han propuesto establecer restricciones de orden directamente en el sistema TM [109]. De este modo se permite que varias transacciones se ejecuten en paralelo restringiendo sus fases de *commit* para que éstas se lleven a cabo en el orden lógico de la aplicación. Si las transacciones no tienen conflictos entre sí este orden permite explotar el paralelismo del sistema, a la vez que mantiene la corrección de la aplicación en caso de que existan dependencias.

En la figura 2.8 (derecha) se muestra esta solución: un parámetro en la instrucción `TmBegin` permite asignar un número de orden que coincide con la iteración que se está ejecutando en cada momento. Una transacción ordenada de este modo sólo podrá finalizar si no ha tenido conflictos y el resto de transacciones con un orden menor ya han finalizado. Si el cuerpo del bucle no tenía dependencias entre iteraciones, la mayor parte del cómputo se realizará en paralelo, si bien será necesario serializar la fase de *commit* de todas las transacciones para que las modificaciones en los objetos del sistema se realicen en el orden secuencial del bucle original.

Aunque un sistema TM ordenado implica que el punto de serialización $t*$ de sus transacciones está dentro de su intervalo de ejecución, el hecho de que un sistema TM garantice un criterio de consistencia como *strict serializability*, *strict VWC* u *opacity* no implica que sea ordenado. Esto se debe a que dos transacciones $t_{1,n}$ y $t_{2,m}$ concurrentes ejecutándose en distintos hilos admiten las ejecuciones secuenciales $\sigma = \{t_{1,n}*, t_{2,m}*\}$ y $\sigma' = \{t_{2,m}*, t_{1,n}*\}$ bajo cualquiera de los criterios antes mencionados. Un sistema TM ordenado, por el contrario, prohibiría una de estas ejecuciones.

2.5. Sistemas de memoria transaccional

En esta sección se describen brevemente algunos sistemas TM de la literatura. Dada la cantidad de sistemas existentes desde la propuesta original de Herlihy y Moss se ha realizado una selección primando sistemas a la vez relevantes para esta tesis y variados en su diseño, de manera que sirvan como muestra de distintas opciones dentro del espacio de diseño descrito en la sección 2.3. Los sistemas incluidos aquí se presentan de forma sintética con el objetivo de dar una visión general su funcionamiento.

UNIVERSIDAD DE MÁLAGA

2.5.1. Sistemas TM software

Los sistemas TM software (STM) permiten implementar la funcionalidad transaccional en procesadores convencionales. Desde su introducción a mediados de la década de los 90 [102], han surgido numerosas propuestas basadas en distintas estrategias para extraer el mayor rendimiento posible.

Al no depender de soporte específico en el procesador, los STM utilizan *metadatos* para dar soporte al sistema transaccional. Estos metadatos pueden ser tanto globales como locales a cada transacción. Los últimos se suelen englobar en una estructura denominada *descriptor de transacción* que incluye, entre otros, los búferes que almacenan las escrituras tentativas en sistemas con gestión de versiones *lazy* (*redo-log*), o las escrituras anteriores en sistemas con gestión de versiones *eager* (*undo-log*) tal y como se describió en la sección 2.3.2. Además, en la mayoría de STM suele ser necesario mantener sendos conjuntos de las posiciones de memoria leídas y escritas por cada transacción para realizar la detección de conflictos (*read-set* y *write-set*). En la práctica es común que los STM combinen las estructuras de datos utilizadas para los *undo-log* y *redo-log* con las de los *read-set* y *write-set*.

Dado que los STM requieren instrumentar mediante software cada acceso transaccional para mantener la consistencia del sistema, llevan asociado un *overhead* a cada acceso transaccional que puede impedir una ganancia real de rendimiento. Algunos autores, de hecho, no consideran STM una alternativa viable [15] debido principalmente al coste derivado de la instrumentación y a la falta de garantías fuertes de consistencia con códigos no transaccionales, que normalmente vienen causadas por la necesidad de reducir la instrumentación adicional al mínimo imprescindible. Los STM analizados en esta sección, de hecho, son todos explícitos y de *aislamiento débil*. Esto se traduce en que es responsabilidad del programador —o del compilador— especificar qué accesos de memoria dentro de una transacción deben considerarse transaccionales. Estos STM sólo garantizarán la atomicidad y el aislamiento entre transacciones y, dentro de éstas, entre los accesos marcados como transaccionales. El motivo no es otro que reducir estos accesos en la medida de lo posible y con ello la instrumentación necesaria para mantener la consistencia del sistema.

Por otra parte las menores limitaciones del software permiten algoritmos más sofisticados y un prototipado más rápido, lo que ha permitido una mayor evolución en estos sistemas. Los algoritmos son más flexibles al no depender de límites rígidos determinados por las estructuras hardware. Un ejemplo de esto es la granularidad a la hora de detectar conflictos, que suele ser a nivel de palabra o incluso de objeto en STM, mientras que los HTM actuales necesitan ampliarla a

UNIVERSIDAD DE MÁLAGA

	Detección conflictos	Gestión de versiones	Método de validación	Instrumentación	Commit concurrente
TML	Eager	Eager	Flag *writer*	Mínima	No
TL-2	Eager	Lazy	Versioned-locks	Media	Sí
TinySTM	Eager	Lazy	Versioned-locks	Media	Sí
NoRec	Eager	Lazy	Basado en valor (VBV)	Baja	No
InvalSTM	Lazy	Lazy	Filtros de Bloom[1]	Media	No

[1] En este caso se refiere al método de invalidación.

Tabla 2.2: Resumen de propuestas STM analizadas en esta sección. Instrumentación hace referencia a las instrucciones adicionales que introduce el STM para preservar la consistencia del sistema. Commit concurrente indica si dos transacciones con *write-set* disjuntos pueden finalizar en paralelo.

bloques completos de caché, aumentando la posibilidad de conflictos debidos a falsos positivos. Por otra parte, el número de eventos en las transacciones puede ser virtualmente ilimitado en software, mientras que en hardware estará limitado por las estructuras utilizadas para mantener los valores transaccionales.

Los STM deben mantener un equilibrio entre el nivel de instrumentación y el número de conflictos, intentando minimizar ambos. Para un código concreto, y garantizando un determinado criterio de consistencia, existe un conjunto de historias consideradas correctas. Aunque idealmente un STM debería considerar como válidas cualquiera de estas historias, en la práctica esto no es así, ya que la necesidad de minimizar la instrumentación originará abortos innecesarios que impedirán la ejecución de un subconjunto de estas historias. Como veremos a continuación, distintos diseños de STM representan soluciones de compromiso entre estos dos objetivos. La tabla 2.2 resume las características más relevantes de las propuestas STM analizadas en esta sección.

Transactional Mutex Locks

Uno de los diseños software más sencillos es *Transactional Mutex Locks* (TML) [22]. Este STM utiliza un control de concurrencia pesimista junto a una gestión de versiones *eager*, y garantiza *opacity*[5]. El nivel de metadatos e instrumentación de TML es mínimo, no siendo necesario almacenar los valores modificados en ningún búfer. Esta sencillez, sin embargo, limita sensiblemente la concurrencia, ya que sólo permite que una transacción activa actualice la memoria. Cada tran-

[5] En puridad, y salvo que se explicite lo contrario, los TM garantizan una forma débil de *opacity* que no implica que $t*$ deba serializarse durante el intervalo de ejecución de t.

UNIVERSIDAD DE MÁLAGA

sacción comienza con un flag $t.writer = 0$, que indica que la transacción t no ha realizado ninguna escritura. Las transacciones pueden leer datos concurrentemente, pero deben promocionar a *writers* antes de poder realizar una operación de escritura. Esto se realiza mediante una operación *compare&swap* (CAS) sobre una variable global, inicialmente a cero, que indica si existe alguna transacción en modo escritura, garantizando así que sólo una transacción activa podrá realizar escrituras. El resto de las transacciones $t \in Live(H)$ deben abortar cuando esto ocurra, para lo cual comprueban la variable global antes de cada lectura y abortan si está activa. La fase de *commit* no requiere ninguna acción adicional, salvo si la transacción involucrada había promocionado a *writer*, en cuyo caso restablece la variable global para permitir la ejecución del resto de transacciones. El diseño de TML es sencillo, pero la mejora potencial respecto a un esquema que utilice *coarse-grain locks* es limitada, ya que serializa cualquier transacción que realice escrituras y éstas causan a su vez el aborto de cualquier otra transacción activa durante su intervalo de ejecución. Su escenario de aplicación depende por tanto del porcentaje de transacciones de sólo lectura que ejecute el sistema.

Transactional Locking II

La mayoría de STM, a diferencia de TML, permiten que diferentes hilos de ejecución procesen transacciones que incluyan escrituras de forma concurrente. Es necesario para ello algún algoritmo que permita determinar eficientemente si un determinado dato está en el *write-set* de alguna transacción. Uno de los métodos más populares es combinar el uso de un reloj global con *versioned-locks*. Un ejemplo de este sistema es TL-2 [28], que introdujo este algoritmo. Este STM utiliza un control de concurrencia pesimista en las operaciones de escritura y optimista en las operaciones de lectura, junto a una gestión de versiones *lazy*, y garantiza *opacity*[6].

TL-2 requiere un contador entero compartido, que sirve como reloj global; y un *redo-log* para almacenar las escrituras transaccionales y una variable *rv* (*read-version*) locales a cada hilo. Adicionalmente se necesita un *versioned-lock* por cada objeto transaccional a monitorizar, que no es más que una estructura compartida que almacena un flag para indicar el estado del *lock*, y un entero que almacena un número de versión. El reloj global es incrementado atómicamente por cada transacción al finalizar su *commit*. Las transacciones almacenan en su inicio el valor del reloj global en su variable *rv*. Desde ese momento se considera el valor de *rv* como la versión correcta de los datos de esa transacción. Antes de leer un

[6]Realmente las garantías de TL-2 se ajustan a *strict VWC*, pero la formalización de este criterio es posterior al artículo de TL-2.

UNIVERSIDAD DE MÁLAGA

dato, la transacción consulta el *redo-log* por si lo ha escrito anteriormente y puede obtener de ahí el valor. En caso contrario, deberá verificar que el *versioned-lock* asociado al dato no esté bloqueado y que su número de versión sea igual o menor que *rv* para poder continuar. Las escrituras no se realizan sobre los datos compartidos, sino que se almacenan (dirección y valor) en el *redo-log* hasta llegar a la fase de *commit*. En dicha fase se intenta adquirir el *versioned-lock* de todos los datos almacenados en el *redo-log* usando *bounded-spinning*[7], se intenta revalidar el *read-set* y sólo entonces se actualizan los datos con el contenido del *redo-lock* y se liberan los *versioned-lock* previa actualización de sus números de versión, que se establecen a *rv+1*. La transacción finaliza incrementando atómicamente el contador global. De este modo las transacciones concurrentes con la que acaba de actualizar los datos, cuyo *rv* es ahora menor que el contador global, abortarán si intentan leer o validar alguno de los datos actualizados, que tendrán un número de versión mayor.

TinySTM

La combinación de un reloj global y un sistema de *locks* se usa en otros STM. TinySTM [34] se basa en el algoritmo LSA [94], similar a TL-2, pero que introduce algunas mejoras. Por una parte permite mantener varias versiones de un mismo objeto transaccional en el sistema. De este modo, una transacción t que no contenga operaciones de escritura y que haya leído una versión n de un objeto transaccional no tiene por qué abortar si otra transacción t' actualiza ese objeto a una versión $n + 1$. Si el resto de objetos accedidos por t siguen siendo válidos (es decir, no han sido actualizados a una versión posterior a n), la transacción puede continuar y finalizar, serializándose en este caso antes de t'. Por otra parte, una transacción t que contenga escrituras y acceda a un dato actualizado por una transacción posterior t' puede intentar *extender* su validez verificando si el conjunto de objetos accedidos por t no ha sido actualizado desde el último acceso de t. Si es así, la transacción puede continuar y finalizar, serializándose en este caso después de t'.

La figura 2.9 ilustra estas dos mejoras: en ambos casos suponemos que no existen otras transacciones en el sistema y que los objetos A, B, C comienzan con una versión inicial A_0, B_0, C_0 que representa el valor inicial de estos objetos. En la situación Ⓐ, t_0 y t_1 comienzan concurrentemente (mismo reloj global). La transacción t_0 lee la versión 0 de los objetos A y B (A_0, B_0). Posteriormente,

[7]Similar a un *spin lock* pero con un límite de intentos. Si el *lock* sigue bloqueado pasado el límite, la transacción libera sus *locks* y aborta, generalmente esperando un tiempo aleatorio o creciente para prevenir que se repitan las mismas interacciones entre transacciones.

UNIVERSIDAD DE MÁLAGA

t_1 actualiza el valor de A y finaliza, incrementando el reloj global. Desde ese momento, existe una nueva versión A_1 del objeto A. Finalmente, la primera transacción vuelve a acceder a A. Al encontrarlo actualizado, TL-2 abortaría la transacción. TinySTM, sin embargo, puede comprobar que B sigue siendo válido, devolver A_0 y finalizar con éxito. En la situación Ⓑ, t_0 y t_1 comienzan concurrentemente (mismo reloj global). t_0 lee A y escribe B. Mientras tanto, t_1 actualiza C y finaliza, incrementando el reloj global. A continuación t_0 intenta acceder a C. Al encontrarlo actualizado, TL-2 abortaría t_0. TinySTM, sin embargo, puede intentar extender su validez (es decir, su versión local rv) comprobando si las versiones de sus datos accedidos (A, B) siguen siendo válidas en la etapa actual del reloj global. Si es así, t_0 puede devolver C_1 y finalizar la transacción, que se serializaría después de t_1.

Otra diferencia reseñable de este STM es el uso de una política *encounter-time locking*, donde las transacciones intentan adquirir los *versioned-locks* de sus escrituras al invocar la operación transaccional, en lugar de esperar hasta la fase de *commit*.

Al igual que TL-2, TinySTM utiliza un control de concurrencia pesimista en las operaciones de escritura y optimista en las operaciones de lectura, junto a una gestión de versiones *lazy*, y garantiza *opacity*.

NOrec

Como alternativa a los *versioned-locks* a la hora de validar, NOrec [23] utiliza *value-based validation* (VBV), un sistema que permite prescindir de los *locks* asociados a los objetos transaccionales. En este caso se require que el *read-set* almacene los valores leídos por la transacción además de su dirección. Al igual que TL-2 o TinySTM, NOrec utiliza un control de concurrencia optimista en lecturas y pesimista en escrituras, combinado con una gestión de versiones *lazy*, y garantiza *opacity*. Los metadatos incluyen un contador entero compartido, que se utiliza como reloj global; y un entero por transacción a modo de reloj local, que almacena el valor del reloj global al inicio de la transacción.

El funcionamiento del reloj global en NOrec es diferente a los STM anteriores: en este caso, un valor impar indica que una transacción está en su fase de *commit*. Las transacciones listas para finalizar intentan incrementar el reloj atómicamente mediante una operación CAS, impidiendo que otra transacción pueda finalizar o comenzar y serializando por tanto la fase de *commit* de cualquier transacción que contenga escrituras. Cualquier otra transacción activa debe confirmar en cada operación de lectura que el valor del reloj global sea par, que éste valor coincida con el reloj local que leyó en su inicio y que no varíe durante la operación,

UNIVERSIDAD DE MÁLAGA

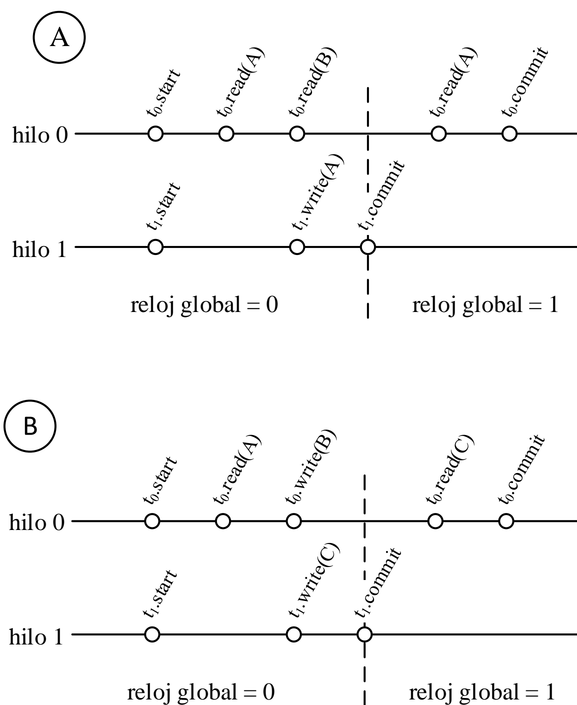


Figura 2.9: Mejoras de TinySTM respecto a TL-2. En este diagrama t_0 y t_1 representan los eventos de dos transacciones ejecutadas en hilos diferentes. En cada línea temporal, los círculos corresponden a distintos eventos de la transacción asociada (en este caso sólo se muestran las invocaciones). La línea discontinua indica el instante en que se incrementa el reloj global, tras el *commit* de alguna transacción. A, B, C hacen referencia a objetos compartidos.

UNIVERSIDAD DE MÁLAGA

asegurándose así que los datos de su *read-set* siguen siendo válidos. Si alguna de estas condiciones no se cumple, debe esperar a que el reloj global vuelva a ser par (es decir, a que el estado de la memoria compartida sea consistente), y revalidar su *read-set*, que en VBV implica verificar que todos los pares dirección-valor almacenados por la transacción coincidan con el valor que existe en ese momento en memoria. Si es así, la transacción actualiza su reloj local a la versión global actual; en caso contrario, la transacción debe abortar. Por su parte, las transacciones que sólo realicen lecturas, pueden finalizar sin incrementar el reloj global.

NOrec combina por tanto dos ideas clave: un contador global que se utiliza a modo de *lock* de secuencia [65] y el uso de VBV en lugar de *versioned-locks*. Su mayor ventaja es la baja instrumentación que introduce en escenarios con pocas escrituras y pocos conflictos. Sus principales limitaciones son la serialización de la fase de *commit* de las transacciones y la mayor demanda de espacio requerida por la necesidad de almacenar el valor de las posiciones leídas para poder realizar la validación.

InvalSTM

Un enfoque diferente a las propuestas anteriores es utilizar un algoritmo de *invalidación*. En lugar de que cada transacción intente validar sus datos cada vez que otra transacción finalice, es ésta última la encargada de invalidar a las transacciones activas con las que pueda existir algún conflicto. InvalSTM [45] implementa esta idea de forma eficiente. Sus autores destacan dos ventajas de usar un protocolo de invalidación: por una parte, al realizarse ésta antes del *commit* de una transacción, hay más información disponible para que el gestor de conflictos pueda tomar la mejor decisión posible ante un conflicto. Una segunda ventaja es que es menos costoso computacionalmente garantizar *opacity*, ya que las transacciones $t \in Live(H)$ sólo deben comprobar si un *flag* local de invalidación ha sido activado antes de cada lectura, en lugar de tener que revalidar su *read-set* completo. InvalSTM utiliza un control de concurrencia optimista en lecturas y escrituras, combinado con una gestión de versiones *lazy*, y garantiza *opacity*.

En lugar de usar *versioned-locks* para proteger las distintas posiciones de memoria, este STM utiliza un *lock* por transacción y un par de *locks* globales (*commit lock* e *in-flight lock*) para garantizar que sólo una transacción pueda realizar su *commit* de forma concurrente y que sólo una transacción pueda bloquear los *locks* asociados al resto de transacciones. Adicionalmente, cada transacción utiliza dos

UNIVERSIDAD DE MÁLAGA

filtros de Bloom[8] para comprimir sus *read-set* y *write-set*; y un *flag* de validez, que indica si la transacción debe o no abortar. Las transacciones deben comprobar este *flag* en cada operación transaccional, abortando si deja de estar activo. Al llegar a su fase de *commit*, las transacciones intentan adquirir el *commit lock*, garantizando su acceso exclusivo. Durante esta fase se impide que el resto de transacciones puedan realizar lecturas o escrituras adquiriendo sus *locks* asociados, y se compara el conjunto de escritura de la transacción en fase de *commit* con los conjuntos de lectura y escritura del resto de transacciones. Si existen conflictos, InvalSTM puede invalidar la transacción activa implicada, abortar la transacción en fase de *commit* o bloquearla temporalmente.

El algoritmo de invalidación de InvalSTM tiene como principal ventaja la información adicional con la que cuenta la transacción encargada de invalidar, lo que permite minimizar el número de abortos del sistema: InvalSTM puede por ejemplo abortar la transacción en fase de *commit* si con ello evita una serie de abortos en cascada de otras transacciones activas. La principal desventaja de este STM es la necesidad de serializar la fase de *commit* unida a que, debido a su diseño, dicha fase es más larga que en otros STM debido a la necesidad de realizar las invalidaciones.

2.5.2. Sistemas TM hardware

Los sistemas TM hardware (HTM) implementan la mayor parte de la funcionalidad transaccional directamente en el procesador. Esto conlleva una serie de diferencias respecto a los STM descritos en la sección anterior. En general, y desde el punto de vista del programador, los HTM se caracterizan por ofrecer mayores garantías de corrección y un menor *overhead*. En contraparte, son menos flexibles debido a las limitaciones impuestas por el hardware.

Todos los HTM analizados en esta tesis proporcionan *aislamiento fuerte*, lo que permite detectar posibles interacciones entre código transaccional y no transaccional. Además —y esto es una diferencia respecto a los sistemas STM— la gran mayoría de HTM son *implícitos*: todos los accesos a memoria por parte de una transacción se consideran transaccionales. La combinación de estas características resulta muy útil de cara a proporcionar compatibilidad con programas y librerías ya existentes sin necesidad de realizar cambios en el código.

El diseño de los sistemas HTM analizados se basa en modificaciones en el sistema de coherencia de caché junto con estructuras hardware y lógica adicio-

[8]Estructura probabilística que permite verificar la pertenencia de un elemento a un conjunto de forma eficiente [8]. Se describirá con más detalle en la sección 3.3.3.

UNIVERSIDAD DE MÁLAGA

nal para mantener la semántica transaccional. Como se verá más adelante, las propuestas teóricas introducen cambios más disruptivos, mientras que las implementaciones reales reutilizan gran parte de sus diseños, intentando alterar la arquitectura existente lo menos posible. En cualquier caso, la decisión de utilizar el sistema de cachés para implementar la funcionalidad transaccional, si bien resulta conveniente dado que se pueden aprovechar protocolos y estructuras existentes, introduce un par de limitaciones que no están presentes en STM. Por una parte, la granularidad a la hora de detectar conflictos es a nivel de bloque de caché, algo que puede ocasionar conflictos espurios cuando dos transacciones acceden a datos diferentes, pero alojados en un mismo bloque de caché (*false sharing*). Por otra parte, existe un límite físico a la hora de almacenar los datos tentativos accedidos por cada transacción. Si se sobrepasa dicho límite la transacción tendrá que abortar o controlar su conjunto de datos de alguna forma. Estos dos factores provocan que los sistemas HTM funcionen especialmente bien cuando las transacciones son cortas y acceden a objetos con poca contención. La mayoría de HTM analizados, de hecho, no garantizan que una transacción hardware finalice (por ejemplo, si dicha transacción accede a más datos de lo que la estructura hardware es capaz de almacenar o si contiene operaciones irrevocables, interrupciones o llamadas al sistema). En este caso hablamos de *best-effort* HTM, y el programador es responsable de implementar una solución que tenga en cuenta esta limitación, que se suele denominar código *fallback*.

A continuación se describe brevemente una selección de diseños HTM. Dado el enfoque de esta tesis, nos hemos centrado en diseños HTM sobre procesadores reales que se utilizarán como base en el capítulo 4. Sin embargo, se mencionan además algunas propuestas teóricas relevantes en la literatura para ofrecer una perspectiva más general. La tabla 2.3 presenta un resumen de los sistemas incluidos en esta sección.

Herlihy HTM

La primera idea de diseño HTM la proporcionan M. Herlihy y otros en el mismo trabajo que introduce el paradigma de memoria transaccional [56]. Este HTM combina una gestión de versiones *lazy* con una detección de conflictos *eager* y sirvió de base para diseños posteriores. Se introducen una serie de instrucciones transaccionales para comenzar, abortar y terminar transacciones, acceder a datos transaccionales y validar la transacción. Para soportarlas, se propone un diseño que aprovecha el sistema de cachés bajo la premisa de que el mismo protocolo de coherencia que se encarga de detectar conflictos de datos puede detectar conflictos entre transacciones sin coste adicional.

UNIVERSIDAD DE MÁLAGA

En este HTM las operaciones no transaccionales utilizan las mismas cachés y protocolos que en un procesador convencional, pero se añade una *caché transaccional* totalmente asociativa a cada núcleo que opera de forma exclusiva con la caché L1 (un bloque puede estar en la caché transaccional o en la caché L1, pero no en ambas). Un par de *flags* en cada núcleo indican si está ejecutando o no una transacción y el estado de la misma. Durante una transacción, la caché transaccional almacena las escrituras que realice la misma sin propagarlas al resto de procesadores o a la memoria principal. Si la transacción finaliza con éxito se permite al resto de procesadores acceder a los datos. Si la transacción aborta simplemente se invalidan las líneas que contenían los datos transaccionales. Una serie de etiquetas asociadas a los bloques de la caché transaccional permiten distinguir si el bloque está libre, contiene datos transaccionales o contiene datos ya confirmados. El diseño propuesto asume que las transacciones serán pequeñas y accederán a conjuntos de datos reducidos. Esta propuesta no proporciona garantías de progreso, por lo que se puede catalogar como *best-effort*.

Virtualized Transactional Memory

Con el objetivo de proporcionar mayores garantías al usuario, Virtualized Transactional Memory (VTM) [90] presenta un diseño capaz de controlar de forma transparente al programador las transacciones que exceden las capacidades hardware del procesador, garantizando así que las transacciones acabarán finalizando independientemente de su tamaño. La propuesta se centra en controlar los posibles fallos por exceso de datos transaccionales y por exceso de tiempo (que ocasionaría una interrupción del sistema operativo).

VTM desacopla las transacciones del hardware evitando enlentecer el caso común (es decir, las transacciones que se podrían ejecutar sin problemas en un HTM). Para ello utiliza dos modos de operación: el primero es similar a otros HTM y está pensado para mayoría de transacciones que no son interrumpidas ni exceden los límites de capacidad del hardware donde se ejecutan. Un segundo modo se encarga de controlar posibles problemas como fallos de página, exceso de accesos transaccionales o migración de hilos de ejecución. Cada transacción tiene una variable de estado que mantiene el modo activo en cada momento. Una tabla almacenada en memoria virtual, transparente al usuario y compartida por todas las transacciones (*Transaction Address Data Table* o XADT) almacena los accesos tentativos de las transacciones que sobrepasan los recursos hardware. Las transacciones que sufran un fallo de caché al intentar acceder a un dato transaccional deben verificar posibles conflictos con las entradas de la XADT. Para acelerar esta verificación VTM utiliza filtros de Bloom [8] para acelerar las

UNIVERSIDAD DE MÁLAGA

consultas. Además mantiene un contador de entradas almacenado localmente por cada transacción de modo que sólo se comprobará la XADT cuando esta tabla contenga entradas a verificar. En la mayoría de situaciones la XADT estará vacía y no será necesaria ninguna comprobación. Al igual que en la propuesta de Herlihy, VTM utiliza detección de conflictos *eager* y gestión de versiones *lazy*.

LogTM

LogTM [76] utiliza también detección de conflictos *eager* pero añade una gestión de versiones *eager* con el objetivo de acelerar el caso común; esto es, las transacciones que finalizan sin conflictos. Las escrituras transaccionales modifican directamente los datos en memoria y un *log* compartido en memoria virtual con punteros hardware independientes para cada núcleo almacena los valores anteriores para poder restaurarlos en caso de aborto.

LogTM extiende un protocolo MOESI de coherencia de caché añadiendo bits de lectura y escritura transaccional asociados a los bloques caché y estados adicionales para detectar conflictos entre distintas transacciones siguiendo una política *requester-loses* —la transacción que detecta un conflicto al acceder a un dato es la encargada de resolverlo, ya sea esperando o abortando si detecta una situación de *deadlock*—. Si la transacción puede finalizar sin abortos, la fase de *commit* simplemente descarta el *log* asociado a la transacción y limpia los bits transaccionales de los bloques caché utilizados. Un aborto, sin embargo, requiere de una rutina software que restaure los valores del *log*, por lo que resulta mucho más costoso.

Transactional Coherence and Consistency

Como ejemplo de detección de conflictos *lazy* y gestión de versiones *lazy*, Transactional Coherence and Consistency (TCC) [52] es un HTM que propone utilizar transacciones como unidad básica de comunicación paralela. TCC sustituye los modelos de consistencia de memoria de los CMP por un modelo de consistencia entre transacciones completas, permitiendo simplificar los protocolos de coherencia de caché.

La información de los bloques caché accedidos por las transacciones se mantiene mediante un par de bits de lectura y escritura por bloque caché, que pueden ampliarse para aumentar la precisión a la hora de detectar conflictos (por ejemplo, para realizar las detecciones a nivel de palabra). Cuando una transacción finaliza, un búfer de escritura separado de la caché empaqueta todas las escrituras de la transacción y las difunde al resto del sistema mediante *broadcast*. TCC añade es-

UNIVERSIDAD DE MÁLAGA

tructuras hardware para arbitrar la fase de *commit* de las transacciones habilitada por estos *broadcasts* y permite establecer diferentes fases entre transacciones para garantizar que una transacción con fase n no pueda finalizar hasta que cualquier transacción con fase $k < n$ haya terminado. El uso de diferentes fases permite establecer restricciones de orden entre transacciones, lo que puede resultar útil cuando existen dependencias de datos entre las mismas. TCC propone además una estructura de *double-buffering* que permite a cada núcleo de ejecución seguir ejecutando una segunda transacción manteniendo otra en espera. Esto resulta útil para no dejar el núcleo inactivo ante el eventual bloqueo de alguna transacción.

Intel Transactional Syncronization Extensions

Intel añadió soporte transaccional a sus procesadores de consumo mediante *Transactional Synchronization Extensions* (TSX) a partir de la microarquitectura Haswell [51]. Aunque los detalles arquitecturales no han sido publicados, se trata de un HTM con detección de conflictos *eager* y gestión de versiones *lazy*, y utiliza una política de resolución de conflictos *requester-wins*, donde la transacción que produce un conflicto causa el aborto de la otra transacción implicada [12].

Intel utiliza la caché L1 para almacenar los valores tentativos de los accesos transaccionales y detecta conflictos con granularidad de bloque caché (64 Bytes). Si un bloque que contenía alguna escritura transaccional es desalojado de L1, la transacción abortará. Algunos autores han mostrado experimentalmente que este diseño limita el tamaño del *write-set* a unos 22 KB en Haswell [80]. En el caso de que se desalojen de L1 bloques que contengan lecturas transaccionales, es posible seguir ejecutando la transacción gracias a lógica de control adicional, probablemente en las cachés de niveles inferiores, o a una estructura hardware adicional, que en la práctica permite que el *read-set* pueda ampliarse hasta unos 4 MB. El uso de *simultaneous multithreading* (SMT) reduce estos límites significativamente, ya que la caché se reparte entre los procesadores lógicos del mismo núcleo físico. Adicionalmente existen ciertas restricciones respecto a las instrucciones permitidas en el cuerpo de una transacción: instrucciones que operen con x87 (ciertas operaciones en coma flotante) o MMX (soporte SIMD), instrucciones de entrada/salida o llamadas al sistema abortarán la transacción. Otros eventos como interrupciones también abortan la transacción, lo que implica que, en el mejor de los casos, la vida de la misma estará limitada por las interrupciones del planificador del sistema operativo. El anidamiento de transacciones está soportado vía *flattening* hasta un máximo de ocho niveles. En cuanto a rendimiento, Intel informa de latencias adicionales a la hora de iniciar y finalizar las transacciones, y de un *overhead* variable dependiente del número de accesos a

UNIVERSIDAD DE MÁLAGA

memoria que se realicen dentro de la transacción, probablemente debido a las operaciones adicionales que se deben llevar a cabo para la detección de conflictos y gestión de versiones. Estas penalizaciones se enmascaran parcialmente por las optimizaciones *out-of-order* de la arquitectura, y se espera que sean reducidas en microarquitecturas más recientes [19].

Intel proporciona dos interfaces para trabajar con TM: *Hardware Lock Elision* (HLE) permite ejecutar secciones críticas protegidas con *locks* de forma especulativa. Para ello se añaden a la ISA un par de instrucciones `xacquire` y `xrelease` que preceden a la adquisición y liberación de un *lock* tradicional y ejecutan las instrucciones protegidas por el mismo sin bloquearlo, encapsulando las instrucciones protegidas dentro de una transacción. Una ejecución exitosa permite ejecutar la sección crítica sin necesidad de usar directivas de sincronización, resultando mucho más eficiente. Si se detecta un conflicto de datos o algún otro hilo intenta adquirir el *lock* de manera convencional, los cambios tentativos son descartados y la ejecución se realiza de forma no transaccional. Esta interfaz está pensada para trabajar con código *legacy*, ya que proporciona compatibilidad con procesadores sin soporte transaccional, que ignoran las nuevas instrucciones y utilizan el *lock* de forma tradicional. Por su parte, *Restricted Transactional Memory* (RTM) añade soporte a *best-effort* HTM y proporciona un par de instrucciones `xbegin`, `xend` que permiten delimitar transacciones. En este caso el código no es compatible con procesadores que no soporten TSX. A diferencia de HLE, una transacción ejecutada usando RTM puede reiniciarse indefinidamente, por lo que el programador es responsable de proporcionar un código alternativo no transaccional y con garantías de progreso y utilizarlo cuando sea necesario. Intel proporciona una serie de códigos de error para determinar la causa de los abortos que pueden utilizarse como apoyo a la hora de determinar si una transacción abortada debe ser reejecutada o no. En ambos casos el HTM es implícito, por lo que cualquier acceso dentro de la transacción será susceptible a conflictos.

IBM Blue Gene/Q

El supercomputador Blue Gene/Q (BG/Q) implementa uno de los primeros diseños HTM sobre un sistema real [110]. Este HTM utiliza una gestión de versiones *lazy*, una detección de conflictos *eager* con granularidad de bloque de caché y soporta anidamiento de transacciones mediante *flattening*. Aunque este HTM es *best-effort*, un *runtime software* se encarga de ejecutar de modo irrevocable las transacciones que no se pueden ejecutar por el sistema transaccional, evitando que el programador tenga que proporcionar un código alternativo.

La CPU de BG/Q se compone de 16 núcleos cada uno de los cuales puede

UNIVERSIDAD DE MÁLAGA

ejecutar 4 hilos SMT. Cada núcleo tiene acceso a una caché L1 privada de 16 KB y a un búfer de escritura de 2 KB. Adicionalmente, todos los núcleos comparten una caché multiversión L2 de 32 MB. BG/Q implementa la mayoría del mecanismo HTM en su caché L2: cada escritura transaccional produce un nuevo valor que se almacena junto con el dato no transaccional en dos versiones que se almacenan en sendas vías de la L2. Estos bloques son privatizados por el sistema para que sólo sean accesibles por el hilo que contiene la transacción hasta que ésta finalice. Una serie de etiquetas asociadas a los bloques permiten anotar las lecturas y escrituras transaccionales con un identificador vinculado al hilo que está ejecutando la transacción correspondiente. El manejo de estos identificadores y la utilización de dos vías para almacenar los valores reales y tentativos de las escrituras permiten detectar conflictos entre transacciones y accesos no transaccionales.

Dado que BG/Q no añade lógica transaccional a nivel de caché L1 existen dos modos de ejecución para las transacciones que siguen distintas estrategias para interactuar con la misma en diferentes escenarios. El primer modo se denomina *short-running mode* y evita por completo el uso de L1: cuando se produce una escritura transaccional, la línea correspondiente de L1 es desalojada, por lo que cualquier lectura posterior debe obtener el dato de L2, que lo almacenará en un registro, pero no en L1. Por su parte, cualquier lectura transaccional que esté en L1 necesita notificar a L2 al contener ésta los metadatos necesarios para la detección de conflictos. Un segundo modo denominado *long-running mode* permite que los accesos especulativos puedan mantenerse en la caché L1, que puede almacenar cuatro versiones transaccionales diferentes de una línea (dando soporte a los cuatro procesadores lógicos de cada núcleo) y una versión no transaccional. Para ello, la unidad de control de memoria (MMU) crea alias de las direcciones de memoria físicas en el *Translation Lookaside Buffer* (TLB). Si una línea transaccional se desaloja de L1 estos alias son revertidos ya que son innecesarios en la caché L2 al ser multiversión. Este modo requiere invalidar todos los bloques de la caché L1 cada vez que se inicia una transacción con el objetivo de producir fallos de caché en los accesos transaccionales, forzando así el acceso a L2, que contiene los metadatos necesarios para mantener la coherencia del sistema transaccional. El primer modo resulta apropiado para transacciones pequeñas con pocos accesos de memoria, ya que no requiere invalidar la caché L1, pero no permite aprovechar esta caché para acelerar los accesos a memoria. El segundo modo permite aprovechar la caché L1, pero la invalida en cada inicio de transacción, por lo que resulta más apropiado para transacciones más largas, con un mayor número de accesos transaccionales que puedan aprovechar la localidad espacial y temporal. BG/Q inicia las transacciones en *long-running mode*.

El rendimiento de este HTM es modesto en las pruebas realizadas [111]. En

UNIVERSIDAD DE MÁLAGA

particular existe una penalización de rendimiento notable debido al aumento de fallos de caché, especialmente en la caché L1, debido al diseño del HTM. Por otra parte el inicio y la finalización de transacciones requieren una serie de operaciones adicionales que incluyen el guardado de los registros del núcleo correspondiente vía software. Si bien la penalización de rendimiento es menor que en un STM, sigue siendo significativa en comparación a propuestas hardware posteriores.

IBM System Z

IBM introdujo HTM en su familia de *mainframes* System Z con el modelo zEC12 [62]. Este HTM utiliza una gestión de versiones *lazy*, una detección de conflictos *eager* con granularidad de bloque de caché y soporta anidamiento de transacciones mediante *flattening*.

La CPU de System Z se compone de 6 núcleos, cada uno de los cuales accede a una caché L1 privada de 96 KB y a una caché L2 también privada de 1 MB, ambas *write-through*. Los núcleos comparten además una caché L3 de 48 MB y están conectados a una caché externa L4 de 384 MB. Esta CPU implementa la lógica transaccional añadiendo un par de etiquetas a los bloques de la caché L1 para notificar si el bloque contiene lecturas o escrituras transaccionales. La unidad de memoria se extiende con lógica adicional para la detección de conflictos y con una estructura que permite mantener bloques transaccionales de lectura en caché L2. Un búfer intermedio entre L1 y L2 permite almacenar bloques transaccionales de escritura si son desalojados de L1 sin abortar la transacción [59].

La política de resolución de conflictos es *requester-loses*: la unidad de memoria de la CPU se encarga de abortar la transacción si detecta un conflicto con un nuevo acceso transaccional. Los accesos de escritura transaccionales se almacenan en una cola de escritura con una etiqueta transaccional y se transfieren a la caché L1 en la etapa de *writeback*. Cada transacción puede acceder a sus escrituras tentativas desde cualquiera de las dos estructuras. Si se produce un aborto, todos los bloques de caché L1 marcados como escrituras transaccionales son invalidados para poder obtener los datos anteriores de la caché L2. En caso de *commit* se limpian las etiquetas transaccionales de la caché L1, validando así los datos tentativos. Dado que la lógica de detección de conflictos se encuentra en la caché L1, no se permite que las escrituras transaccionales accedan a las cachés inferiores hasta que la transacción finalice con éxito. Para no limitar los accesos de escritura a la caché L1, un búfer circular de 64 entradas denominado *store gathering cache* permite almacenar bloques transaccionales desalojados de L1. Los accesos de lectura transaccionales pueden almacenarse en L2 gracias a

UNIVERSIDAD DE MÁLAGA

una extensión en el sistema de memoria que almacena en un vector de bits qué bloques de L1 estaban marcados como transaccionales. Esto permite mantener la coherencia de las lecturas transaccionales en L2 de manera imprecisa: una operación no transaccional sobre un bloque de L2 diferente producir un aborto de la transacción si existía un alias.

System Z presenta un par de características novedosas respecto a los HTM analizados hasta ahora. Aunque sigue siendo *best-effort*, permite ejecutar *constrained transactions* con garantía de finalización. Estas transacciones no necesitan una rutina *fallback software*, pero tienen una serie de restricciones que incluyen ejecutar un máximo de 32 instrucciones, no contener bucles o llamadas a función, acceder a un máximo de 32 Bytes de memoria y ejecutar sólo un subconjunto de instrucciones de la ISA en el cuerpo de la transacción. Por otra parte, se incluye una instrucción `ntstg` que permite realizar una escritura no transaccional en el cuerpo de una transacción: los datos escritos con `ntstg` sobreviven a un eventual aborto, siendo útiles para depuración.

IBM POWER8

IBM introdujo HTM en su línea de procesadores POWER8 en 2015 [66]. Este HTM utiliza gestión de versiones *lazy* y gestión de conflictos *eager*. La detección de conflictos se realiza con granularidad de bloque caché (128 Bytes) y soporta anidamiento de transacciones de hasta 62 niveles mediante *flattening*.

La CPU de POWER8 se compone de 12 núcleos, cada uno de los cuales puede ejecutar 8 hilos SMT. Cada núcleo tiene acceso a una caché L1 privada de 64 KB y a una caché L2 privada de 512 KB con comunicación *write-through* entre ambas. Todos los núcleos comparten una caché víctima L3 de 8 MB de forma semi privada (cada núcleo utiliza una partición de la caché, pudiendo aceptar datos desalojados de particiones colindantes). La política de resolución de conflictos es mixta: si una transacción realiza una lectura a un bloque que se encuentre en el *write-set* de otra, aborta (*requester-loses*). Si una transacción realiza una escritura a un bloque que se encuentre en el *read-set* o *write-set* de otra transacción, es ésta última la que aborta (*requester-wins*).

El diseño *write-through* entre las cachés L1 y L2 hace que la mayoría de la lógica transaccional esté implementada a nivel de caché L2. Para ello se añade una memoria totalmente asociativa de 64 entradas (L2 TMCAM) que almacenará los metadatos transaccionales necesarios para mantener la coherencia del HTM sin modificar sustancialmente el diseño de la caché. Las escrituras tentativas almacenarán los nuevos datos en la caché L2, desplazando los datos anteriores

UNIVERSIDAD DE MÁLAGA

a L3 —si otro hilo del mismo núcleo tiene el dato actualizado— o a memoria principal; y se marcará el bloque como transaccional en la L2 TMCAM. Del mismo modo, las lecturas transaccionales almacenarán metadatos en L2 TMCAM para que el bloque correspondiente pueda ser monitorizado por el HTM. La caché L3 contiene una estructura similar (L3 TMCAM) para controlar qué bloques corresponden a datos transaccionales guardados desde L2 e impedir que se hagan visibles al resto de núcleos. La caché L1, por su parte, incluye lógica adicional para guardar el valor de los registros antes de iniciar una transacción, y añade una serie de etiquetas al directorio de la caché para controlar qué hilo SMT del núcleo tiene la propiedad de cada bloque y controlar así las escrituras transaccionales entre hilos del mismo núcleo.

Debido a este diseño, la capacidad máxima de datos transaccionales del HTM viene limitada por el tamaño de L2 TMCAM. Esta memoria puede almacenar metadatos transaccionales para un máximo de 64 bloques de 128 Bytes, que resulta en una capacidad máxima de 8 KB de datos transaccionales. El uso de SMT divide esta capacidad entre los hilos utilizados, ya que comparten esta estructura, que contiene 8 bits adicionales para controlar qué hilo SMT tiene la propiedad de cada bloque.

POWER8 añade dos características interesantes respecto al resto de sistemas analizados. Por una parte incluye un modelo limitado de soporte transaccional por medio de *Rollback-Only Transactions* (ROT). Los accesos de lectura de las transacciones iniciadas en este modo no son monitorizados por el sistema, ni garantizan por tanto atomicidad. Las escrituras sí son almacenadas tentativamente por el HTM hasta que la transacción finalice. El programador será por tanto el encargado de detectar posibles conflictos y abortar explícitamente estas transacciones. Las ROT están enfocadas a dar soporte a técnicas de especulación monohilo [38]. Por otra parte, este HTM permite la ejecución de código no transaccional dentro de una transacción por medio del denominado *suspended mode*: un par de instrucciones `tsuspend/tresume` habilitan y deshabilitan este modo dentro de una transacción. Las lecturas y escrituras que se realicen en modo *suspended* son no transaccionales a todos los efectos, pudiendo causar abortos con la misma transacción que ha activado este modo. Además, una lectura en este modo de un dato que haya sido escrito tentativamente por la transacción devolverá el valor tentativo, algo que resulta útil para depuración. Si una transacción aborta estando en modo *suspended*, el aborto no se hace efectivo hasta que vuelva a modo transaccional. El modo *suspended* puede utilizarse también para establecer cierta comunicación entre transacciones sin producir abortos, a diferencia del resto de HTM analizados. En esta tesis se hará uso de esta característica para desarrollar el trabajo expuesto en el capítulo 4.

UNIVERSIDAD DE MÁLAGA

HTM	Diseño	Detección de conflictos	Gestión de versiones	Granularidad	Acceso	Capacidad	Características adicionales
Herlihy HTM	Teórico	Eager	Lazy	Bloque[1]	Explícito	Caché L1	Primera propuesta de HTM.
VTM	Teórico	Eager	Lazy	Bloque[1]	Implícito	Ilimitado[3]	Virtualización de transacciones para evitar abortos.
LogTM	Teórico	Eager	Eager	Bloque[1]	Implícito	Ilimitado[4]	Fase de rollback implementada en software.
TCC	Teórico	Lazy	Lazy	Bloque[1]	Implícito	Limitado[5]	Permite restricciones de orden. Propuesta para sustituir el modelo de memoria por transacciones.
Intel TSX	Real	Eager	Lazy	64 Bytes	Implícito	22 KB wset 4 MB rset	Interfaz HLE para lock elision con compatibilidad a códigos sin soporte transaccional.
IBM BG/Q	Real	Eager	Lazy	8 – 64 Bytes[2]	Implícito	16 KB wset 20 MB rset	Runtime adicional que implementa automáticamente el fallback transaccional.
IBM System Z	Real	Eager	Lazy	256 Bytes	Implícito	8 KB wset L2[6] rset	Restricted transactions con garantía de finalización. Escrituras no transaccionales.
IBM POWER8	Real	Eager	Lazy	128 Bytes	Implícito	8 KB wset+rset	Suspended mode para ejecutar código no transaccional. Rollback-Only Transactions para especulación.

[1] Al ser un diseño teórico, no se especifica ningún tamaño específico de bloque caché.
[2] La granularidad es de 8 Bytes si se cumplen una serie de condiciones, por defecto es de 64 Bytes.
[3] Limitado por la memoria virtual.
[4] Limitado por el tamaño del log.
[5] Limitado por las estructuras hardware. Los autores proponen dividir transacciones mayores.
[6] Detección de conflictos imprecisa. Pueden producirse abortos adicionales si el rset llega a caché L2.

Tabla 2.3: Resumen de propuestas HTM analizadas en esta sección.

UNIVERSIDAD DE MÁLAGA

2.5.3. Sistemas TM híbridos

Los sistemas HTM reales descritos anteriormente son todos *best-effort*, ya que el sistema no ofrece garantías de que una transacción hardware finalice en algún momento. Desde un punto de vista de diseño hardware, esta limitación tiene sentido si en la práctica la mayoría de las transacciones consiguen ejecutarse con éxito, ya que el HTM se encarga de ejecutar rápido el caso común y deja en manos del programador la resolución de transacciones que implicarían un diseño hardware más complejo. Un sistema *best-effort* requiere por tanto que el programador defina un código *fallback* alternativo para las transacciones hardware que no consigan finalizar. En este sentido, tanto Intel como IBM recomiendan el uso de un *lock* global para automatizar este código *fallback*: Las transacciones hardware leen el estado del *lock* para confirmar que está libre y lo añaden a su *read-set*. Si una transacción falla demasiadas veces, en lugar de reintentar su ejecución puede optar por adquirir ese *lock* y ejecutar el cuerpo de la transacción de forma irrevocable. Al adquirir el *lock* de forma no transaccional, cualquier transacción hardware que se esté ejecutando concurrentemente debe abortar al detectar un conflicto y no podrá reejecutarse hasta que el hilo que adquirió el *lock* lo libere después de ejecutar la sección crítica. Este enfoque es sencillo de implementar pero tiene dos inconvenientes: un hilo que esté ejecutando el código *fallback* impide que cualquier transacción hardware pueda ejecutarse aunque no existan conflictos de datos con ellas. Además, el uso de un *lock* global de grano grueso serializa todos los *fallbacks* de transacciones que haya que ejecutar, impidiendo cualquier ganancia de rendimiento.

Ante la posibilidad de optimizar este *fallback*, algunos trabajos han propuesto sistemas TM híbridos, que combinan una ejecución HTM para la mayoría de transacciones con un STM para el resto. Un aspecto clave a la hora de combinar ambos sistemas es la detección y resolución de los conflictos que puedan producirse entre transacciones controladas por el HTM y transacciones controladas por el STM. En esta sección se describen brevemente algunas de las propuestas de la literatura y sus limitaciones.

HyTM

La propuesta que introduce los sistemas híbridos es HyTM [24], que se propone como solución al problema de proporcionar garantías transaccionales para el programador independientemente del soporte HTM que pueda tener su sistema. HyTM está pensado para soportar un HTM *best-effort* con garantías similares a los diseños descritos en la sección 2.5.2. Para permitir la interacción entre

UNIVERSIDAD DE MÁLAGA

transacciones hardware y software, se introduce código adicional a las primeras para asegurarse de que no exista un conflicto con una transacción software. Si se detecta un conflicto, la transacción hardware abortará, pudiéndose reejecutar en hardware o en software.

HyTM propone el uso de un STM basado en *ownership records*[9] (*orec*) para las transacciones software; una tabla compartida de *locks* asociados a direcciones de memoria mediante una función *hash* almacena qué transacciones han realizado accesos de lectura o escritura a la memoria compartida. Para permitir ejecutar concurrentemente transacciones hardware, éstas se instrumentan para que accedan al *orec* asociado a cada acceso transaccional y verifiquen que está libre. De este modo, si una transacción software modifica uno de estos *locks* causa inmediatamente el aborto de cualquier transacción hardware que lo tenga en su *read-set*.

Este diseño es sencillo de implementar, pero la instrumentación de las transacciones hardware supone una penalización notable en el rendimiento del caso común, ya que esencialmente está instrumentando cada acceso transaccional. Además, la necesidades de interacción entre las transacciones hardware y software impone ciertas restricciones al STM utilizado como *fallback* que lo hacen poco competitivo respecto a otros diseños.

Phased Transactional Memory

Una alternativa al enfoque anterior la encontramos en Phased Transactional Memory (PhTM) [68]. La idea de este sistema es soportar varios modos de funcionamiento que se adapten a diversos escenarios de ejecución y combinarlos mediante un *runtime* que permita alternar entre ellos de forma transparente al programador. PhTM contempla un modo hardware para escenarios donde la mayoría de transacciones pueden ejecutarse por un HTM *best-effort* sin necesidad de abortar, un modo software para ejecutar las transacciones en un STM en escenarios donde las transacciones hardware no son efectivas y una serie de modos mixtos que incluyen una ejecución en HTM con *fallback* transaccional similar a HyTM o un modo puramente secuencial.

La implementación de PhTM se limita a dos modos: uno puramente hardware, donde las transacciones se ejecutan sólo en HTM, y un modo software, que utiliza un STM basado en TL-2 (ver sección 2.5.1) que no interacciona con el HTM. Cada hilo lleva asociado una variable que informa del modo de ejecución actual y puede

[9]Estructura que contiene al menos un *lock* y un identificador de transacción. Cuando el *lock* está adquirido, el identificador contiene la transacción propietaria del espacio protegido por el *lock*.

UNIVERSIDAD DE MÁLAGA

actualizarse mediante una llamada al *runtime*, que decide si seguir en el modo actual o no. El HTM incluye una comprobación del modo actual de ejecución al inicio de cada transacción hardware para garantizar que cualquier cambio de modo abortará dichas transacciones. PhTM comienza las transacciones en modo hardware y monitoriza los posibles abortos y sus causas para determinar si es necesario ejecutarlas en software. Si es así, la transacción incrementa un contador compartido (`deferredCount`) y espera un cierto tiempo antes de cambiar el modo de ejecución. Una vez en modo software se ejecutarán al menos `deferredCount` transacciones software exitosas antes de intentar cambiar de nuevo a modo hardware.

PhTM consigue un sistema híbrido con un overhead mucho menor que HyTM a costa de impedir la ejecución concurrente de transacciones hardware y software. De este modo, códigos donde un HTM no es efectivo pueden ejecutarse utilizando transacciones software y se evita instrumentar cada acceso de las transacciones hardware y penalizar el rendimiento del HTM.

Hybrid NOrec

Hybrid NOrec es un TM híbrido que combina el STM NOrec (ver sección 2.5.1) con un HTM *best-effort* [21]. Este TM intenta combinar las ventajas de HyTM, donde transacciones hardware y software pueden estar en ejecución de forma concurrente; con las de PhTM, que no requiere instrumentar los accesos de las transacciones hardware. NOrec, a diferencia de la mayoría de propuestas STM, realiza la validación de sus datos transaccionales por valor (es decir, sin necesidad de metadatos asociados a los elementos de memoria accedidos), y utiliza muy pocos metadatos. Estos dos motivos lo hacen especialmente adecuado para combinarlo con un HTM. La idea de este algoritmo es que las transacciones hardware notifiquen a las software cuando finalicen su fase de *commit* para forzar a éstas últimas a revalidar sus datos.

Hybrid NOrec utiliza el algoritmo basado en *lock* de secuencia de NOrec y lo combina con un contador global para habilitar la comunicación entre transacciones hardware y software. Cada transacción hardware lee el *lock* de secuencia al iniciarse y se asegura de que sea par (un valor impar indica que hay una transacción software en fase de *commit*). Esta lectura añade dicho *lock* al *read-set* de la transacción hardware, de modo que abortará si cualquier transacción software finaliza durante su ejecución. Antes de que la transacción hardware llegue a su fase de *commit*, incrementa el contador global para notificar a posibles transacciones software concurrentes de la necesidad de revalidar sus datos. Por su parte, cada transacción software debe comprobar que el valor de este contador global en cada

UNIVERSIDAD DE MÁLAGA

lectura —revalidando sus datos si detecta alguna variación—, y antes de adquirir el *lock* de secuencia en su fase de *commit*. El hecho de que todas las transacciones hardware compartan un mismo contador global puede ocasionar abortos si varias transacciones intentan incrementarlo mientras una de ellas está en fase de *commit*. Los autores proponen utilizar un *array* de contadores para que las transacciones hardware de distintos hilos incrementen contadores distintos a costa de penalizar a las transacciones software, que deberán comprobar los contadores adicionales. Otra optimización, sólo posible si el HTM permite lecturas no transaccionales dentro de una transacción, es utilizar un esquema de *lazy subscription* [13] al *lock* de secuencia: en el algoritmo anterior cualquier *commit* de una transacción software aborta a las transacciones hardware activas aunque no existan conflictos de datos, debido al incremento del *lock* de secuencia. Las transacciones hardware pueden evitar suscribirse al inicio a este *lock* comprobando su valor —mediante una lectura no transaccional— antes de cada acceso transaccional y realizando una validación en caso de que alguna transacción software haya finalizado. Este sistema evita abortos innecesarios de transacciones hardware a costa de añadir instrumentación a cada acceso que realicen.

Hybrid NOrec consigue cierto nivel de concurrencia entre transacciones hardware y software sin añadir demasiada instrumentación. Cuando los abortos son espurios permite iniciar unas pocas transacciones software sin necesidad de abortar a las hardware, lo que le da ventaja en ciertos escenarios respecto a PhTM. Sin embargo, las transacciones software no sólo tardan más tiempo en ejecutarse, sino que afectan al rendimiento de las transacciones hardware, teniendo éstas últimas que realizar comprobaciones adicionales.

Invyswell

Invyswell [12] es una propuesta híbrida específicamente pensada para adaptarse a las limitaciones del HTM Intel TSX, el cual no permite operaciones no transaccionales en el contexto de una transacción (ver sección 2.5.2). Esta propuesta utiliza una versión modificada de InvalSTM (ver sección 2.5.1) como STM debido a que se complementa bien con TSX: mientras que el HTM funciona especialmente bien con transacciones pequeñas con poca contención, InvalSTM permite la ejecución de transacciones largas o en escenarios de contención elevada de forma eficiente gracias a su sistema de detección de conflictos *lazy*.

Invyswell distingue cinco tipos diferentes de transacciones, cada uno de ellos optimizado para distintos escenarios de ejecución: Las transacciones *SpecSW* se ejecutan en software y son similares a las de InvalSTM, utilizando filtros de Bloom para la detección de conflictos. Su contraparte son las transacciones

UNIVERSIDAD DE MÁLAGA

BFHW, ejecutadas en hardware pero con sus accesos transaccionales instrumentados para utilizar filtros de Bloom, permitiendo la detección de conflictos con las transacciones *SpecSW*. Un par de variables compartidas permiten la coordinación entre estos dos tipos de transacciones de forma similar a la utilizada en Hybrid NOrec, aunque en este caso las transacciones se abortan usando un sistema de invalidación como el utilizado en InvalSTM. Dado que las transacciones *BFHW* requieren instrumentación, se añade un tipo adicional, *LiteHW*, que se ejecuta en hardware sin necesidad de instrumentación adicional, pero que no puede finalizar si cualquier otro tipo de transacción está activa. Los dos últimos tipos de transacciones se ejecutan en software: *IrrevocSW* garantiza irrevocabilidad[10], pero impide que otras transacciones puedan finalizar durante su ejecución; *SglSW* restringe la ejecución de cualquier otra transacción adquiriendo un *lock* global, pero garantiza la irrevocabilidad sin necesidad de instrumentación adicional. Una máquina de estados permite transicionar entre los diferentes tipos de transacción dependiendo del escenario de ejecución que encuentre el sistema en cada momento.

Las limitaciones de TSX complican el diseño a la hora de comunicar los distintos tipos de transacciones y, especialmente, a la hora de conseguir garantizar la consistencia entre transacciones de distinto tipo. Los autores han comparado este modelo con una versión de Hybrid NOrec sin algunas optimizaciones debido a la falta de soporte para código no especulativo dentro de transacciones de TSX. Invyswell consigue mejores resultados que Hybrid NOrec en la mayoría de pruebas realizadas. Sin embargo, diseños puramente STM como NOrec resultan competitivos respecto a modelos híbridos, superándolos incluso en algunos escenarios.

En general, aunque los enfoques híbridos parecen tener sentido para complementar los escenarios menos favorables de las implementaciones hardware y software de TM, las propuestas vistas hasta ahora no superan claramente a diseños puramente hardware o software en multitud de escenarios. Actualmente, Intel e IBM, principales fabricantes de procesadores con soporte HTM, recomiendan en sus manuales el uso de un simple *lock* global para implementar un *fallback* software de forma sencilla.

UNIVERSIDAD DE MÁLAGA

[10]Una transacción irrevocable no puede ser abortada una vez iniciada.

UNIVERSIDAD DE MÁLAGA

Capítulo 3

Soporte de reducciones y orden en memoria transaccional software

Este capítulo analiza la extracción de paralelismo en aplicaciones irregulares utilizando modelos basados en memoria transaccional. La sección 3.1 introduce el concepto de aplicaciones irregulares. La sección 3.2 introduce las reducciones, un tipo de operación que aparece con frecuencia en este tipo de aplicaciones. La sección 3.2.1 ofrece una perspectiva de diferentes aproximaciones a la hora de extraer paralelismo de este tipo de aplicaciones, incluyendo técnicas clásicas y enfoques más recientes basados en técnicas especulativas y en el uso de memoria transaccional. La sección 3.3 presenta nuestra propuesta basada en memoria transaccional para extraer paralelismo de aplicaciones irregulares cuando éstas contienen patrones de reducción. La sección 3.4 evalúa nuestra propuesta en una serie de escenarios de interés. Por último, la sección 3.5 analiza otros trabajos relacionados de la literatura.

3.1. Aplicaciones irregulares

El término irregular, en el contexto de códigos de programación, hace referencia a la complejidad de las estructuras de datos que contiene dicho código y al modo en que estas estructuras son accedidas a lo largo de su ejecución. En este tipo de

UNIVERSIDAD DE MÁLAGA

códigos podemos encontrar estructuras como grafos, listas enlazadas o matrices dispersas que son accedidas mediante punteros o funciones de indirección. Por contraposición, hablamos de códigos regulares cuando éstos contienen estructuras simples que son accedidas siguiendo patrones sencillos, como matrices densas o vectores recorridos mediante índices. El carácter irregular de un código viene a menudo determinado por el uso de estructuras que permiten trabajar con determinados problemas de forma más eficiente, si bien dicho problema podría ser recodificado en estructuras y patrones de acceso más sencillos. Por ejemplo, una estructura de matriz dispersa con funciones de indirección permite ahorrar una gran cantidad de memoria respecto a una estructura típica de matriz densa. Esta clase de códigos se encuentra a menudo en aplicaciones de campos relacionados con la ciencia y la ingeniería.

Normalmente extraer paralelismo de un programa secuencial requiere descomponerlo en diferentes tareas que puedan ejecutarse de forma concurrente y resolver las dependencias que puedan producirse entre las mismas. Las aplicaciones irregulares suponen una dificultad añadida ya que sus dependencias no siguen patrones sencillos, siendo frecuente que no se conozcan hasta el momento de la ejecución. Esto conlleva que un análisis estático (en tiempo de compilación) de estas dependencias no siempre es posible o efectivo en este tipo de aplicaciones. En este contexto, modelos de concurrencia optimista como *Thread-Level Speculation* (TLS) o *Speculative Multithreading* (SpMT) pueden resultar útiles a la hora de extraer paralelismo [87]. La especulación es particularmente útil en aplicaciones cuando la información de las dependencias de datos no está disponible de antemano, ya que permite analizar y resolver estas dependencias dinámicamente en lugar de optar por alternativas más conservadoras que implicarían serializar la ejecución.

Es posible aprovechar TM para proporcionar soporte a la ejecución especulativa de códigos secuenciales [109]. El modelo de concurrencia optimista que aporta el uso de TM puede ayudar a paralelizar aplicaciones irregulares [74]: los programadores pueden paralelizar una aplicación secuencial repartiendo adecuadamente la computación entre los hilos de ejecución disponibles y ejecutando las tareas con posibles dependencias de datos en transacciones concurrentes. El sistema transaccional será el encargado de monitorizar los accesos a memoria y de detectar y resolver los eventuales conflictos (dependencias de datos) que puedan producirse entre las distintas tareas durante la ejecución. Hay que señalar, no obstante, que es necesario establecer una serie de restricciones de orden entre las transacciones para asegurarse de que estas dependencias de datos se resuelven adecuadamente, preservando la corrección de los resultados. En general, las transacciones deben finalizar en un orden que preserve la semántica secuencial del problema.

UNIVERSIDAD DE MÁLAGA

3.2. Patrones de reducción

Las reducciones son un conjunto de operaciones que aparecen con frecuencia en el núcleo de muchas aplicaciones científicas como operaciones con matrices dispersas o métodos numéricos para la resolución de ecuaciones diferenciales [113, 44, 69, 54, 60, 49]. Formalmente, una operación o sentencia de reducción puede definirse como sigue.

Definición. *Una sentencia de reducción es una asignación de la forma* $O = O \oplus \xi$*, donde* $\oplus$ *es un operador asociativo y conmutativo que se aplica al objeto de memoria* O*, y* ξ *es una expresión computada sin usar objetos que dependan de* O*.*

Ejemplos de operadores de reducción son la suma, el producto o el cálculo de máximos o mínimos.

El hecho de que un operador de reducción sea conmutativo y asociativo permite cierta libertad en la ejecución concurrente de estas operaciones, ya que el orden de ejecución de varias sentencias de reducción puede alterarse sin afectar al resultado final. Esto introduce oportunidades de extraer paralelismo, particularmente en bucles donde el acceso a los datos compartidos se realice mediante operaciones de reducción. Sin embargo, el hecho de que un bucle acceda a datos compartidos mediante estas operaciones no es condición suficiente para poder reordenar sus iteraciones libremente. Denominamos *bucle de reducción* a aquellos bucles que sí pueden reordenarse libremente sin alterar el resultado final.

Definición. *Un bucle de reducción es un conjunto de instrucciones que contiene al menos una sentencia de reducción aplicada a uno o más objetos en memoria manteniendo siempre el mismo operador para cada objeto, y de tal manera que las únicas dependencias reales entre iteraciones*[11] *son aquellas originadas por las sentencias de reducción.*

La definición anterior implica que no deben existir accesos de lectura o escritura a las variables de reducción O en otras partes del bucle fuera de las sentencias de reducción (ya que ocasionaría una dependencia de datos) [60]. Gracias a las propiedades de conmutatividad y asociatividad de las sentencias de reducción, las iteraciones de un bucle de reducción pueden reordenarse libremente sin afectar al resultado final. La figura 3.1 muestra un par de ejemplos de bucles de reducción: en el código de la izquierda un operador de reducción $\oplus$ se aplica a la variable de reducción `A` (reducción escalar). En el código de la derecha las operaciones de reducción se aplican a elementos de un *array* de reducción `A[]`, lo que se conoce como *reducción de histograma* [101].

[11] Denominadas *loop-carried dependences* en la literatura [36].

UNIVERSIDAD DE MÁLAGA

```
1  float A;
2  for (i=0; i<N; i++){
3      Calculate ξ;
4      A = A ⊕ ξ;
5  }
6
7
8
```

```
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  }
```

Figura 3.1: Ejemplos de bucles de reducción: sentencia de reducción escalar simple (izquierda), múltiples sentencias de reducción irregulares (derecha).

En el último caso nos encontramos además ante un código irregular, donde la complejidad de la operación viene determinada por el patrón de acceso al *array* de reducción: una serie de *arrays de indirección* (f1 ... fn), actúan como subíndices del *array* de reducción, el cuál, a su vez, depende del índice i del bucle. Este tipo de bucles con patrones de indirección aparecen con cierta frecuencia los *kernels* de aplicaciones irregulares, por lo que su paralelización eficiente es motivo de estudio. La figura 3.2 muestra algunos ejemplos de este tipo de bucles en aplicaciones reales.

Las aproximaciones automáticas o semi automáticas a la hora de paralelizar aplicaciones irregulares suelen ser conservadoras, especialmente si se basan en un análisis de dependencias estático (durante la compilación). Esto se debe a que las dependencias que pueden producirse entre las iteraciones del bucle vienen determinadas por el valor de los *arrays* de indirección, que es desconocido hasta la ejecución. De hecho, a menudo no será posible determinar si un bucle es o no de reducción *a priori*. Ejemplos de estas situaciones ocurren cuando el objeto de reducción puede ser accedido fuera de la sentencia de reducción, cuando se aplican operadores de reducción distintos a los mismos objetos de reducción o cuando el bucle incluye otras dependencias además de aquellas producidas por las sentencias de reducción.

Reducciones parciales

Desde el punto de vista de las dependencias de datos, cualquier acceso a memoria compartida dentro de un bucle de reducción potencial puede introducir dependencias entre iteraciones. En el mejor de los casos las únicas dependencias verdaderas serán aquellas causadas por las sentencias de reducción. En estas situaciones, las iteraciones del bucle pueden ser reordenadas arbitrariamente

UNIVERSIDAD DE MÁLAGA

```
1  ! Leer arrays de subíndices:
2  edge(1,*), edge(2,*)
3
4  do itime=1,nTimes
5     do i=1,nEdges
6        n1 = edge(1,i)
7        n2 = edge(2,i)
8
9        Compute ζ1, ζ2, ζ3
10
11       vel(1,n1)=vel(1,n1)+ζ1
12       vel(2,n1)=vel(2,n1)+ζ2
13       vel(3,n1)=vel(3,n1)+ζ3
14
15       vel(1,n2)=vel(1,n2)-ζ1
16       vel(2,n2)=vel(2,n2)-ζ2
17       vel(3,n2)=vel(3,n2)-ζ3
18    enddo
19    ...
20 enddo
21
```

(a) Ecuaciones de Euler (extracto).

```
1  ! Calcular arrays de subíndices:
2  m(*),
3  mbeg(*),
4  mend(*)
5
6  do irow=1, nRows
7     do i=1,jdt+1
8        im =m(i)
9        imb=mbeg(i)
10       ime=mend(i)
11       do is=imb,ime,2
12          Compute ζ1, ζ2...
13          do ilev=1,2*jdlev
14             f1(ilev,im)=f1(ilev,im)+ζ1
15             f2(ilev,im)=f2(ilev,im)+ζ2
16             ...
17          enddo
18       enddo
19    enddo
20    ...
21 enddo
```

(b) Transformada de Legendre.

```
1  do ihop=1, nHops
2     ! Actualizar subscripts: B1(*),B2(*)
3     do itime=1,nTimes
4        do ih=1,nParticles
5           i=B1(ih)
6           j=B2(ih)
7           Compute r(i,j), ζ(i,j), η(i,j), θ(i,j)
8
9           if (r .lt. CutOff) then
10             AX(i)=AX(i) + ζ
11             AX(j)=AX(j) - ζ
12             AY(i)=AY(i) + ζ
13             AY(j)=AY(j) - ζ
14             U = U + η
15             P = P + θ
16          endif
17       enddo
18       ...
19    enddo
20 enddo
```

(c) Simulación de dinámica molecular 2D.

Figura 3.2: Algunos *kernels* reales con bucles de reducción: (a) Ecuaciones de Euler para simulación de dinámica de fluidos, (b) Transformada de Legendre para predicción meteorológica, (c) Simulación de dinámica molecular 2D para interacciones de corto alcance.

UNIVERSIDAD DE MÁLAGA

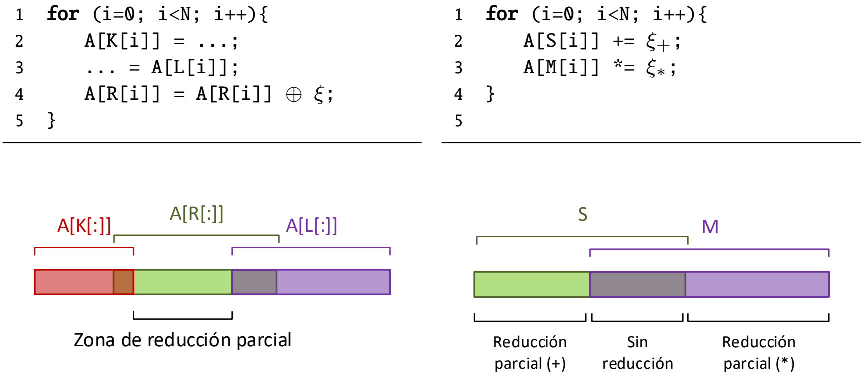


Figura 3.3: Ejemplos de reducciones parciales. La imagen bajo cada código representa el subconjunto del *array* `A` accedido mediante los *arrays* de indirección `K`, `L`, `R`, `S` y `M`.

sin afectar al resultado final, como consecuencia de las propiedades de conmutatividad y asociatividad del operador de reducción. Por otra parte, se pueden producir situaciones donde las condiciones de reducción no se cumplan en su totalidad. Ejemplos de estas situaciones ocurren cuando la variable de reducción es accedida en el bucle fuera de la sentencias de reducción, cuando varios operadores de reducción diferentes se aplican a la misma variable de reducción, o cuando el bucle incluye otras dependencias entre iteraciones además de las propias sentencias de reducción [92, 54]. En los primeros dos casos es posible que las condiciones de reducción se cumplan solamente en un subconjunto de los accesos a memoria al objeto de reducción, pero no para la totalidad de estos accesos. Inspirados por [54] denominamos a esta situación *reducción parcial*.

Definición. *Una reducción parcial es un bucle potencial de reducción que sólo cumple las condiciones de reducción en un subconjunto de los accesos de reducción del bucle.*

La figura 3.3 muestra dos ejemplos de este tipo de reducciones. El bucle de la izquierda presenta accesos de lectura, escritura y reducción al objeto `A` mediante *arrays* de indirección. En este código, identificar el bucle como *bucle de reducción* implica garantizar que los elementos de los *arrays* de indirección `K`, `L` y `R` son disjuntos en el intervalo `0..N` que recorre el bucle. En caso contrario existirán dependencias entre iteraciones que requerirán que el bucle se ejecute siguiendo —al menos parcialmente— el orden secuencial original. El ejemplo de la derecha requiere de las mismas garantías en los *arrays* de indirección `S` y `M`. En este caso los

UNIVERSIDAD DE MÁLAGA

dos accesos del bucle son de reducción, pero cada uno utiliza un operador distinto que no es asociativo y conmutativo con el otro. Si existen índices repetidos en `S` y `M` no se puede clasificar el bucle como de reducción.

La presencia en aplicaciones de reducciones parciales —o de bucles de reducción potenciales cuyas dependencias no son analizables estáticamente— impide extraer paralelismo utilizando técnicas clásicas. En este contexto la memoria transaccional puede proporcionar un soporte para explotar la concurrencia de estas aplicaciones durante la ejecución, permitiendo de este modo la explotación de paralelismo en aplicaciones irregulares, donde la información de las dependencias de datos entre los distintos hilos de ejecución no es fácilmente analizable.

3.2.1. Técnicas clásicas para la paralelización de reducciones

La paralelización de bucles de reducción es un campo ampliamente estudiado en la literatura dada su relevancia en códigos científicos. A la hora de clasificar las distintas propuestas podemos distinguir tres grupos de técnicas: las basadas en exclusión mutua, las basadas en la privatización de las variables de reducción y las basadas en el particionamiento de los objetos de reducción. Una limitación común a estas técnicas es que requieren que el bucle donde se aplican haya sido identificado previamente como *bucle de reducción*. Esta identificación se puede llevar a cabo en tiempo de compilación siempre que la complejidad no sea excesiva [72]. Si esta premisa no se cumple se obtendrán resultados incorrectos.

Técnicas basadas en exclusión mutua

Estas técnicas intentan extraer paralelismo ejecutando la mayor parte del bucle de reducción en paralelo siguiendo un esquema DoAll y utilizando algún método de exclusión mutua para proteger los accesos de reducción a memoria compartida, encapsulando dichos accesos en una sección crítica. Las aproximaciones basadas en exclusión mutua son sencillas de implementar, pero tienen el inconveniente del grado de serialización que introducen y el coste de la sincronización en los procesadores CMP actuales. El grado de serialización viene determinado por el número de iteraciones que haya en conflicto y por el método utilizado para garantizar la exclusión mutua.

La técnica más trivial es proteger la sección completa mediante un mecanismo de bloqueo como un *lock*, serializando todos los accesos de reducción (*coarse-grain locks*). Para mitigar el coste que introduce este método se pueden utilizar varios *locks*, de forma que cada uno de los ellos proteja el acceso a un subconjunto del

UNIVERSIDAD DE MÁLAGA

conjunto de reducción (*fine-grain locks*). Si dos hilos acceden a subconjuntos disjuntos del conjunto de reducción pueden realizar sus accesos simultáneamente, aumentando así el nivel de concurrencia. Aunque este método es eficiente, soportar un gran número de *locks* en el caso de conjuntos de reducción grandes puede repercutir significativamente en los requerimientos de memoria del algoritmo.

Otras técnicas contemplan el uso de instrucciones atómicas hardware disponibles en procesadores modernos. Estas instrucciones bloquean el bus de memoria mientras se ejecutan, evitando que otros hilos puedan acceder a los datos que utilizan de forma simultánea. Aunque estas técnicas suelen ser eficientes tanto en el coste de la sincronización como en el uso de memoria, dependen de que el operador de reducción y el tipo de dato de las variables de reducción estén soportados de forma atómica por la arquitectura del hardware. Por ejemplo, la mayoría de procesadores convencionales no soportan el uso de variables de punto flotante en este tipo de instrucciones. Por otra parte, estas técnicas no permiten agrupar varias operaciones de forma atómica, algo que puede ser necesario para garantizar la ejecución correcta del algoritmo.

Los modelos basados en tareas, que se pueden utilizar también para paralelizar bucles de reducción, pueden incluirse en este conjunto de técnicas. Por ejemplo, en modelos de soporte para dependencias entre tareas como *OpenMP task depend* o propuestas como [86, 41], las variables de reducción se pueden expresar como dependencias de entrada-salida de la tarea. La limitación principal de estas técnicas, en el contexto de aplicaciones irregulares, es la capacidad de estos modelos a la hora de expresar dependencias complejas con suficiente precisión, así como su aplicabilidad cuando estas dependencias se computan dentro de la propia tarea.

Técnicas basadas en la privatización de los objetos de reducción

Este conjunto de técnicas reparte las iteraciones del bucle de reducción entre los distintos hilos de ejecución. Cada hilo realiza sus operaciones de reducción sobre una copia privada del objeto de reducción que debe inicializarse con el elemento neutro del operador correspondiente antes de ejecutar el bucle de reducción (*copy-in*). Una vez los hilos han terminado la computación, una fase adicional debe acumular los resultados locales de cada hilo en el objeto global de reducción (*copy-out*). Dos ejemplos representativos de este conjunto de técnicas son *Replicated Buffer* [50], que crea la réplica privada en el espacio de memoria de cada hilo y requiere sincronizar la fase de acumulación final; y *Array Expansion* [31], que aumenta la dimensión del objeto de reducción, pudiendo ejecutar en paralelo la fase de acumulación.

UNIVERSIDAD DE MÁLAGA

Las técnicas basadas en privatización son sencillas de implementar y evitan cualquier tipo de sincronización durante la ejecución del bucle, que se reserva para las fases adicionales *copy-in* y *copy-out*. Suelen funcionar bien cuando el patrón de accesos al objeto de reducción es denso, es decir, cuando la mayor parte del objeto de reducción se actualiza durante la computación del bucle. En patrones dispersos, la penalización que introducen las fases *copy-in* y *copy-out* puede no verse compensada si un porcentaje alto de los elementos del objeto de reducción no son modificados durante la ejecución del bucle. Otro aspecto a tener en cuenta es el hecho de que la duración de estas fases adicionales se incrementa proporcionalmente al número de hilos de ejecución. No obstante, el mayor inconveniente de estas técnicas es su elevado uso de memoria, ya que el espacio de memoria del objeto de reducción se multiplica por el número de hilos de ejecución, al replicar el conjunto de reducción completo. Para paliar estos requerimientos, otras técnicas como *Selective Privatization* [113] intentan reducir el uso de memoria replicando selectivamente el subconjunto del objeto de reducción que necesite cada hilo de ejecución a costa de implementaciones más complejas.

Técnicas basadas en el particionado del espacio de reducción

Con el objetivo de reducir el consumo de memoria y aumentar la localidad de los accesos, estas técnicas garantizan que cada hilo de ejecución accede a un subconjunto privado del objeto de reducción. Para ello es necesaria una fase de análisis previa que determine qué iteraciones asignar a cada uno de los hilos de ejecución y una segunda fase donde se lleve a cabo la ejecución. Ejemplos de estas técnicas son LocalWrite [53] y SynchWrite [49].

Aunque estas técnicas pueden mejorar la localidad en el acceso al objeto de reducción, especialmente en códigos irregulares, son más complejas al requerir una fase previa de inspección para determinar el particionamiento correcto del objeto de reducción. El coste adicional de esta fase puede ser demasiado elevado para obtener una mejora en el tiempo de ejecución. Además pueden producirse desbalanceos de carga entre los diferentes hilos si los accesos al objeto de reducción no se distribuyen de forma uniforme a lo largo del bucle.

3.2.2. Paralelización de reducciones con soporte TM

La memoria transaccional puede resultar útil para extraer paralelismo en las situaciones anteriores. TM combina las garantías de las técnicas clásicas para

UNIVERSIDAD DE MÁLAGA

	Uso de memoria	Paralelismo potencial	Coste de la sincroniz.	Etapas adicionales
Coarse-grain locks	Muy bajo	Bajo	Alto	—
Fine-grain locks	Medio	Alto	Bajo	—
Instrucciones atómicas	Nulo	Alto[3]	Bajo	—
Modelos basados en tareas	Bajo	Medio	Variable	Análisis de dependencias
Privatización[1]	Muy alto	Alto	Bajo	Inicialización + Reducción final
Particionamiento del objeto de reducción[2]	Bajo	Alto	Bajo	Inspector
TM	Bajo	Alto	Variable	—

[1] Engloba a las técnicas de *Replicated Buffer* y *Array Expansion*.
[2] Engloba a las técnicas *LocalWrite* y *SynchWrite*.
[3] Siempre que el operador y el tipo de datos estén soportados por el hardware.

Tabla 3.1: Comparativa entre diferentes técnicas de paralelización de reducciones.

paralelizar bucles de reducción (atomicidad y privatización selectiva) con un entorno especulativo y una mayor facilidad de programación. En el caso de aplicaciones irregulares, el sistema TM será el encargado de detectar posibles conflictos en accesos con indirecciones. La tabla 3.1 resume las diferentes técnicas de reducción revisadas hasta el momento incluyendo el uso de TM.

Las transacciones pueden ser un sustituto natural de las secciones críticas a la hora de paralelizar reducciones. Una reducción es, al fin y al cabo, una lectura seguida de una escritura sobre el mismo objeto de memoria, y puede adaptarse a un entorno TM explícito tal y como muestra la figura 3.4 (a). Este modelo de ejecución es similar al uso de *locks* de grano fino para proteger objetos de memoria individuales [101, 93, 64]. En escenarios de baja contención, TM debería obtener una mayor escalabilidad respecto a otras técnicas basadas en el uso de secciones críticas.

La figura 3.4 (b,c,d) muestra cómo las sentencias de reducción pueden incluirse dentro de transacciones utilizando distintos niveles de granularidad; ya sea utilizando una transacción por sentencia (b), una transacción por iteración del bucle (c) o una transacción por conjunto (chunk) de iteraciones (d). El cálculo de las variables que serán reducidas (ξ) se considera privado en estos ejemplos, por lo que puede ejecutarse fuera de la sección crítica. Sin embargo, las indirecciones

UNIVERSIDAD DE MÁLAGA

```
1  #pragma omp critical
2  {
3    ...
4    A = A ⊕ ξ
5  }
6                 ⇓
7  TmBegin();
8  ...
9  TmWrite(A, TmRead(A) ⊕ ξ);
10 TM_end();
11
```

(a) Reducción en un TM explícito.

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

(b) Transacciones de grano fino.

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

(c) Transacciones de grano medio.

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

(d) Transacciones de grano grueso.

Figura 3.4: Paralelización de bucles de reducción utilizando TM.

que contienen las sentencias de reducción pueden causar abortos de transacciones si existen dependencias entre los accesos. El coste de estos abortos aumenta con el tamaño de la transacción, si bien el uso de transacciones más reducidas (b) y (c) conlleva una penalización producida por el mayor número de fases de inicio y finalización de las transacciones.

Si es posible determinar en tiempo de compilación que no existe ninguna otra dependencia en el bucle a paralelizar además de las originadas por las sentencias de reducción, el sistema TM puede ser modificado para deshabilitar la detección de conflictos dentro del bucle [42]. Sin embargo, si los bucles contienen otros accesos a memoria compartida que puedan ocasionar dependencias o si la contención sobre las variables de reducción es elevada, el rendimiento obtenible utilizando TM puede verse afectado. La figura 3.5 muestra ejemplos de estas situaciones. En el código (a) el acceso a los objetos de reducción se realiza

UNIVERSIDAD DE MÁLAGA

a través de punteros, lo que puede causar alias con otros objetos del código que sean accedidos fuera de las sentencias de reducción. En el código (b) los accesos al objeto de reducción se realizan mediante indirecciones y el número de sentencias de reducción, que determina la densidad de los accesos, no se conoce de antemano. En estos casos, las técnicas clásicas vistas en la sección 3.2.1 no siempre son aplicables, debido a la dificultad en el análisis de dependencias, y pueden resultar poco efectivas si el número de accesos a los objetos de reducción es escaso respecto al tamaño de dichos objetos. En tales situaciones, explotar el paralelismo puede requerir el uso de técnicas especulativas [92, 91, 54]. Es aquí donde centramos nuestra propuesta.

3.3. ReduxSTM

En esta sección introducimos *ReduxSTM*, una implementación STM capaz de aprovechar las propiedades de conmutatividad y asociatividad de las sentencias de reducción para mejorar el rendimiento de aplicaciones que presentan este tipo de operaciones. La idea que proponemos es aprovechar la privatización selectiva implícita que realiza TM para evitar conflictos debidos a accesos de reducción, junto con ciertas restricciones de orden que garantizan la corrección del sistema en caso de que produzcan interacciones con otros accesos transaccionales. El programador —o el compilador— únicamente tiene que reconocer patrones de reducción potenciales, no siendo necesario garantizar que el bucle completo cumpla las propiedades de reducción. De este modo, tanto reducciones clásicas como reducciones parciales se pueden tratar con una interfaz común, y el sistema TM detectará y resolverá los conflictos potenciales para asegurar la correcta ejecución del código.

3.3.1. Características

ReduxSTM combina dos características principales:

Soporte transaccional explícito para reducciones Se añade la operación de reducción como una primitiva más del sistema TM. Este soporte requiere añadir nuevos *conjuntos de reducción* para cada operador de reducción soportado, así como definir nuevas reglas a la hora de detectar y resolver posibles conflictos asociados a los nuevos conjuntos.

Restricciones de orden en la fase de *commit* de las transacciones Esta característica permite mantener la corrección del sistema en presencia de reduccio-

UNIVERSIDAD DE MÁLAGA

```
1  new_dbox_a(..., *costptr) {
2    ...
3    for (termptr = ...; termptr = termptr->nextterm) {
4      ...
5      for (netptr = ... ; netptr=netptr->nterm) {
6        ...
7        *costptr += ABS(newx - new_mean) - ...; /* Sentencia de reducción */
8      }
9      ...
10     /* Accesos de lectura al puntero rowptr que pueden contener alias
11      * con los objetos de reducción */
12     rowsptr = tmp_rows[net]; /* tmp_rows es global */
13     for (row = 0; rowsptr[row] == 0 ; row++) {
14       ...
15     }
16     ...
17     /* Accesos de escritura al objetos globales que pueden contener alias
18      * con los objetos de reducción */
19     tmp_num_feeds[net] = f;
20     ...
21     tmp_missing_rows[net] = -m;
22     ...
23     delta_vert_cost += ( ... ); /* Sentencia de reducción */
24   }
25   return;
26 }
```

(a) Alias en punteros en el código 300.twolf (SPEC2000).

```
1  do {
2    ...
3    delta = 0.0;
4    for (i = 0; i < npoints; i++) {
5      index = findNearestPoint(feature[i], nfeatures, clusters, nclusters);
6      /* Si el punto cambia de cluster, incrementa delta */
7      if (membership[i] != index) {
8        delta += 1.0;
9      }
10     /* Asigna la pertenencia del objeto i. No es modificado por otro hilo */
11     membership[i] = index;
12     /* Actualiza los nuevos centros de los clusters */
13     new_centers_len[index][0] += 1;
14     for (j = 0; j < nfeatures; j++) {
15       new_centers[index][j] += feature[i][j];
16     }
17   }
18   ...
19 } while (delta > threshold);
```

(b) Patrones de reducción en el código *Kmeans* (STAMP).

Figura 3.5: Ejemplos de bucles de reducción de interés. Este tipo de patrones dificultan un análisis estático de dependencias.

UNIVERSIDAD DE MÁLAGA

nes parciales y permite preservar la consistencia secuencial siempre que sea necesario, como ocurre cuando se usa TM para propósitos TLS. ReduxSTM utiliza la información adicional de orden para optimizar la resolución de conflictos entre transacciones.

ReduxSTM puede implementarse como un STM independiente o como un soporte adicional para mejorar una implementación STM existente. No necesita de soporte específico a nivel de compilador ni a nivel arquitectural. A continuación se detallan sus características diferenciales:

Primitivas de reducción ReduxSTM permite trabajar con sentencias de reducción sin requerir información adicional sobre el contexto en el que aparecen dichas sentencias —que podría ser o no un bucle de reducción—. Con este propósito, se introduce una operación transaccional adicional para las reducciones que recibe tres argumentos: una posición de memoria A, un operador de reducción *op* (suma, producto, máximo, mínimo, ...), y un valor ξ que será reducido en A:

`TmRdx(A,` ξ`, op).`

La acción llevada a cabo por `TmRdx` es semánticamente equivalente a:

`TmWrite(A, TmRead(A)` $\oplus$ ξ`)` ($\oplus$ es el operador de reducción *op*),

pero con la salvedad de que las operaciones de reducción que utilicen el mismo operador no producirán conflictos entre sí. Esta primitiva está disponible al programador junto con las operaciones transaccionales habituales `TmRead(A)` y `TmWrite(A, val)` descritas en la sección 2.2, y su uso es opcional.

Conjuntos de reducción Además de los conjuntos de datos habituales definidos en la sección 2.3.3 (*read-set* y *write-set*), es necesario introducir un conjunto de reducción para cada operador *op* de reducción que soporte el sistema TM (*rdx-set*^{op}). Cuando se invoque a una primitiva de reducción transaccional, la dirección de memoria de la variable de reducción y ξ se añadirán al conjunto de reducción correspondiente.

Deben tenerse en cuenta dos particularidades:

(1) Una dirección de memoria permanecerá en su conjunto de reducción a condición de que ninguna lectura, escritura o reducción transaccional con *op* diferente opere en la misma dirección de memoria en esa misma transacción. Si se produce esta situación, la entrada del conjunto de reducción deberá migrar a los conjuntos de lectura y escritura, debido a que las propiedades de la reducción han dejado de cumplirse.

UNIVERSIDAD DE MÁLAGA

(2) Para cada dirección de memoria almacenada en el conjunto de reducción, el valor parcial a reducir hasta ese momento se almacena asociado a la misma. La primera vez que se aplica una reducción a una dirección de memoria en una transacción, el valor parcial ξ simplemente se almacena (virtualmente es reducido con el elemento neutro de la operación de reducción). En las operaciones de reducción subsiguientes que utilicen el mismo operador, este valor será actualizado en el conjunto de reducción acumulándolo mediante el operador correspondiente con el valor tentativo almacenado en dicho conjunto.

Gestión de versiones Debido al soporte de operaciones de reducción —que pueden producir conflictos con otras operaciones transaccionales— y de restricciones de orden, el diseño de ReduxSTM se ajusta naturalmente a un sistema *lazy* de gestión de versiones. Las versiones tentativas, almacenadas en los conjuntos de escritura y reducción, se consolidarán en memoria durante la fase de *commit*. Hasta entonces los valores estarán almacenados en sendos búferes privados a cada transacción, que almacenaran conjuntos disjuntos de datos (un objeto transaccional puede estar en cualquiera de los búferes, pero sólo en uno de ellos). Nótese que en este caso existen valores a consolidar tanto en el conjunto de escritura como en los conjuntos de reducción. Durante la fase de *commit* los valores del conjunto de escritura son copiados directamente a sus posiciones de memoria. Por su parte, los valores de cada conjunto de reducción deberán ser acumulados (mediante el operador correspondiente) a los existentes en las posiciones correspondientes de memoria compartida. En este sentido, cada transacción comienza con las entradas de sus conjuntos de reducción inicializadas al elemento neutro del operador correspondiente. Desde el punto de vista de las variables de reducción, las transacciones están realizando una privatización selectiva de forma especulativa.

Commit transaccional ordenado Otra característica destacable de ReduxSTM es el mantenimiento de restricciones de orden entre transacciones. ReduxSTM utiliza este mecanismo para garantizar la atomicidad del sistema y para preservar la semántica del código secuencial cuando se utiliza este sistema para paralelizar bucles de reducción potenciales. Aunque las iteraciones de estos bucles podrían reordenarse sin alterar la semántica del código, las operaciones de reducción pueden coexistir con otras operaciones de lectura y escritura en los objetos de reducción que no sean analizables hasta la ejecución. En tal caso, las restricciones de orden entre transacciones garantizan la corrección de los resultados.

ReduxSTM ejecuta la fase de *commit* de cada transacción de acuerdo al

UNIVERSIDAD DE MÁLAGA

orden secuencial de ejecución. Esta restricción tiene un cierto coste en rendimiento debido a la necesidad de serializar de la fase de *commit* de las transacciones y a los posibles retrasos que puedan sufrir las transacciones que finalicen su ejecución antes de que puedan confirmar sus cambios. Aunque esto limita la concurrencia potencial —las transacciones con conjuntos de datos disjuntos deben serializar su fase de *commit*—, la introducción de *commits* ordenados tiene algunas ventajas adicionales en nuestro modelo. Por una parte simplifica el control de la contención, ya que la transacción en fase de *commit* será la encargada de resolver los posibles conflictos que puedan producirse, ya sea invalidando a la otra transacción implicada o abortando, dependiendo de la política de resolución de conflictos. Por otra parte, dado que el gestor de conflictos tiene disponible información adicional sobre el orden de cada transacción, puede filtrar aquellos conflictos que involucren patrones *write-after-write* y *write-after-read* tratándolos como falsas dependencias. Los detalles de implementación de la fase de *commit* están sujetos a las características particulares del algoritmo utilizado, y se detallan en la sección 3.3.3. Independientemente de la implementación, la fase de *commit* de cada transacción es responsable de: (1) esperar su turno (orden) para iniciar el *commit*; (2) comprobar y resolver posibles conflictos con otras transacciones; y (3) actualizar los valores tentativos de las escrituras y reducciones transaccionales en la memoria compartida.

ReduxSTM define dos modelos para especificar estas restricciones de orden. Un parámetro adicional en el mecanismo de inicio de la transacción puede definir un orden explícito de la misma, de modo que la transacción n-ésima no podrá ejecutar su fase de *commit* hasta que las transacciones precedentes $t_1, ..., t_{n-1}$ hayan finalizado. Este modelo resulta especialmente útil para paralelizar especulativamente las iteraciones de un bucle, vinculando el orden transaccional a los iteradores del mismo. Opcionalmente ReduxSTM admite un orden implícito, en el que las transacciones obtienen su orden automáticamente antes de iniciar su fase de *commit*. Este modo es similar a un TM sin restricciones de orden, pero en este caso, una vez se asigna orden a una transacción, este orden se mantiene incluso si dicha transacción aborta y necesita volver a ejecutarse debido a un conflicto.

La tabla 3.2 muestra las interacciones que pueden producirse entre diferentes transacciones y los casos en los que la combinación de soporte de reducciones, *commit* ordenado y gestión de conflictos y versiones *lazy* pueden evitar abortos. Básicamente, el soporte de reducciones elimina conflictos entre reducciones del mismo tipo entre transacciones distintas, y el orden entre *commits* permite resolver las situaciones producidas cuando dos tran-

UNIVERSIDAD DE MÁLAGA

	TM	TM + Orden	TM + Reducción	TM + Orden + Reducción
R – R	Sin conflicto	Sin conflicto	Sin conflicto	Sin conflicto
R – W	Aborto	Sin conflicto	Aborto	Sin conflicto
R – RDX	Aborto[1]	Sin conflicto	Aborto	Sin conflicto
W – R	Aborto	Aborto	Aborto	Aborto
W – W	Aborto	Sin conflicto	Aborto	Sin conflicto
W – RDX	Aborto[1]	Aborto[1]	Aborto	Sin conflicto
RDX – R	Aborto[1]	Aborto[1]	Aborto	Aborto
RDX – W	Aborto[1]	Sin conflicto	Aborto	Sin conflicto
RDX – RDX	Aborto[1]	Aborto[1]	Sin conflicto	Sin conflicto
RDX – RDX′[2]	Aborto[1]	Aborto[1]	Aborto	Aborto

[1] No existe una primitiva específica para las operaciones de reducción, éstas se traducen a una combinación de lectura y escritura transaccional.
[2] RDX – RDX′ hace referencia a la interacción entre reducciones de diferente clase (con operadores diferentes).

Tabla 3.2: Conflictos entre transacciones en varios diseños STM. R, W y RDX indican operaciones transaccionales de lectura, escritura y reducción respectivamente. La primera columna indica situaciones en las que dos transacciones diferentes realizan sendos accesos sobre un mismo objeto.

sacciones realizan escrituras en la misma posición de memoria. Conviene señalar que las reducciones pueden producir conflictos entre operaciones dentro de la misma transacción. Esto ocurre cuando una variable de reducción colisiona con una operación de escritura o lectura transaccional, o con otra reducción que use un operador diferente. Si se da cualquiera de estas situaciones, la variable de reducción deberá dejar de considerarse como tal y transformarse en una combinación de lectura y escritura transaccional, como se ha comentado anteriormente.

Como observación adicional, la primitiva de reducción transaccional propuesta no devuelve el valor tentativo de la variable de reducción. Las reducciones en ReduxSTM están asociadas a meras actualizaciones de las posiciones de memoria asociadas. De este modo, interacciones W – RDX entre dos transacciones no causan un conflicto real (dependencia *read-after-write*) en nuestro modelo.

Resolución de conflictos La política de resolución de conflictos viene condicionada por el hecho de que existe una restricción de orden entre las fases de *commit* de las transacciones del sistema. En caso de conflicto entre dos transacciones, la manera natural de resolverlo es mediante el aborto y posterior reinicio de la transacción con el orden más alto. El diseño de ReduxSTM implica que la transacción en fase de *commit* sea la encargada de detectar y resolver posibles conflictos. Se han considerado dos estrategias:

UNIVERSIDAD DE MÁLAGA

(1) *Kill itself*: en este caso el conflicto ha ocurrido con una transacción precedente que ya ha finalizado. Esta estrategia resulta adecuada para algoritmos de detección de conflictos basados en *versioned locks* o *value-based validation* ya que, si bien las transacciones anteriores no existen, sus cambios son visibles gracias a los números de versión (*versioned-locks*) o a los propios valores actualizados (*value-based validation*).

(2) *Kill others*: en este caso, las transacciones en conflicto estarán en ejecución o esperando su turno para iniciar la fase de *commit*, pero su número de orden será necesariamente mayor. La transacción en fase de *commit* será la encargada de marcar las transacciones en conflicto como inválidas, que serán abortadas y reiniciadas cuando corresponda. Esta estrategia es adecuada para esquemas de invalidación similares al descrito en la sección 2.5.1.

3.3.2. Diseño

ReduxSTM se ofrece como una librería disponible al programador o al compilador. Se compone de: funciones de inicialización (TmInit, que debe ser invocada por el hilo principal antes de comenzar a ejecutar código transaccional y TmInitThread, que será invocada por cada hilo de ejecución antes de comenzar a ejecutar transacciones); las correspondientes funciones de cierre (TmExit y TmExitThread), y el resto de primitivas básicas de un sistema STM explícito (TmBegin, TmRead, TmWrite, TmEnd, TmRollback). A estas funciones se añade una primitiva explícita para los accesos de reducción, TmRdx. Con objeto de facilitar las descripciones, de aquí en adelante se considerará un operador único para las operaciones de reducción (representado como $\oplus$), aunque en la implementación real se pueden considerar varios.

El diseño de ReduxSTM intenta mantener el funcionamiento esencial de un STM clásico añadiendo las características antes descritas. Con ello perseguimos dos objetivos: poder evaluar el impacto del soporte de reducciones respecto a otros STM y mostrar la posibilidades de adaptación de las características de ReduxSTM a algoritmos STM reales. El soporte de las nuevas características se ha implementado sobre dos aproximaciones STM diferentes que cubren dos de las políticas de resolución de conflictos descritas en la sección anterior.

ReduxSTM-TS Esta aproximación utiliza el concepto de *timestamps* o *versioned-locks*, una técnica popular utilizada en varios diseños STM como TL-2 o TinySTM (ver sección 2.5.1). Este diseño se muestra en el algoritmo 1.

UNIVERSIDAD DE MÁLAGA

ReduxSTM-CTI La segunda aproximación sigue un esquema de *commit-time invalidation* similar al utilizado en propuestas como InvalSTM (ver sección 2.5.1). En este caso la detección de conflictos se realiza a partir de las posiciones de memoria accedidas. Para optimizar esta detección se han utilizado filtros de Bloom. Este diseño se muestra en el algoritmo 2.

La tabla 3.3 resume la notación común utilizada en la descripción de ambos algoritmos.

Ambos diseños incluyen soporte a operaciones de reducción y restricciones de orden. El soporte para reducciones implica la implementación de la nueva primitiva TmRdx y las modificaciones pertinentes a las funciones TmRead, TmWrite y TmEnd de modo que las operaciones de reducción se tengan en cuenta tanto a la hora de realizar una operación de lectura o escritura como a la hora de consolidar los valores tentativos en memoria. La información adicional de cada operación de reducción se almacena en el conjunto de reducción y su búfer de reducción asociado (RdxSet), el cual almacena en cada entrada el valor *parcial* acumulado de las reducciones tentativas que se hayan realizado en la dirección asociada a dicha entrada durante la ejecución de la transacción. Una escritura transaccional sobre un objeto al que se había aplicado anteriormente una operación de reducción implica la migración de entrada asociada del conjunto y búfer de reducción al conjunto y búfer de escritura, al dejarse de cumplir las propiedades de reducción.

Por su parte, las restricciones de orden en la fase de *commit* de las transacciones sirven de base para la sincronización de las mismas. Un contador de orden global, inicializado a 1, se incrementa cada vez que una transacción finaliza su *commit*. El orden de cada transacción se puede fijar implícitamente al invocar TmBegin o explícitamente al invocar TmBegin(orden). Cuando una transacción ha completado su ejecución, la primera acción de TmEnd es precisamente esperar hasta que el contador de orden global coincida con el orden de la transacción, pudiendo entonces continuar la fase de *commit* si no se detectan conflictos. En caso contrario alguna de las transacciones implicadas debe ser abortada. En el diseño basado en *timestamps*, la política de resolución de conflictos es *kill-itself* y en el diseño basado en *commit-time invalidation* es *kill-others*. La restricción de orden total descrita implica que sólo una transacción puede ejecutar la fase de *commit* más allá de la espera inicial, que será la que tenga un orden menor de entre todas las que estén activas en un instante dado de la ejecución.

Para garantizar el progreso del sistema, la asignación de orden a las transacciones deberá llevarse a cabo de un modo tal que el conjunto de transacciones en ejecución en cualquier instante pueda formar una secuencia de orden consecutiva. Si se omite un número de orden, las transacciones de orden superior se quedarán

UNIVERSIDAD DE MÁLAGA

Notación	Significado
nThreads	Número de hilos de ejecución.
glOrder	Contador global de orden. Se incrementa tras cada *commit* transaccional.
glTx	Array global que almacena los descriptores de las transacciones. Existe un descriptor por cada hilo de ejecución.
tx	Puntero local al hilo que apunta al descriptor de transacción asociado al mismo.
tx.status	Estado de la transacción *tx*. Se consideran los estados TxIdle, TxActive, TxDoomed, TxCommiting.
tx.ReadSet	Conjunto de lectura de la transacción tx. Almacena las direcciones accedidas por las lecturas transaccionales.
tx.WriteSet	Conjunto de escritura de la transacción tx. Incluye un búfer de escritura, por lo que almacena las direcciones y valores de las escrituras transaccionales. De este modo, la asignación $tx.WriteSet[addr] \leftarrow value$ denota la introducción del valor $value$ y la dirección de memoria asociada $addr$ en el búfer de escritura. Reiniciar el conjunto de escritura ($tx.WriteSet \leftarrow \emptyset$) implica vaciar el búfer de escritura asociado. Si $addr \notin tx.WriteSet$, entonces $tx.WriteSet[addr]$ devuelve `null`. La misma notación se aplica a los conjuntos de reducción y sus búferes de reducción asociados.
tx.RdxSet	Conjunto de reducción de la transacción tx. Debe definirse un conjunto diferente para cada operador de reducción soportado. En la descripción del modelo se considerará un único operador de reducción $\oplus$.
glWTS	Array de *timestamps* de escritura compartido por todos los hilos de ejecución. La asignación $glWTS[addr] \leftarrow t$ denota que el *timestamp* asociado a la posición de memoria $addr$ es t. El valor de los *timestamps* depende del contador de orden global. Se utiliza una notación similar a la de los conjuntos de escritura y reducción a la hora de su reinicio y consulta. Si una posición a no ha sido escrita durante el *commit* de alguna transacción $glWTS[a] = 0$.
tx.RTS	Array de *timestamps* de lectura locales a cada hilo de ejecución. Dada una transacción tx, si $a \notin tx.RS$ entonces $tx.RTS[a] = 0$.
tx.rBf	Filtro de Bloom de lectura local a cada hilo de ejecución que permite representar las posiciones accedidas en el conjunto de lectura. Dada una transacción tx, la operación $tx.rBf \leftarrow addr$ inserta en el filtro la dirección $addr$. Los filtros pueden vaciarse $tx.rBf \leftarrow \emptyset$ y permiten consultar si una determinada dirección ha sido accedida en una lectura transaccional ($addr \in rBf$).
tx.uBf	Filtro de Bloom de actualización local a cada hilo de ejecución que permite representar las posiciones accedidas en los conjuntos de reducción y escritura. Dada una transacción tx, la operación $tx.uBf \leftarrow addr$ inserta en el filtro la dirección $addr$. Los filtros pueden vaciarse $tx.uBf \leftarrow \emptyset$ y permiten consultar si una determinada dirección ha sido accedida en una escritura o reducción transaccional ($addr \in uBf$).
$*addr$	Contenido de la posición de memoria apuntada por *addr*.
savePC&Env	Procedimiento que guarda el contador de programa, el puntero de pila y el entorno de ejecución. Se utiliza para crear un punto de restauración del sistema al que volver en caso de aborto. Similar a la función `setjmp()` del lenguaje C.
restorePC&Env	Procedimiento que restaura el contador de programa, el puntero de pila y el entorno de ejecución. Similar a la función `longjmp()` del lenguaje C.
MemoryFence	Primitiva que introduce una barrera de memoria, impidiendo reordenar las operaciones de memoria antes y después de la barrera. Similar a `intrinsic __sync_synchronize()` de GCC.

Tabla 3.3: Notación utilizada en la descripción de los algoritmos 1 y 2.

UNIVERSIDAD DE MÁLAGA

esperando indefinidamente una transacción —más prioritaria— inexistente. Si las transacciones se ejecutan de forma implícita, ReduxSTM garantiza el progreso del sistema asignando un orden de forma atómica basado en el momento de inicio o de llegada al *commit* de cada transacción.

ReduxSTM-TS

Denominamos ReduxSTM-TS al diseño basado en *timestamps*, que se muestra en el algoritmo 1.

ReduxSTM-TS utiliza una variable compartida *glOrder* a modo de reloj global entero. Dicha variable se incrementa cada vez que finaliza el *commit* de una transacción (línea 20 de TmEnd) y tiene un doble propósito: por una parte permite establecer las restricciones de orden entre transacciones, ya que cada transacción individual se lanza con un número de orden total y ascendente que debe coincidir con *glOrder* para que ésta pueda consolidar sus datos en memoria. Por otra parte permite determinar las versiones de los objetos accedidos transaccionalmente, sirviendo de base para mantener la coherencia del STM.

Las versiones de los objetos accedidos transaccionalmente se almacenan en estructuras tipo *array* accedidas mediante una función *hash* de la dirección de memoria correspondiente. En concreto, un *timestamp global de escritura*, $glWTS$, se encarga de almacenar un *timestamp* (número de versión) cada vez que un nuevo valor se consolida en memoria durante el *commit* de alguna transacción. Además existe un *timestamp local de lectura* por cada transacción, $tx.RTS$, que almacena el valor de *glOrder* cada vez que se realiza una operación de lectura. Los conflictos se comprueban durante la fase de *commit* comparando ambos *arrays*: si alguno de los objetos leídos por la transacción ($tx.RTS$) ha sido actualizado en un instante posterior a la versión almacenada, alguna otra transacción ha consolidado nuevos datos, por lo que la transacción en fase de *commit* debe abortar.

La figura 3.6 ilustra el funcionamiento de este STM. Tres transacciones concurrentes realizan cambios sobre los objetos compartidos A, B y C, inicialmente con valor 0, y suponemos $glOrder = 1$. Cada transacción se inicia con un número local de orden, que determina el orden en que deben realizarse la consolidación de los datos en memoria. Dicho de otro modo, la única ejecución permitida en este caso es $\alpha = \{tx1*, tx2*, tx3*\}$ o un prefijo de ésta.

La transacción 1 realiza dos lecturas a los objetos A y B utilizando la primitiva TxRead del algoritmo 1. Dado que no existía una escritura previa de los objetos por parte de la transacción —que permitiría obtener el valor directamente del *write-set*—, guarda la versión de los objetos leídos en su *timestamp* local RTS (línea 6

UNIVERSIDAD DE MÁLAGA

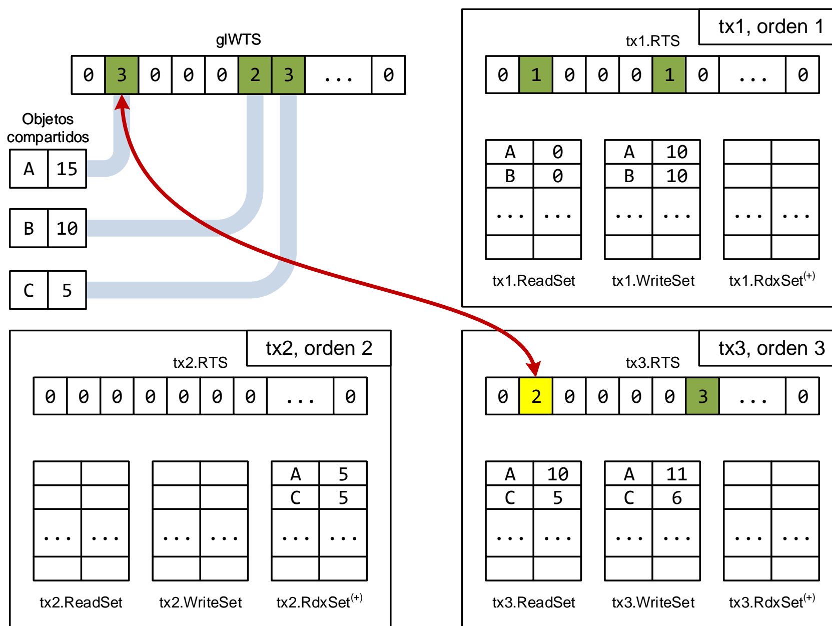


Figura 3.6: Ejemplo de funcionamiento de ReduxSTM-TS. Las transacciones 1 y 2 han finalizado realizando sus correspondientes operaciones sobre los objetos compartidos A, B y C. La transacción 3 ha detectado un conflicto dado que leyó un valor de A antiguo, posteriormente actualizado por la transacción 2. Se utiliza la misma función *hash* para mapear las direcciones en glWTS y tx.RTS. Las líneas azules que unen los objetos A, B y C con las celdas de glWTS representan este mapeo.

UNIVERSIDAD DE MÁLAGA

de TmRead). Esta acción permite saber el momento en que dichos datos fueron accedidos y validar el conjunto de lectura de la transacción en cualquier instante posterior: comparando las entradas correspondientes del *timestamp* global *glWTS* puede determinarse si los objetos leídos han sido actualizados posteriormente. Esta transacción también actualiza los objetos A y B con sendas operaciones de escritura. Dado que la gestión de versiones de ReduxSTM es *lazy*, los nuevos valores son almacenados en el *write-set* de la transacción hasta la fase de *commit* (función TmEnd). En dicha fase, la transacción ha de esperar a que su orden local coincida con *glOrder* (línea 2), validar su conjunto de lectura (líneas 5 a 9), actualizar los *timestamps* de escritura de A y B en *glWTS* —permitiendo que cualquier transacción pueda validar los nuevos cambios— (líneas 10 a 12), y consolidar los nuevos valores de A y B en memoria (líneas 13 a 18). La fase de *commit* finaliza incrementando *glOrder*, lo que tiene dos consecuencias: (1) la siguiente transacción (*tx2*) puede iniciar su fase de *commit* y (2) cualquier transacción que realice una lectura guardará en sus *timestamps* la *segunda* versión de los datos (*glOrder = 2*).

La transacción 2 es similar a la primera, pero en este caso la lectura y posterior actualización de los objetos A y C se realiza por medio de la nueva primitiva de reducción TmRdx. Esta primitiva verifica si existía una escritura anterior en la transacción (línea 2), en cuyo caso calcula el nuevo valor a consolidar en memoria, pero mantiene la entrada en el *write-set*, ya que el acceso de reducción no cumple una de las propiedades de reducción (el objeto de reducción no debe estar afectado por otras operaciones). Del mismo modo se comprueba si existe una operación de reducción anterior (línea 4) para actualizar el valor ξ en *RdxSet*. Si no se da ninguno de los casos anteriores, se copia el valor ξ a una nueva entrada del *RdxSet*. Es importante destacar que, a diferencia de lo que ocurrió en la transacción 1, en este caso no se actualiza el *timestamp* de lectura local, ya que no ha sido necesario acceder al valor en memoria del objeto de reducción. Dicho acceso se realizará durante la fase de *commit* permitiendo la coexistencia de operaciones de reducción sin producir conflictos.

La transacción 3 ilustra cómo se realiza la detección de conflictos en ReduxSTM-TS. La transacción realiza dos lecturas seguidas de sendas escrituras a los objetos A y C. En este caso la transacción obtiene el valor de A actualizado por *tx1* (10), lo que se ve reflejado en el *timestamp* de lectura correspondiente (2) que indica que la versión del objeto A accedida por *tx2* es la que existía cuando $glOrder = 2$. Tras acceder a C (esta vez a su valor actualizado por *tx3*), la transacción llega a la fase de *commit*, donde debe validar su conjunto de lectura. Es en esta validación donde se detecta que el valor de A —que fue accedido en su versión 2— se encuentra en ese momento en la versión 3, ya que fue actualizado por la reducción de *tx2*.

UNIVERSIDAD DE MÁLAGA

Algoritmo 1 ReduxSTM-TS — Versión basada en *timestamps*

```
1  function TmInit(nThreads)
2      glOrder ← 1                              ▷ Inicializa orden global
3      glTx ← AllocateDescriptors(nThreads)     ▷ Reserva descriptores para las transacciones
4      glWTS ← ∅                                ▷ Inicializa el array de timestamps global
5  end function
```

```
1  function TmInitThread( )
2      tx ← glTx[myThreadId]                    ▷ Asigna descriptor de transacción al hilo
3      tx.status ← TxIdle
4  end function
```

```
1  function TmBegin(order)
2      tx.env ← savePC&Env( )                   ▷ Guarda estado de la ejecución
3      tx.ReadSet ← ∅                           ▷ Vacía conjuntos de lectura, escritura y reducción
4      tx.WriteSet ← ∅
5      tx.RdxSet ← ∅
6      tx.RTS ← ∅                               ▷ Inicializa el array de timestamps local
7      tx.order ← order                         ▷ Asigna orden explícito a la transacción
8      tx.status ← TxActive
9  end function
```

```
1  function TmRead(addr)
2      if addr ∈ tx.WriteSet then               ▷ Si el objeto ha sido escrito, devuelve el valor
3          value ← tx.WriteSet[addr]            ▷ directamente, sin añadir la dirección al conjunto
4          return value                         ▷ de lectura
5      end if
6      tx.RTS[addr] ← glOrder                   ▷ Actualiza timestamp local de lectura
7      tx.ReadSet ← tx.ReadSet ⋃ {addr}         ▷ Añade la dirección al conjunto de lectura
8      if addr ∈ tx.RdxSet then                 ▷ Si existía reducción previa, genera valor correcto
9          value ← (*addr) ⊕ tx.RdxSet[addr]
10     else                                     ▷ En caso contrario, lee el valor de memoria
11         value ← (*addr)
12     end if
13     return value
14 end function
```

```
1  function TmWrite(addr, value)
2      if addr ∈ tx.RdxSet then                 ▷ Si existía una reducción previa, se descarta
3          tx.RdxSet ← tx.RdxSet − { addr }
4      end if
5      tx.WriteSet[addr] ← value                ▷ Añade o actualiza entrada en el WriteSet
6  end function
```

```
1  function TmRdx(addr, ξ)
2      if addr ∈ tx.WriteSet then               ▷ Si existía una escritura previa, actualiza valor
3          tx.WriteSet[addr] ← ξ ⊕ tx.WriteSet[addr]
4      else if addr ∈ tx.RdxSet then            ▷ Si existía una reducción previa, actualiza valor
5          tx.RdxSet[addr] ← ξ ⊕ tx.RdxSet[addr]
6      else                                     ▷ Si no existía una entrada previa, copia valor
7          tx.RdxSet[addr] ← ξ
8      end if
9  end function
```

UNIVERSIDAD DE MÁLAGA

Algoritmo 1 ReduxSTM-TS — Versión basada en *timestamps* (continuación)

```
1  function TmRollback( )                              ▷ Aborto y reinicio
2      tx.status ← TxIdle
3      restorePC&Env(tx.env)                           ▷ Restaura ejecución al inicio de TmBegin
4  end function
```

```
1  function TmEnd( )
2      while tx.order ≠ glOrder do                     ▷ Espera del turno
3      end while

4      tx.status ← TxCommitting                        ▷ Inicio de la fase de commit

5      for all addr ∈ tx.ReadSet do                    ▷ Validación
6          if tx.RTS[addr] < glWTS[addr] then
7              StmRollback( )
8          end if
9      end for

10     for all addr ∈ tx.RdxSet ⋃ tx.WriteSet do
11         glWTS[addr] ← glOrder +1                    ▷ Actualización del timestamp global
12     end for

13     for all addr ∈ tx.RdxSet do                     ▷ Consolidación de reducciones en memoria
14         *addr ← *addr ⊕ tx.RdxSet[addr]
15     end for
16     for all addr ∈ tx.WriteSet do                   ▷ Consolidación de escrituras en memoria
17         *addr ← tx.WriteSet[addr]
18     end for

19     tx.status ← TxIdle                              ▷ Finaliza fase de commit
20     Fetch&Add(glOrder, 1)                           ▷ La siguiente transacción puede finalizar
21 end function
```

```
1  function TmExitThread( )
2      tx.status ← TxNone
3  end function
```

```
1  function TmExit( )
2      DeallocateDescriptors(glTx)
3  end function
```

UNIVERSIDAD DE MÁLAGA

Esto produce un fallo en la validación que ocasiona el aborto de la transacción 3 (líneas 6 y 7 de TmEnd).

Finalmente la ejecución realizada corresponde a $\alpha' = \{tx1*, tx2*\}$, que es un prefijo de α. Posteriormente *tx3* será reejecutada con éxito completando la ejecución. Esta restricción a una única ejecución α válida, combinada con la gestión de versiones *lazy*, es la que permite la coexistencia entre sí de operaciones de escritura y reducción, así como de falsas dependencias R – W y R – RDX sin originar conflictos (ver tabla 3.2). Dos transacciones que realicen operaciones de actualización (escrituras y reducciones) a un mismo objeto no comprometen la consistencia del sistema ya que (1) existe una relación estricta de orden entre las mismas, lo que impide que la transacción posterior actualice previamente el objeto y (2) la consolidación de los datos en memoria se produce durante la fase de *commit* —que implica la finalización de todas las transacciones anteriores—, lo que impide que otras transacciones puedan ver una ejecución incorrecta. La contrapartida es la necesidad de serializar la fase de *commit* de las transacciones, y la pérdida potencial de rendimiento derivada de considerar válida una única ejecución.

ReduxSTM-CTI

Denominamos ReduxSTM-CTI (*commit-time invalidation*) al diseño basado en invalidación, que se muestra en el algoritmo 2.

En ReduxSTM-CTI la transacción en fase de *commit* tiene la posibilidad de invalidar al resto de transacciones activas si detecta un conflicto con ellas. Dado que las transacciones coexistentes con una transacción en fase de *commit* son necesariamente posteriores —debido a las restricciones de orden—, no es necesario mantener las versiones accedidas de cada objeto, pudiéndose utilizar directamente la información de los conjuntos de datos de las transacciones.

ReduxSTM-CTI utiliza un reloj global entero *glOrder* con un funcionamiento idéntico al de ReduxSTM-TS. Cuando una transacción llega a su fase de *commit* (TmEnd) y puede continuar (líneas 2 a 4), comienza actualizando los objetos tentativos en la memoria compartida con los datos almacenados en sus búferes de escritura y de reducción (líneas 7 a 12). Una vez actualizada la memoria, la transacción debe comprobar si los datos consolidados afectan a alguna otra transacción activa, comprobando si alguna otra transacción en ejecución contiene alguna de las posiciones actualizadas en su conjunto de lectura (líneas 14 a 20). En caso afirmativo, la transacción en cuestión es marcada como *inválida* (línea 17), y será abortada en una llamada posterior a la función CheckKilled. Dado que cada

UNIVERSIDAD DE MÁLAGA

hilo ejecuta una única transacción en un instante determinado, la transacción en fase de *commit* puede invalidar un máximo de $nThreads - 1$ transacciones.

Al no requerir números de versión, ReduxSTM-CTI no necesita las estructuras adicionales del algoritmo basado en *timestamps*. Sin embargo, para aumentar la eficiencia en la detección de conflictos, cada transacción utiliza dos filtros de Bloom, que almacenan el conjunto de posiciones leídas (*rBf*) y actualizadas (*uBf*). Estas estructuras son simples *arrays* de bits que se inicializan a cero al comienzo de la transacción y que se actualizan con cada operación transaccional. La actualización consiste en activar uno de sus bits que se determina mediante una función *hash* de la dirección de memoria del objeto accedido. Durante el *commit*, la detección de conflictos se lleva a cabo mediante la intersección del *uBf* de la transacción en fase de *commit* con el *rBf* del resto de transacciones activas. Si el resultado de la intersección no es un conjunto vacío, al menos una de las posiciones actualizadas por la transacción en fase de *commit* ha sido leída por una transacción posterior, por lo que ésta última será invalidada. Los detalles de implementación de estos filtros y sus ventajas y limitaciones se detallan en la sección 3.3.3.

La figura 3.7 ilustra el funcionamiento de ReduxSTM-CTI: tres transacciones concurrentes realizan cambios sobre los objetos compartidos A, B y C, inicialmente con valor 0, y suponemos $glOrder = 1$. Cada transacción se inicia con un número local de orden, que determina el orden en que debe realizarse la consolidación de los datos en memoria. La única ejecución permitida en este caso es $\alpha = \{tx1*, tx2*, tx3*\}$ o un prefijo de ésta.

La transacción 1 realiza dos lecturas a los objetos A y B utilizando la primitiva TmRead del algoritmo 2. Dado que no existía una escritura previa de los objetos por parte de la transacción —que permitiría obtener su valor directamente del *write-set*—, añade las direcciones de los mismos al filtro de lectura *rBf* (línea 7 de la función TmRead). Esta inserción permite que cualquier transacción anterior en fase de *commit* pueda detectar posibles conflictos. La transacción 1 también actualiza los objetos A y B con sendas operaciones de escritura, lo que implica añadir las direcciones al filtro de actualización *uBf* (línea 2 de la función TmWrite). Dado que la gestión de versiones de ReduxSTM es *lazy*, los nuevos valores son almacenados en el *write-set* de la transacción hasta la fase de *commit* (función TmEnd). En dicha fase, la transacción ha de esperar a que su orden local coincida con *glOrder* (línea 2) y comprobar si ha sido invalidada por alguna transacción anterior (líneas 3 y 5). Si la transacción sigue siendo válida una vez tiene el turno de *commit*, ninguna otra transacción puede invalidarla, por lo que procede a consolidar los nuevos valores de A y B en memoria (líneas 7 a 12). Una vez actualizada la memoria, se realiza la detección de conflictos con el resto de

UNIVERSIDAD DE MÁLAGA

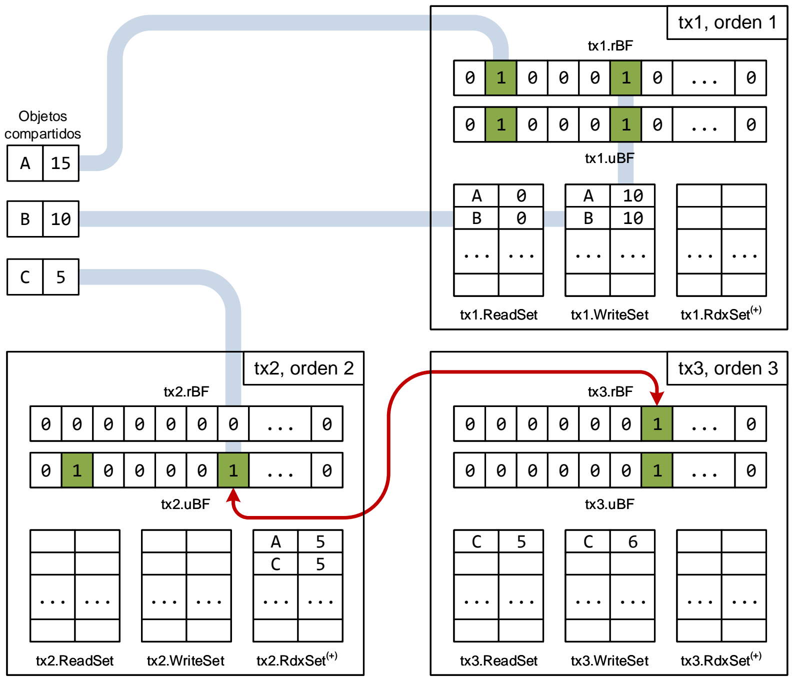


Figura 3.7: Ejemplo de funcionamiento de ReduxSTM-CTI. Las transacciones 1 y 2 han finalizado realizando sus correspondientes operaciones sobre los objetos compartidos A, B y C. La transacción 2 detecta un conflicto con la transacción 3 a mediante la intersección de sus filtros de Bloom y la invalida. Se utiliza la misma función *hash* para mapear los objetos accedidos en rBf y uBf. Las líneas azules que unen los objetos A, B y C con las celdas de rBf y uBf representan este mapeo.

UNIVERSIDAD DE MÁLAGA

Algoritmo 2 ReduxSTM-CTI — Versión basada en invalidación

```
1  function TmInit(nThreads)
2      glOrder ← 1                                  ▷ Inicializa orden global
3      glTx ← AllocateDescriptors(nThreads)
4  end function
```

```
1  function TmInitThread( )
2      tx ← glTx[myThreadId]                        ▷ Asigna descriptor de transacción al hilo
3      tx.status ← TxIdle
4  end function
```

```
1  function TmBegin(order)
2      tx.ReadSet ← ∅                               ▷ Vacía conjuntos de lectura, escritura y reducción
3      tx.WriteSet ← ∅
4      tx.RdxSet ← ∅
5      tx.rBf ← ∅                                   ▷ Vacía filtros de lectura y actualización
6      tx.uBf ← ∅
7      tx.order ← order                             ▷ Asigna orden a la transacción
8      tx.env ← savePC&Env( )                       ▷ Guarda estado de la ejecución
9      tx.status ← TxActive
10 end function
```

```
1  function TmRead(addr)
2      if addr ∈ tx.WriteSet then                   ▷ Si el objeto ha sido escrito, devuelve el valor
3          value ← tx.WriteSet[addr]                ▷ directamente, sin añadir la dirección al conjunto
4          return value                             ▷ de lectura
5      end if
6      tx.ReadSet ← tx.ReadSet ⋃ {addr}             ▷ Añade dirección al conjunto de lectura
7      tx.rBf ← addr                                ▷ Inserta dirección en el filtro de lectura
8      MemoryFence( )                               ▷ Evita reordenación en el resto de hilos
9      if addr ∈ tx.RdxSet then                     ▷ Si existía una reducción previa, genera el valor
                                                      correcto
10         value ← (*addr) ⊕ tx.RdxSet[addr]
11     else
12         value ← (*addr)                          ▷ Lee el valor de memoria
13     end if
14     return value
15 end function
```

```
1  function TmWrite(addr, value)
2      uBf ← addr                                   ▷ Inserta dirección al conjunto de actualización
3      if addr ∈ tx.RdxSet then
4          tx.RdxSet ← tx.RdxSet − { addr }         ▷ Si existía una reducción, migra entrada al wset
5      end if
6      tx.WriteSet[addr] ← value (⇒ tx.WriteSet ← tx.WriteSet ⋃ {addr} )
7  end function
```

```
1  function TmRdx(addr, ξ)
2      uBf ← addr                                   ▷ Inserta dirección al conjunto de actualización
3      if addr ∈ tx.WriteSet then                   ▷ Si existía una escritura, actualiza valor en wset
4          tx.WriteSet[addr] ← ξ ⊕ tx.WriteSet[addr]
5      else if addr ∈ tx.RdxSet then                ▷ Si existía una reducción, actualiza valor en rdxset
6          tx.RdxSet[addr] ← ξ ⊕ tx.RdxSet[addr]
7      else
8          tx.RdxSet[addr] ← ξ (⇒ tx.RdxSet ← tx.RdxSet ⋃ {addr})
9      end if
10 end function
```

UNIVERSIDAD DE MÁLAGA

Algoritmo 2 ReduxSTM-CTI — Versión basada en invalidación (continuación)

```
 1 function TmRollback( )                          ▷ Aborto y reinicio
 2     tx.status ← TxIdle
 3     restorePC&Env(tx.env)                       ▷ Restaura ejecución al inicio de TmBegin
 4 end function
```

```
 1 function CheckKilled( )                         ▷ Validación
 2     if tx.status = TxDoomed then
 3         StmRollback( )
 4     end if
 5 end function
```

```
 1 function TmEnd( )
 2     while tx.order ≠ glOrder do                 ▷ Espera del turno
 3         CheckKilled( )
 4     end while

 5     CheckKilled( )                              ▷ Validación
 6     tx.status ← TxCommitting                    ▷ Inicio de la fase de commit

 7     for all addr ∈ tx.RdxSet do                 ▷ Consolidación de reducciones en memoria
 8         *addr ← *addr ⊕ tx.RdxSet[addr]
 9     end for
10     for all addr ∈ tx.WriteSet do               ▷ Consolidación de escrituras en memoria
11         *addr ← tx.WriteSet[addr]
12     end for

13     MemoryFence( )

14     for all otherTx ∈ globalTx do               ▷ Detección de conflictos e invalidación
15         if otherTx.status = TxActive then
16             if tx.uBf ⋂ otherTx.rBf ≠ ∅ then
17                 otherTx.status ← TxDoomed       ▷ Invalidación de la transacción implicada
18             end if
19         end if
20     end for

21     tx.status ← TxIdle                          ▷ Finaliza fase de commit
22     Fetch&Add(glOrder, 1)                       ▷ La siguiente transacción puede finalizar
23 end function
```

```
 1 function TmExitThread( )
 2     tx.status ← TxNone
 3 end function
```

```
 1 function TmExit( )
 2     DeallocateDescriptors(glTx)
 3 end function
```

UNIVERSIDAD DE MÁLAGA

transacciones del sistema (líneas 14 a 20). Para ello se realiza la intersección del filtro de actualización *uBf* de la transacción 1 con el filtro de lectura de cada una de las transacciones activas en el sistema en ese momento (tx2 y tx3). Si existe intersección, la transacción correspondiente debe ser invalidada cambiando su estado a TmDoomed. Una transacción en este estado será abortada la próxima vez que compruebe su validez con la función CheckKilled. El *commit* de la transacción 1 finaliza incrementando *glOrder*, lo que permite que la siguiente transacción (*tx2*) pueda iniciar su fase de *commit* una vez finalice sus operaciones.

La transacción 2 es similar a la primera, pero en este caso la lectura y posterior actualización de los objetos A y C se realiza por medio de la nueva primitiva de reducción TmRdx. Esta primitiva verifica si existía una escritura anterior en la transacción (línea 3), en cuyo caso calcula el nuevo valor a consolidar en memoria, pero mantiene la entrada en el *write-set*, ya que el acceso de reducción no cumple una de las propiedades de reducción (el objeto de reducción no debe estar afectado por otras operaciones). Del mismo modo se verifica que si existe una operación de reducción anterior (línea 5) para actualizar el valor ξ en *RdxSet*. Si no se da ninguno de los casos anteriores, se copia el valor ξ a una nueva entrada del *RdxSet*. En cualquier caso, la dirección se inserta en el filtro de actualización *uBf* para la posterior detección de conflictos. Esta detección puede observarse una vez la transacción 2 lleva a la fase de *commit*. En este caso, al realizar la intersección entre su *uBf* y el *rBf* de la transacción 3 detectará un conflicto con una lectura anterior del objeto C, por lo que la transacción 3 es invalidada, y será abortada una vez llegue al inicio de su fase de *commit* (línea 5 de la función TmEnd).

En este ejemplo, la ejecución realizada corresponde a $\alpha' = \{tx1*, tx2*\}$, que es un prefijo de α. Eventualmente *tx3* será reejecutada con éxito completando la ejecución. Del mismo modo que en ReduxSTM-TS, la restricción a una única ejecución α válida, combinada con la gestión de versiones *lazy*, permite la coexistencia entre sí de operaciones de escritura y reducción, así como de falsas dependencias R – W y R – RDX sin originar conflictos. Es importante señalar que, si bien el uso de los filtros de Bloom no es estrictamente necesario, ya que es posible realizar la fase de detección de conflictos verificando directamente el contenido de los conjuntos de datos de las transacciones; su uso resulta crucial para obtener un rendimiento competitivo, al reducir la complejidad de la fase de detección de conflictos de $O(n^3)$ a $O(n)$.[12]

[12]Si no se usan los filtros de Bloom, es necesario comprobar cada lectura de cada transacción de cada hilo con cada escritura de la transacción en fase de *commit*.

UNIVERSIDAD DE MÁLAGA

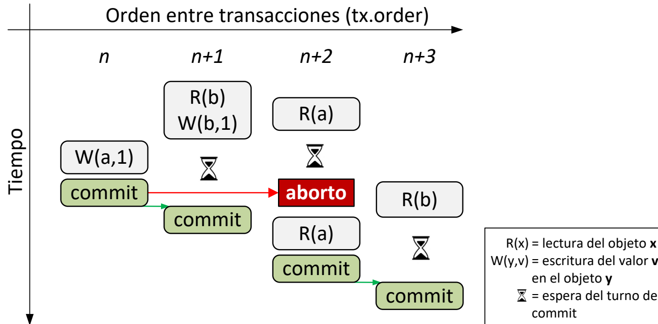


Figura 3.8: Sincronización en ReduxSTM: *commits* ordenados.

3.3.3. Implementación

En el diseño de un sistema TM software, además del algoritmo utilizado, buena parte del rendimiento proviene de los detalles particulares de la implementación. En esta sección discutimos algunos detalles de nuestra propuesta.

Mecanismos de sincronización

En ReduxSTM las transacciones no utilizan *locks* para adquirir la propiedad de las posiciones de memoria. En este sentido ambos diseños pueden calificarse como non-OREC [55]. El mecanismo que garantiza la atomicidad del sistema es la combinación de su diseño con gestión de versiones *lazy* y la espera del turno de las transacciones durante el inicio de su fase de *commit*. Esta espera es la que habilita la secuencialización de los *commits* aplicando las restricciones de orden, como muestra la figura 3.8.

De hecho este bucle de espera se comporta en la práctica como un *spin-lock* que evita conflictos a la hora de actualizar la memoria compartida. Esta solución se adapta a las restricciones establecidas en la sección 3.3.1 a expensas de serializar la fase de *commit* de las transacciones, por lo que la duración de la misma se convierte en un aspecto crítico de cara al rendimiento final de la implementación.

UNIVERSIDAD
DE MÁLAGA

Barreras de memoria

Los sistemas multinúcleo actuales utilizan modelos de memoria relajados que no garantizan la consistencia secuencial entre instrucciones ejecutadas por núcleos diferentes [1]. Esto hace necesario incluir ciertas sincronizaciones adicionales vía barreras de memoria [103].

Observemos el protocolo de *commit-time invalidation* del algoritmo 2. Consideremos una transacción t_n en fase de commit y, simultáneamente, otra transacción t_m que está realizando una lectura transaccional. Para mantener la consistencia del sistema TM, el par de acciones:

- $t_n.commit$: (1) consolidar los valores tentativos en memoria compartida, (2) comprobar el conjunto de lectura del resto de transacciones.
- $t_m.read$: (1) actualizar el conjunto de lectura, (2) leer valor tentativo de memoria.

deben ser vistas exactamente en ese orden por cada hilo respecto al otro.

La figura 3.9 ilustra esta idea: la transacción $t_{0,1}$ escribe transaccionalmente la variable A y llega a su fase de *commit*, donde (1) hace visibles sus cambios y (2) comprueba —e invalida en su caso— el resto de transacciones concurrentes. El resto de transacciones[13] representan las tres posibles situaciones que pueden ocurrir simultáneamente:

- La transacción $t_{1,1}$ realiza una lectura en A (incluyendo la actualización de su rBf) previa al *commit* de $t_{0,1}$. En este caso, el conflicto es detectado por $t_{0,1}$, que invalidará $t_{1,1}$ provocando su aborto la próxima vez que ésta compruebe su validez (en el algoritmo 2 dicha comprobación se produce al iniciar su fase de *commit*).
- La transacción $t_{2,1}$ realiza una lectura en A coincidiendo con la fase de *commit* de $t_{0,1}$. De este modo, la inserción de la dirección de A en el rBf se produce después de que el A haya sido actualizado en memoria, pero antes de la fase de invalidación de $t_{0,1}$. En este caso, aunque $t_{2,1}$ lee el valor correcto de A, es igualmente invalidada por $t_{0,1}$. Este comportamiento puede invalidar ocasionalmente una transacción correcta, pero evita el uso de *locks* que podrían enlentecer el caso común.

[13]En este caso, las transacciones $t_{1,1}$, $t_{2,1}$ y $t_{3,1}$ se inician con orden 1 para simplificar el ejemplo. En una ejecución real el número de orden de las transacciones debe ser único.

UNIVERSIDAD DE MÁLAGA

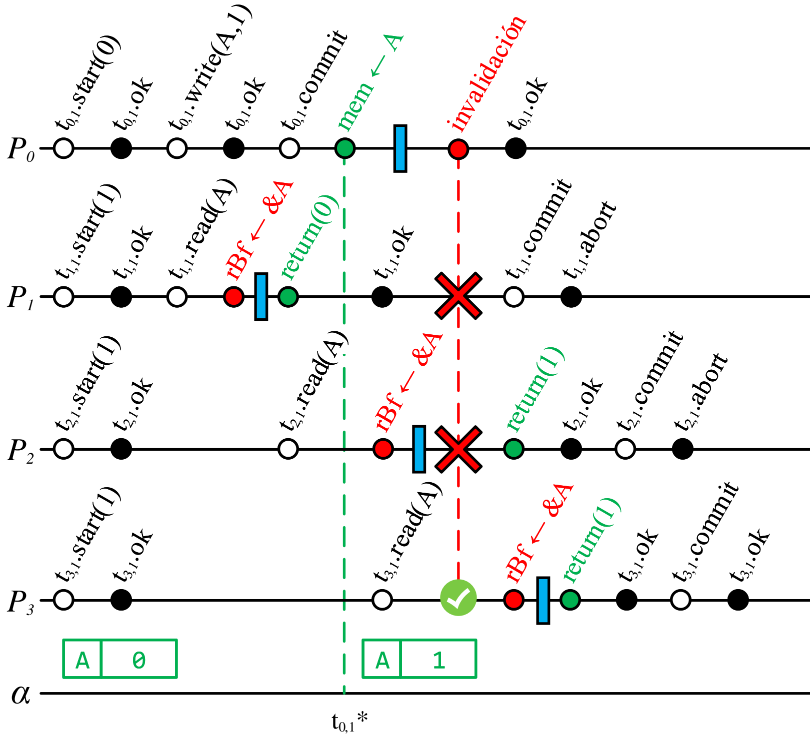


Figura 3.9: Sincronización en ReduxSTM-CTI. La línea verde representa el instante en que A es actualizado en memoria, la línea roja representa el instante en que se produce la invalidación de las transacciones por parte de $t_{0,1}$. Las barreras de memoria se representan mediante barras azules. Además de las invocaciones y respuestas representadas por círculos blancos y negros, se incluyen eventos adicionales representados por círculos rojos (invalidación de las transacciones y actualización de los filtros de Bloom de lectura) y círculos verdes (volcado de los datos a memoria principal y valor tentativo leído por las transacciones). Una actualización del filtro de Bloom anterior a la línea roja será detectada durante la invalidación. Un valor tentativo leído antes de la línea verde devolverá el valor inicial de A.

UNIVERSIDAD DE MÁLAGA

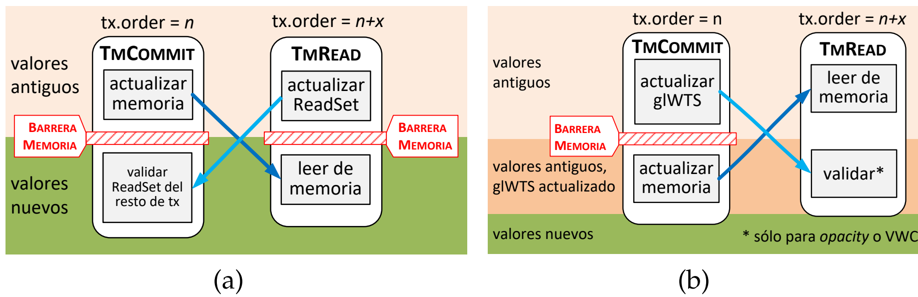


Figura 3.10: Sincronización en ReduxSTM: (a) Barreras de memoria en ReduxSTM-CTI. (b) Barreras de memoria para garantizar *opacity* o *strict VWC* en ReduxSTM-TS.

- La transacción $t_{3,1}$ realiza una lectura de A (incluyendo la actualización de su rBf) después del *commit* de $t_{0,1}$. En este caso $t_{0,1}$ no invalidará la transacción debido a que la actualización del rBf de $t_{3,1}$ no se ha llevado a cabo aún. Por tanto, $t_{3,1}$ lee un valor correcto y puede finalizar sin conflictos.

En cualquiera de los tres casos, si se produce un conflicto potencial, o bien la comprobación de conflictos se realiza después de actualizar el conjunto de lectura (inserción de la dirección en el rBf) o bien se lee un valor correcto de memoria compartida, obteniendo en cualquier caso una historia correcta. Para que esto ocurra, sin embargo, es necesario garantizar que cada hilo percibe estas acciones en su orden correcto. Para ello se utilizan el par de barreras de memoria que muestra la figura 3.10 (a). Estas barreras aparecen representadas mediante barras azules en el ejemplo de la figura 3.9.

La variante de ReduxSTM basada en *timestamps* (algoritmo 1) no requiere ninguna barrera de memoria salvo para añadir criterios de consistencia como *opacity* o *VWC*, como se detalla en la sección 3.3.3. Esto se debe a que en este algoritmo las transacciones actualizan y validan su conjunto transaccional durante la fase de *commit* —respecto a transacciones ya finalizadas— (ver figura 3.10 (b)). En este caso la serialización de la fase de *commit* por medio de la instrucción FETCH&ADD (línea 20 del algoritmo 1) lleva implícita una barrera de memoria que es suficiente para garantizar la consistencia del sistema.

La figura 3.11 ilustra esta idea: la transacción $t_{0,1}$ escribe transaccionalmente la variable A y llega a su fase de *commit*, donde (1) valida sus operaciones de lectura comparando su *tx.RTS* con *glWTS*, (2) actualiza *glWTS* con las direcciones de sus conjuntos de escritura y reducción y (3) hace visibles sus cambios. Dado

UNIVERSIDAD DE MÁLAGA

que el orden global (*ord*) serializa la fase de *commit* de las transacciones, el resto de transacciones[14] no pueden comenzar su *commit* hasta que $t_{0,1}$ finalice:

- La transacción $t_{1,1}$ realiza una lectura en A (incluyendo la actualización de su RTS) previa al *commit* de $t_{0,1}$. Esto implica que (1) el orden global que inserta en RTS es 0 y (2) lee un valor de A incorrecto. El conflicto será detectado por $t_{1,1}$ en su fase de validación, durante su *commit*, ya que $tx.RTS(\&A) < glWTS(\&A)$.
- La transacción $t_{2,1}$ realiza una lectura en A coincidiendo con la fase de *commit* de $t_{0,1}$. Esta vez la actualización del RTS se produce antes de que $t_{0,1}$ actualice el orden global —se almacena el valor de orden global previo en RTS—, pero $t_{2,1}$ lee el valor actualizado de A. En este caso $t_{2,1}$ detectará un conflicto durante su fase de validación aun habiendo accedido a un valor correcto. Este comportamiento es similar al del algoritmo ReduxSTM-CTI, permitiendo abortar ocasionalmente una transacción correcta para evitar el uso de *locks*.
- La transacción $t_{3,1}$ realiza una lectura de A (incluyendo la actualización de su RTS) después del *commit* de $t_{0,1}$. En este caso el orden insertado en RTS será el correcto, por lo que $t_{3,1}$ puede finalizar sin conflictos.

En este algoritmo, la serialización de la fase de *commit* de las transacciones, que se muestra en la figura mediante las barras azules al inicio y al final de la fase de *commit*, es suficiente para garantizar la consistencia del sistema.

Estructuras de datos para búferes de escritura y reducción

Hasta el momento los búferes de escritura y de reducción han sido descritos como estructuras de datos con el objeto de almacenar los valores tentativos que serán consolidados en memoria si la transacción finaliza correctamente. Nuestra implementación utiliza una estructura unificada para ambos búferes, que denominamos *OpSet* y se muestra en la figura 3.12 (a). Dicha estructura se compone de una tabla *hash* implementada en forma de matriz de pares *(dirección, valor)* con estructuras adicionales para optimizar las operaciones más comunes. Dado que la granularidad utilizada por ReduxSTM es de 8 bytes alineados[15], los tres bits menos significativos del campo de dirección no aportan información

[14]En este caso, las transacciones $t_{1,1}$, $t_{2,1}$ y $t_{3,1}$ se inician con orden 1 para simplificar el ejemplo. En una ejecución real el número de orden de las transacciones debe ser único.

[15]Por ser el tamaño de palabra estándar en sistemas de 64 bits.

UNIVERSIDAD DE MÁLAGA

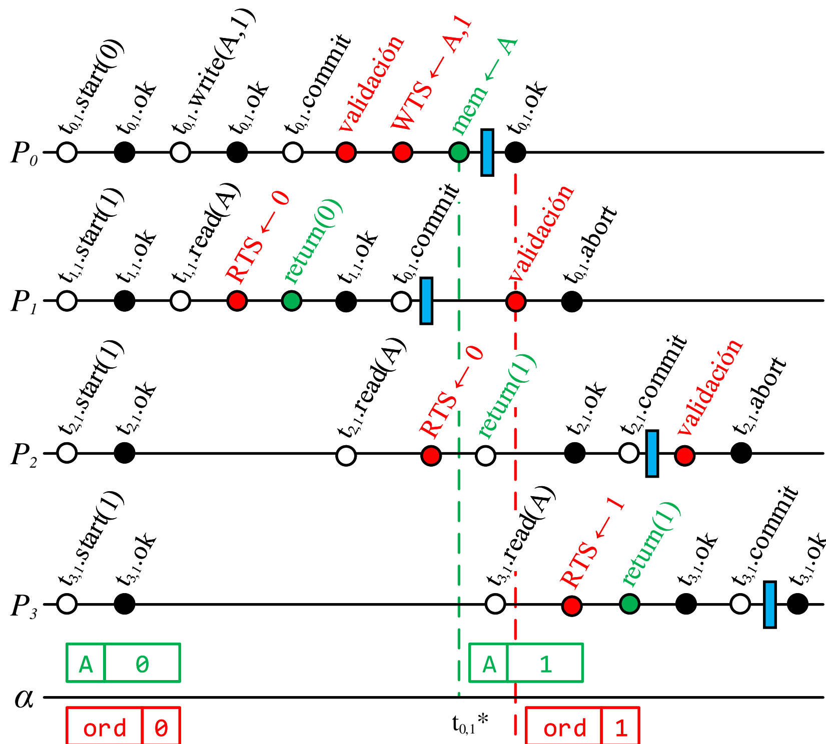


Figura 3.11: Sincronización en ReduxSTM-TS. La línea verde representa el instante en que A es actualizado en memoria, la línea roja actualiza el orden global, permitiendo a la siguiente transacción comenzar a validar sus lecturas transaccionales. Las barreras de memoria se representan mediante barras azules. Además de las invocaciones y respuestas representadas por círculos blancos y negros, se incluyen eventos adicionales representados por círculos rojos (validación de las transacciones y actualizaciones de los *timestamps* de lectura y escritura) y círculos verdes (volcado de los datos a memoria principal y valor tentativo leído por las transacciones). Una actualización del *timestamp* de lectura anterior a la línea roja producirá un conflicto durante la validación de esa transacción. Un valor tentativo leído antes de la línea verde devolverá el valor inicial de A.

UNIVERSIDAD DE MÁLAGA

adicional, y se han utilizado para codificar el conjunto (escritura o reducción) al que pertenece el valor almacenado en el par correspondiente. De este modo, los búferes de escritura y de reducción quedan unificados en la misma estructura[16], lo que resulta conveniente no sólo a efectos de ahorro de espacio, sino a la hora de migrar de elementos de un conjunto a otro, al requerir sólo cambiar éstos últimos tres bits.

Las operaciones necesarias sobre esta estructura de datos son principalmente la inserción y la búsqueda de pares dada una dirección y el recorrido secuencial de todos los pares de la misma (utilizado para la consolidación de los datos durante la fase de *commit*). Cada línea de la tabla se selecciona aplicando una función *hash* a la dirección de memoria sobre la que se quiera operar. Las inserciones implican reutilizar una entrada si la dirección asociada estaba ya en el conjunto: esta opción proporciona un mayor rendimiento que la alternativa de añadir todas las entradas con la misma dirección a modo de *log*. En cada *commit* los conjuntos de escritura y reducción deben ser recorridos secuencialmente. Para ello, los *arrays* auxiliares `row_count` e `inserted` se utilizan como índices para recorrer la estructura de forma eficiente: el primero almacena el número de elementos que contiene cada fila de la tabla, facilitando la inserción y la búsqueda de elementos; el segundo indexa qué líneas de la tabla contienen datos, evitando tener que recorrer toda la tabla durante la fase de consolidación. Nótese que, aunque los algoritmos 1 y 2 muestran la consolidación de los conjuntos de escritura y reducción como operaciones diferenciadas, en la implementación estas acciones ocurren simultáneamente, determinando para cada par la operación a realizar durante su actualización en memoria.

Las dimensiones de esta estructura tienen una clara influencia en el rendimiento del sistema transaccional. Idealmente, aumentar el número de filas del *OpSet* reduce el número de colisiones de la función *hash* y, por tanto, la necesidad de recorrer la fila secuencialmente para buscar una dirección concreta. Sin embargo, aumentar el número de filas incrementa el coste de reiniciar la estructura (lo que implica reiniciar el *array* `row_count` y la variable `count`) cada vez que una transacción aborta o se inicia, además de reducir la explotación de la localidad espacial y temporal de la caché.

Con el objeto de aumentar la escalabilidad y reducir los fallos de página y de caché, la matriz de pares está organizada en *planos*, como muestra la figura 3.12 (a). De esta forma, el acceso al elemento `OpSet`$[f][c]$ se realiza como `OpSet`$[c/2^p][f][c \mod 2^p]$, donde 2^p indica el número de entradas de la fila

[16]Los 3 bits permiten 1 conjunto de escritura y hasta 7 conjuntos de reducción para operadores de reducción diferentes.

UNIVERSIDAD DE MÁLAGA

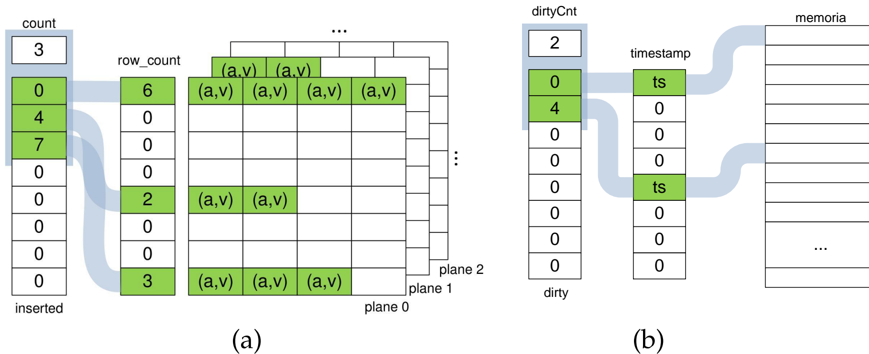


Figura 3.12: (a) Estructura unificada de tabla *hash* para almacenar los conjuntos de escritura y de reducción (OpSet). (b) Estructura para almacenar los *timestamps* en ReduxSTM-TS.

de un plano dado. Experimentalmente los mejores resultados en cuanto a rendimiento se han obtenido con un tamaño de plano de una o dos páginas de memoria.

Estructuras de datos para timestamps (ReduxSTM-TS)

Los *timestamps* del algoritmo 1 ($glWTS$ y $tx.RTS$) se almacenan en una estructura similar al *OpSet*, que se muestra en la figura 3.12 (b). En este caso sólo se necesita almacenar un valor por entrada (el *timestamp* asociado a la misma) y ésta se calcula aplicando una función *hash* a la dirección de memoria asociada. En esta estructura, el *array* `dirty` tiene una función similar al `inserted` del *OpSet*, y permite acelerar el proceso de validación, comprobando únicamente las entradas del *timestamp* que se hayan actualizado durante la ejecución de la transacción. Durante la fase de *commit* la estructura asociada a $tx.RTS$ se recorre en orden para validar el conjunto de lectura.

Dado el uso de funciones *hash*, todas las direcciones que produzcan el mismo *hash* se consideran alias a la hora de asociarles un *timestamp*. Consecuentemente pueden producirse falsos conflictos debidos a este alias, cuya probabilidad de aparición será inversamente proporcional al tamaño de la estructura que almacene los *timestamps*.

Como consideración adicional, si las estructuras de datos para el *OpSet* y los *timestamps* tienen las mismas dimensiones, se pueden añadir dos optimizaciones:

UNIVERSIDAD DE MÁLAGA

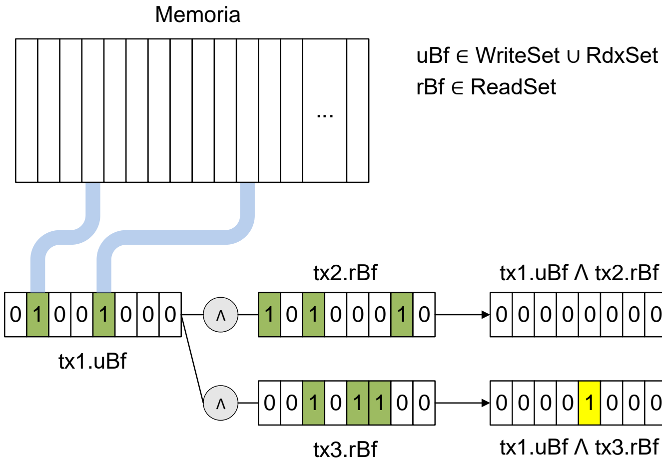


Figura 3.13: Detección de conflictos mediante filtros de Bloom en ReduxSTM-CTI.

por una parte se puede utilizar la misma función *hash* para ambos escenarios; además se puede usar el mismo bucle durante la fase de *commit* para consolidar los datos en memoria y actualizar el *timestamp* global *glWTS* en el algoritmo 1, permitiendo fusionar los tres bucles de las líneas 10–18 de la función TmEnd.

Estructuras para los conjuntos de datos (ReduxSTM-CTI)

Aunque cada *OpSet* contiene información de pertenencia en los conjuntos de escritura y reducción, recorrerlo completamente durante la etapa de invalidación resulta ineficiente en la implementación *commit-time invalidation*. Para aumentar el rendimiento, esta implementación utiliza filtros de Bloom [8, 16, 104, 88]. Cada transacción utiliza un filtro para representar su conjunto de lectura $tx.rBf$ y otro para sus conjuntos de escritura y reducción $tx.uBf \hookleftarrow tx.WriteSet \cup tx.RdxSet$. Estos filtros se han implementado como un *array* de valores sin signo, donde cada bit individual representa un conjunto de direcciones de memoria mapeadas a través de una única función *hash*. Es necesario realizar una inserción en el filtro correspondiente en cada operación de lectura, escritura o reducción transaccional. No obstante la comprobación de conflictos resulta muy rápida, ya que consiste en realizar una intersección de los filtros a verificar.

UNIVERSIDAD DE MÁLAGA

En la figura 3.13 se muestra el funcionamiento de estos filtros. Una transacción *tx1* se encuentra en su fase de *commit* y existen un par de transacciones *tx2*, *tx3* ejecutándose concurrentemente. Una vez consolidados los datos tentativos, *tx1* realiza una operación AND bit a bit entre su filtro de escritura y reducción y los filtros de lectura de *tx2* y *tx3*. Si el resultado es cero, no existen conflictos entre las transacciones. En el caso de *tx3* se obtiene un positivo, lo que implica una dependencia *read-after-write*, que se resolverá invalidando *tx3*.

El rendimiento de los filtros de Bloom depende de su tamaño: filtros pequeños aumentan la probabilidad de falsos conflictos debido a los alias de las direcciones de memoria, mientras que filtros mayores requieren más tiempo a la hora de comprobar conflictos durante la fase de *commit* o de reiniciarlos durante un eventual aborto.

Criterios de corrección: Opacidad y VWC

Los algoritmos de ReduxSTM descritos hasta el momento verifican *strict Serializability*[17]. Cualquier criterio más garantista (ver sección 2.4.2) requiere que el estado que observen las transacciones sea válido en todo momento, lo que implica la necesidad de comprobar la validez de las lecturas que realiza la transacción antes de hacerlas visibles a la misma. En esta sección se describen los cambios necesarios para soportar estos criterios en ReduxSTM.

El algoritmo 3 muestra cómo incorporar la comprobación de *strict VWC* y *opacity* a ReduxSTM-TS. Al iniciar la transacción (TMBEGIN) se añade el valor del reloj global a una nueva variable *tx.start* local a cada transacción (línea 2). En adelante, si un valor leído por la transacción ha sido actualizado posteriormente al valor de *tx.start*, la transacción debe abortar. Antes de finalizar la función TMREAD se incluye una llamada a una función de validación parcial (línea 11) que anticipa la validación total que deberá realizarse igualmente durante la fase de *commit*. La función de validación depende del criterio a garantizar:

- VALIDATEVWC comprueba la consistencia para *strict VWC*. En este caso sólo es necesario comprobar si el valor al que se acaba de acceder se actualizó o no con posterioridad a *tx.start*. Nótese que, aunque otros valores ya leídos por la transacción hayan podido cambiar desde su lectura, mientras esta validación se cumpla la transacción puede serializarse en algún instante anterior a la actualización de dichos valores, como muestra la línea roja de la figura 3.14.

[17] La condición *strict*, tanto en éste como en otros criterios de corrección viene implícita por las garantías de orden que mantiene ReduxSTM.

UNIVERSIDAD DE MÁLAGA

- ValidateOpacity comprueba la consistencia para *opacity* comprobando la validez de todos los objetos leídos hasta ese instante por la transacción.

Como se analizó en la sección 2.4.2, *VWC* es una alternativa a *opacity* que resulta adecuada para la gran mayoría de situaciones e implica una menor penalización debido a las validaciones. La diferencia se ilustra en la figura 3.14. La transacción $t_{1,1}$ ha accedido a un valor de A posteriormente actualizado por $t_{0,0}$. Garantizar *opacity* implica que $t_{1,1}$ debe abortar en el siguiente acceso transaccional, ya que su vista de la ejecución es inconsistente con α. Esto requiere comprobar la validez del *read-set* parcial en cada acceso transaccional. *Strict VWC*, sin embargo, admite que $t_{1,1}$ tenga una vista de la ejecución distinta, siempre que sea válida y $t_{1,1} \notin Comm(H)$. En este caso se admite que $t_{1,1}$ vea $\alpha' = \{t_{1,1}*, t_{0,1}*\}$ durante su ejecución, siempre que acabe abortando[18].

Dado que garantizar cualquiera de estos criterios requiere realizar validaciones fuera de la fase de *commit*, es necesario añadir una barrera de memoria (línea 13 de TmEnd) para garantizar que las transacciones que realizan las validaciones adicionales vean el sistema en el orden correcto respecto a las posibles transacciones que pudiera haber en fase de *commit* en ese instante, como muestra la figura 3.10 (b).

En ReduxSTM-CTI es necesario un enfoque distinto a la hora de garantizar estos criterios, ya que el coste que implicaría que cada transacción accediese a los conjuntos de datos del resto de transacciones en cada lectura sin producir condiciones de carrera sería prohibitivo de cara al rendimiento. En este caso, la solución pasa por tratar la fase de *commit* de las transacciones como atómica respecto a las *lecturas* del resto de transacciones activas [45]. A tal efecto se ha utilizado un mecanismo *lock-free* basado en un *lock de secuencia* [65, 23] que se muestra en el algoritmo 4.

La variable global *glOpacitySeqLock* impide que se realicen operaciones de lectura si hay alguna transacción en fase de *commit*. Esta variable se inicializa a cero en TmInit (línea 2) y se incrementa atómicamente antes y después de cada *commit* (líneas 7 y 22 de TmEnd). De este modo un valor impar indica que hay una transacción en fase de *commit* en ese momento. En cada lectura (TmRead), las transacciones deben comprobar su validez (línea 3) y esperar a que no haya un *commit* simultáneo (líneas 5 a 7). Una vez obtenido el dato tentativo, se comprueba si el valor de *glOpacitySeqLock* se ha mantenido constante (líneas 16 a 18), lo que indica que ninguna otra transacción ha escrito en memoria durante la lectura. En caso contrario, debe repetirse el proceso previa validación, para comprobar si el *commit* existente ha invalidado la transacción (línea 17).

[18]Debido a que *strict VWC* sí requiere que toda $t \in Comm(H)$ vea la misma ejecución α.

UNIVERSIDAD DE MÁLAGA

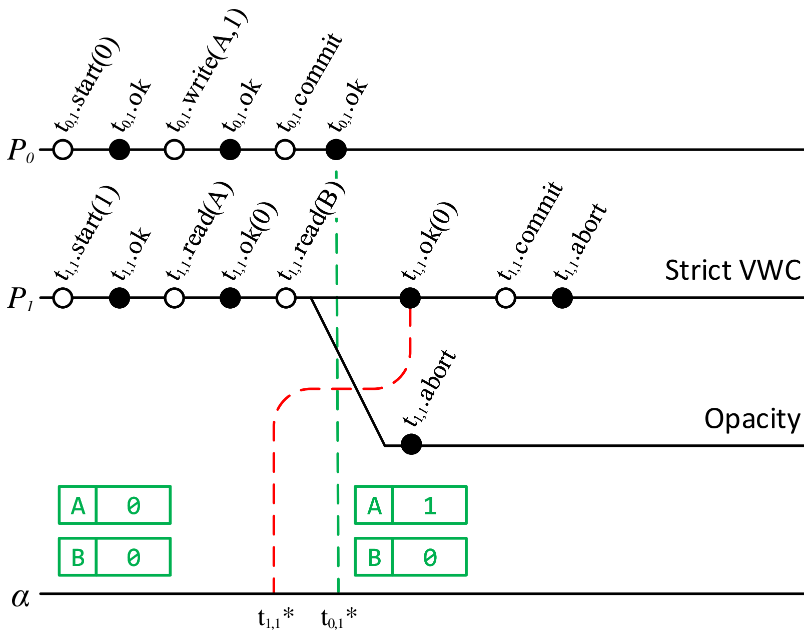


Figura 3.14: Garantías de *opacity* y *strict VWC* en ReduxSTM. La bifurcación en P_1 ilustra dos posibles historias válidas que garantizan distintos criterios de consistencia. La línea roja discontinua indica un posible punto de serialización $t_{1,1}*$ para $t_{1,1} \in Live(H)$. La línea verde indica el instante en que el nuevo valor de A, actualizado por $t_{0,1}$, se hace visible al resto de transacciones.

UNIVERSIDAD DE MÁLAGA

Algoritmo 3 Garantizando *opacity* y VWC en ReduxSTM-TS

```
1  function TmBegin(order)
       ...
2      | tx.start ← glOrder |                  ▷ Timestamp de inicio
3  end function
```

```
1  function ValidateVWC(addr)
2      if tx.start < glWTS[addr] then          ▷ Aborta si addr ha sido escrita después del
3          StmRollback( )                      ▷ inicio de la transacción
4      end if
5  end function
```

```
1  function ValidateFullOpacity( )
2      for all addr ∈ tx.ReadSet do
3          if tx.RTS[addr] < glWTS[addr] then  ▷ Aborta si cualquier objeto del read-set
4              StmRollback( )                  ▷ ha sido actualizado por otra transacción
5          end if
6      end for
7  end function
```

```
1  function TmRead(addr)
2      if addr ∈ tx.WriteSet then              ▷ Si el objeto ha sido escrito, devuelve el valor
3          value ← tx.WriteSet[addr]           ▷ directamente, sin añadir la dirección al conjunto
4          return value                        ▷ de lectura
5      end if
6      tx.RTS [addr] ← glOrder                 ▷ Actualiza timestamp local de lectura
7      tx.ReadSet ← tx.ReadSet ⋃ {addr}        ▷ Añade la dirección al conjunto de lectura
8      if addr ∈ tx.RdxSet then                ▷ Si existía reducción previa, genera valor correcto
9          value ← (*addr) ⊕ tx.RdxSet[addr]
10     else                                    ▷ En caso contrario, lee el valor de memoria
11         value ← (*addr)
12     end if
13     | Validate(addr) |                      ▷ Llamada a una de las funciones de validación
14     return value
15 end function
```

UNIVERSIDAD DE MÁLAGA

Algoritmo 3 Garantizando *opacity* y VWC en ReduxSTM-TS (continuación)

```
 1 function TmEnd( )
 2     while tx.order ≠ glOrder do                   ▷ Espera del turno
 3     end while

 4     tx.status ← TxCommitting                      ▷ Inicio de la fase de commit

 5     for all addr ∈ tx.ReadSet do                  ▷ Validación
 6         if tx.RTS[addr] < glWTS[addr] then
 7             StmRollback( )
 8         end if
 9     end for

10     for all addr ∈ tx.RdxSet ⋃ tx.WriteSet do
11         glWTS[addr] ← glOrder +1                  ▷ Actualización del timestamp global
12     end for
13     [MemoryFence( )]                              ▷ Barrera de memoria, evita reordenamiento

14     for all addr ∈ tx.RdxSet do                   ▷ Consolidación de reducciones en memoria
15         *addr ← *addr ⊕ tx.RdxSet[addr]
16     end for
17     for all addr ∈ tx.WriteSet do                 ▷ Consolidación de escrituras en memoria
18         *addr ← tx.WriteSet[addr]
19     end for

20     tx.status ← TxIdle                            ▷ Finaliza fase de commit
21     Fetch&Add(glOrder, 1)                         ▷ La siguiente transacción puede finalizar
22 end function
```

UNIVERSIDAD DE MÁLAGA

Algoritmo 4 Garantizando *opacity* y VWC en ReduxSTM-CTI

```
 1 function TmInit(nThreads)
       ...
 2     glOpacitySeqLock ← 0                      ▷ Inicializa lock de secuencia
 3 end function
```

```
 1 function TmRead(addr)
 2     if addr ∈ tx.WriteSet then                ▷ Si el objeto ha sido escrito, devuelve el valor
 3         value ← tx.WriteSet[addr]             ▷ directamente, sin añadir la dirección al conjunto
 4         return value                          ▷ de lectura
 5     end if

 6 re_read:
 7     CheckKilled( )
 8     do
 9         start ← glOpacitySeqLock              ▷ Valor del lock de secuencia antes del acceso
10     while start mod 2                         ▷ Verifica que no haya un commit concurrente

11     tx.ReadSet ← tx.ReadSet ⋃ {addr}          ▷ Añade dirección al conjunto de lectura
12     tx.rBf ← addr                             ▷ Inserta dirección en el filtro de lectura

13     MemoryFence( )                            ▷ Evita reordenación en el resto de hilos

14     if addr ∈ tx.RdxSet then                  ▷ Si existía una reducción previa, genera el valor
                                                   correcto
15         value ← (*addr) ⊕ tx.RdxSet[addr]
16     else
17         value ← (*addr)                       ▷ Lee el valor de memoria
18     end if

19     if start ≠ glOpacitySeqLock then          ▷ Comprueba que no haya habido un commit
20         goto re_read                          ▷ mientras se realizaba la lectura
21     end if

22     return value
23 end function
```

UNIVERSIDAD DE MÁLAGA

Algoritmo 4 Garantizando *opacity* y VWC en ReduxSTM-CTI (continuación)

```
 1 function TmEnd( )
 2     while tx.order ≠ glOrder do                       ▷ Espera del turno
 3         CheckKilled( )
 4     end while

 5     CheckKilled( )                                    ▷ Validación
 6     tx.status ← TxCommitting                          ▷ Inicio de la fase de commit

 7     [Fetch&Add(glOpacitySeqLock, 1)]                  ▷ El lock de secuencia es impar mientras haya una
                                                           transacción en fase de commit
 8     for all addr ∈ tx.RdxSet do                       ▷ Consolidación de reducciones en memoria
 9         *addr ← *addr ⊕ tx.RdxSet[addr]
10     end for
11     for all addr ∈ tx.WriteSet do                     ▷ Consolidación de escrituras en memoria
12         *addr ← tx.WriteSet[addr]
13     end for

14     MemoryFence( )

15     for all otherTx ∈ globalTx do                     ▷ Detección de conflictos e invalidación
16         if otherTx.status = TxActive then
17             if tx.uBf ⋂ otherTx.rBf ≠ ∅ then
18                 otherTx.status ← TxDoomed             ▷ Invalidación de la transacción implicada
19             end if
20         end if
21     end for

22     [Fetch&Add(glOpacitySeqLock, 1)]                  ▷ El lock de secuencia es par una vez finalizada la
                                                           fase de commit
23     tx.status ← TxIdle                                ▷ Finaliza fase de commit
24     Fetch&Add(glOrder, 1)                             ▷ La siguiente transacción puede finalizar
25 end function
```

UNIVERSIDAD DE MÁLAGA

Optimizaciones adicionales

Ambas implementaciones de ReduxSTM incluyen algunas optimizaciones adicionales:

- Uso de la librería de optimización *asmlib* [40].

  La inicialización de estructuras de datos relativamente grandes (*OpSet*, *timestamp*, filtros de Bloom) es una operación frecuente (inicio, *commit* y aborto de las transacciones), que puede penalizar el rendimiento. Por esta razón se ha utilizado una librería optimizada para realizar operaciones como `memset`. *Asmlib* contiene versiones específicas de funciones básicas del estándar de C capaces de aprovechar características del procesador como SSE o AVX para acelerar la ejecución.

- Irrevocabilidad en el reinicio de transacciones en ReduxSTM-TS.

  Dado que una transacción que haya llegado a su fase de *commit* sólo puede abortar una vez —los conflictos se detectan con transacciones ya finalizadas—, la reejecución ante un eventual aborto puede llevarse a cabo de forma irrevocable [23]. De esta forma no se necesita realizar ninguna validación durante la reejecución de la transacción. No obstante, si el soporte a *opacity* o *strict VWC* está habilitado, seguirá siendo necesaria la barrera de memoria después de actualizar cada entrada del *timestamp* global de escritura durante la fase de *commit*.

- Adquisición relajada de orden en ReduxSTM-TS.

  Para garantizar *opacity* o *strict VWC* se adquiere el orden global al inicio de la transacción. Esta adquisición puede retrasarse al momento de la primera lectura transaccional reduciendo la ventana de vulnerabilidad a conflictos de la transacción.

- Aborto anticipado en ReduxSTM-CTI.

  Cualquier transacción activa con un número de orden mayor al del reloj global es susceptible de ser invalidada si se produce un conflicto. La comprobación de esta invalidación apenas introduce penalización ya que sólo requiere verificar un *flag* que estará en caché en la mayoría de accesos. Consultar el estado de la transacción en cada operación transaccional (lectura, escritura, reducción) puede ser conveniente para anticipar el aborto antes de la fase de *commit*, especialmente en transacciones largas, donde un aborto tardío descarta una gran cantidad de trabajo.

UNIVERSIDAD DE MÁLAGA

Parámetro	Descripción
Procesador	Intel Xeon E5-2698 2.30 GHz
Núcleos	16 físicos. Hyperthreading (SMT) deshabilitado.
Memoria	126 GB
Sistema	Ubuntu Server 14.04.4 LTS, Kernel 3.13.0 x86_64
Compilador	GNU GCC v4.8.2

Tabla 3.4: Plataforma de evaluación de ReduxSTM

Cabe mencionar que con este sistema pueden producirse abortos innecesarios si una transacción en fase de *commit* comprueba el conjunto de lectura de otra transacción que está reiniciando sus estructuras debido a un aborto anterior. Sin embargo, este escenario es muy poco frecuente en casos reales, por lo que no compensa la adición de sincronización adicional para evitarlo.

3.4. Evaluación

ReduxSTM proporciona un sistema de memoria transaccional, soporte adicional para operaciones de reducción y restricciones de orden entre transacciones. Con el objetivo de evaluar estas características se ha considerado una selección de códigos que se resumen en la tabla 3.5. Este estudio cubre diferentes escenarios que hemos considerado de interés, incluyendo un análisis de sensibilidad de ReduxSTM, bucles de reducción irregulares, escenarios con reducciones parciales —que necesitan restricciones de orden para garantizar resultados correctos—, posibilidad de usar técnicas de especulación y rendimiento en escenarios generales, que no requieran soporte de operaciones de reducción o restricciones de orden. Nuestro objetivo en esta sección es medir el impacto que puede suponer la adición de soporte transaccional explícito a operaciones de reducción y restricciones de orden en un sistema TM, así como analizar el rendimiento de nuestra propuesta en contextos más generales.

Como sistema transaccional de referencia se ha considerado TinySTM [32], una de las librerías STM más utilizadas en el estado del arte. Dado que la última versión disponible (1.0.5) ofrece soporte para transacciones ordenadas, se han considerado dos configuraciones: la configuración por defecto del STM, que denominaremos TinySTM de aquí en adelante, y una configuración adicional con restricciones de orden (activando el módulo `mod_order` de la librería), a la que nos referiremos como TinySTM-Ordered. En cuanto a ReduxSTM, se han analizado las dos implementaciones propuestas en este trabajo: la basada en *timestamps* (ReduxSTM-TS) y la basada en *commit-time invalidation* (ReduxSTM-CTI).

UNIVERSIDAD DE MÁLAGA

Los experimentos se han realizado en el sistema descrito en la tabla 3.4. Los códigos se han compilado usando GCC 4.8.2 con nivel de optimización `-O2`.

El tiempo de ejecución mostrado se ha obtenido considerando el tiempo mínimo de ejecución de al menos diez ejecuciones por experimento. De este modo reducimos la interferencia del sistema de acuerdo con lo indicado en [46]. Los experimentos que exceden un tiempo de ejecución razonable (diez veces el tiempo de la ejecución secuencial) se consideran fallidos.

UNIVERSIDAD DE MÁLAGA

Escenario	Benchmark	Descripción	Reducc. irregulares	Reducc. parciales	Orden requerido
Análisis de sensibilidad	RXasRW	Código de cálculo de histograma con una fracción de las reducciones implementadas como lecturas y escrituras (ver figura 3.15).	Sí	Sí	No
	Eigenbench	Benchmark transaccional [81] modificado para incluir reducciones.	No[1]	Sí[1]	No
	WormBench	Benchmark para STM [114] modificado para incluir reducciones.	Sí[1]	No[1]	No
Códigos irregulares	FluidAnimate	Aplicación de simulación de fluidos mediante el método Smoothed Particle Hydrodynamics para aplicaciones interactivas y en tiempo real. Parte de la suite PARSEC.	Sí	No	No
	Euler	Resolución de las ecuaciones de Euler para simulación física. Utilizado en aplicaciones de dinámica de fluidos y cálculo de deformaciones.	Sí	No	No
	Legendre	Kernel que utiliza la transformada de Legendre para aplicaciones de predicción meteorológica.	Sí	No	No
	MD2	Bucle para simulación molecular 2D en interacciones de corto alcance basado en lista de vecinos.	Sí	No	No
Reducciones parciales	Twolf	Función `new_dbox_a()` del código de la aplicación 300.twolf. Parte de la suite SPEC CPU2000 [47].	Sí	Sí	Sí
Rendimiento general	STAMP	Todos los códigos de la suite STAMP para memoria transaccional [14].	No[2]	No[2]	No
Soporte para especulación	SPEC2006	Bucles de interés para aplicar *Thread-Level Speculation*. Parte de la suite SPEC CPU2006 [80].	No	No	Sí

[1] Asumiendo las modificaciones realizadas.
[2] Excepto KMeans.

Tabla 3.5: Resumen de los *benchmarks* utilizados en la evaluación.

UNIVERSIDAD DE MÁLAGA

3.4.1. Análisis de sensibilidad

En esta sección se pretende evaluar el potencial que ofrece el soporte explícito de reducciones y el impacto de las restricciones de orden en el rendimiento de los sistemas STM. Los *benchmarks* utilizados en esta evaluación son sintéticos y permiten controlar de forma precisa determinados aspectos de interés para el análisis.

El impacto resultante del soporte a operaciones de reducción se evalúa en RXasRW. Este *benchmark* permite implementar un porcentaje variable de las operaciones de reducción como lecturas y escrituras, dando lugar a patrones de reducciones parciales y permitiendo evaluar la penalización de rendimiento de los distintos sistemas en presencia de las mismas. Adicionalmente se ha analizado cómo afecta el tamaño de las estructuras de datos de ReduxSTM a la tasa de abortos debidos a los falsos conflictos causados por alias de accesos diferentes en dichas estructuras. Eigenbench nos permite analizar cómo afecta la distribución de las operaciones transaccionales y el tamaño de la transacción al rendimiento del STM. En este análisis se han evaluado sistemáticamente diferentes distribuciones de las operaciones transaccionales en cuatro tamaños de transacción diferentes. Por último, y con el objetivo de estudiar cómo afecta la contención de un problema al rendimiento de los sistemas analizados, WormBench nos ha permitido analizar un mismo patrón de accesos con diferentes niveles de contención y observar el comportamiento de ReduxSTM respecto a otros sistemas STM de la literatura.

RXasRW (Reductions as Read+Write)

RXasRW es un código sintético que calcula un histograma siguiendo el bucle de reducción que se muestra en la figura 3.15. Su propósito es medir el impacto del soporte explícito de reducciones en una fracción del total de las sentencias que pueden realmente beneficiarse del mismo.

En el código se realiza una operación de reducción a través de una función de indirección en cada iteración. Esta indirección se realiza de forma aleatoria siguiendo una distribución uniforme. Los parámetros configurables de este *benchmark* incluyen el tamaño del *array* de reducción `NR`, que determina el nivel de contención del problema, el tamaño de la transacción (medido en iteraciones ejecutadas en cada transacción) y la carga computacional asociada a la generación del valor ξ que debe ser reducido. *RXasRW* incluye además un umbral configurable, θ, que determina la probabilidad de que la operación de reducción se implemente en el STM como una lectura seguida de una escritura. De este modo, para un STM con soporte de operaciones de reducción, un valor de θ=0% significa que todas

UNIVERSIDAD DE MÁLAGA

```
1  uint64_t A[NR];               /* Array de reducción */
2  int Ind[N/xactSize][xactSize]; /* Array de indirecciones */
3
4  /* Los arrays de reducción (A) e indirección (Ind) han sido
5   * inicializados previamente */
6
7  /* Se asume que el espacio de iteraciones (N) es múltiplo del
8   * tamaño de la transacción (xactSize) */
9
10 for (i=0; i<N/xactSize; i++){
11     TmBegin();
12     for (ii=0; ii<xactSize; ii++){
13         Compute ξ
14         idx = Ind[i][ii]; /* Indirección para acceder al array de reducción */
15         prw = rand();
16         if (prw > θ)
17             TmRdx(A[idx], ξ);
18         else
19             TmWrite(A[idx], TmRead(A[idx]) ⊕ ξ );
20     }
21     TmEnd();
22 }
```

Figura 3.15: Extracto del *benchmark RXasRW*. Bucle de reducción de histograma donde una fracción de los accesos de reducción se implementan en el código sin soporte transaccional explícito.

las reducciones se transformarán a operaciones de reducción transaccional `TmRdx` (línea 17), mientras que un θ=100% indica que todas las reducciones se realizarán mediante una lectura transaccional seguida de una escritura transaccional (línea 19). Para el resto de STM, todas las operaciones de reducción se transforman en una lectura seguida de una escritura independientemente del valor de θ.

La figura 3.16 muestra el rendimiento obtenido en términos de *speedup* respecto a la ejecución secuencial sin instrumentar y el TCR (Transactional Commit Rate), definido como el cociente entre el número de transacciones finalizadas con éxito respecto al total de transacciones (exitosas y abortadas). Los experimentos se han realizado a partir de una configuración inicial con un bucle de 10^7 iteraciones, que se reduce en un *array* `NR` de 1000 elementos; un tamaño de transacción de 10 iteraciones y una carga computacional de 50 FLOPs por iteración. El tamaño del bucle se ha escogido de modo que sea suficiente para obtener tiempos consistentes (tamaños menores presentan mayor variabilidad en los resultados). El resto de parámetros se han establecido en una zona intermedia de la ventana de resultados que se analizan en este *benchmark*, para observar más adelante cómo afecta la variación de los mismos al rendimiento final.

UNIVERSIDAD DE MÁLAGA

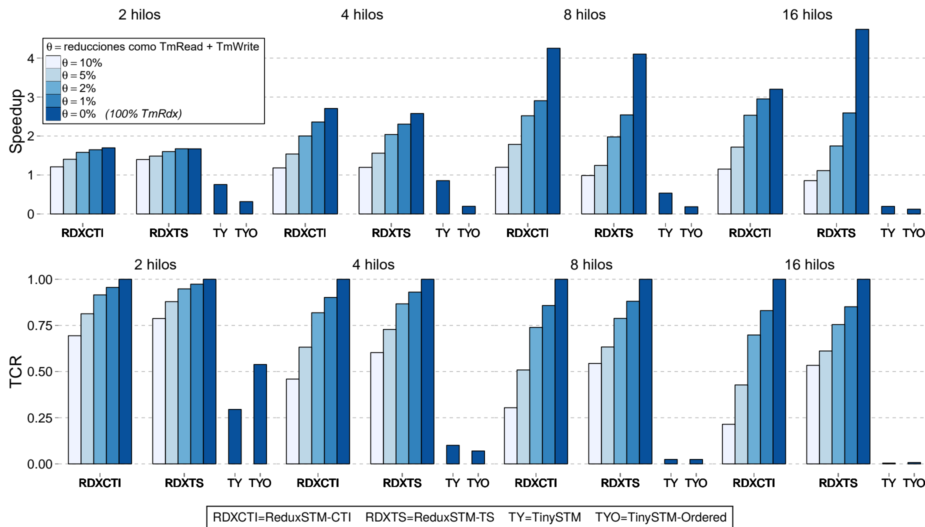


Figura 3.16: Rendimiento de *RXasRW* en términos de *speedup* (arriba) y TCR (abajo).

Se puede observar que los valores de *speedup* y TCR alcanzados por ReduxSTM son sensiblemente mejores que los alcanzados por TinySTM, éste último tanto en su configuración por defecto como en la ordenada. Al aumentar el número de hilos de ejecución aumenta también la contención sobre `NR`, y TinySTM baja su rendimiento por no poder filtrar los conflictos derivados de las sentencias de reducción. ReduxSTM necesita serializar la fase de *commit* de todas sus transacciones debido a las restricciones de orden, pero mantiene un buen rendimiento que aumenta con el número de hilos de ejecución. Por una parte, esto verifica la capacidad de los diseños propuestos a la hora de extraer rendimiento de las operaciones de reducción. Por otra, el TCR indica la efectividad del soporte para operaciones de reducción a la hora de filtrar un número considerable de conflictos causados por las propias sentencias de reducción.

Si nos centramos en las implementaciones de ReduxSTM, ReduxSTM-TS obtiene un mejor rendimiento comparado con ReduxSTM-CTI en escenarios con un número elevado de hilos de ejecución. Este comportamiento es esperado, y se explica por el protocolo de invalidación de la versión CTI para implementar la política de resolución de conflictos *kill-others*, que obliga a las transacciones a verificar un número de filtros de Bloom proporcional al número de transacciones concurrentes.

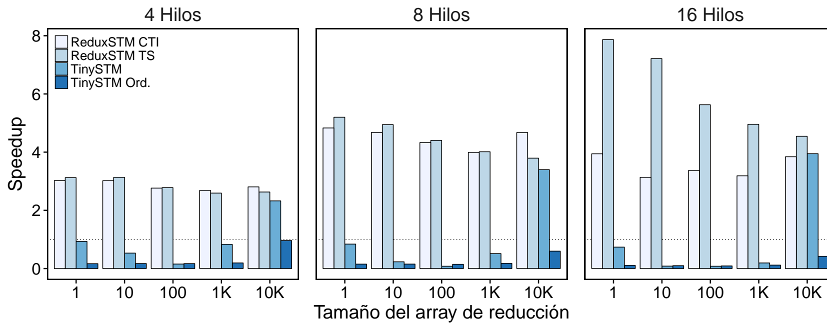
UNIVERSIDAD DE MÁLAGA

Las discrepancias relativas entre las medidas de TCR y *speedup* se explican por el coste asociado a la fase de *commit* de los dos algoritmos de ReduxSTM. De este modo, aunque ReduxSTM-CTI y ReduxSTM-TS obtienen un valor de TCR similar, el *speedup* puede variar ligeramente. Hay que mencionar además la existencia de reducciones parciales, generadas por el hecho de implementar una fracción de las sentencias de reducción como lecturas y escrituras transaccionales de acuerdo con el valor de θ. Este factor permite observar cómo la ventaja que ofrece ReduxSTM aumenta al reducirse el número de reducciones parciales del escenario correspondiente (menor θ). De hecho, un escenario con θ=100% no permitiría aprovechar el soporte explícito de reducciones de ReduxSTM.

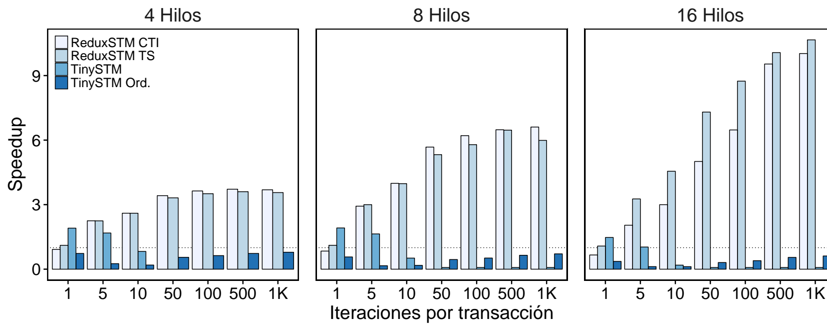
A partir de la configuración inicial descrita, la figura 3.17 muestra la sensibilidad de los distintos STM al resto de parámetros del *benchmark*. Estos experimentos se han ejecutado con un valor θ=0%, es decir, máximo TCR para ReduxSTM, ya que la totalidad de los conflictos son filtrados por el soporte de reducciones.

La relación entre el número de accesos totales `N` y el tamaño del array de reducción `NR` determina el nivel de contención del algoritmo. Dicha contención es inversamente proporcional al tamaño del array de reducción, y su efecto puede observarse en la figura 3.17 (a). Los escenarios de alta contención derivan en una cantidad elevada de conflictos potenciales que pueden prevenir cualquier ganancia de rendimiento si no se resuelven adecuadamente. Es el caso de TinySTM, que necesita un `NR` de 10K para mejorar el tiempo de la ejecución secuencial debido a los conflictos derivados de la falta de soporte a operaciones de reducción. Por su parte los escenarios de baja contención pueden incrementar el coste asociado a la instrumentación en STM debido a la necesidad de monitorizar un rango más amplio de objetos transaccionales, que se traduce en más consultas e inserciones en las estructuras de datos. Este hecho es especialmente notable en ReduxSTM-TS ya que, en su caso, la complejidad de la fase de validación durante el *commit* depende del número de direcciones únicas accedidas por la transacción. Este fenómeno se observa claramente en la gráfica correspondiente a 16 hilos de ejecución, donde el rendimiento de ReduxSTM-TS disminuye conforme aumenta el tamaño de `NR`. Es precisamente en los escenarios de mayor contención donde ReduxSTM tiene una mayor ventaja sobre la propuesta STM de referencia. Cuando la contención disminuye, la ventaja sobre TinySTM disminuye ligeramente, aunque continúa siendo alta respecto a TinySTM-Ordered, que tiene restricciones de orden similares a ReduxSTM. Como se mencionó anteriormente, ReduxSTM-TS es capaz de escalar mejor al aumentar el número de hilos de ejecución, ya que no necesita consultar los conjuntos de datos de las transacciones activas en el resto de hilos durante su validación.

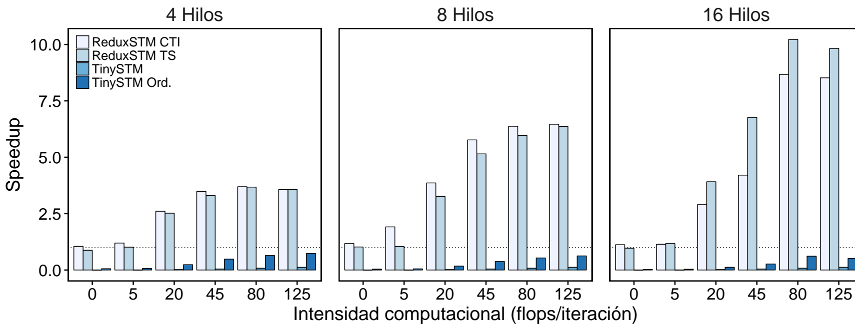
UNIVERSIDAD DE MÁLAGA



(a)



(b)



(c)

Figura 3.17: RXasRW: Sensibilidad de las propuestas STM al tamaño del array de reducción `NR` (a), al tamaño de las transacciones (b), y a la intensidad computacional (c).

UNIVERSIDAD DE MÁLAGA

El tamaño de las transacciones debe mantener un compromiso entre la penalización asociada a las fases de inicio y *commit* de las mismas —que se incrementa con el número de transacciones ejecutadas—, y el coste y probabilidad de los abortos, que aumenta en transacciones más grandes. La figura 3.17 (b) muestra este comportamiento en un escenario sin conflictos reales en el caso de ReduxSTM. El uso de un tamaño de transacción elevado implica un menor número de transacciones para una configuración dada, lo que disminuye el coste asociado a las fases de inicio y *commit*. El rendimiento de TinySTM es bajo de por sí debido a los conflictos producidos en las sentencias de reducción, por lo que no se muestra demasiado afectado por el tamaño de la transacción, tanto en su versión por defecto como en la ordenada. De hecho en varios de estos experimentos los resultados de TinySTM llegaron al límite de tiempo establecido (diez veces el tiempo de la ejecución secuencial). En cuanto a ReduxSTM, en este experimento se puede observar con claridad la penalización de rendimiento de la versión CTI respecto a la versión TS al aumentar el número de hilos de ejecución a 16.

La intensidad computacional, definida como el ratio entre las operaciones de cómputo y de memoria, es otro factor a considerar, especialmente en sistemas TM software. En el caso de bucles de reducción, esta intensidad se suele asociar con la computación de las variables que serán acumuladas en las sentencias de reducción. En general, aunque una mayor intensidad computacional aumenta el coste de las recuperaciones en caso de aborto, no aumenta la probabilidad de conflicto, y enmascara los ciclos adicionales asociados a la instrumentación del sistema transaccional, por lo que suele traducirse en una mayor explotación del paralelismo disponible. Este comportamiento se muestra en la figura 3.17 (c). Aunque los resultados mejoran en todos los STM, la tasa de conflictos elevada penaliza a TinySTM. ReduxSTM se beneficia notablemente de una intensidad computacional elevada en escenarios donde predominan las operaciones de reducción.

La necesidad de representar la totalidad del espacio de memoria en los conjuntos de datos de forma eficiente implica el uso estructuras de datos finitas (filtros de Bloom en ReduxSTM-CTI y *arrays* de *timestamps* en ReduxSTM-TS) accedidas mediante funciones *hash* a partir de la posición de memoria asociada. El tamaño finito de dichas estructuras conlleva una cierta probabilidad de conflictos entre transacciones debidos a falsos positivos, es decir, a accesos a posiciones de memoria diferentes que el STM considera como iguales debido a una colisión en el *hash*.

Los efectos de estos falsos positivos se muestran en la figura 3.18, que considera un θ=3% para que dichas colisiones puedan producir abortos en las transacciones. Puede observarse un rendimiento óptimo para estructuras de en torno a 2^{10}

UNIVERSIDAD DE MÁLAGA

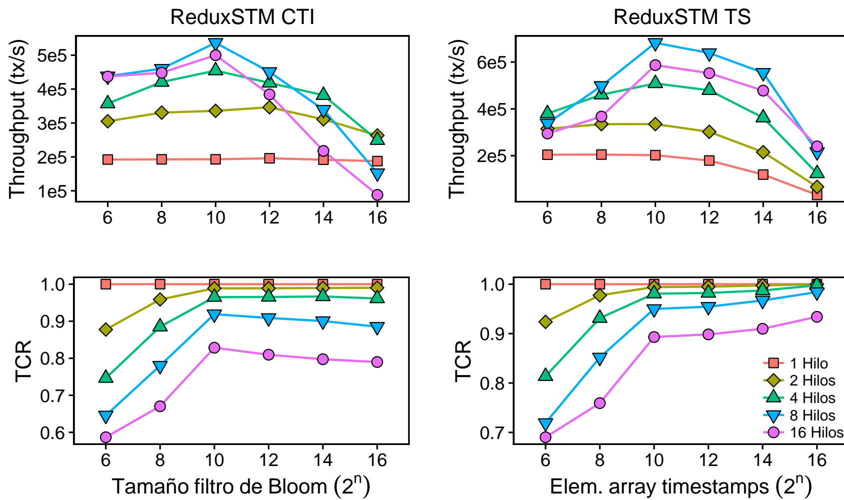


Figura 3.18: RXasRW: Sensibilidad de ReduxSTM al tamaño de las estructuras de datos en términos de rendimiento (arriba) y TCR (abajo).

elementos para el problema analizado, tanto en los filtros de Bloom como en los *arrays* de *timestamps*. Aumentar más el tamaño de estas estructuras apenas repercute en el TCR. Dado que el aumento de tamaño de las estructuras añade un coste adicional a sus operaciones (reinicio, inserción y búsqueda de elementos), una mejora marginal en la tasa de falsos positivos puede no compensar este coste, como puede observarse en las medidas de *throughput*[19] con tamaños elevados. En el caso de ReduxSTM-CTI, que debe comprobar conflictos con transacciones activas en el sistema, puede incluso observarse cómo el aumento del tiempo de ejecución asociado a tamaños de estructuras elevados penaliza ligeramente el TCR en las ejecuciones con un mayor número de hilos, al aumentar la ventana de tiempo en la que se pueden producir y detectar los conflictos.

Como observación adicional, los costes derivados del aumento del tamaño de las estructuras están dominados por las operaciones de reinicio en el caso de ReduxSTM-TS (inicio y aborto de transacciones) y por las operaciones de intersección de los filtros de Bloom (fase de *commit*) en el caso de ReduxSTM-CTI.

UNIVERSIDAD DE MÁLAGA

[19]Transacciones ejecutadas por unidad de tiempo.

```
1  void test_core(tid, loops, nbWrites, nbReads, nbRedux, ...) {
2      ...
3
4      for (i=0; i<loops; i++) {
5          TmBegin();
6          ...
7          /* Para el número total de operaciones de la transacción */
8          for (j=0; j< (nbWrites + nbReads + nbRedux) ; j++) {
9              /* Seleccionar una acción a realizar de las disponibles */
10             switch(rand_action(...) {
11                 index = rand_index(tid, ...);
12                 case READ:
13                     val += TmRead(hotArray[index]);
14                 case WRITE:
15                     TmWrite(hotArray[index], val);
16                 case REDUX:
17                 #ifdef RDX_EXPLICIT_SUPPORT
18                     TmRdx(hotArray[index], val);
19                 #else
20                     TmWrite(TmRead(hotArray[index])+val);
21                 ...
22             }
23         }
24         ...
25         TmEnd();
26
27         val += local_ops(R_OUT, W_OUT, val, tid);
28     }
29 }
```

Figura 3.19: Extracto del *kernel* de Eigenbench, modificado para incluir operaciones de reducción además de lecturas y escrituras.

Eigenbench

Eigenbench [81] es un *benchmark* sintético configurable cuyo objetivo es aislar aspectos ortogonales de un STM para analizar su rendimiento. Su *kernel* realiza una serie de operaciones transaccionales en posiciones aleatorias (a partir de una configuración predefinida) sobre un conjunto de tres *arrays*. El denominado *hot-array* está compartido por todos los hilos de ejecución y las transacciones pueden acceder a él libremente. El segundo, *mild-array*, es también compartido, pero cada hilo de ejecución accede a un subconjunto privado del mismo. Los conflictos que se produzcan como resultado de interacciones en este *array* se deben únicamente a falsos positivos del STM. Un tercer *array*, *cold-array*, es accedido de forma similar al *mild-array*, pero en este caso de forma no transaccional, permitiendo controlar

UNIVERSIDAD DE MÁLAGA

la cantidad de accesos no transaccionales de la configuración. Variando el número de accesos y su distribución a cada uno de estos *arrays*, este *benchmark* permite analizar el rendimiento de un STM en una gran variedad de escenarios.

Con el objetivo analizar el soporte a operaciones de reducción, el código original de Eigenbench se ha modificado para introducir sentencias de reducción en el conjunto de operaciones soportadas sobre el *hot-array*, tal y como se muestra en la figura 3.19. La función `rand_action` de la línea 10 devuelve una operación aleatoria a realizar a partir de una probabilidad dada para cada una de las operaciones soportadas. Para garantizar la compatibilidad con STM sin soporte a operaciones de reducción, éstas se transformarán en una lectura y una escritura transaccional a la misma posición de memoria, conservando así la semántica de la operación (línea 17).

Nuestro objetivo aquí es evaluar el comportamiento de ReduxSTM en distintas configuraciones de tamaño y distribución de operaciones (lecturas, escrituras y reducciones). Esta evaluación se ha realizado utilizando el *hot-array*, con un tamaño establecido en 10^5 elementos, y considerando un tamaño fijo (número de operaciones) en todas las transacciones de un mismo experimento. Se han realizado cuatro series de experimentos variando el número de operaciones en cada transacción de 10 a 40. En cada serie se han probado un total de 66 configuraciones variando la distribución del tipo de operaciones en la transacción.

La figura 3.20 muestra varios diagramas ternarios que muestrean el espacio de operaciones de memoria (lectura, escritura y reducción) para diferentes tamaños de transacción. Cada punto del diagrama indica el STM que produce el máximo *speedup* en un escenario concreto. Dicho escenario viene determinado por el porcentaje de lecturas, escrituras y reducciones de la transacción, que corresponden a la intersección de punto con cada uno de los tres ejes del diagrama.

En transacciones cortas (10 operaciones por transacción) TinySTM obtiene los mejores resultados salvo en los escenarios donde el número de lecturas es prácticamente nulo, en cuyo caso ReduxSTM-CTI trabaja mejor. Conforme la probabilidad de conflicto aumenta (en transacciones más largas), ReduxSTM-TS empieza a relevar a TinySTM, especialmente en los escenarios con más operaciones que impliquen actualizaciones de memoria (escrituras y reducciones). TinySTM queda relegado a los escenarios con un mayor número de lecturas. En tamaños de transacción a partir de 40 operaciones de memoria, TinySTM sólo obtiene el mejor resultado con transacciones de sólo lectura[20]. Por su parte, TinySTM-Ordered no obtiene el mejor resultado en ningún escenario.

[20]Debido a su diseño y a la ausencia de restricciones de orden, TinySTM permite evitar la fase de *commit* en transacciones de sólo lectura.

UNIVERSIDAD DE MÁLAGA

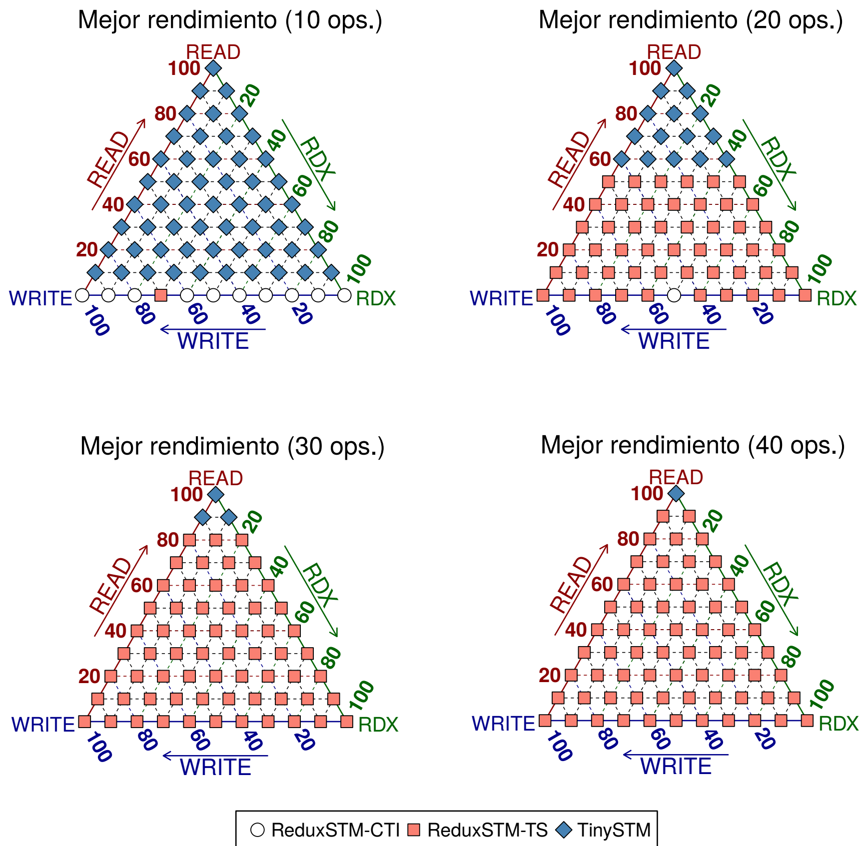


Figura 3.20: Eigenbench: STM con el mejor rendimiento para un porcentaje dado de operaciones de lectura, escritura y reducción. Se consideran 4 hilos de ejecución.

Wormbench

Wormbench [114] es un *benchmark* diseñado para evaluar el comportamiento de los diseños STM en escenarios de sincronización asociados a las aplicaciones multihilo. Su código simula un conjunto de gusanos moviéndose en un mundo representando por una matriz compartida de valores. Cada gusano se compone de un cuerpo y una cabeza, que cubre un subconjunto del mundo, y está controlado

UNIVERSIDAD DE MÁLAGA

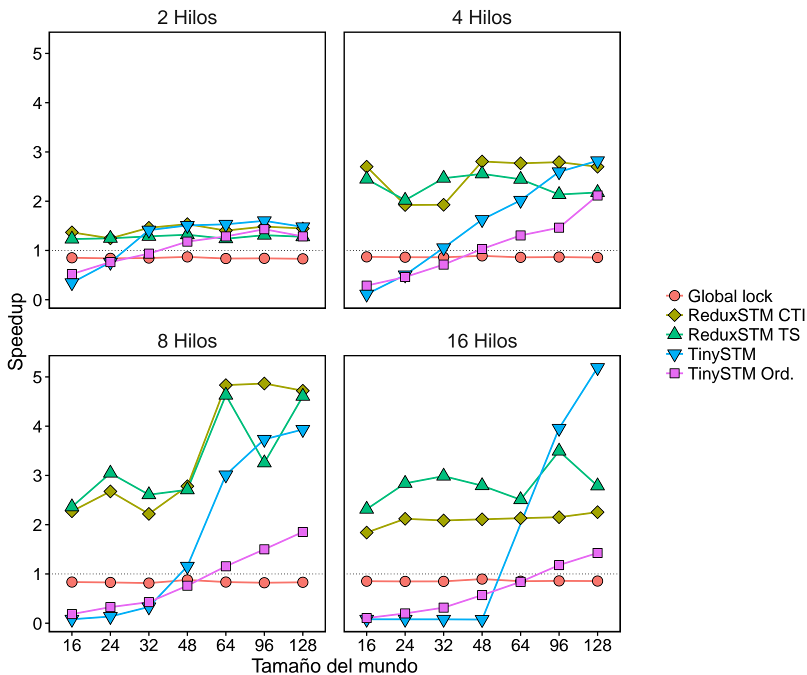


Figura 3.21: Wormbench: sensibilidad al tamaño del mundo.

por un hilo de ejecución. En cada paso de la simulación todos los gusanos realizan un movimiento y una acción, que implica una serie de operaciones que deben realizarse de forma atómica sobre el subconjunto del mundo cubierto por la cabeza del gusano. Pueden simularse diferentes escenarios configurando el tamaño del mundo y de los gusanos, la velocidad de los mismos y la secuencia de operaciones realizadas en cada paso de la simulación.

Wormbench está programado originalmente en C#, y se ha portado a C para compatibilizarlo con las librerías STM utilizadas en esta evaluación. Además se ha introducido una nueva operación para los gusanos que lleva a cabo una reducción de histograma de los elementos del mundo cubiertos por la cabeza del gusano en cada paso de la simulación. Los experimentos realizados se han centrado en esta nueva operación, que no implica reducciones parciales; las dependencias entre los distintos hilos provienen en su totalidad de las sentencias de reducción.

UNIVERSIDAD DE MÁLAGA

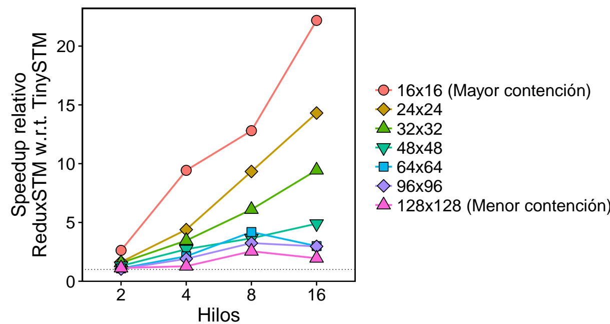


Figura 3.22: Wormbench: *speedup* relativo (mejor versión de ReduxSTM respecto a TinySTM-Ordered) para diferentes grados de contención.

Los experimentos realizados pretenden evaluar la utilidad del soporte explícito de reducciones de ReduxSTM en escenarios con distintos niveles de contención. Los resultados se muestran en la figura 3.21 y corresponden a una configuración con un tamaño constante de ocho elementos para las cabezas de los gusanos y un rango de tamaños para el mundo que oscila entre 16x16 y 128x128. El tamaño del mundo determina el grado de contención del problema: mayor cuanto más pequeño sea el mundo, lo que lleva asociado una mayor probabilidad de conflicto.

Ambas versiones de TinySTM presentan dificultades a la hora de obtener rendimiento en los escenarios de mayor contención. De hecho, TinySTM-Ordered presenta un rendimiento muy pobre, y TinySTM alcanza el límite de tiempo en tamaños de mundo pequeños y 16 hilos de ejecución, algo que se ve representado por un valor nulo en el *speedup* en mundos de hasta 48x48 elementos. Conforme aumenta el tamaño del mundo la contención del sistema disminuye, por lo que la ventaja que supone el filtrado de los conflictos debidos a reducciones se hace menos relevante. Este fenómeno se observa con mayor claridad en la figura 3.22, que compara el *speedup* relativo entre TinySTM-Ordered y ReduxSTM para mostrar precisamente cómo la ganancia de rendimiento de ReduxSTM aumenta en los escenarios de mayor contención.

Un hecho interesante es que el tamaño del mundo en el que ReduxSTM y TinySTM igualan su rendimiento aumenta con el número de hilos concurrentes: en experimentos con dos hilos de ejecución ocurre en mundos de 24x24, mientras que al utilizar 16 hilos no llega hasta mundos de 64x64, lo que implica un nivel menor de contención. Otro hecho destacable es que ReduxSTM-CTI rinde ligeramente mejor que ReduxSTM-TS en ejecuciones con 2 y 4 hilos, mientras que

UNIVERSIDAD DE MÁLAGA

Benchmark	S	Accesos	Bucles	Reducciones por bucle
FluidAnimate	104M	182M	2	2, 6
MD2	320K	572M	1	6
Euler	40K	9M	6	6, 6, 6, 9, 9, 6
Legendre	390K	2M	2	8, 8

Tabla 3.6: Características y carga de trabajo de los códigos con reducciones irregulares.

ReduxSTM-TS lo supera en los escenarios con 16 hilos. Este comportamiento es esperable dado que la duración de la fase de *commit* de ReduxSTM-CTI aumenta con el número de hilos, mientras que la de ReduxSTM-TS depende del número de elementos únicos accedidos.

3.4.2. Reducciones irregulares

Los *benchmarks* analizados en esta sección se caracterizan por contener bucles de reducción completos, cuyos accesos siguen patrones irregulares. El hecho de tener garantías de que las dependencias de los bucles sólo proceden de las sentencias de reducción permite el uso de técnicas clásicas de paralelización como las descritas en la sección 3.2.1. Todos los códigos analizados emplean una parte significativa de su tiempo de ejecución ejecutando bucles de reducción.

La tabla 3.6 resume algunas características de la carga de trabajo para los *benchmarks* seleccionados. La columna *S* corresponde al total de elementos contenidos en todos los *arrays* de reducción, y sirve como referencia de los requerimientos de memoria del problema. *Accesos* indica el número total de operaciones transaccionales sobre los *arrays* de reducción. *Bucles* indica el número de bucles de reducción presentes en cada *benchmark*. La última columna indica el número de sentencias de reducción que contiene cada uno de los bucles de reducción del código. En códigos con más de un bucle de reducción, las sentencias de cada bucle se muestran separadas por comas.

En estos experimentos se han aplicado una serie de técnicas adicionales para paralelizar los bucles de reducción presentes en los códigos:

- *Coarse-grained locks* (CGL), donde se protege la sección crítica completa con un *lock* de grano grueso.
- *Fine-grained locks* (FGL), donde se utiliza un *lock* individual para proteger cada elemento de los *arrays* de reducción compartidos.

UNIVERSIDAD DE MÁLAGA

- *Privatización* completa de las variables de reducción mediante el método de *Array Expansion* (ver sección 3.2.1).

Para los perfiles ejecutados con soporte TM, la paralelización de las reducciones se ha llevado a cabo descomponiendo los bucles en *chunks* de iteraciones consecutivas y asignando dichos *chunks* a las transacciones. Hay que recordar que ReduxSTM-TS, ReduxSTM-CTI y TinySTM-Ordered presentan restricciones de orden, a diferencia de TinySTM, que puede reordenar las iteraciones arbitrariamente.

Debido a la necesidad de instrumentación en los perfiles STM, el número de iteraciones ejecutadas en cada transacción es un parámetro relevante, que debe ser calibrado estableciendo un compromiso entre la penalización que introduce la instrumentación adicional y la mayor probabilidad de conflictos derivada del uso de transacciones mayores. En tales perfiles se han realizado pruebas adicionales para determinar un tamaño de transacción adecuado para cada *benchmark*. Los resultados de estas pruebas se muestran en gráficas de sensibilidad del *speedup* respecto al tamaño de la transacción para 16 hilos de ejecución. En las gráficas de rendimiento, los resultados mostrados corresponden al tamaño de transacción más adecuado para cada sistema de entre los probados para cada configuración.

En los perfiles basados en *locks* y *privatización* los bucles han sido particionados de manera uniforme entre los hilos de ejecución, sin agrupar varias iteraciones en *chunks*. Conviene recordar que la privatización requiere una fase de inicialización previa de los *arrays* auxiliares (*copy-in*) y una fase de reducción final de los mismos en el *array* de reducción original (*copy-out*). Por su parte, los perfiles basados en *locks* se han implementado utilizando *spinlocks* de POSIX. En las versiones FGL, el número de *locks* se ha optimizado utilizando un *lock* por cada indirección al array en lugar de por cada elemento del array. De este modo, grupos de reducciones que comparten el mismo acceso de indirección (por ejemplo, `vel(*,n1)` en la figura 3.2 (a) para Euler) utilizan el mismo *lock*. Esta implementación reduce el número de *locks* requeridos y el número de llamadas *lock/unlock*, lo que se traduce en una menor penalización.

FluidAnimate

La figura 3.23 muestra el rendimiento para FluidAnimate utilizando dos *datasets* que corresponden a mallas de 100K y 500K partículas. Como se esperaba, CGL no consigue explotar paralelismo debido al uso de un solo *lock* global. Este método puede ser viable para aplicaciones donde el tiempo empleado en las reducciones sea despreciable respecto al resto de la computación fuera de la sección crítica,

UNIVERSIDAD DE MÁLAGA

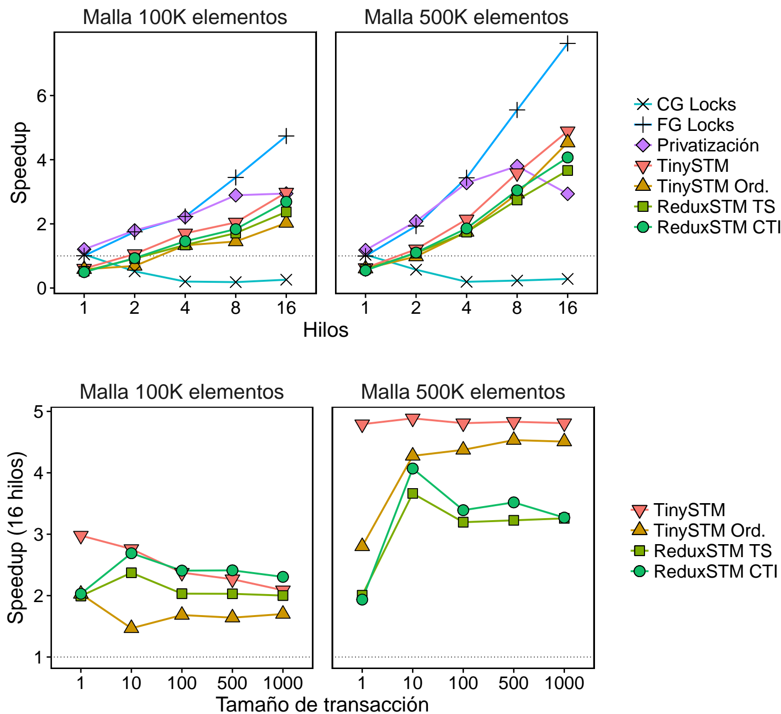


Figura 3.23: FluidAnimate: comparativa de rendimiento (arriba) e influencia del tamaño de transacción con 16 hilos de ejecución (abajo).

pero no es el caso. Los mejores resultados se obtienen mediante el uso de FGL o esquemas de privatización, gracias en parte a la baja contención de este problema. Es interesante observar que para la malla más grande, el rendimiento obtenido mediante privatización no escala tan bien como cabría esperar conforme crece el número de hilos. La causa la encontramos en los elevados requerimientos de memoria de esta técnica en algoritmos con conjuntos de reducción grandes y la arquitectura NUMA de memoria del servidor donde se han realizado los experimentos. Todos los sistemas STM se comportan de forma similar aquí, ya que el ratio de abortos es extremadamente bajo en cualquiera de las situaciones analizadas. Esto hace que ReduxSTM no obtenga ventaja del soporte de las operaciones de reducción. En este escenario, el uso de TM permite obtener un buen rendimiento y constituye una buena solución de compromiso entre la explotación del paralelismo y los requisitos de memoria. Respecto a la influencia

UNIVERSIDAD DE MÁLAGA

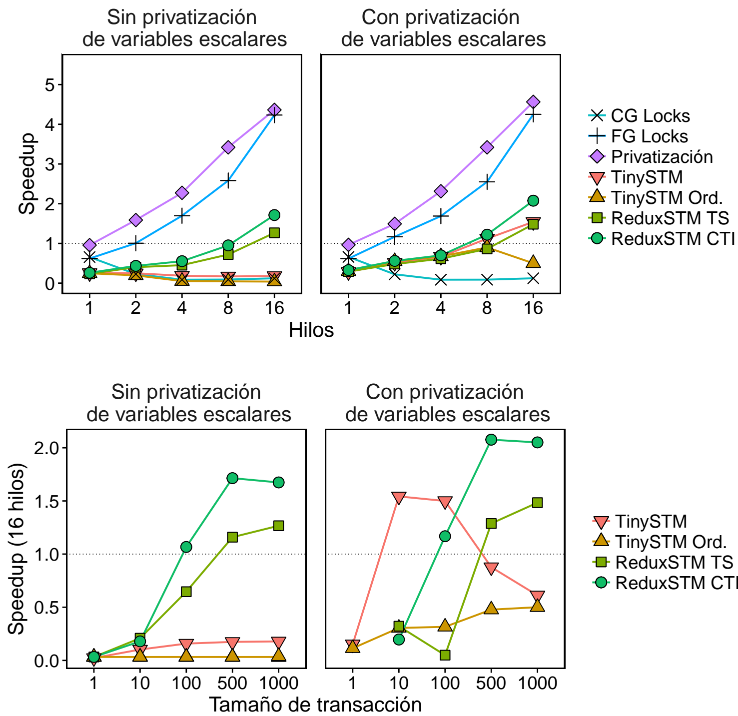


Figura 3.24: MD2: comparativa de rendimiento (arriba) e influencia del tamaño de transacción con 16 hilos de ejecución (abajo).

del tamaño de la transacción en el rendimiento, el comportamiento de los STM depende en gran medida del *dataset* utilizado. De este modo, al usar la malla más pequeña la penalización que implica el uso de transacciones grandes es más significativa que al usar la malla grande, donde el uso de transacciones demasiado pequeñas conlleva una penalización notable. En las gráficas que muestran el rendimiento respecto al tamaño de transacción puede observarse que el *speedup* se mantiene más o menos estable en tamaños de transacción muy variados, lo que da una idea de la baja contención de este problema.

UNIVERSIDAD
DE MÁLAGA

MD2

MD2 incluye tanto bucles de histograma como reducciones escalares en el bucle de reducción, como se muestra en la figura 3.2 (c). En este código se han realizado dos paralelizaciones diferentes. En la primera todas las variables compartidas de reducción (escalares y vectores) han sido privatizadas, de modo que el número de sentencias de reducción se reduce a cuatro (en lugar de seis) dentro de la iteración. Esta optimización se encuentra a menudo en la literatura [20] aplicada a esta clase de códigos, pero requiere conocimiento adicional de la aplicación. Sin esta privatización escalar, TinySTM y TinySTM-Ordered empeoran su rendimiento debido al aumento en la tasa de abortos —la contención en reducciones escalares es máxima—. ReduxSTM, sin embargo, es capaz de filtrar los conflictos derivados de estas sentencias de reducción, por lo que su rendimiento apenas varía si no se realiza esta optimización. En cualquier caso, el tamaño reducido de los *arrays* de reducción hace que el uso de privatización y, en menor medida, FGL obtengan los mejores resultados aquí.

Euler

La figura 3.25 muestra los resultados para Euler. En estos experimentos, el soporte de reducciones permite a ReduxSTM obtener un mejor rendimiento respecto a TinySTM. El rendimiento de éste último se degrada rápidamente debido al aumento de los conflictos con transacciones medianas y grandes. Por su parte, ReduxSTM aprovecha la ausencia de conflictos causados por reducciones para mantener un buen rendimiento con tamaños de transacción mayores obteniendo mejores resultados. Como en experimentos anteriores, las técnicas de privatización y FGL obtienen los mejores resultados.

Además de la carga computacional original, se ha incluido un perfil con una carga computacional adicional diez veces superior asociada al cálculo de las variables de reducción (función `Compute` en la línea 9 de la figura 3.2). Este experimento permite observar cómo los todos los perfiles analizados obtienen un mayor rendimiento en este tipo de escenarios. El caso de ReduxSTM es destacable, ya que es capaz de alcanzar *speedups* similares a las técnicas de privatización y FGL en problemas con cargas computacionales elevadas.

UNIVERSIDAD DE MÁLAGA

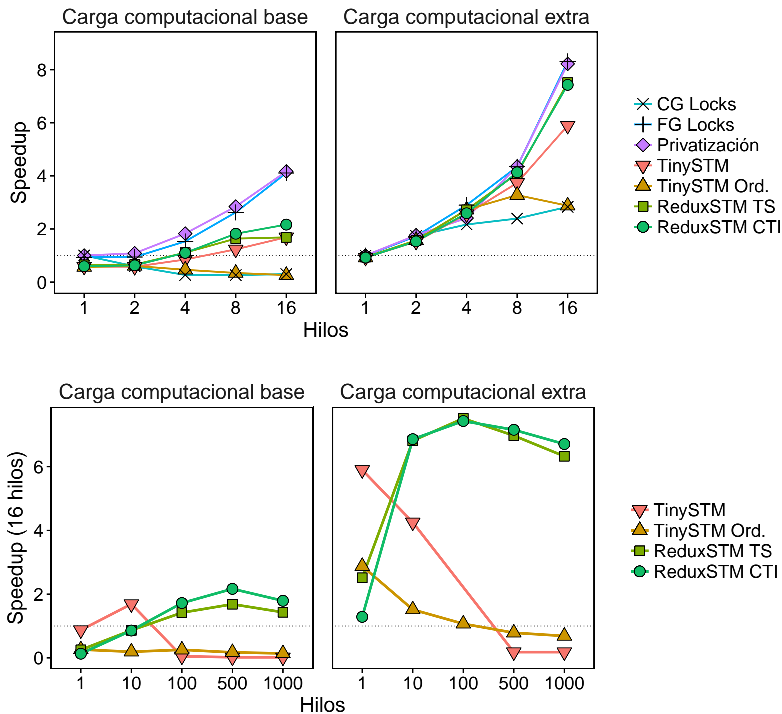


Figura 3.25: Euler: comparativa de rendimiento (arriba) e influencia del tamaño de transacción con 16 hilos de ejecución (abajo).

Legendre

Los resultados de Legendre se muestran en la figura 3.26. En este caso no se han considerado varios tamaños de transacción debido al bajo número de iteraciones del bucle a paralelizar. Ninguna de las técnicas analizadas consigue buenos resultados en este *benchmark*, dado que la baja carga computacional del *kernel* hace mucho más notable la penalización introducida por la instrumentación adicional requerida.

UNIVERSIDAD DE MÁLAGA

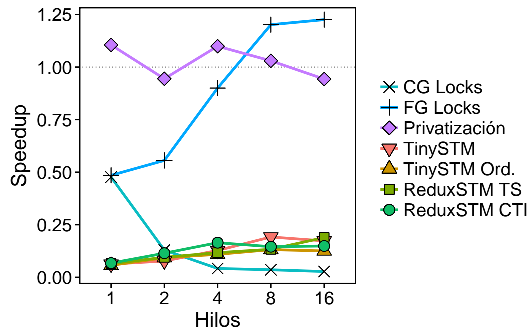


Figura 3.26: Legendre: comparativa de rendimiento (los perfiles STM ejecutan una sola iteración por transacción).

Requerimientos de memoria

Para finalizar esta evaluación, se ha realizado una estimación del uso adicional de memoria que requieren las técnicas analizadas respecto a la ejecución secuencial sin instrumentar en cada uno de los *benchmarks* analizados. La comparativa se muestra en la figura 3.27. En este análisis se ha considerado un tamaño de 8 bytes para variables y punteros y de 4 bytes para las estructuras de *locks* (*spinlocks* de POSIX).

Las técnicas de privatización utilizan *arrays* de reducción privados para cada hilo, por lo que sus requerimientos de memoria son proporcionales al número de hilos de ejecución y al tamaño del *array* de reducción del código original. Normalmente es el método con mayores requerimientos de memoria. Dichos requerimientos pueden calcularse como $S \cdot T$, donde S es el tamaño del conjunto de *arrays* de reducción del problema y T es el número de hilos de ejecución (ver tabla 3.6).

El uso de *locks* de grano fino (FGL) requiere un uso de memoria comparable a las técnicas de privatización, ya implica un *array* de *locks* de S elementos. Sin embargo los requerimientos de memoria no aumentan con el número de hilos de ejecución. En las gráficas se observa un uso de memoria ligeramente inferior al de la privatización para un sólo hilo de ejecución debido al tamaño considerado para los *locks* y el resto de variables. En el caso de usar *locks* de grano grueso (CGL) sólo es necesario un *lock*, siendo la técnica con menores requisitos de memoria.

En los perfiles que involucran el uso de STM, los requerimientos de memoria adicionales provienen en su mayor parte de los metadatos y estructuras que

UNIVERSIDAD DE MÁLAGA

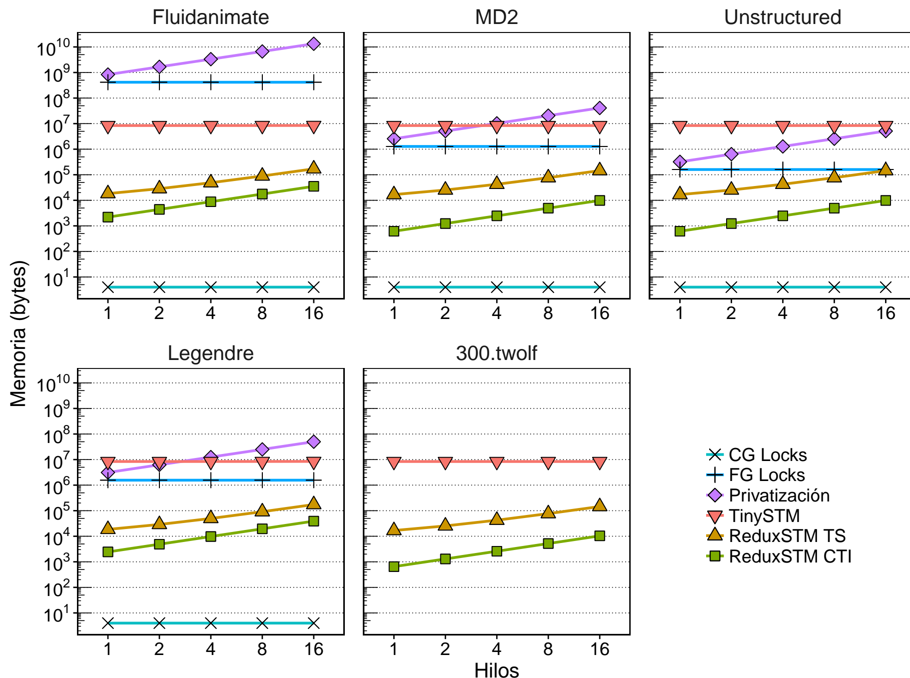


Figura 3.27: Uso de memoria de las diferentes técnicas de paralelización.

almacenan los búferes y los conjuntos de datos de cada transacción. Estas estructuras se reciclan entre todas las transacciones que ejecuta cada hilo, por lo que los requerimientos de memoria dependen del número de accesos de memoria *diferentes* realizados por las transacciones. Para calcular esta ocupación se ha monitorizado el número de direcciones únicas accedidas por cada transacción durante la ejecución de cada código. Para cada una de estas direcciones se asume que la entrada correspondiente contendrá como mínimo dos valores de 8 bytes: la dirección accedida y su valor asociado.

Además de los conjuntos de datos accedidos, TinySTM declara un array constante de 2^{20} elementos de 8 bytes para almacenar los *versioned locks* que utiliza para la detección de conflictos. También necesita 24 bytes adicionales por cada entrada del conjunto de escritura que almacenan los metadatos adicionales para la gestión de versiones. Estos valores han sido incluidos en los cálculos para obtener un límite conservador de los requerimientos de memoria. De este modo, la memoria adicional utilizada en TinySTM y TinySTM-Ordered se puede

UNIVERSIDAD DE MÁLAGA

estimar como $8 \cdot 2^{20} + \eta\,(2+3)\,8\,T$, donde η representa el número máximo de direcciones diferentes accedidas durante una transacción en cada *benchmark*, y $8 \cdot (2+3)$ corresponde al uso de dos elementos de 8 bytes para almacenar la dirección y el valor de cada entrada y 3 elementos de 8 bytes para almacenar los metadatos adicionales.

ReduxSTM-CTI emplea dos filtros de Bloom por transacción de 2^{10} entradas (1 bit por entrada) para la detección de conflictos. Por su parte, la implementación del *OpSet* necesita 2 variables de 8 bytes por entrada (los metadatos adicionales aprovechan los tres últimos bits del campo de dirección) y dos arrays de 2^8 elementos de 2 bytes para las estructuras adicionales que permiten acelerar los accesos al mismo. Sus requerimientos de memoria en bytes se pueden obtener como $(2 \cdot 2^7 + 16\,\eta + 2 \cdot 2^8 \cdot 2)\,T$.

ReduxSTM-TS necesita un *array* de *timestamps* de lectura por cada hilo de ejecución y un array de *timestamps* de escritura compartido por todos los hilos. Ambos *arrays* contienen 2^{10} elementos de 8 bytes. El *OpSet* es similar al utilizado en ReduxSTM-CTI, y también utiliza los *arrays* de 2 bytes para acelerar el acceso al conjunto de escritura. Los requerimientos de memoria se pueden calcular como $(2^{10} \cdot 8 + 16\,\eta + 2 \cdot 2^8 \cdot 2)\,T + 2^{10} \cdot 8$.

Conviene señalar que, en aplicaciones con alta contención, la probabilidad de obtener falsos positivos en ReduxSTM depende en buena parte del tamaño de los metadatos, lo que puede afectar al rendimiento. En estos escenarios, aumentar el tamaño de los filtros (ReduxSTM-CTI) o de los *timestamps* (ReduxSTM-TS) puede ser conveniente de acuerdo a los requerimientos de memoria de la aplicación particular. En esta evaluación se ha mantenido el tamaño de 2^{10} elementos en ambos STM como solución de compromiso entre la tasa de falsos positivos y el tamaño de los metadatos, tal y como se mostró en el análisis de sensibilidad de la figura 3.18 (abajo).

Aunque considerando únicamente el rendimiento las técnicas de privatización y FGL han obtenido los mejores resultados a la hora de paralelizar aplicaciones con reducciones irregulares, los requisitos de memoria pueden ser demasiado altos para determinados problemas respecto a cualquiera de los sistemas STM analizados, como muestra la figura 3.27. En estos casos, el uso de técnicas TM puede ser un método viable a la hora de extraer una cantidad significativa de paralelismo manteniendo un uso de memoria contenido.

UNIVERSIDAD DE MÁLAGA

3.4.3. Reducciones Parciales

Este escenario corresponde a bucles que contienen sentencias de reducción pero que no cumplen las propiedades de reducción en todas sus iteraciones debido a la interferencia de otros accesos (lecturas y escrituras) en las variables de reducción (ver sección 3.2). En estas situaciones los métodos clásicos de paralelización de reducciones no son aplicables. En el caso de utilizar TM es necesario establecer restricciones de orden para garantizar resultados correctos, dado que se pueden producir dependencias *read-after-write* en algunas iteraciones del bucle a paralelizar, que deben ser resueltas de acuerdo al orden secuencial de ejecución.

TWolf

Para evaluar el comportamiento de ReduxSTM a la hora extraer paralelismo en estas situaciones se ha elegido la función `new_dbox_a()`, que forma parte del *kernel* de la aplicación *300.twolf* (SPEC CPU2000). Su interés reside en el bucle que se muestra en la figura 3.5 (a), que contiene sentencias potenciales de reducción. Sin embargo, como las variables compartidas son accedidas mediante punteros, pueden existir alias en las referencias que no pueden ser detectados hasta la ejecución [54].

Estos experimentos se han evaluado utilizando la siguiente metodología: en una primera fase, el código secuencial original se ha instrumentado para obtener una traza de los accesos a memoria de la función de interés `new_dbox_a()`. Esta traza contiene una relación temporal de cada posición accedida y el tipo de acceso realizado (lectura, escritura o reducción). En una segunda fase, la traza es simulada recreando el patrón de accesos instrumentado previamente y la carga computacional original. La simulación se realiza en paralelo utilizando TM particionando el bucle más externo de la función simulada (líneas 3 a 35 de la figura 3.5 (a)) en *chunks* de iteraciones consecutivas y asignando estos *chunks* a transacciones mapeadas a su vez en los distintos hilos de ejecución mediante un protocolo *round-robin*. En este caso es necesario establecer restricciones de orden para preservar la semántica secuencial. El apéndice A.2 ofrece detalles adicionales de esta metodología.

Los experimentos mostrados se han llevado a cabo con una carga de trabajo media (correspondiente al perfil de ejecución *training* de SPEC2000). En esta configuración se ejecutan un total de 12 millones de iteraciones del bucle externo en la función `new_dbox_a()`. La traza de memoria obtenida contiene alrededor de 500 millones de lecturas, 46 millones de escrituras y 70 millones de reducciones.

UNIVERSIDAD DE MÁLAGA

Del total de operaciones de memoria se ha observado que sólo el 0.15% de las referencias son a direcciones diferentes, lo que indica un nivel alto de contención en la ejecución. Por último hay que destacar que paralelismo explotable viene limitado por la baja intensidad computacional del código (*memory-bounded*).

La figura 3.28 muestra los resultados de *speedup* y TCR obtenidos ejecutando una y dos iteraciones del bucle original en cada transacción. Aunque TinySTM ha sido incluido en la comparativa en su versión no ordenada, los resultados obtenidos con este STM son incorrectos ya que en este caso las restricciones de orden son necesarias para garantizar la corrección de los resultados. El resto de perfiles, incluyendo TinySTM-Ordered sí garantizan resultados correctos. Como referencia se muestra un perfil adicional de paralelización mediante el uso de un *lock* de grano grueso para proteger la sección crítica, que no consigue escalar con ninguna de la configuraciones debido a la contención del problema.

En este experimento los mejores resultados se obtienen con el tamaño de *chunk* más pequeño (una sola iteración por transacción). Este fenómeno se explica por la gran cantidad de operaciones de memoria que contiene cada iteración unido a la elevada contención de código. Al utilizar *chunks* mayores el número de conflictos aumenta y las transacciones descartan una cantidad considerable de trabajo al abortar, lo que reduce demasiado el rendimiento. La capacidad de filtrado de conflictos de ReduxSTM se evidencia en la gráfica de TCR, donde podemos observar valores por encima de 0.8 para la versión ReduxSTM-TS con 16 hilos de ejecución. ReduxSTM-CTI reduce este valor a 0.6 e incluso obtiene valores por debajo de 0.5 si se ejecutan dos iteraciones por transacción. Este comportamiento coincide con el observado en el *benchmark* RXasRW (sección 3.4.1), donde transacciones demasiado grandes penalizan más a este perfil debido a que debe comprobar conflictos con transacciones activas. No obstante podemos observar que el valor del TCR no se traduce directamente en el *speedup* obtenido. ReduxSTM-CTI, aunque obtiene peores resultados que la versión TS, es menos sensible a la variación del tamaño de la transacción. TinySTM, por su parte, no consigue un buen rendimiento en este escenario: la alta contención y la imposibilidad de filtrar conflictos derivados de las operaciones de reducción dan lugar a un TCR demasiado bajo incluso en el escenario más favorable (una iteración por transacción y dos hilos de ejecución).

3.4.4. Códigos generales

Aunque ReduxSTM está orientado a la optimización de códigos con restricciones de orden y reducciones, en esta sección se evalúa el rendimiento de las dos

UNIVERSIDAD DE MÁLAGA

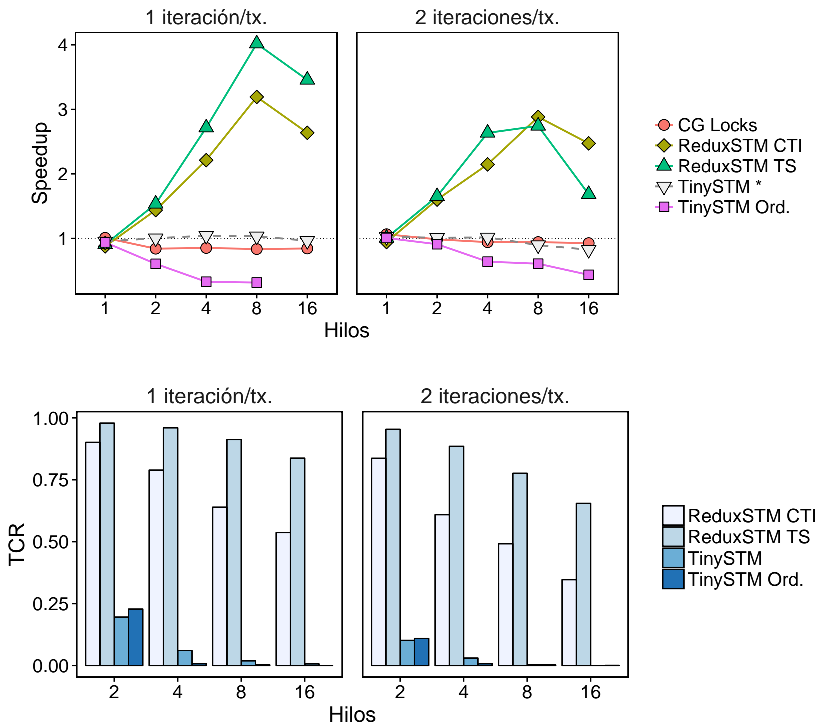


Figura 3.28: *Speedup* (arriba) y TCR (abajo) de la rutina `new_dbox_a()` del código *300.twolf* (SPEC CPU2000).

implementaciones como sistemas TM generales. Dado que ReduxSTM incluye restricciones de orden, este análisis puede dar una idea de la penalización que supone dicho orden en aplicaciones donde no se requiera para garantizar una ejecución correcta y donde esta penalización no sea enmascarada con la ganancia potencial de rendimiento derivada del tratamiento explícito de patrones de reducción.

STAMP

La suite de Stanford para memoria transaccional STAMP [14] se compone de un conjunto de *benchmarks* diseñados para TM que trata de cubrir un amplio espectro de algoritmos y dominios, haciendo especial énfasis en aquellos donde no es

UNIVERSIDAD DE MÁLAGA

Aplicación	Número de Transacciones	Tamaño de transacción	Tiempo en transacciones	Contención	Dominio
Bayes	2518	Grande	Alto	Alta	Aprendizaje automático
Genome	2139692	Medio	Alto	Baja	Bioinformática
Intruder	23428126	Pequeño	Medio	Alta	Seguridad
KMeans	87382	Pequeño	Bajo	Baja	Data Mining
Labyrinth	1026	Grande	Alto	Alta	Ingeniería
SSCA2	22362279	Pequeño	Bajo	Baja	Ciencia
Vacation	4194304	Medio	Alto	Baja/Media	Procesamiento online
Yada	2415298	Grande	Alto	Media	Ciencia

Tabla 3.7: STAMP: características de las aplicaciones de la suite.

trivial realizar una paralelización eficiente, y presentando diversas situaciones en cuanto a tamaños de transacción, nivel de contención o porcentaje del tiempo de ejecución empleado en código transaccional. Incluye ocho programas basados en aplicaciones reales cuyas características que se resumen en la tabla 3.7. Los experimentos mostrados utilizan la configuración recomendada en el artículo original para sistemas reales[21][14].

En la figura 3.29 se puede observar que el rendimiento de ReduxSTM se sitúa entre TinySTM y TinySTM-Ordered. Algunas razones de este comportamiento son la penalización introducida por las restricciones de orden, la baja tasa de abortos obtenida en los *benchmarks* con la configuración utilizada y la ausencia de patrones de reducción que puedan ser explotados por nuestra propuesta. En códigos como SSCA2, KMeans e Intruder, el tamaño reducido de las transacciones limita especialmente el rendimiento de ReduxSTM-CTI, que resulta mucho más efectivo en escenarios con transacciones largas y con mayor contención, como en Bayes o Labyrinth. ReduxSTM-TS, por otra parte, obtiene mejores resultados, que son competitivos con TinySTM-Ordered en la mayoría de casos.

El único código de la suite que contiene patrones de reducción es KMeans, mostrado en la figura 3.5 (b). Sin embargo, la explotación de dichos patrones no ha sido tan significativa como se esperaba. Dos hechos explican este comportamiento: primero, las transacciones que contienen estos accesos son especialmente reducidas, lo que, unido a la restricciones de orden, causa un efecto barrera entre las transacciones ejecutadas por hilos diferentes debido a la serialización de las fases de *commit*, lo que perjudica notablemente el rendimiento de ReduxSTM y TinySTM-Ordered. Además la implementación de KMeans de STAMP utiliza

[21]STAMP dispone de tres configuraciones para cada *benchmark*. Dos de ellas utilizan conjuntos de datos reducidos y están pensadas para su uso en entornos de simulación. La tercera, usada en esta tesis, utiliza conjuntos de datos mayores y está pensada para su uso en sistemas reales.

UNIVERSIDAD DE MÁLAGA

una cola para distribuir la carga de trabajo entre los hilos, lo que añade una serie de dependencias R – RDX debidas a la comprobación de una condición dentro del bucle de reducción. Esta comprobación viola las condiciones de reducción en el bucle, lo que disminuye la capacidad de ReduxSTM de filtrar los conflictos derivados de esos accesos. No obstante, ambos problemas son causados por la implementación específica de STAMP, y podrían evitarse si KMeans fuese reprogramado, ya que el algoritmo original sí se puede considerar un bucle de reducción completo. En este caso hemos optado por no modificar el código de STAMP al estar evaluando un escenario de rendimiento general, sin beneficios derivados del filtrado de reducciones.

Otro hecho destacable es que algunos códigos requieren al menos garantías de *VWC* por parte del STM para poder ejecutarse correctamente (Bayes, Yada e Intruder) [95]. La figura 3.30 muestra la ganancia de rendimiento obtenida en ReduxSTM-TS al relajar las garantías de consistencia a cuando es posible hacerlo. Por una parte, garantizar *serializability* produce una ganancia notable de rendimiento en varios de los *benchmarks*, al ser una condición de consistencia más relajada (ver sección 2.4.2). En el extremo opuesto, *opacity* ofrece la mayor garantía posible a costa de degradar notablemente el rendimiento en *benchmarks* con tamaños de transacciones medios y altos, debido a la necesidad de validaciones adicionales. VWC es un buen compromiso en ReduxSTM, ya que ofrece unas garantías similares a *opacity* en la práctica, sin introducir una penalización excesiva en el rendimiento en la mayoría de casos.

3.4.5. Soporte para técnicas de especulación

El último escenario analizado es la posibilidad de explotación de paralelismo en códigos secuenciales mediante técnicas especulativas. La idea aquí es mantener al menos uno de los hilos de ejecución ejecutando código no especulativo (*safe thread*) [73] y lanzar hilos que intenten paralelizar estructuras como bucles ejecutándolos en paralelo y de forma especulativa siguiendo un esquema DO-ALL. Un *runtime* —en nuestro caso el sistema transaccional—, monitoriza los hilos especulativos para detectar posibles conflictos, de modo que descarte el trabajo si los encuentra. Si el bucle se pudo ejecutar sin conflictos, los resultados pueden ser validados y actualizados en memoria principal.

Los STM con restricciones de orden pueden resultar adecuados para soportar este tipo de técnicas, ya que las propias transacciones dan soporte a la ejecución especulativa, el gestor de conflictos puede detectar dependencias en la especulación y las restricciones de orden proporcionan de forma natural un hilo *safe*, que será el que tenga menor prioridad de entre los activos en el sistema. El manteni-

UNIVERSIDAD DE MÁLAGA

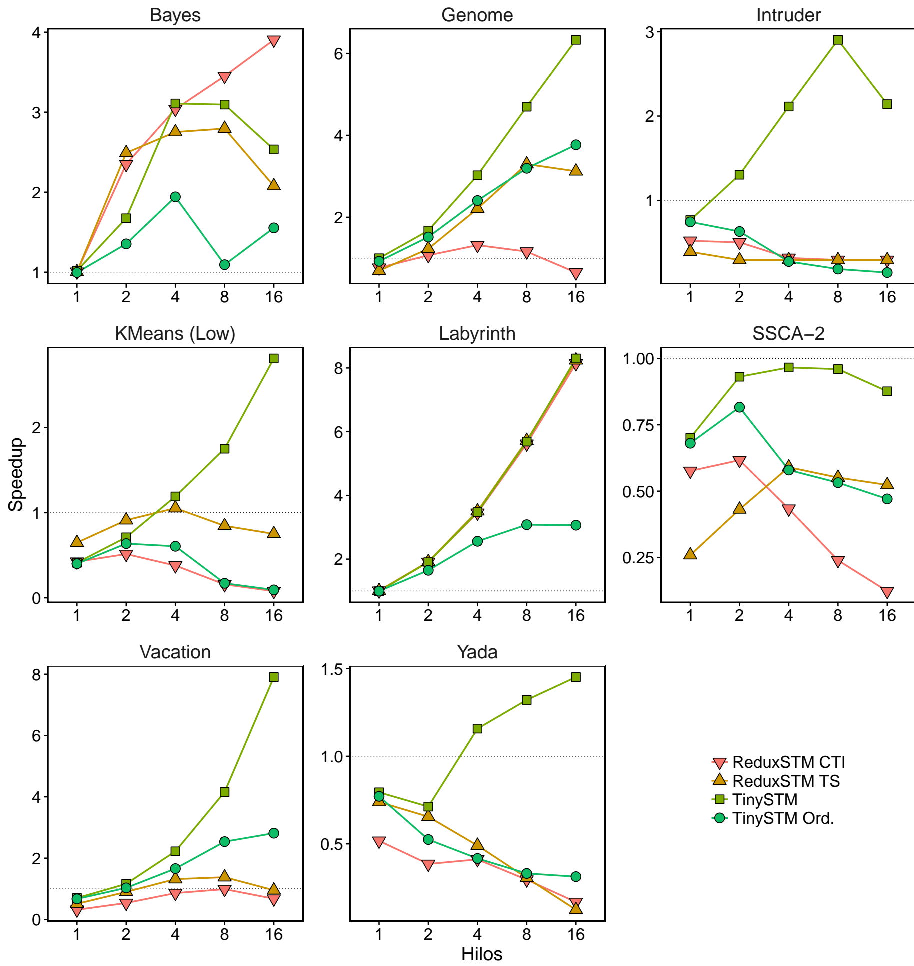


Figura 3.29: STAMP: comparativa de rendimiento.

UNIVERSIDAD DE MÁLAGA

miento de la consistencia secuencial se consigue ordenando adecuadamente las fases de *commit* de las transacciones.

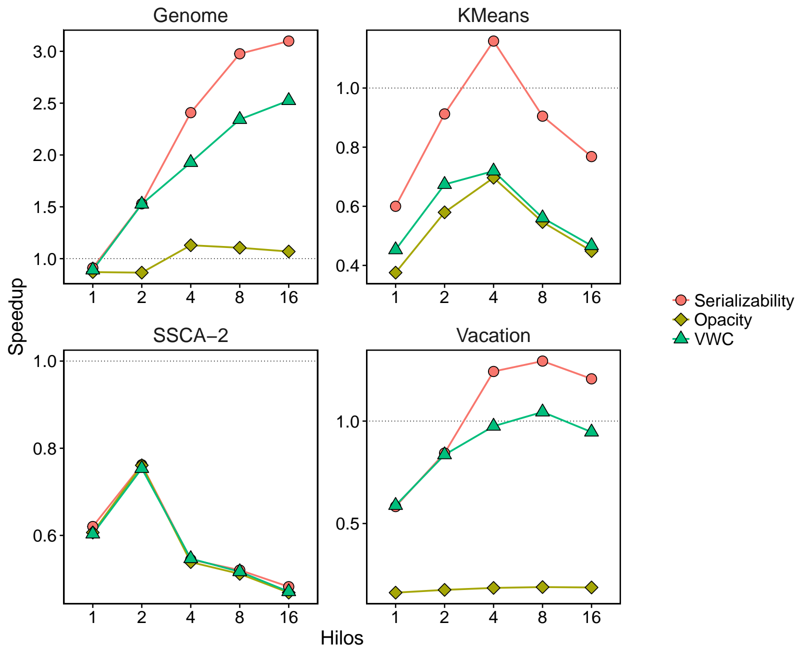


Figura 3.30: ReduxSTM-TS: *speedup* con diferentes criterios de consistencia para los códigos de STAMP que pueden ejecutarse con criterios relajados.

SPEC2006

Este análisis incluye experimentos realizados con algunos bucles de interés de la suite SPEC CPU2006. Los bucles han sido seleccionados a partir del estudio realizado en [80], que evalúa el potencial para aplicar técnicas de especulación en esta suite a partir de estudios previos [82]. Todos ellos representan un porcentaje significativo del tiempo de ejecución del *benchmark* asociado y se detallan en la tabla 3.8. Nuestro objetivo es evaluar la capacidad de ReduxSTM como soporte para realizar técnicas TLS, es decir, cómo se comporta ReduxSTM en códigos sin un énfasis especial en reducciones, pero que resultan adecuados para a la hora de aplicar técnicas de especulación. En este contexto, garantizar que el *commit* de las transacciones respete el orden secuencial equivalente es necesario para garantizar la corrección de los resultados.

UNIVERSIDAD DE MÁLAGA

Benchmark	Fichero	Línea
429.mcf	`pbeampp.c`	165
433.milc	`quark_stuff.c`	1523
456.hmmer	`fast_algorithms.c`	133
464.h264ref	`mv-search.c`	394
482.sphinx3	`vector.c`	513

Tabla 3.8: Selección de bucles de la suite SPEC CPU2006 con potencial para aplicar técnicas de especulación.

En esta evaluación se ha utilizado una metodología similar a la descrita en la sección 3.4.3 para el *benchmark 300.twolf*. Los experimentos se han realizado utilizando los parámetros por defecto (*reference workload*) de SPEC CPU2006. Aunque la versión no ordenada de TinySTM se ha incluido en la evaluación a efectos comparativos, hay que recalcar que los resultados obtenidos con este STM no serán correctos al no incluir restricciones de orden entre las transacciones.

La figura 3.31 muestra los resultados obtenidos. El valor del *speedup* mostrado se corresponde al obtenido en los bucles analizados.

El bucle 429.mcf contiene una dependencia *read-after-write* entre iteraciones que serializa las transacciones de todos los STM evaluados. Aunque ReduxSTM podría explotar una sentencia de reducción presente en el cuerpo del bucle, el uso de la variable de reducción como índice para un acceso posterior a un *array* viola las condiciones de reducción al crear una interacción *read-after-reduction* que impide cualquier ganancia de rendimiento (ver tabla 3.2). En este caso, el uso de optimizaciones adicionales como *data-forwarding* podrían ser necesarias para mejorar el rendimiento [80].

El bucle 433.milc no presenta dependencias reales entre iteraciones. En este caso la escalabilidad del los STM está limitada por la presencia de conflictos debidos a falsos positivos, especialmente en TinySTM, que muestra un TCR especialmente bajo. Las diferencias TCR entre TinySTM-Ordered y ambas versiones de ReduxSTM evidencian la importancia de adaptar el protocolo de detección y resolución de conflictos a las restricciones de orden. Aun así, la dificultad de los diseños STM a la hora de explotar la localidad de los datos y la instrumentación adicional requerida también limitan su rendimiento respecto al código secuencial. Ambos diseños de ReduxSTM escalan progresivamente con el número de hilos, pero es la versión ReduxSTM-TS la que produce mejores resultados.

El bucle 456.hmmer incluye una recurrencia que hace que cada iteración

UNIVERSIDAD DE MÁLAGA

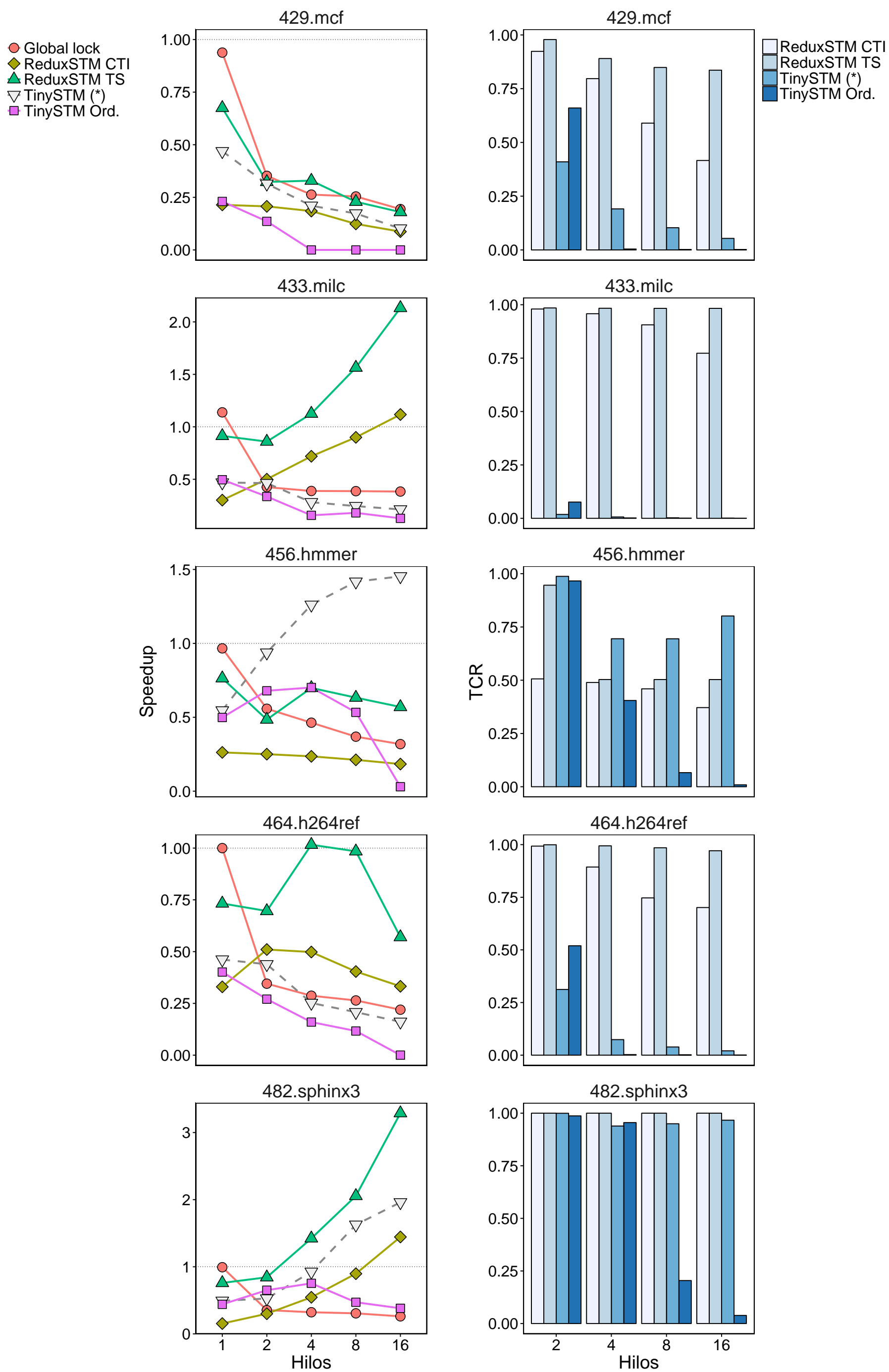


Figura 3.31: *Speedup* y TCR para una selección de bucles de SPEC CPU2006.

UNIVERSIDAD DE MÁLAGA

del bucle dependa de la anterior (dependencia *read-after-write*). Esto causa la serialización de las transacciones en los sistemas STM ordenados a menos que se incluyan optimizaciones como *data-forwarding*. En este caso, TinySTM mantiene un valor elevado de TCR al aumentar el número de hilos, ya que no considera las dependencias derivadas de la necesidad de mantener el orden secuencial de ejecución.

El bucle 464.h264ref presenta dependencias *write-after-write* debidas a falsos positivos. Aunque ReduxSTM consigue filtrar dichas dependencias y mantiene un valor alto de TCR, la baja carga computacional de este algoritmo impide alcanzar un *speedup* competitivo.

El bucle 482.sphinx3 presenta pocos conflictos, causados en su mayoría por dependencias *write-after-write* debidas a falsos positivos. En este bucle ambas versiones de ReduxSTM mantienen un valor alto de TCR, ya que los falsos positivos que podrían limitar su rendimiento corresponden a dependencias *write-after-write*, que pueden ser filtradas de forma efectiva. Este no es el caso de TinySTM-Ordered, cuyo TCR es mucho menor. Por su parte, TinySTM consigue mantener un TCR elevado al evitar las dependencias derivadas de las restricciones de orden.

En el análisis realizado en [80] se destacan ciertas desventajas de utilizar HTM para técnicas de especulación, como abortos causados por situaciones de inversión de orden y dependencias *write-after-write* y *write-after-read*. En su mayoría, estas desventajas se deben a la falta de características avanzadas en las implementaciones HTM comerciales. ReduxSTM no ve afectado su rendimiento en este tipo de situaciones.

3.5. Trabajos relacionados

Durante los últimos años los sistemas STM han sido objeto de un estudio intensivo, que ha dado lugar a un gran número de diseños y algoritmos. Una buena taxonomía puede encontrarse en [111], que señala que la gran cantidad de algoritmos disponibles puede incluso dificultar su utilización, ya que cada uno presenta ventajas para una serie de códigos o patrones de memoria específicos. El diseño de ReduxSTM, aunque introduce nuevas operaciones, hereda algunas características de diseños existentes como algoritmos *time-based* (TinySTM/LSA [34, 33], TL2[28]), TML [106], NOrec [23] e InvalSTM [45].

El uso de TM como soporte para técnicas de paralelización especulativa ha sido explorado previamente en el campo de la computación científica [79, 70, 107, 7] y

UNIVERSIDAD DE MÁLAGA

como método de paralelización para códigos binarios o *legacy* [26, 98]. Desde hace décadas se han propuesto técnicas especulativas para la paralelización de bucles como LRPD [91], basadas en un paradigma inspector/ejecutor, donde las posibles dependencias entre iteraciones son analizadas para variables estáticas antes de la ejecución, y un sistema ejecutor se encarga de lanzar grupos de iteraciones que no contengan dependencias entre sí. LRPD es capaz de privatizar variables de reducción adecuadamente para evitar dependencias derivadas de las mismas.

Más recientemente, estas ideas han sido reformuladas en *Privateer* [60], un sistema TLS capaz de analizar dependencias que involucren punteros y estructuras de datos dinámicas. *Privateer* introduce un criterio de privatización, que determina si un bucle puede ejecutarse completamente en paralelo, y añade un criterio de reducción para evitar las posibles dependencias causadas por operaciones de reducción. Cualquier otra dependencia entre iteraciones invalida el criterio de privatización y descarta la ejecución especulativa, reejecutando el bucle de forma secuencial.

En el contexto de sistemas TLS ordenados, IPOT (*Implicit Parallelism with Ordered Transactions*) [109] se propone como un modelo de programación enfocado en la paralelización de códigos secuenciales que permite definir grupos de iteraciones, las cuales son ejecutadas en paralelo con restricciones de orden utilizando estructuras similares a las transacciones. IPOT requiere soporte adicional tanto a nivel de compilador como a nivel arquitectural para soportar el entorno de ejecución especulativa. IPOT proporciona una serie de anotaciones que son tratadas como *hints*, permitiendo categorizar variables (*read-only, private, reduction*) y relajar algunos criterios de consistencia para aumentar el rendimiento.

De modo similar, ALTER [108] propone otro esquema TLS con estructuras similares a las transacciones. Las variables pueden ser anotadas para habilitar criterios de consistencia más relajados como *out-of-order*, donde se deshabilitan las restricciones de orden para TLS; o *stale-read*, que ignora las dependencias de lectura, permitiendo que el algoritmo lea valores incorrectos (con ciertas restricciones) durante la ejecución. Adicionalmente, una variable puede ser anotada como reducción. ALTER utiliza un esquema de ejecución *fork-join* para los bucles anotados, y ejecuta grupos de iteraciones en estructuras similares a transacciones, que ejecutan un proceso de validación al final de su ejecución.

Otra propuesta TLS para patrones de reducción se presenta en [54], y se centra específicamente en variables de reducción parcial (PRV). Un algoritmo de detección de PRV permite crear tareas especulativas y ejecutarlas en paralelo. Los hilos se bloquean cuando una variable de reducción es accedida por un operador que viola las condiciones de la reducción. Los conflictos entre reducciones no

UNIVERSIDAD DE MÁLAGA

producen bloqueos, ya que se privatizan en réplicas privadas que serán publicadas en una fase posterior. Esta propuesta requiere soporte específico en la arquitectura hardware.

En [42] se explora la posibilidad de paralelizar bucles de reducción de histograma utilizando sistemas TM mediante una privatización selectiva de las operaciones. Esta propuesta requiere garantías de que las condiciones de reducción se mantengan a lo largo del bucle y de que no existan dependencias adicionales entre las iteraciones, ya que deshabilita la detección de conflictos durante la ejecución de estos bucles. Esta propuesta está soportada por TEPO [44], un gestor de transacciones basado en TinySTM que permite la ejecución de transacciones con restricciones de orden.

Read-Modify-Write (RMW) *without aborts* [96] es otra propuesta STM reciente que optimiza patrones que implican la lectura (*read*) y posterior modificación (*write*) de un objeto mediante una función (*modify*). Las reducciones son un caso particular de este patrón. RMW y ReduxSTM comparten algunas características, como el uso de conjuntos específicos para estas operaciones, la transición de las reducciones/RMW a lecturas y escrituras cuando los objetos implicados son accedidos por otras operaciones y la publicación de los cambios durante la fase de *commit*. Sin embargo, esta propuesta no puede aprovechar las propiedades de asociatividad y conmutatividad que sí tienen las operaciones de reducción: las RMW no pueden combinarse entre sí, siendo necesario añadirlas a un *log* para replicar la función *modify* de cada RMW individual durante la fase de *commit* de cada transacción. De este modo la escalabilidad de esta propuesta está fuertemente determinada por la cantidad de operaciones RMW y por el coste asociado a la función *modify* de las mismas. Además, la política de detección y resolución de conflictos no puede aprovechar las características de las operaciones de reducción, lo que impide filtrar conflictos que involucren a RMW; ni la información de orden para filtrar conflictos RDX – W y W – W.

Los conceptos en los que se basa ReduxSTM son suficientemente generales para ser aplicables a cualquier sistema STM. A diferencia de otras propuestas como IPOT o ALTER, que introducen su propia notación, ReduxSTM define una primitiva estándar a añadir a un sistema TM que simplifica su programabilidad.

Aunque es cierto que el concepto de RMW es más general que el de las operaciones de reducción, ReduxSTM consigue un buen compromiso a la hora de proporcionar oportunidades adicionales para la explotación del paralelismo en códigos irregulares a la vez que mantiene un nivel de instrumentación razonable, limitando la penalización causada por la misma.

UNIVERSIDAD DE MÁLAGA

3.6. Conclusiones

La memoria transaccional es un paradigma adecuado para aplicarse a aplicaciones irregulares, donde la explotación optimista de la concurrencia puede ser efectiva a la hora de extraer paralelismo debido a que algunas dependencias no son conocidas *a priori*. En este contexto, nuestro trabajo opta por mejorar los sistemas TM añadiendo un soporte específico para tratar de forma efectiva algunos patrones comunes de acceso a los datos. En concreto, nos hemos enfocado en patrones de reducción, que aparecen a menudo en el núcleo de este tipo de aplicaciones. Con este propósito hemos introducido ReduxSTM, un sistema STM que combina soporte específico para operadores conmutativos y asociativos con un mecanismo de garantía de orden secuencial, que resulta necesario para garantizar resultados correctos en situaciones en las que los patrones de reducción pueden coexistir con otros patrones de acceso a memoria. Gracias a este soporte, el gestor de conflictos es capaz de evitar varias situaciones relacionadas con los patrones de reducción que supondrían un aborto en sistemas TM tradicionales. Además, el sistema puede aplicarse a bucles de reducción potenciales, sin la necesidad de identificarlos previamente como bucles de reducción.

Se ha realizado una extensa evaluación para mostrar que las ideas que propone ReduxSTM pueden mejorar el rendimiento en diseños STM. ReduxSTM se ha comparado con un STM del estado del arte (TinySTM) y con otras técnicas de paralelización de reducciones en una variedad de escenarios que combinan patrones de reducción con otros tipos de acceso en diferentes grados. Los resultados obtenidos alientan a incluir este tipo de soporte en futuros sistemas STM.

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Capítulo 4

Soporte de orden parcial en memoria transaccional hardware

Este capítulo presenta una alternativa relajada a las restricciones totales de orden entre transacciones que incluyen ReduxSTM y otros sistemas que utilizan TM como soporte para técnicas de especulación. Además propone TMbarrier, una implementación para soportar el modelo de orden parcial en sistemas HTM reales. La sección 4.1 contextualiza y motiva nuestra propuesta. La sección 4.2 analiza las limitaciones más comunes de los sistemas HTM presentes en procesadores de consumo. La sección 4.3 detalla el funcionamiento de TMbarrier y su soporte a restricciones parciales de orden. La sección 4.4 detalla el diseño e implementación de TMbarrier en el HTM del procesador IBM POWER8. La sección 4.5 evalúa TMbarrier en varios escenarios de interés. Por último, la sección 4.6 analiza otros trabajos relacionados de la literatura.

4.1. Introducción

La memoria transaccional puede utilizarse como soporte para técnicas de especulación a nivel de hilo (TLS), algo especialmente interesante a la hora de paralelizar código ya existente [26, 80, 6, 99]. Estas técnicas permiten ejecutar en paralelo secciones de código de forma optimista —esto es, bajo la asunción de

UNIVERSIDAD DE MÁLAGA

que no existen dependencias entre los distintos hilos de ejecución— y posponer las escrituras a memoria hasta que todos los hilos hayan finalizado su sección sin producir conflictos de datos. Si se detecta un conflicto, el sistema debe actuar para preservar la consistencia del código (por ejemplo, descartando el trabajo especulativo de todos los hilos y reejecutando la sección completa de forma secuencial). Los sistemas de detección de conflictos y gestión de versiones consustanciales a TM pueden aprovecharse como soporte para técnicas de especulación. Normalmente esto conlleva la imposición de restricciones de orden secuencial entre las transacciones de modo que sus puntos de serialización sigan el orden del código secuencial, preservando así la corrección de la ejecución en caso de que se detecte algún conflicto.

El concepto de transacción en TM no contempla restricciones de orden, lo que genera dos problemas a la hora de soportar este tipo de técnicas: por una parte, cuando dos transacciones producen un conflicto, el programador no puede asumir cuál de ellas será descartada y reiniciada y cuál continuará su ejecución, impidiendo preservar el orden del código secuencial. Por otra parte no se puede impedir que una transacción haga visibles sus cambios una vez haya finalizado, incluso si el algoritmo así lo requiere para su correcta ejecución. Algunas propuestas TM introducen restricciones totales de orden [52, 34, 74, 37]. En estos casos cada transacción se inicia con un número de orden exclusivo y creciente. Cuando una transacción termina su ejecución debe esperar a que todas las transacciones previas hayan finalizado su *commit* antes de que pueda hacer visibles sus cambios en el sistema. Esto fuerza una serialización de la fase de *commit* transaccional que, si bien preserva la equivalencia con el orden lexicográfico del algoritmo, puede tener un impacto notable en el rendimiento. ReduxSTM utiliza estas restricciones de orden para dar soporte a escenarios con reducciones parciales [84] y como soporte para especulación [85].

Existen algoritmos que sólo necesitan garantizar un orden entre ciertas fases del código [5]. En estos escenarios se suelen usar barreras para sincronizar dichas fases. Una barrera sincroniza cierto número de hilos en un punto del programa, bloqueando su ejecución hasta que todos hayan alcanzado dicho punto. Al tratarse de una abstracción muy común, los diferentes entornos para programación paralela suelen incluir soporte para barreras [61]. TM, sin embargo, no incluye una definición específica para las barreras, al estar pensado para utilizarse en conjunción con estos entornos.

El uso de barreras representa una solución pesimista al problema de sincronizar las distintas fases de un algoritmo: en previsión de que existan dependencias de datos antes y después de un determinado punto de ejecución, se impide que cualquier hilo pueda continuar más allá de dicho punto hasta que se pueda

UNIVERSIDAD DE MÁLAGA

garantizar —con la llegada del resto de hilos— una ejecución libre de conflictos. Aunque esta sincronización es necesaria, si las dependencias entre código anterior y posterior a la barrera no ocurren a menudo y los puntos de sincronización son frecuentes, el uso de barreras puede afectar sensiblemente al rendimiento final de la aplicación [78].

En este capítulo proponemos TMbarrier, una alternativa optimista a las barreras tradicionales diseñada para utilizarse en códigos transaccionales. La idea en TMbarrier es conservar la semántica de dichas barreras y a la vez permitir que un hilo pueda continuar su ejecución de forma especulativa sin necesidad de esperar al resto de hilos. TMbarrier utiliza soporte transaccional para detectar y resolver conflictos potenciales entre código anterior y posterior al punto de sincronización, estableciendo restricciones de orden *parciales* en tiempo de ejecución entre las transacciones ejecutadas antes y después de dicho punto. Esta directiva ha sido concebida para sustituir las barreras tradicionales en un contexto de ejecución transaccional con el objetivo de reducir el tiempo que los hilos desaprovechan esperando en barreras.

Aunque TMbarrier no requiere un sistema transaccional específico, la eficiencia final depende en buena parte del TM utilizado como soporte. TMbarrier requiere cierta flexibilidad por parte del mismo para introducir las restricciones de orden. Además, una menor instrumentación y la posibilidad de modificar la política de resolución de conflictos son características deseables de cara al rendimiento potencial de nuestra propuesta. Vista la penalización introducida por las restricciones totales de orden en un sistema TM software, en este capítulo nos hemos planteado realizar una implementación de TMbarrier con soporte transaccional hardware. Los sistemas HTM, si bien requieren considerar limitaciones adicionales, especialmente a la hora de implementar las restricciones parciales de orden y de establecer comunicación entre las transacciones, presentan un *overhead* mucho menor a la hora de llevar a cabo la detección de conflictos y en la fases de inicio y *commit* de las transacciones.

Hemos realizado una implementación de TMbarrier utilizando el sistema HTM del procesador POWER8 y llevado a cabo una evaluación experimental. Los resultados obtenidos muestran que nuestra propuesta es capaz de reducir significativamente el tiempo de bloqueo en barreras, mejorando así el rendimiento de las aplicaciones analizadas a pesar de las limitaciones de las implementaciones HTM actuales.

UNIVERSIDAD DE MÁLAGA

4.2. Limitaciones de los HTM comerciales

Recientemente algunas de las principales empresas de semiconductores han empezado a incluir soporte HTM en sus procesadores de consumo. Intel introdujo un conjunto de instrucciones denominadas *Transactional Extensions* (TSX) para dar soporte HTM a partir de su microarquitectura Haswell. IBM, por su parte, añadió soporte transaccional a su gama de procesadores de consumo POWER en el procesador POWER8, aunque previamente ya había incluido HTM en sus líneas de supercomputadores (Blue Gene) y de *mainframes* (System Z)[22]. AMD presentó la extensión ASF para dar soporte HTM en su arquitectura, si bien aún no ha presentado procesadores que la soporten [18].

Cualquiera de las implementaciones anteriores ofrece soporte básico para TM, incluyendo soporte hardware para la detección de conflictos, almacenamiento tentativo para las escrituras transaccionales y restauración del estado del sistema en caso de conflicto.

Del mismo modo, todas presentan una serie de limitaciones derivadas en su mayor parte de su implementación basada en el uso de la memoria caché para la implementación de la lógica transaccional:

- La capacidad limitada de la caché producirá abortos —independientemente de si existen o no conflictos— cuando los conjuntos de datos transaccionales sean mayores que el espacio disponible en la caché para su almacenamiento o cuando un bloque caché marcado como transaccional sea desalojado.

- La detección de conflictos, que se realiza aprovechando en parte el protocolo de coherencia de caché, se lleva a cabo con granularidad de bloque caché, lo que da lugar a falsos conflictos si varias transacciones acceden a datos diferentes que se mapean en el mismo bloque (*false-sharing*). Dado que el tamaño de dichos bloques es relativamente grande respecto a los tamaños de la mayoría de variables[23], el impacto de estos falsos positivos es un problema a considerar [99].

- Las implementaciones son *implícitas*: cuando un núcleo del procesador se encuentra en modo transaccional todos los accesos a memoria que realice el hilo de ejecución correspondiente son considerados transaccionales, pasando a formar parte del conjunto de datos transaccional y siendo por tanto susceptibles a producir abortos.

[22]La sección 2.5.2 analiza los sistemas mencionados.

[23]Por ejemplo, 64 bytes en Intel TSX o 128 bytes en IBM POWER8.

UNIVERSIDAD DE MÁLAGA

- El anidamiento de transacciones se realiza mediante *flattening*: cualquier transacción iniciada o finalizada en el contexto de otra es esencialmente ignorada por el sistema transaccional. Aunque este sistema es correcto desde un punto de vista semántico, un conflicto en cualquier transacción interna abortará todas las transacciones anidadas del hilo.

- El gestor de conflictos, que determina qué transacción continúa y cuál aborta en caso de producirse un conflicto, es fijo; el programador no puede modificarlo para adaptarlo a diferentes escenarios.

- Ninguna de las implementaciones HTM actuales garantiza la eventual finalización de una transacción hardware (HTM *best-effort*). Es responsabilidad del programador implementar una solución software para garantizar el progreso del sistema en última instancia.

Comunicación entre transacciones y suspended mode

Una consecuencia de que los sistemas HTM sean implícitos es la imposibilidad establecer cualquier tipo de comunicación entre transacciones activas sin causar abortos entre las mismas. Por ejemplo, implementar restricciones de orden entre transacciones implica un contador compartido que contenga el turno de *commit* actual, que deberá ser: (1) accedido por todas las transacciones activas, (2) comparado con el turno de *commit* de cada una de ellas, y (3) incrementado cada vez que finalice una transacción. El acceso al contador registra la variable en el *read-set* de todas las transacciones, por lo que su eventual incremento producirá un conflicto que abortará cualquier transacción cuyo turno de *commit* no coincida con el turno actual. Esto es una limitación conocida en propuestas que combinan TLS con soporte TM [99].

El HTM de POWER8 soporta una característica denominada *suspended mode*, que permite desactivar el modo transaccional temporalmente durante la ejecución de una transacción (ver sección 2.5.2). Todos los accesos a memoria realizados en *suspended mode* son registrados como no transaccionales por el sistema, y pueden producir abortos con cualquier transacción activa, incluyendo la que está en este modo.

Aunque el *suspended mode* está pensado para propósitos de depuración [57], se puede utilizar para proporcionar un mecanismo básico de comunicación entre transacciones activas sin producir abortos. Esta característica se utilizará como base para el diseño de TMbarrier, y es el principal motivo para utilizar el HTM de POWER8 como sistema transaccional de apoyo para nuestra propuesta.

UNIVERSIDAD DE MÁLAGA

4.2.1. Garantías de progreso en HTM *best-effort*

Las transacciones hardware en un HTM *best-effort* no tienen por qué finalizar [66]. Incluso si no hay conflictos aparentes entre las mismas, hay varios eventos que pueden producir un aborto de las transacciones, algunos de los cuales pueden impedir siempre que una transacción hardware finalice. Estos eventos[24] pueden resumirse en: (1) exceder el tamaño máximo del conjunto de datos transaccional soportado por el hardware, (2) un conflicto con otra transacción, (3) un conflicto con un acceso no transaccional, (4) la ejecución de una instrucción no permitida en modo transaccional, como una llamada a sistema o una interrupción, (5) una llamada explícita a la primitiva de aborto transaccional y (6) otras causas. Aunque algunos de estos eventos no tienen por qué impedir que una transacción abortada pueda reejecutarse con éxito, ante los eventos de los grupos (1), (4), (5) y (6) es muy probable que una eventual reejecución falle igualmente. Por esta razón los HTM *best-effort* requieren que el programador implemente un código *fallback* alternativo para garantizar el progreso de la ejecución en estas situaciones.

La alternativa más típica involucra el uso de un *lock* global siguiendo un modelo similar al mostrado en los algoritmos 5 y 6. Para mayor claridad, las líneas de color rojo indican que su ejecución se produce en el contexto de una transacción hardware, mientras que las líneas negras se ejecutan fuera del modo transaccional. La tabla 4.1 resume la notación utilizada en los algoritmos de este capítulo.

Una transacción se inicia mediante la invocación de la función `TmBegin` (algoritmo 5), donde se invoca a la instrucción hardware `HtmBegin`, que lanza una transacción hardware y devuelve `true` si se ha podido iniciar. En caso de éxito, la transacción debe suscribirse al *lock* global `fallbackLock` realizando una lectura del mismo y añadiéndolo por tanto a su conjunto de lectura (línea 8). Si la transacción no ha podido iniciarse —o ha sido abortada—, el procedimiento continúa por la rama `else` (línea 12), que comienza decrementando un contador de reintentos `tx.retries` (línea 13). Si este contador llega a cero el sistema asumirá que la transacción no puede continuar en modo hardware, por lo que intentará adquirir el *lock* global (línea 15) y pasa a ejecutarse en software de modo irrevocable. La adquisición de este *lock* produce un aborto en todas las transacciones hardware que haya ejecutándose en ese momento debido precisamente a la lectura del mismo que realizaron al iniciarse. De este modo, una transacción que aborte demasiadas veces pasa a ejecutarse en exclusión mutua de modo similar a un *lock* de grano grueso, impidiendo la ejecución concurrente del resto de hilos, pero garantizando el progreso del sistema.

[24]Esta clasificación se ha realizado a partir de un análisis de las posibles causas de aborto consideradas en las propuestas HTM de Intel, AMD e IBM.

UNIVERSIDAD DE MÁLAGA

Notación	Significado
glOrder	Número de fase global del sistema.
fallbackLock	Lock global que permite ejecutar un código *fallback software* en exclusión mutua con el resto de transacciones.
tx	Descriptor de transacción asociado a un hilo de ejecución. Las variables con prefijo tx que aparecen en la tabla conforman el contenido de este descriptor.
tx.status	Indica si una transacción hardware ha sido iniciada con éxito por el HTM.
tx.mode	Estado transaccional asociado al hilo de ejecución. La figura 4.4 indica los estados posibles y sus transiciones.
tx.retries	Número de reintentos restantes de una transacción hardware. Si llega a cero, el hilo intentará adquirir el lock global para ejecutar el cuerpo de la transacción en software.
tx.order	Número de fase local de la transacción.
tx.specLevel	Número de transacciones anidadas restantes de una transacción especulativa. Si llega a cero, el hilo no podrá continuar la especulación, bloqueándose hasta poder confirmar sus cambios o descartarlos en caso de conflicto.
tx.specMax	Número máximo de transacciones anidadas soportadas por un hilo. Este valor se ajusta dinámicamente durante la ejecución.
MaxRetries	Número máximo de reintentos permitidos para una transacción hardware.
HtmBegin	Instrucción de la ISA del procesador que intenta iniciar una transacción hardware asociada al hilo de ejecución y devuelve un booleano indicado si tuvo éxito.
HtmEnd	Instrucción de la ISA del procesador que intenta finalizar una transacción (commit). Si se ejecuta con éxito, la transacción hardware asociada al hilo hará visibles sus cambios en el sistema. En caso contrario la transacción abortará, restaurando el estado de la ejecución al inicio de la transacción.
HtmAbort	Instrucción de la ISA del procesador que aborta incondicionalmente la transacción actual asociada al hilo de ejecución.
HtmSuspend	Instrucción de la ISA del procesador que inicia el *suspended mode* en el contexto de una transacción hardware.
HtmResume	Instrucción de la ISA del procesador que finaliza el *suspended mode* en el contexto de una transacción hardware.

Tabla 4.1: Notación utilizada en la descripción de los algoritmos 5, 6, 9, 10 y 11.

Cuando la transacción finaliza se invoca el procedimiento `TmEnd` (algoritmo 6), que comprueba si la transacción se estaba ejecutando en modo hardware o software. En el primer caso se ejecuta la instrucción de *commit* hardware del HTM (`HtmEnd`), que intenta confirmar los cambios de la transacción a memoria principal. Si la transacción estaba en modo software, estos cambios se han realizado ya en modo no transaccional, por lo que sólo es necesario liberar el *lock* global `fallbackLock`, permitiendo que puedan ejecutarse el resto de transacciones (línea 5). En cualquiera de los casos el contador `tx.retries` de la transacción se restablece a su valor inicial.

El bucle de la línea 3 del algoritmo 5 evita un escenario perjudicial de cara al rendimiento denominado *lemming effect* [27, 89], que se produce cuando una

UNIVERSIDAD DE MÁLAGA

Algoritmo 5 Inicio de transacción en HTM *best-effort*

```
 1 function TmBegin( )
 2     while True do
 3         while IsLocked(fallbackLock) do        ▷ Evita lemming effect
 4         end while
 5         tx.status ← HtmBegin                   ▷ Instrucción de inicio HTM
 6         if tx.status = Success then            ▷ Transacción iniciada con éxito
 7             tx.mode ← HW                       ▷ Modo hardware
 8             if IsLocked(fallbackLock) then     ▷ Añade fallbackLock al read-set
 9                 HtmAbort                       ▷ Instrucción de aborto HTM
10             end if
11             Break                              ▷ Salto al cuerpo de la transacción (HW)
12         else                                   ▷ Transacción abortada o fallo al iniciarla
13             tx.retries ← tx.retries −1
14             if tx.retries = 0 then             ▷ Límite de reintentos. Modo software
15                 Lock(fallbackLock)             ▷ Adquisición del lock global
16                 tx.mode ← SW
17                 Break                          ▷ Salto al cuerpo de la transacción (SW)
18             end if
19         end if
20     end while
21 end function
```

Algoritmo 6 Commit de transacción en HTM *best-effort*

```
1 function TmEnd( )
2     if tx.mode = HW then                       ▷ Commit desde una transacción hardware
3         HtmEnd                                 ▷ Instrucción de commit HTM
4     else                                       ▷ Commit desde un fallback software
5         Unlock(fallbackLock)                   ▷ Liberación del lock global
6     end if
7     tx.retries ← MaxRetries                    ▷ Restaurar reintentos
8 end function
```

transacción ejecutándose en modo software causa que las transacciones hardware aborten continuamente (línea 8), saturen sus contadores de reintento y pasen en cascada a modo software. Por esta razón, los hilos deben esperar a que `fallbackLock` esté libre antes de intentar iniciar una transacción hardware.

4.3. Barreras Transaccionales

Como ejemplo motivador, y para ilustrar el funcionamiento de nuestra propuesta, consideremos la ecuación general de recurrencia lineal extraída de los Livermore Loops [35] que se muestra en el algoritmo 7. El código anterior puede paralelizarse como se muestra en el algoritmo 8 [100]. En esta paralelización, el bucle interno distribuye el trabajo por bloques y el paralelismo explotable del algoritmo se reduce en cada iteración del bucle externo. Es necesario el uso de una barrera antes de iniciar cada nueva iteración del bucle externo para garantizar que los hilos

UNIVERSIDAD DE MÁLAGA

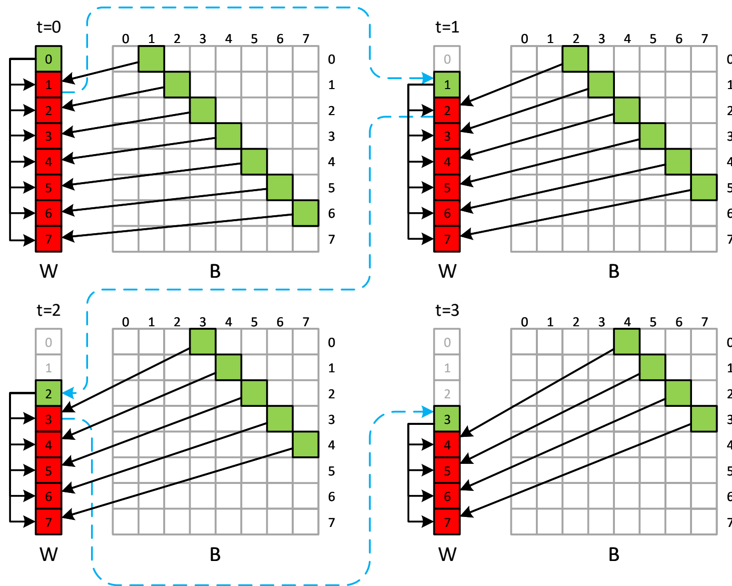


Figura 4.1: Gráfico de dependencias en Livermore Loop 6. Las flechas azules indican dependencias *read-after-write* entre iteraciones del bucle externo. Cada subfigura corresponde a una iteración externa con el valor de `t` indicado. Las lecturas y escrituras de los objetos en cada iteración se indican con los colores verde y rojo respectivamente. Las flechas negras indican dependencias en la misma iteración para generar los valores de `W`.

de ejecución acceden a valores actualizados en `W`. Sin embargo, las dependencias reales entre iteraciones de este algoritmo son mínimas, como se muestra en la figura 4.1. En cada iteración del bucle interno, sólo existe un acceso al vector `W` que provoca una dependencia real. En este escenario, el uso de barreras tradicionales puede tener un impacto considerable en el rendimiento debido al tiempo que los hilos permanecen bloqueados y a la gestión conservadora que realiza el compilador [78].

UNIVERSIDAD DE MÁLAGA

Algoritmo 7 Livermore Loop 6: Recurrencia lineal general

```
1 for i = 1 to N do
2     for k = 0 to i do
3         w[i] = w[i] + b[k][i] * w[(i − k) − 1]
4     end for
5 end for
```

Algoritmo 8 Livermore Loop 6: Versión paralela

```
1 for t = 0 to N - 2 do
2     for k = tid*chunk to (tid+1)*chunk do
3         if k < (N − t) then
4             w[t + k + 1] = w[t + k + 1] + b[k][t + k + 1] * w[t]
5         end if
6     end for
7     BARRIER
8 end for
```

Reemplazando barreras con transacciones ordenadas

Una alternativa al uso de barreras es combinar memoria transaccional con restricciones de orden total. Podemos encapsular el cuerpo de cada iteración en una transacción y establecer un orden de precedencia estricto entre las mismas que corresponda al orden de ejecución del código secuencial original. La figura 4.2 ilustra este método: el diagrama de la izquierda corresponde a un código transaccional que hace uso de barreras para sincronizar las distintas fases del algoritmo, mientras que el diagrama de la derecha puede prescindir de esta sincronización mediante el uso de restricciones de orden. Esta estrategia se ha utilizado en la literatura para habilitar sistemas TLS utilizando TM [80]. La lógica subyacente es suponer que los hilos pueden ejecutar en paralelo la mayor parte del trabajo y que sólo será necesario serializar la fase de *commit* de las transacciones, comparativamente más corta. En la práctica, sin embargo, la penalización que introduce esta serialización es a menudo demasiado alta, especialmente si se usan transacciones de tamaño reducido.

TMbarrier

Con el objetivo de reducir esta penalización, las restricciones estrictas de orden pueden relajarse en aquellos códigos donde las dependencias originales puedan ser resueltas mediante el uso de barreras. Tomando de nuevo el algoritmo 8, si el cuerpo del bucle interno es ejecutado dentro de una transacción, las transacciones ejecutadas entre las mismas barreras (aquellas que comparten una misma iteración del bucle externo), pueden finalizar en cualquier orden sin alterar el

UNIVERSIDAD DE MÁLAGA

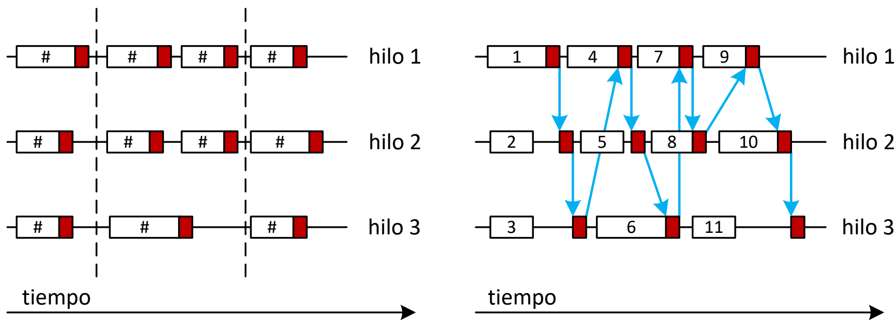


Figura 4.2: Reemplazando la sincronización con barreras mediante el uso de TM con restricciones de orden. Los rectángulos blancos representan el cuerpo de cada transacción y los rectángulos rojos representan su fase de *commit*. Las líneas discontinuas representan barreras. Las flechas azules indican el orden estricto entre la fase de *commit* de las transacciones.

resultado final. Así pues podemos definir un *orden parcial* que garantice que todas las transacciones *anteriores* a una determinada barrera finalicen antes de que cualquier otra transacción *posterior* a esa barrera pueda hacer visibles sus cambios. Esta restricción mantiene la equivalencia secuencial del código pero permite más oportunidades para la explotación del paralelismo, ya que las transacciones anteriores a la barrera pueden finalizar en paralelo.

Para dar soporte a las restricciones de orden parcial nuestra propuesta es la introducción de una nueva primitiva, de aquí en adelante `tmbarrier`[25], que proporciona información adicional a un sistema TM para habilitar dinámicamente dichas restricciones durante la ejecución.

Cuando un hilo llega a `tmbarrier`, transiciona a un modo especulativo iniciando una nueva transacción a partir de la invocación de la primitiva. Desde ese momento, todas las escrituras a memoria que realice el hilo serán retenidas por el TM hasta que (a) se detecte un conflicto o (b) el resto de hilos haya llegado a la instrucción `tmbarrier`. En el caso (a), la transacción deshace los cambios especulativos —aborta— y vuelve al punto de inicio justo después de la llamada a `tmbarrier`. En el caso (b), la transacción finaliza haciendo visibles sus cambios especulativos al resto de hilos y regresa a un modo no especulativo para continuar con la ejecución.

[25]Distinguimos entre TMbarrier, que hace referencia al sistema completo que soporta el orden parcial y `tmbarrier`, que se refiere a la primitiva que proporciona la información de orden al sistema.

UNIVERSIDAD DE MÁLAGA

Aunque la especulación podría extenderse indefinidamente mientras el resto de hilos llegan a `tmbarrier`, en la práctica no tiene sentido continuar la especulación más allá de una segunda llamada consecutiva a la primitiva. El uso de barreras en el código original es de por sí indicativo de la existencia de dependencias entre el código anterior y posterior a las mismas, por lo que especular demasiado conduciría muy probablemente a conflictos con el código no especulativo.

Por esta razón, si un hilo en modo especulativo se encuentra con una nueva invocación a `tmbarrier` es bloqueado hasta que pueda confirmar sus cambios a memoria o hasta que aborte. Esto limita implícitamente la ventana de especulación entre dos llamadas a `tmbarrier`, permitiendo simplificar el funcionamiento del algoritmo y mantener a la vez las oportunidades reales de especulación.

4.3.1. Restricciones de orden

Para soportar el modelo de orden parcial en TMbarrier cada hilo mantiene un número local de fase en sus metadatos (`tx.order`), y una variable global (`glOrder`) mantiene la fase real del sistema. Todas las fases son inicializadas a 1 al inicio de la aplicación y se actualizan durante la ejecución ante los siguientes eventos:

- `tx.order` se incrementa cuando su correspondiente hilo llega en modo no especulativo a una llamada a `tmbarrier`.
- `glOrder` se incrementa cuando el último hilo no especulativo termina su llamada a `tmbarrier`. Una transacción sólo podrá finalizar si su fase local `tx.order` coincide con `glOrder`.

Estas fases permiten al sistema identificar qué transacciones se están ejecutando antes y después de una barrera en un instante determinado. De ahora en adelante nos referiremos como transacciones *especulativas*[26] a aquellas que se inician *después* de la barrera actual en un instante determinado de la ejecución y transacciones *no especulativas* al resto. Dado que el número de fase de las transacciones especulativas no coincide con `glOrder`, no podrán finalizar hasta que todos los hilos de ejecución hayan llegado a la barrera actual y se produzca el incremento de `glOrder`.

[26] Aunque toda transacción constituye una ejecución especulativa, ya que puede abortar, mediante esta nomenclatura distinguimos el uso de transacciones para paralelización (no especulativas) de aquellas utilizadas para propósitos TLS (especulativas).

UNIVERSIDAD DE MÁLAGA

4.3.2. Transacciones anidadas

El esquema descrito hasta ahora admite una única transacción especulativa por hilo de ejecución, ya que no permite que dicha transacción pueda finalizar hasta que `glOrder` haya sido incrementado y coincida con el número de fase local de las transacciones especulativas. Mejorar la eficiencia de la especulación implica permitir la ejecución de varias transacciones especulativas en cada hilo. En nuestra propuesta esto se lleva a cabo utilizando anidamiento de transacciones [77].

El sistema TM iniciará una transacción (externa) tras la llamada a `tmbarrier`, de modo que las transacciones subsiguientes que aparezcan en el código serán iniciadas como transacciones internas anidadas a la primera. Es importante destacar que la transacción externa es instrumental (es decir, no estaba presente en el código original), mientras que las transacciones anidadas son aquellas que existían previamente en el código. Con este diseño se cumple la condición de mantener en ejecución una única transacción por hilo tras la barrera, pero esta transacción (externa) puede contener varias transacciones internas, permitiendo incrementar el grado de especulación del sistema.

4.3.3. Funcionamiento de TMbarrier

Consideremos el escenario con tres hilos de ejecución que se ilustra en la figura 4.3. Los hilos ejecutan un código paralelo en un instante en el que la fase global `glOrder` es n. El hilo 1 ejecuta una transacción con `tx.order` $= n$ y, acto seguido, llega a una instrucción `tmbarrier`. En lugar de bloquearse, el hilo incrementa su fase local `tx.order` e inicia una transacción externa (que se muestra en rojo en la figura) justo al terminar la llamada a `tmbarrier`. La ejecución del hilo 1 continúa de modo que las transacciones subsiguientes se iniciarán con un número de fase mayor `tx.order` $= n + 1$. Lo mismo ocurre en el hilo 3. El hilo 2 es el último en llegar a `tmbarrier`, y produce el incremento de `glOrder` además del de su fase local. Este hilo no cambia a modo especulativo, ya que su fase local sigue coincidiendo con el valor actualizado de `glOrder`, y continúa por tanto ejecutando el resto de transacciones normalmente. A partir de este momento, cuando el hilo 1 termina su segunda transacción especulativa, verifica que su fase local `tx.order` coincide con el valor actualizado de `glOrder`, y finaliza su transacción externa, volcando los cambios hasta ahora tentativos en la memoria compartida. Lo mismo ocurre en el hilo 3 tras finalizar su primera transacción especulativa. El hilo 1 encuentra una nueva llamada a `tmbarrier`, pero en este caso la transacción especulativa detecta un conflicto con otra transacción en el hilo 2. Esto produce un aborto en las transacciones interna y externa del hilo 1,

UNIVERSIDAD DE MÁLAGA

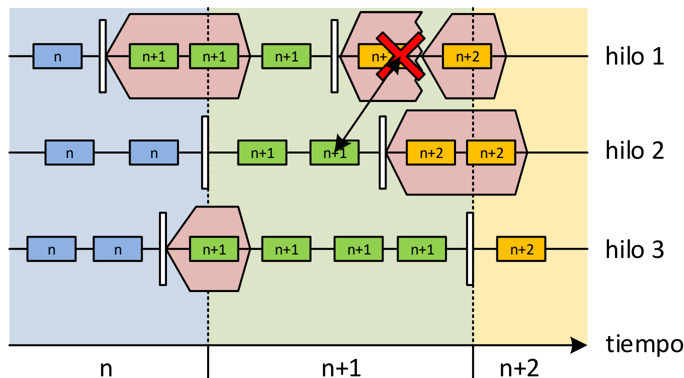


Figura 4.3: Funcionamiento de TMbarrier. Los rectángulos coloreados representan transacciones, y las barras blancas representan barreras. Los hexágonos rojos representan transacciones externas que contienen a su vez una o varias transacciones internas. Cada n del interior de una transacción representa su fase local (`tx.order`). La fase global `glOrder` se representa en la parte inferior del diagrama y cambia con la ejecución de la última barrera.

descartando los cambios especulativos y devolviendo la ejecución al instante posterior a la segunda barrera. En ese momento el hilo 3 aún no ha llegado a la segunda barrera, por lo que el hilo 1 vuelve a iniciar una transacción externa y a recomenzar la ejecución especulativa. Esta vez, tras finalizar una transacción interna, el hilo 1 puede volver a modo no especulativo de nuevo ya que el resto de hilos han llegado a la segunda llamada a `tmbarrier`. Lo mismo ocurre con el hilo 2, que ejecuta dos transacciones especulativas antes de volver a modo no especulativo.

4.4. Diseño de TMbarrier

Una vez explicado el funcionamiento general de TMbarrier, esta sección detalla su diseño y la implementación llevada a cabo tomando como base los algoritmos 5 y 6 descritos en la sección 4.2.1. Uno de los objetivos a la hora de diseñar TMbarrier es dotarlo de compatibilidad con implementaciones HTM reales. En este caso se ha optado por utilizar el soporte ofrecido en IBM POWER8 debido a

UNIVERSIDAD DE MÁLAGA

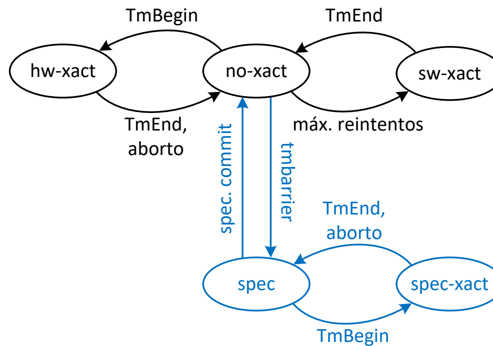


Figura 4.4: Diferentes estados para cada hilo de ejecución en TMbarrier. Los nodos representan estados y las aristas los eventos asociados a las transiciones. Los nodos y aristas de color azul representan los estados y transiciones que añade TMbarrier.

las posibilidades del *suspended mode* a la hora de establecer cierta comunicación entre transacciones activas sin producir abortos.

4.4.1. Nuevos estados en los hilos de ejecución

En un HTM *best-effort* se pueden distinguir tres estados para un hilo de ejecución determinado: (1) *no-xact*, que corresponde a un hilo que no está ejecutando una transacción, (2) *hw-xact*, que corresponde a un hilo que está ejecutando una transacción hardware, y (3) *sw-xact*, que corresponde a un hilo ejecutando código *fallback* software tras haber abortado una transacción hardware. Nuestra propuesta de TMbarrier introduce dos estados adicionales, concretamente (4) *spec*, que corresponde a un hilo ejecutando código especulativo —tras haber llegado a una llamada a `tmbarrier`— en una transacción hardware externa que no existía originalmente; y (5) *spec-xact*, que corresponde a un hilo ejecutando una transacción hardware interna tras completar una llamada a `tmbarrier`. La figura 4.4 muestra estos estados y los eventos que producen transiciones entre los mismos.

Un hilo siempre comienza su ejecución en estado *no-xact*. Cuando se inicia una transacción hardware, el hilo transiciona al estado *hw-xact*, y se mantiene en dicho estado hasta que dicha transacción aborte o finalice, regresando de

UNIVERSIDAD DE MÁLAGA

Algoritmo 9 Barrera transaccional en TMbarrier con soporte HTM *best-effort*

```
 1 function TmBarrier( )
 2     if tx.mode = spec then                       ▷ Limita especulación a una llamada a tmbarrier
 3         HtmSuspend                               ▷ Inicio de suspended mode
 4         while tx.order ≠ glOrder do              ▷ Espera al resto de hilos no especulativos
 5         end while
 6         HtmResume                                ▷ Fin de suspended mode
 7         HtmEnd                                   ▷ Finaliza especulación
 8         tx.mode ← no-xact
 9         tx.retries ← MaxRetries
10         tx.specLevel ← tx.specMax
11     end if
12     tx.order ← tx.order + 1                      ▷ Incrementa fase local
13     if atomicSub&Fetch(barrier.remain, 1) = 0 then
14         barrier.remain ← barrier.nb_threads      ▷ Si es el último hilo, restaura estado de la barrera
15         atomicFetch&Add(glOrder,1)               ▷ Si es el último hilo, incrementa fase global
16     else                                         ▷ Si no es el último hilo en llegar a tmbarrier:
17         tx.mode ← spec                           ▷ Cambia a modo especulativo
18         TmBegin                                  ▷ Inicia transacción externa
19     end if
20 end function
```

nuevo al estado *no-xact*. Si una transacción hardware aborta demasiadas veces, se producirá una transición del estado *no-xact* al estado *sw-xact* para ejecutar el código transaccional mediante un *fallback* software. Esto se produce tras adquirir el *lock* global `fallbackLock` (línea 15 del algoritmo 5). Una transacción en estado *sw-xact* es irrevocable: las operaciones transaccionales se realizan directamente en memoria compartida y, una vez finalizada, el hilo regresa al estado *no-xact*.

Cuando un hilo en estado *no-xact* llega a una instrucción `tmbarrier`, comienza a especular mediante el inicio de una transacción hardware (externa), lo que origina un cambio al estado *spec*. En este estado, el inicio de cualquier transacción (interna) presente originalmente en el código modifica el estado del hilo de *spec* a *spec-xact*. Las transacciones internas iniciadas en este modo se denominan *especulativas*, y son siempre hardware, es decir, no pueden ejecutar un *fallback* software. Cuando una transacción especulativa finaliza o aborta, se comparan la fase local del hilo `tx.order` y la fase global del sistema `glOrder` con el objetivo de finalizar la especulación si fuera posible. Si ambas fases coinciden el hilo en estado *spec* transiciona al estado *no-xact* ejecutando la fase de *commit* de la transacción externa (spec. commit en la figura 4.4).

UNIVERSIDAD DE MÁLAGA

4.4.2. La primitiva `tmbarrier`

El algoritmo 9 muestra la implementación de la barrera transaccional[27]. La primitiva utiliza un contador interno para mantener un registro del número de hilos que la han ejecutado (`barrier.remain`). Su valor inicial será el número de hilos que deben sincronizarse en la barrera, y este valor se decrementará de forma atómica cada vez que un hilo invoque la primitiva (línea 13). Cuando el último hilo ejecute `tmbarrier`, el número global de fase (`glOrder`) se incrementa atómicamente, permitiendo finalizar a los hilos especulativos que haya en ejecución en ese momento (línea 15). Si el hilo que ejecuta `tmbarrier` no era el último, cambiará su modo a *spec* (línea 17), de acuerdo a lo indicado en figura 4.4, e iniciará una transacción anidada externa. Por último, si un hilo ejecuta `tmbarrier` mientras está en modo *spec*, es bloqueado en *suspended mode* hasta que pueda finalizar (cuando su fase local `tx.order` coincida con la fase global), o hasta que detecte un conflicto, abortando en este caso. Con este método se limita la ventana de especulación entre dos llamadas consecutivas a `tmbarrier`, de acuerdo a lo expuesto en la sección 4.3.

4.4.3. Iniciando transacciones

El inicio de una transacción en TMbarrier se detalla en el algoritmo 10. El procedimiento es similar al visto en el algoritmo 5, pero incluye reglas adicionales para los nuevos estados descritos en la sección 4.4.1. En caso de aborto (líneas 12 a 31), si el hilo está en modo especulativo (estados *spec* o *spec-xact*) se impide la transición al *fallback* software en cualquier caso (*sw-xact*). En su lugar se decrementa la variable `specMax` para ajustar la ventana de especulación y se reinicia el contador de reintentos (`tx.retries`) (líneas 25 a 29). Con este sistema se preservan las garantías de corrección del código, ya que ninguna transacción especulativa hace visibles sus cambios hasta que todos los hilos de ejecución hayan llegado a la barrera. La razón del decremento de `specMax` es reducir el riesgo de aborto en posteriores ejecuciones disminuyendo el número máximo de transacciones especulativas que pueden ejecutarse. Además, antes de volver a reejecutar transacciones en modo especulativo, el sistema determina si es posible volver al estado *no-xact* (ver figura 4.4) comprobando si su fase local coincide con la fase global (líneas 21 a 24). Si es así la llamada a `TmBegin` retorna sin iniciar ninguna transacción, ya que el aborto devuelve el punto de ejecución del

[27] En los algoritmos de esta sección se ha utilizado un código de colores a efectos de mayor claridad. Las sentencias en color rojo se ejecutan en el sistema en el contexto de una transacción hardware. Las sentencias de color azul se ejecutan en *suspended mode*. Por último, las sentencias en color negro se ejecutan fuera de una transacción.

UNIVERSIDAD DE MÁLAGA

Algoritmo 10 Inicio de transacción en TMbarrier con soporte HTM *best-effort*

```
1  function TmBegin( )
2      while True do
3          while IsLocked(fallbackLock) do         ▷ Evita lemming effect
4          end while
5          tx.started ← HtmBegin                   ▷ Instrucción de inicio HTM
6          if tx.started = Success then            ▷ Transacción iniciada con éxito
7              tx.mode ← hw-xact                   ▷ Modo transaccional hardware
8              if IsLocked(fallbackLock) then      ▷ Añade fallbackLock al read-set
9                  HtmAbort                        ▷ Instrucción de aborto HTM
10             end if
11             Break                               ▷ Salto al cuerpo de la transacción (HW)
12         else                                    ▷ Transacción abortada o fallo al iniciarla
13             tx.retries ← tx.retries - 1
14             if tx.mode = hw-xact then           ▷ Gestión de transacciones no especulativas
15                 if tx.retries = 0 then          ▷ Límite de reintentos. Transición a modo software
16                     Lock(fallbackLock)          ▷ Adquisición del lock global
17                     tx.mode ← sw-xact
18                     Break                       ▷ Salto al cuerpo de la transacción (SW)
19                 end if
20             else                                ▷ Gestión de transacciones especulativas
21                 if tx.order = glOrder then      ▷ Transición a modo no especulativo
22                     tx.mode ← no-xact
23                     Return                      ▷ Salto a código posterior a tmbarrier
24                 end if
25                 if tx.retries = 0 then          ▷ Ajuste del nivel de especulación
26                     tx.specMax ← Min(tx.specMax - 1, 1)
27                     tx.retries ← MaxRetries
28                     tx.specLevel ← tx.specMax
29                 end if
30             end if
31         end if
32     end while
33 end function
```

hilo al instante inmediatamente posterior a la última llamada a `tmbarrier`. La línea 7 indica la transición del hilo a modo *hw-xact* a efectos de claridad. En una implementación real la transición a este modo debe realizarse fuera de la transacción, ya que un eventual aborto descartaría también el cambio de modo.

Aunque en el algoritmo 10 las líneas 1 a 5 aparecen como ejecutadas en modo no transaccional, podrían ejecutarse en modo *spec* si la llamada a `TmBegin` se produce en un hilo que se encuentra especulando tras una barrera. En este caso las líneas 1 a 11 se ejecutarían en modo *spec*, en el contexto de la transacción externa tras la barrera, y la línea 7 realizaría una transición a *spec-xact*.

4.4.4. Finalizando transacciones

El procedimiento de *commit* de las transacciones en TMbarrier se muestra en el algoritmo 11. Existen tres escenarios posibles a considerar: la finalización de

UNIVERSIDAD DE MÁLAGA

Algoritmo 11 Commit de transacción en TMbarrier con soporte HTM *best-effort*

```
 1 function TmEnd( )
 2     if tx.mode = sw-xact then                      ▷ Commit desde un fallback software
 3         Unlock(fallbackLock)                       ▷ Liberación del lock global
 4     else if tx.mode = hw-xact then                 ▷ Commit desde una transacción hardware
 5         HtmEnd                                     ▷ Instrucción de commit HTM
 6     else if tx.mode = spec-xact then               ▷ Commit desde una transacción especulativa
 7         HtmSuspend                                 ▷ Inicio de suspended mode
 8         if tx.order = glOrder then                 ▷ ¿Es posible salir de la especulación?
 9             HtmResume                              ▷ Fin de suspended mode
10             HtmEnd                                 ▷ Commit transacción especulativa (interna)
11             HtmEnd                                 ▷ Commit transacción especulativa (externa)
12             tx.mode ← non-xact
13         else                                       ▷ No es posible abandonar la especulación
14             HtmResume
15             tx.specLevel ← tx.specLevel - 1
16             if tx.specLevel > 0 then               ▷ Es posible seguir especulando
17                 HtmEnd                             ▷ Commit transacción especulativa (interna)
18                 tx.mode ← spec                     ▷ La transacción externa continúa activa
19             else                                   ▷ Alcanzado límite de especulación
20                 HtmSuspend                         ▷ Inicio de suspended mode
21                 while tx.order ≠ glOrder do        ▷ Espera al resto de hilos no especulativos
22                 end while
23                 HtmResume                          ▷ Fin de suspended mode
24                 HtmEnd                             ▷ Commit transacción especulativa (interna)
25                 HtmEnd                             ▷ Commit transacción especulativa (externa)
26                 tx.mode ← non-xact
27             end if
28         end if
29     end if
30     tx.retries ← MaxRetries
31     tx.specLevel ← tx.specMax
32 end function
```

una transacción hardware no especulativa (líneas 4 y 5), la finalización de una transacción ejecutada mediante el *fallback* software (líneas 2 y 3) y la finalización de una transacción especulativa (líneas 6 a 29). El sistema utiliza el estado del hilo para decidir cómo proceder. En los dos primeros casos el funcionamiento es similar al utilizado en el algoritmo 6. Si, por el contrario, el hilo está en modo *spec-xact*, la transacción entra en *suspended mode* para comprobar si `glOrder` y `tx.order` coinciden (líneas 7 a 9). Conviene recordar que los accesos a `glOrder` en el contexto de una transacción han de realizarse siempre en *suspended mode* con el objetivo de no añadir esta variable al conjunto de lectura de la misma. No hacerlo implicaría que todos los hilos especulativos abortarían al actualizarse `glOrder`, impidiendo cualquier ganancia de rendimiento.

Si ambas fases coinciden la transacción puede abandonar la especulación. Para ello realiza un doble *commit*: el primero para finalizar la transacción anidada, y el segundo para finalizar la transacción externa correspondiente, actualizando así los cambios especulativos en la memoria principal y volviendo a continuación al estado *no-spec* (líneas 10 a 12).

UNIVERSIDAD DE MÁLAGA

Metadatos	No transaccional	Transaccional	Suspended
tx.order	W, R	—	R
tx.retries	W, R	R	—
tx.specLevel	W	W, R	—
tx.specMax	W, R	—	—
tx.mode	W, R	R	—
tx.stats	W, R	—	—
glOrder	W, R	—	R
fallbackLock	W, R	R	—

Tabla 4.2: Operaciones de lectura y escritura sobre los metadatos de TMbarrier. Las variables `glOrder` y `fallbackLock` son globales. El resto son metadatos locales a cada transacción. Cada columna representa las operaciones que se realizan sobre los metadatos en modo transaccional, *suspended* o no transaccional.

Si las fases no coinciden, el hilo realiza un único *commit* para finalizar la transacción anidada (transicionando a estado *spec*) y se decrementa un contador local al hilo (`specLevel`) hasta llegar a cero. Si la especulación puede continuar (líneas 16 a 18) no es necesario realizar ninguna acción adicional; la siguiente transacción presente en el código iniciará otra transacción interna. Si `specLevel` ha llegado a cero se ha alcanzado el límite de la especulación; el hilo será bloqueado en *suspended mode* hasta que su fase local coincida con la fase global o hasta que se detecte un conflicto, abortando en este caso (líneas 20 a 26).

4.4.5. Gestionando las limitaciones del HTM de POWER8

Algunas características específicas de la implementación *best-effort* del HTM de POWER8 deben ser tenidas en cuenta a la hora de implementar el diseño de TMbarrier de forma eficiente.

Granularidad de la detección de conflictos

El primer problema lo encontramos en el hecho de que este HTM es implícito y detecta los conflictos con granularidad de bloque caché (128 bytes). Esto requiere considerar qué tipo de accesos se realizan a cada uno de los metadatos del

UNIVERSIDAD DE MÁLAGA

sistema y en qué modo (transaccional[28], no transaccional o *suspended*), y alinear los metadatos en memoria adecuadamente para evitar conflictos debidos a *false-sharing*. La tabla 4.2 resume las operaciones (lecturas o escrituras) realizadas en cada estado sobre cada uno de los metadatos del sistema.

Las variables globales correspondientes a `glOrder` y a `fallbackLock` deben almacenarse en dos bloques caché diferentes. La razón es que `fallbackLock` se incluye en el conjunto de lectura de todas las transacciones hardware para poder detectar posibles transacciones software en ejecución (línea 8 del algoritmo 10). Sin embargo `glOrder` no debe registrarse en ningún conjunto transaccional, ya que, en caso contrario, su eventual incremento provocaría el aborto de cualquier transacción activa. Aunque algunas transacciones en estado *hw-xact* podrían finalizar a pesar de esto, todo el trabajo especulativo por parte de hilos ejecutando código después de `tmbarrier` sería abortado debido a que las transacciones en estado *spec-xact* no pueden finalizar hasta que `glOrder` se haya actualizado.

Por otra parte la variable `tx.order`, accedida en modo *suspended* durante las comprobaciones del orden parcial, puede interaccionar con la escritura en modo transaccional de `tx.specLevel` si ambas están mapeadas en el mismo bloque de caché, ya que POWER8 considera las operaciones en modo *suspended* como no transaccionales. Sin embargo se ha comprobado experimentalmente que esta interacción no provoca conflictos adicionales.

Los metadatos locales a cada hilo de ejecución deberían combinarse en el menor número de bloques caché para evitar polucionar la caché innecesariamente. Asimismo debe asegurarse que los metadatos locales de hilos distintos estén mapeados en bloques caché distintos para evitar conflictos debidos a *false sharing*.

En conjunto, nuestra implementación utiliza un único bloque caché por hilo (128 bytes) para almacenar los metadatos transaccionales, y dos bloques caché adicionales para almacenar metadatos globales.

Interacciones en suspended mode

El *suspended mode* del HTM debe ser tratado con especial cuidado, ya que puede dar lugar a efectos poco intuitivos [57]. Por ejemplo, el incremento de un contador local dentro de una transacción hardware utilizando el *suspended mode* puede provocar un aborto de la transacción si cualquier otro metadato transaccional (o variable de los conjuntos de lectura y escritura) estaba mapeado en el mismo bloque de caché. Este fenómeno se produce por un acceso no transaccional (el incremento

[28]Consideramos modo transaccional los estados *hw-xact*, *spec* y *spec-xact*, al ejecutarse todos en el contexto de una transacción hardware.

UNIVERSIDAD DE MÁLAGA

en *suspended mode*) a un bloque que había sido marcado como transaccional por el HTM. Una manera de mitigar este problema es prohibir los accesos de escritura en *suspended mode*. Un enfoque alternativo, menos restrictivo, consistiría en separar a nivel de bloque caché los datos que serán actualizados en *suspended mode* de los datos que serán accedidos en modo transaccional. En nuestra propuesta el *suspended mode* se utiliza únicamente para comprobar si la fase local del hilo transaccional coincide con la fase global del sistema, por lo que no es necesario realizar escrituras en este modo. Aunque nuestra implementación registra ciertas estadísticas (`tx.stats`), sólo lo hace cuando el hilo está ejecutando código no transaccional (por ejemplo, las causas de cada aborto después de producirse, o qué camino ha seguido cada transacción después de que ésta haya finalizado). Si se quisieran obtener estadísticas adicionales dentro de la transacción, estos accesos deben realizarse en *suspended mode* —para que los contadores no se reviertan en caso de aborto de la transacción— y las variables actualizadas deben mapearse en un bloque diferente al del resto de metadatos transaccionales de cada hilo.

Limitaciones del anidamiento de transacciones

El anidamiento de transacciones en el HTM de POWER8 está soportado mediante *flattening* de las transacciones internas contenidas en la transacción externa correspondiente (ver sección 2.4.3). Este soporte tiene dos desventajas si lo comparamos con otros modelos más avanzados [77]. La primera es que un conflicto en una de las transacciones internas provoca el aborto de todas las transacciones contenidas en la transacción externa. La segunda es que los accesos transaccionales que realice cualquier transacción interna son acumulados en un mismo conjunto de datos transaccional, lo que incrementa la probabilidad de que se produzca un conflicto y de producir abortos por capacidad.

Para mitigar el impacto de estas limitaciones, el número máximo de transacciones anidadas internas que se permiten en TMbarrier se controla de forma dinámica mediante la variable `specMax`, reduciéndose si se producen demasiados abortos (línea 26 de algoritmo 10). Además, si una transacción especulativa sufre un aborto por exceder el tamaño de los conjuntos transaccionales soportado por el hardware, `specMax` se decrementa inmediatamente para reducir el número máximo transacciones a especular y, por tanto, el número de accesos transaccionales de las transacciones anidadas.

UNIVERSIDAD DE MÁLAGA

Política de resolución de conflictos

El diseño de TMbarrier debería evitar que un conflicto que involucre transacciones especulativas cause una pérdida de trabajo no especulativo. Esto requiere un gestor de conflictos modificado que implemente en su algoritmo una política *abort-speculative*, de modo que cualquier conflicto entre transacciones en modo especulativo y transacciones en modo no especulativo se resuelva siempre en favor de las últimas. Lamentablemente el gestor de conflictos del HTM de POWER8 está implementado directamente en el hardware y no es modificable por el usuario. Por tanto una transacción anterior a `tmbarrier` puede abortar si detecta un conflicto con una transacción especulativa en esta implementación. Esto no es una limitación del algoritmo, sino de la falta de soporte en el HTM utilizado.

4.5. Evaluación

El rendimiento de la implementación de TMbarrier propuesta en la sección anterior ha sido evaluado utilizando tres *benchmarks* representativos. El primero de ellos es un *microbenchmark* sintético extraído de [11] para comprobar el rendimiento potencial de TMbarrier en ausencia de conflictos reales. Este código consiste en una serie de etapas cada una de las cuales se sincroniza con una barrera. En cada una de las etapas sólo la mitad de los hilos realiza trabajo real, generando así un desbalanceo entre los hilos de ejecución. El segundo *benchmark* corresponde a la ecuación de recurrencia general extraída de los *Livermore Loops* [100] que se ha mostrado previamente en el algoritmo 8. Su interés reside en que emplea una fracción notable del tiempo de ejecución en la sincronización de los distintos hilos mediante barreras y presenta una tasa relativamente baja de dependencias de datos entre sus distintas etapas [78]. El tercer *benchmark* corresponde a un algoritmo de coloreado de grafos extraído de [29]. Como en el código anterior, representa un escenario de interés al tener una tasa de dependencias relativamente baja entre el código de sus distintas etapas. La contención en este caso depende del tamaño del grafo y de su conectividad, y nos permite comprobar cómo afecta la misma a nuestra propuesta.

Los experimentos se han llevado a cabo en el sistema descrito en la tabla 4.3. Para reducir la interferencia del sistema en los resultados se ha utilizado la siguiente metodología: cada hilo de ejecución se ha mapeado a un núcleo físico mediante el uso del comando `taskset` con el objetivo de evitar la compartición de recursos transaccionales entre los hilos SMT, lo que reduciría el tamaño efectivo de los conjuntos transaccionales y aumentaría la tasa de abortos. Los resultados

UNIVERSIDAD DE MÁLAGA

Parámetro	Descripción
Procesador	IBM POWER8 3.5 GHz
Núcleos	10 físicos con SMT 8
Memoria	512GB
Sistema	Ubuntu Server 16.04.1 LTS, Kernel 4.4.0-47 ppc64le
Compilador	GNU GCC v5.4

Tabla 4.3: Plataforma de evaluación de TMbarrier

obtenidos corresponden a la mejor de al menos diez ejecuciones para cada perfil mostrado con el objetivo de reducir la interferencia del sistema de acuerdo con lo indicado en [46]. Los experimentos que exceden un tiempo de ejecución razonable (diez veces el tiempo de la ejecución secuencial) se consideran fallidos.

Se han considerado cuatro perfiles diferentes de cara a la evaluación:

- *OMPBar*: corresponde al HTM de POWER8 con el esquema básico representado en los algoritmos 5 y 6. En este perfil la sincronización mediante barreras se ha llevado a cabo utilizando las barreras estándar de OpenMP.
- *NoBar*: es un esquema similar al anterior, pero toda la sincronización mediante barreras ha sido eliminada. Este perfil puede producir resultados incorrectos, pero se utiliza como límite superior del rendimiento potencial obtenible[29].
- *TMOrd*: corresponde a una implementación con restricciones totales de orden similar a la mostrada en la figura 4.2 realizada a partir del perfil *P8*, donde cada transacción debe esperar a que todas las transacciones con un orden anterior hayan finalizado antes de ejecutar su fase de *commit*. En este perfil no se utilizan barreras ya que la sincronización está garantizada implícitamente por las restricciones de orden.
- *TMB*: implementa nuestra propuesta de TMbarrier representada en los algoritmos 10 y 11 utilizando el soporte HTM de POWER8. En este perfil la sincronización mediante barreras utiliza la primitiva `tmbarrier`, implementada según el algoritmo 9. Todos los hilos en este perfil pueden especular un máximo de ocho transacciones anidadas por hilo de ejecución después de una instrucción `tmbarrier`.

[29]Realmente este perfil no tiene por qué obtener una cota superior de rendimiento, ya que permite nuevas interacciones entre transacciones potencialmente conflictivas que podrían degradar el rendimiento. En los experimentos realizados, sin embargo, la tasa de abortos es similar a la obtenida con el perfil *OMPBar*, por lo que no se da esta situación.

UNIVERSIDAD DE MÁLAGA

```
1  #pragma omp parallel
2  {
3    int tid = omp_get_thread_num();
4
5    for(int step = 0; step < nb_steps; step++){
6      TmBegin();
7      volatile int load;
8      for(load = 0; load != nb_load * ((tid + step) % 2); load++){
9        Compute(); // Computación sin dependencias entre llamadas
10     }
11     TmEnd();
12     #pragma omp barrier
13   }
14 }
```

Figura 4.5: Kernel del microbenchmark sintético.

4.5.1. Microbenchmark sintético

El propósito de este *benchmark* es ilustrar el potencial para la especulación de TMbarrier en un escenario favorable, esto es, sin conflictos reales entre el código anterior y posterior a la barrera. Los conflictos entre transacciones en este escenario serán aquellos derivados de la implementación de TMbarrier y de las propias limitaciones del soporte HTM.

El kernel del *benchmark* se muestra en la figura 4.5. La ejecución está organizada en distintas etapas (línea 6), en cada una de las cuales un conjunto de hilos realiza cierto trabajo (líneas 8 a 10) y se lleva a cabo una sincronización al final de la misma utilizando una barrera (línea 12). La carga de trabajo no está distribuida de manera uniforme entre los hilos: en cada etapa sólo la mitad de los hilos de ejecución realiza trabajo útil. Este trabajo no produce dependencias de datos y aumenta proporcionalmente al número de hilos de ejecución. Por este motivo se ha utilizado la *eficiencia*[30] como métrica de rendimiento en lugar del *speedup*, que nos da una idea de la penalización que introduce cada método de sincronización considerado.

La figura 4.6 (arriba) muestra la eficiencia obtenida con la configuración por defecto utilizada en [11] (10000 operaciones en cada transacción), así como en otros escenarios menos favorables, donde se realiza menos trabajo en cada etapa. El límite superior lo establece el perfil *NoBar* y se mantiene en el rango del 90–100% según la carga computacional utilizada. Los resultados obtenidos

[30] $\text{eficiencia} = \frac{\text{throughput(N hilos)}}{N \times \text{throughput(secuencial)}}$, para N hilos, donde el $\text{throughput} = \frac{\#\text{steps}}{\text{tiempo}}$.

UNIVERSIDAD DE MÁLAGA

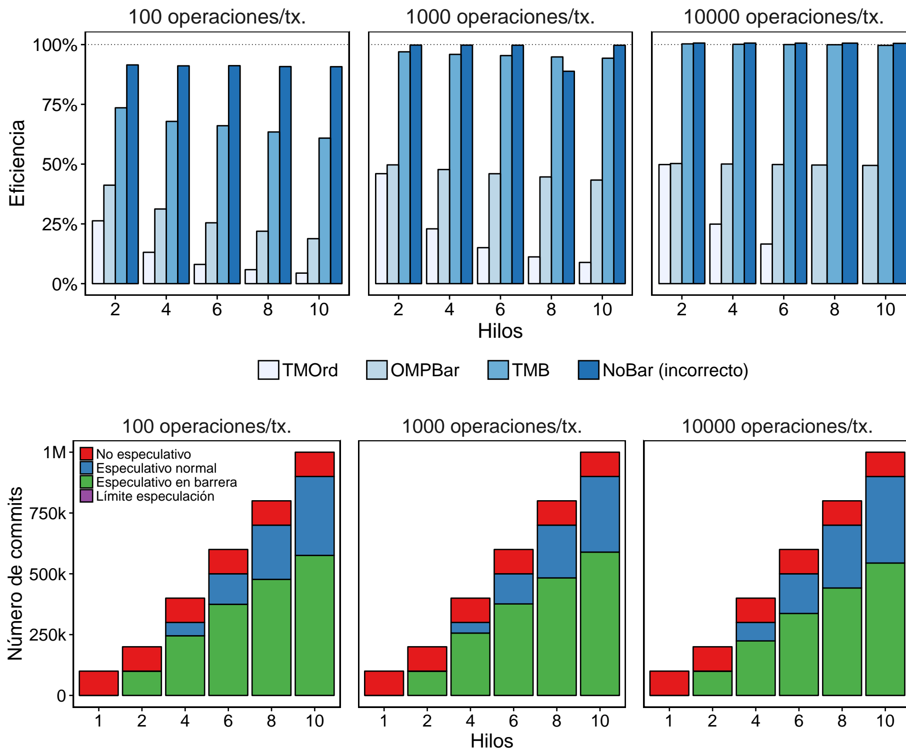


Figura 4.6: Microbenchark: eficiencia sobre la ejecución secuencial (arriba) y tipos de transacciones ejecutados (abajo).

muestran cómo el perfil *TMB* es capaz de mantener una eficiencia elevada en todos los escenarios, mientras que las dos alternativas de sincronización sufren una penalización notable. Hay que recordar que en este *benchmark* los hilos que no realizan trabajo real en cada etapa se benefician de la especulación que ofrece TMbarrier, ya que pueden continuar ejecutando el trabajo de la siguiente etapa, explotando así el paralelismo en situaciones de desbalanceo.

Con el objetivo de analizar el grado de aprovechamiento de la especulación en TMbarrier, la figura 4.6 (abajo) clasifica las transacciones que han finalizado correctamente en el perfil *TMB* según el camino que hayan seguido durante su ejecución.

UNIVERSIDAD DE MÁLAGA

Se han distinguido cuatro caminos diferentes:

(1) *No especulativo* hace referencia a transacciones hardware (estado *hw-xact*) que han finalizado antes de una llamada a TMbarrier (línea 5 del algoritmo 11).

(2) *Especulativo normal* engloba a transacciones especulativas donde los hilos han ejecutado al menos una transacción anidada y han realizado un doble *commit* para abandonar la especulación y confirmar los cambios a memoria (líneas 10 y 11 del algoritmo 11).

(3) *Especulativo en barrera* contiene las transacciones que han vuelto a llegar a una instrucción `tmbarrier` mientras estaban en estado especulativo (línea 7 del algoritmo 9).

(4) *Límite especulación* se refiere a las transacciones especulativas que han llegado al máximo nivel de especulación del hilo correspondiente antes de poder finalizar (líneas 24 y 25 del algoritmo 11).

El camino (1) contendrá un número mínimo de transacciones que vendrá determinado por el último hilo de cada etapa que llegue a TMbarrier, que no tiene la oportunidad de especular. Este límite varía según el algoritmo en cuestión y un subconjunto de dichas transacciones formarán parte del camino crítico de la ejecución. El hecho de que el número de transacciones de este grupo no aumente respecto al escenario con un sólo hilo de ejecución (donde todas las transacciones pertenecen a este grupo) es un indicador del correcto funcionamiento de la especulación y de la ausencia de dependencias entre las distintas etapas del *benchmark*. Al aumentar el número de hilos, aumentan proporcionalmente las transacciones ejecutadas y las oportunidades de especulación. En este caso, el valor constante de transacciones del camino (1) indica que dichas transacciones son las últimas en llegar a la barrera en cada etapa (camino crítico). El resto de hilos, que inician la especulación, consiguen especular sin producir conflictos, por lo que sus transacciones forman parte del resto de caminos considerados.

Por otra parte existe un gran porcentaje de transacciones que encuentran una barrera mientras el hilo está en modo especulativo (3). Esto es esperable ya que cada hilo ejecuta una única transacción en cada etapa, y se sincroniza con el resto de hilos inmediatamente después. En nuestra implementación los hilos en estado *spec.* se bloquean ante una nueva instrucción `tmbarrier`. Esta es también la causa de que no existan transacciones del grupo (4)[31].

[31]Para que existiesen transacciones en este grupo, cada hilo debería ejecutar al menos ocho transacciones especulativas consecutivas, alcanzando el límite de especulación del perfil *TMB*.

UNIVERSIDAD DE MÁLAGA

Por último, las transacciones especulativas que salen de la especulación tras comprobar que el resto de hilos han llegado a la barrera (2) apenas aparecen en la ejecución con dos hilos, pero su número crece linealmente al aumentar el número de hilos de ejecución. Esto se debe a la existencia de más transacciones activas en el sistema, lo que aumenta la posibilidad de cumplir las condiciones de *commit* cuando hayan finalizado su ejecución.

Dada la ausencia de conflictos reales entre transacciones, los cuatro perfiles obtienen un ratio de abortos muy bajo, independientemente del número de hilos utilizados en la ejecución y de la carga computacional utilizada. Aunque esto es esperable dada la naturaleza optimista de este *benchmark*, sirve para confirmar que el diseño e implementación de TMbarrier no introduce conflictos adicionales.

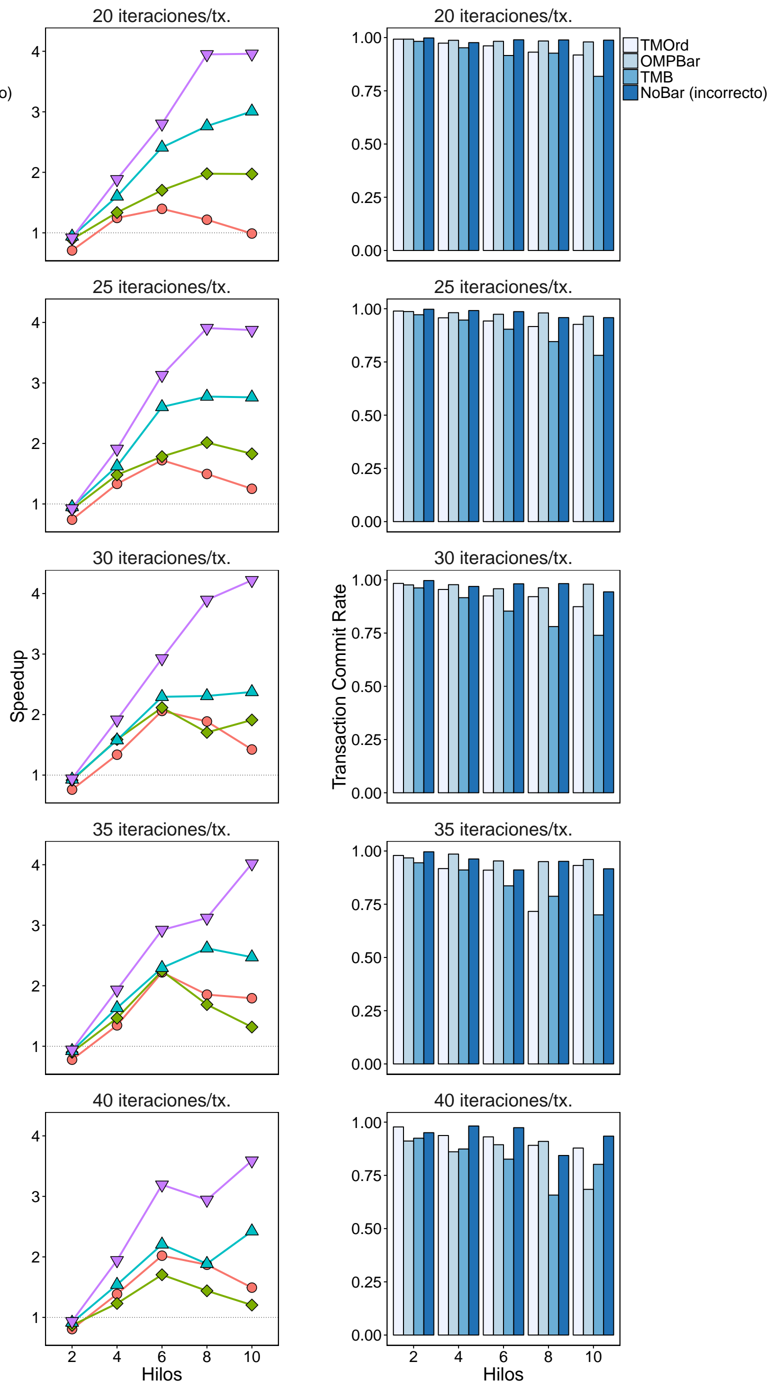
4.5.2. Livermore Loop 6: Recurrencia Lineal

Este código corresponde con el sexto *benchmark* de la suite Livermore Loops (LFK) [35] analizado en la sección 4.3, sobre el que se ha realizado la paralelización mostrada en el algoritmo 8 [100]. Los *speedups* obtenidos se muestran en la figura 4.7 (izquierda). Las diferentes gráficas corresponden a ejecuciones con distintos tamaños de transacción[32]. Dichos tamaños se han seleccionado en un rango de interés que minimiza el número de transacciones a ejecutar manteniendo a la vez una tasa de abortos reducida. Los experimentos corresponden al rango donde se maximiza el rendimiento del perfil *OMPBar*.

Este *benchmark* presenta una cantidad de paralelismo limitada, incluso al eliminar la sincronización que introducen las barreras. Nuestros resultados están en la línea de los reportados por [100], que sugieren un *speedup* máximo de 2x utilizando barreras software optimizadas en un procesador de 16 núcleos. En nuestros experimentos hemos medido un *speedup* máximo de 3x eliminando la sincronización de las barreras (perfil *NoBar*). La penalización que sufre el perfil *TMOrd* limita su *speedup* a 1.4x en el mejor escenario. Además puede observarse cómo esta penalización aumenta rápidamente con el número de hilos. El perfil *OMPBar*, que utiliza barreras estándar, consigue un *speedup* de alrededor de 1.5x con 8 hilos en la mejor configuración. Estos resultados no son mucho mejores que los obtenidos con *TMOrd*, lo que se debe en parte a la penalización que introduce la sincronización y a su dependencia del tamaño de las transacciones. El perfil *TMB* aumenta este *speedup* a 2x, pero además consigue mantener mejores

[32]Los tamaños se miden según el número de iteraciones del bucle que ejecute una sola transacción. Tamaños mayores requerirán un menor número de transacciones pero serán más susceptibles a conflictos.

UNIVERSIDAD DE MÁLAGA



UNIVERSIDAD DE MÁLAGA

Figura 4.7: Recurrence: *speedup* respecto a la ejecución secuencial (izquierda) y TCR (derecha) para diferentes tamaños de transacción.

resultados que el resto de perfiles en configuraciones no óptimas del tamaño de la transacción. Por una parte, estos resultados confirman la capacidad de TMbarrier para reducir el tiempo de sincronización en barreras mediante la especulación. Por otra, podemos observar cómo el ajuste dinámico del nivel de especulación permite ampliar el rango óptimo de tamaños de transacción. El sistema ejecutará más transacciones especulativas en configuraciones que usen transacciones de menor tamaño, y será menos sensible a los desbalanceos de carga que puedan producirse con transacciones relativamente grandes gracias a poder especular al llegar a una barrera.

El límite superior en nuestros experimentos lo establece el perfil *NoBar* con un *speedup* de aproximadamente 3x con 10 hilos, lo que nos da una idea de la limitada cantidad de paralelismo explotable de este código incluso eliminando la sincronización. Las razones de este rendimiento las podemos encontrar en la naturaleza *memory-bound* del código, que necesita de cuatro accesos a memoria y dos operaciones aritméticas en cada iteración del bucle interno; y en la reducción del paralelismo explotable con las sucesivas iteraciones del bucle externo (ver algoritmo 8).

En referencia la tasa de abortos, el *Transaction Commit Rate* (TCR) [3] se muestra en la figura 4.7 (derecha). Esta métrica muestra el ratio entre el número de transacciones finalizadas con éxito respecto al número total de transacciones iniciadas. Su valor máximo es 1, lo que indica que todas las transacciones que se iniciaron en el sistema terminaron sin producir ningún aborto.

El perfil *OMPBar* mantiene la tasa de abortos prácticamente a cero, ya que las transacciones del bucle interno no contienen dependencias de datos reales (por lo que los conflictos sólo pueden producirse por falsos positivos por alias en las referencias de memoria). Las barreras, al bloquear los hilos, impiden los conflictos debidos a dependencias verdaderas.

El perfil *NoBar* obtiene también un número de abortos muy bajo. Aunque en este perfil pueden existir conflictos debido a la ejecución concurrente de iteraciones diferentes del bucle externo, su probabilidad es baja y las transacciones son relativamente cortas, lo que reduce el tiempo de exposición a conflictos. Cabe destacar que el hecho de eliminar las barreras no afectará igual a todos los algoritmos, ya que da pie a nuevas interacciones entre los accesos a memoria de los hilos de ejecución.

El perfil *TMB* obtiene un TCR menor al resto de perfiles analizados, y su valor disminuye al aumentar el número de hilos y el tamaño de las transacciones. Estos abortos son el resultado de conflictos derivados de permitir dependencias reales entre las transacciones ejecutadas antes y después de la barrera [100]. De hecho, permitir la ejecución especulativa al llegar a una barrera produce nuevos patrones

UNIVERSIDAD DE MÁLAGA

de acceso a los datos que, al igual que sucede al eliminar las sincronización en el perfil *NoBar*, pueden producir dependencias que no existían originalmente. Dos factores adicionales agravan el problema en TMBarrier: por una parte, este TM encapsula varias transacciones especulativas en una transacción anidada mayor. Debido a la implementación de anidamiento mediante *flattening* de POWER8, el resultado es una transacción de mayor tamaño, que será más sensible a posibles conflictos debido tanto a su duración como al número de accesos transaccionales. Por otra parte existen situaciones en las que una transacción activa ejecutándose tras una barrera debe esperar para salir de la especulación, ya sea por llegar en modo especulativo a una nueva barrera transaccional o por superar el límite de especulación del sistema mientras haya hilos que aún no han llegado a la barrera correspondiente. En ambos casos aumenta la ventana de tiempo donde la transacción es susceptible a conflictos. Es importante recordar que estas esperas son necesarias para garantizar una ejecución correcta del algoritmo.

En los escenarios con menor TCR se ha observado que un porcentaje notable de los abortos son causados por conflictos con código no transaccional. Este efecto es el resultado de la adquisición del *lock* global para ejecutar el *fallback* software cuando una transacción hardware aborta demasiadas veces. Cuando un hilo adquiere dicho *lock*, produce el aborto de todas las transacciones activas debido a la lectura del mismo durante la fase de inicio (línea 8 del algoritmo 5).

En los escenarios con transacciones de mayor tamaño (mayor número de iteraciones por transacción), pueden aparecer algunos abortos por superar la capacidad hardware para los conjuntos de datos transaccionales. Este problema se acentúa por la implementación del anidamiento de transacciones mediante *flattening* que realiza POWER8, que combina los accesos de todas las transacciones anidadas en el mismo conjunto de datos.

En este *benchmark* el ratio de transacciones especulativas que finalizan con éxito respecto del total de transacciones es de aproximadamente un 2%, aunque varía con el número de hilos de ejecución, el tamaño de las transacciones y la frecuencia de las llamadas a `tmbarrier`. Aunque este ratio no es elevado, el porcentaje se refiere a transacciones anidadas externas que han podido finalizar con éxito, cada una de las cuales puede contener hasta ocho transacciones internas presentes en el código original[33].

UNIVERSIDAD DE MÁLAGA

Algoritmo 12 Extracto del kernel del algoritmo CFL

```
 1  function CFL(steps)
 2      for each t ∈ steps do
 3          for all node ∈ Graph do
 4              TMBEGIN
 5              collision ← CHKEQCOLOR(node.neighbors) ▷ Comprueba colisiones con nodos vecinos
 6              if collision then                     ▷ Modifica probabilidades según proceda
 7                  for each color ∈ colors do
 8                      if color = node.color then
 9                          prob[node][color] ← (1 − β)∗prob[node][color]
10                      else
11                          prob[node][color] ← (1 − β)∗prob[node][color] + (β/colors−1)
12                      end if
13                  end for
14              else
15                  for each color ∈ colors do
16                      if color = node.color then
17                          prob[node][color] ← 1
18                      else
19                          prob[node][color] ← 0
20                      end if
21                  end for
22              end if
23                                                    ▷ Actualiza color del nodo
24              newColor ← RANDCOLOR(prob[node], colors)
25              if newColor ≠ node.color then
26                  node.color ← newColor             ▷ Escritura transaccional
27              end if
28              TMEND
29          end for
30          BARRIER                                   ▷ Sincroniza etapa
31          end ← CHECKSOLUTION                       ▷ Comprueba si se ha llegado a la solución
32          if end or t = tmax then
33              return
34          end if
35          BARRIER
36      end for
37  end function
```

4.5.3. Algoritmo CFL

El algoritmo *Communication-Free Learning* (CFL) es un método estocástico propuesto inicialmente para la asignación de canales en redes inalámbricas [67] que se puede generalizar a un algoritmo para coloreado de grafos [29]. Su característica principal es que las decisiones sobre los nodos individuales no requieren tener la información completa de la estructura del grafo.

CFL trabaja con un vector por nodo que actúa como función de masa de probabilidad y cuyo tamaño coincide con el número de colores disponibles. El algoritmo se ejecuta durante un determinado número de pasos o hasta alcanzar una solución válida. En cada paso, todos los nodos actualizan su vector asociado

[33] Este límite se ha establecido empíricamente para maximizar las oportunidades de especulación y minimizar los abortos por conflictos y capacidad.

UNIVERSIDAD DE MÁLAGA

según el color observado de cada uno de sus vecinos. A continuación se asigna aleatoriamente un nuevo color al nodo a partir de los colores disponibles y su vector de probabilidad actualizado. Este método converge probabilísticamente en una solución válida [29].

El algoritmo 12 muestra el *kernel* de CFL. En cada paso hasta hallar una solución o llegar a `tmax`, cada nodo compara su color con el de sus nodos vecinos (línea 5). Si encuentra alguna coincidencia, actualiza el vector de probabilidades de forma que la probabilidad de conservar el color actual sea reducida y las probabilidades de elegir el resto de colores aumenten proporcionalmente[34] (líneas 6 a 13). El parámetro $\beta \in (0, 1)$ determina la probabilidad de que un nodo conserve un óptimo local una vez hallado. Un valor próximo a cero hace que el nodo tenga una mayor resistencia a cambiar su color actual a pesar de detectar una colisión, mientras que un valor mayor aumenta el efecto de las colisiones, haciendo que sus efectos se propaguen más rápidamente. Si no se detecta una colisión, el color actual del nodo se considera un óptimo local, por lo que se asigna una probabilidad de 1 al color correspondiente del vector de probabilidades del nodo, y un 0 al resto de colores, garantizado que el color actual se mantiene en este paso. (líneas 14 a 21). La última acción es asignar un nuevo color al nodo a partir del vector de probabilidades que se acaba de actualizar (línea 24 a 27). La escritura de la línea 26 es la causa de dependencias reales en este algoritmo. Una barrera garantiza que todos los nodos estén actualizados antes de verificar si se ha alcanzado una solución global (línea 31) o seguir ejecutando el algoritmo.

Este algoritmo es interesante por dos razones. Por una parte, se ajusta bien a un escenario donde TM puede resultar útil a la hora de implementar una paralelización eficiente, especialmente en grafos de tamaño considerable, implementados mediante matrices dispersas. Si protegemos el cómputo de cada nodo en una transacción (líneas 4 y 28), los únicos conflictos reales entre transacciones se producirán entre los nodos que cambien su color actual y sus vecinos. Por otra parte, el uso de barreras transaccionales puede resultar eficiente si hay pocas actualizaciones de los colores de los nodos entre distintas etapas.

La figura 4.8 muestra el rendimiento obtenido en varias configuraciones del *kernel* anterior variando el número de nodos del grafo y su conectividad (número de conexiones entre nodos), de la que depende directamente la probabilidad de conflictos entre transacciones. Los nodos han sido inicializados a un color de forma aleatoria siguiendo una distribución uniforme.

UNIVERSIDAD DE MÁLAGA

[34] Dado que el vector actúa como una función de masa de probabilidad, la suma de las probabilidades de todos los colores debe sumar 1.

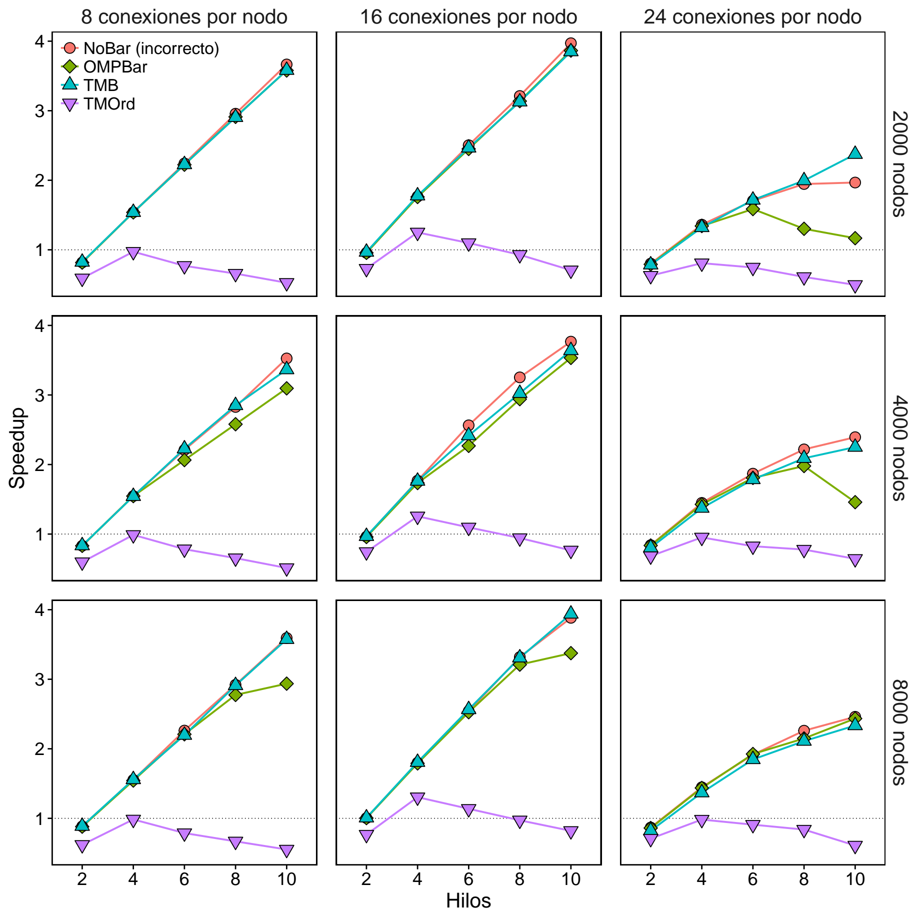


Figura 4.8: Speedup de algoritmo CFL en grafos de distinto tamaño y conectividad.

UNIVERSIDAD DE MÁLAGA

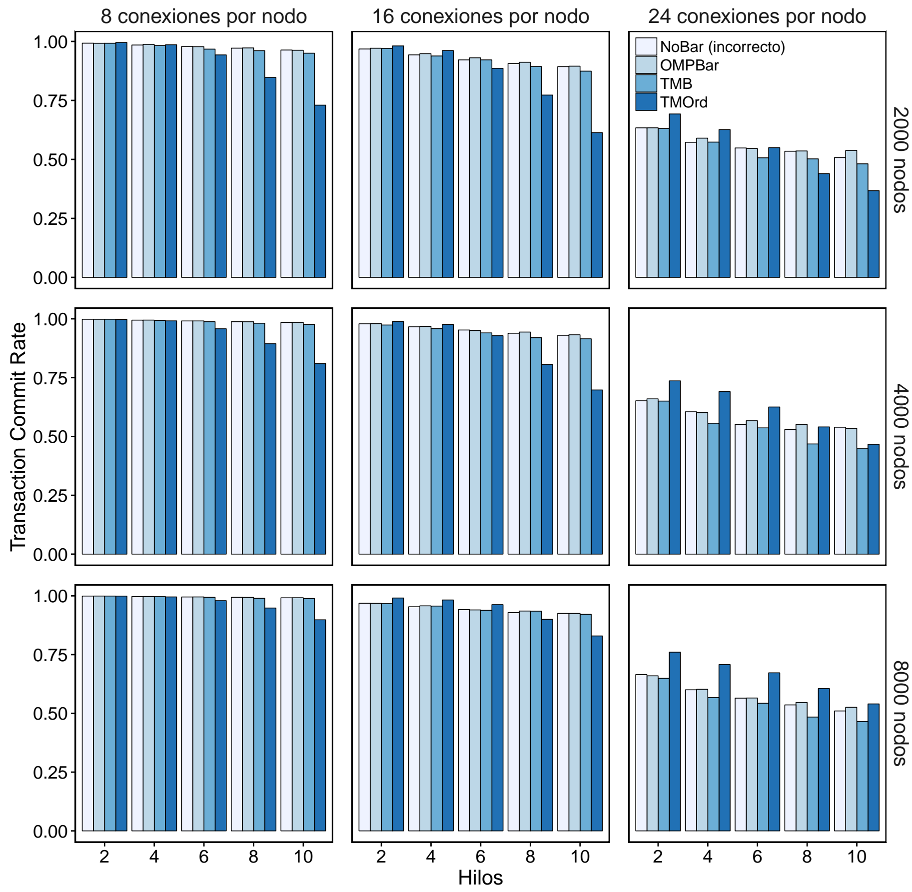


Figura 4.9: TCR de algoritmo CFL en grafos de distinto tamaño y conectividad.

UNIVERSIDAD DE MÁLAGA

En estos experimentos, el rendimiento potencial que puede extraer TMbarrier viene limitado por el *overhead* que introduce el uso de barreras en cada configuración. Con tamaños de grafo reducidos (primera fila de la figura 4.8) esta diferencia apenas es apreciable, por lo que no se observa mejora de rendimiento al utilizar TMbarrier. Por el contrario, con los tamaños intermedio y mayor sí existe una diferencia de rendimiento entre el perfil *OMPBar*, que utiliza barreras tradicionales, y el perfil *NoBar*, que prescinde del uso de barreras para obtener una cota superior del rendimiento obtenible. En estos casos, el rendimiento de TMbarrier se sitúa entre ambos perfiles. En las pruebas realizadas, el número de *steps* se ha mantenido fijo, al no afectar al rendimiento relativo, ya que el patrón de accesos a memoria no depende de este parámetro.

La conectividad del grafo afecta a la probabilidad de conflictos de las transacciones, debido a que cada transacción tiene que acceder a los colores de sus vecinos. En los experimentos con 8 y 16 conexiones por nodo el número de conflictos se mantiene bajo y los resultados de *speedup* son similares en los perfiles analizados. En los experimentos con 24 conexiones por nodo, el número de conflictos aumenta, lo que repercute en el rendimiento obtenible, especialmente en el grafo con menor número de nodos. En estas situaciones TMbarrier mantiene su rendimiento próximo al perfil *NoBar*, mientras que el uso de TM con barreras tradicionales no escala con más de seis hilos de ejecución. Una posible causa de este comportamiento es el desbalanceo de los hilos producido por los abortos de las transacciones. Mientras que TMbarrier puede enmascarar este desbalanceo especulando con los hilos que lleguen antes a la barrera, no ocurre lo mismo en el perfil *OMPBar*, que debe esperar a que los hilos con más abortos terminen su ejecución en cada etapa.

El TCR correspondiente de los experimentos anteriores se muestra en la figura 4.9. El comportamiento de los distintos perfiles es similar al del *benchmark* anterior; con los perfiles *OMPBar* y *NoBar* obteniendo ratios más elevados y un mayor número de abortos al utilizar TMbarrier debido a las interacciones adicionales producidas por la especulación. El perfil *TMOrd*, sin embargo, obtiene un número más reducido de abortos en algunos escenarios, especialmente en aquellos con mayor probabilidad de conflictos. Este comportamiento se debe a las propias restricciones de orden, que limitan la interacción entre transacciones dado que deben esperar en *suspended mode* su turno de *commit*. Aunque se produzcan conflictos, una transacción en *suspended mode* no aborta hasta no restaurar el modo transaccional, lo que reduce el número de abortos. Este mayor TCR, sin embargo, no lleva asociado un mayor rendimiento, como muestra la figura 4.8. Aunque el valor del TCR suele correlarse con el rendimiento, los resultados obtenidos demuestran que no siempre es así, y que el sistema TM utilizado, las

UNIVERSIDAD DE MÁLAGA

interacciones que éste permita entre transacciones y las restricciones de orden parciales o estrictas son otros aspectos a tener en cuenta.

4.6. Trabajos relacionados

Aprovechar el soporte que brinda TM para la explotación optimista de paralelismo usando técnicas de *Thread-Level Speculation* (TLS) o *Speculative Multithreading* (SpMT) es un campo ya investigado en la literatura [87, 80, 109]. El soporte TM para la detección de conflictos y la retención de las escrituras hasta la fase de *commit* son dos características que se adaptan bien a los diseños TLS. En [80] se presenta un análisis del uso de HTM para propósitos TLS usando la implementación de Intel TSX. Los autores analizan el rendimiento obtenible en una selección de *benchmarks* de la suite SPEC CPU2006 y concluyen que, aunque es posible obtener una ganancia de rendimiento en algunos casos, la mayoría de los programas no se benefician de este soporte debido a pérdidas de rendimiento causadas por abortos debidos a dependencias. Se sugiere la implementación de características avanzadas en futuros diseños HTM que incluyen *data-forwarding*, cachés multiversión y detección de conflictos a nivel de palabra. Estas propuestas coinciden con [82], que estudia también la posibilidad de usar técnicas especulativas en la suite SPEC CPU2006 y encuentra algunas oportunidades interesantes para la paralelización usando TLS.

La aplicación de técnicas TLS a códigos ya paralelos puede involucrar la especulación tras directivas de sincronización como barreras o *locks*. En [73] los autores proponen un diseño hardware que permite este tipo de especulación utilizando un hilo *safe* que garantiza el progreso del algoritmo y la monitorización de los accesos para preservar la corrección del sistema.

OpenTM [4] es una propuesta para extender OpenMP con una API transaccional, y considera un conjunto adicional de directivas que permiten expresar garantías de sincronización no bloqueantes. OpenTM soporta restricciones de orden total entre transacciones y en bucles transaccionales, pero no permite utilizar las directivas de sincronización ya existentes en OpenMP con transacciones, ya que puede interferir con las garantías de aislamiento fuerte de algunos sistemas TM. La propuesta requiere el soporte para *transacciones virtualizadas*, un modelo en el que las transacciones no están limitadas por el tiempo de ejecución, el número de accesos transaccionales ni otras restricciones presentes en las implementaciones HTM *best-effort* actuales.

UNIVERSIDAD DE MÁLAGA

En relación a las barreras [78] estudia el efecto de su uso para sincronización y los efectos de cara al rendimiento en un conjunto de programas paralelos. El objetivo es encontrar oportunidades para la aplicación de técnicas TLS en códigos que empleen un porcentaje considerable de su tiempo de ejecución en la sincronización entre los hilos y que contengan pocas dependencias de datos. Los autores proponen una solución basada en la *Advanced Load Address Table* (ALAT), una estructura hardware presente en los procesadores Itanium, para detectar posibles dependencias habilitar la especulación en barreras.

Otra aproximación para especular en directivas de sincronización combinando OpenMP con HTM la encontramos en [11]. La propuesta, pensada para utilizarse con código no transaccional, también utiliza una transacción después de una barrera para almacenar las escrituras especulativas. Sin embargo su diseño no está pensado para aprovechar el *suspended mode* ni para aplicarse en códigos originalmente transaccionales. Por esta razón es necesario especificar manualmente un punto de sincronización tras la barrera. La transacción especulativa comprobará si puede actualizar la memoria al llegar a dicho punto, pero debe abortar si no es así. Los autores presentan una tasa de éxito en la especulación de un 40% en el mejor caso utilizando el mismo *microbenchmark* analizado en la sección 4.5.1. La misma configuración obtiene una tasa de éxito cercana al 100% en TMbarrier gracias al aprovechamiento del *suspended mode*. Este trabajo no considera algunas limitaciones presentes en las implementaciones actuales de HTM, incluyendo el tamaño de los conjuntos transaccionales o la presencia de instrucciones ilegales en las transacciones especulativas. Además la solución propuesta no explota el anidamiento de transacciones para aumentar la ventana de especulación, ni soporta múltiples puntos de sincronización, al no existir la posibilidad de comprobar las condiciones de *commit* sin finalizar la transacción.

4.7. Conclusiones

En este capítulo hemos presentado TMbarrier, un sistema que permite soportar restricciones parciales de orden en códigos transaccionales. Nuestra propuesta implementa una barrera transaccional diseñada para reducir el tiempo de espera debido a la sincronización mediante el uso de especulación. La semántica de dicha barrera es similar a la de una barrera tradicional y su uso es transparente al programador, que únicamente debe reemplazar las barreras estándar por la primitiva `tmbarrier`, permitiendo así la especulación cuando sea posible. El diseño de TMbarrier tiene en cuenta las limitaciones de las implementaciones HTM reales, presentes en procesadores de consumo.

UNIVERSIDAD DE MÁLAGA

Nuestra propuesta tiene como ámbito de aplicación códigos transaccionales que hacen uso de barreras para sincronizar los hilos de ejecución. El motivo principal es que las propias transacciones presentes en el código proporcionan de forma natural puntos de sincronización adecuados que permiten adaptar el grado de especulación automáticamente sin intervención del usuario. Este modelo puede adaptarse fácilmente para su uso en códigos no transaccionales especificando manualmente o con ayuda del compilador posibles puntos de sincronización.

El rendimiento de TMbarrier ha sido analizado experimentalmente mediante el *microbenchmark* descrito en [11], el *kernel* de recurrencia lineal extraído de los *Livermore Loops* [100] y el algoritmo CFL para coloreado de grafos [29]. Estos códigos emplean un porcentaje notable de su tiempo de ejecución en sincronizaciones mediante barreras y contienen un número bajo de dependencias entre los accesos realizados antes y después de las barreras [78]. Los resultados obtenidos muestran que el TMbarrier mejora el rendimiento obtenido respecto al uso de barreras tradicionales en estas aplicaciones y sugiere que es posible reducir el tiempo empleado en la sincronización aprovechando el soporte HTM de los procesadores actuales en otras aplicaciones con características similares. Además, la implementación propuesta puede beneficiarse de futuros avances en los HTM *best-effort*, especialmente los relativos a mayores conjuntos transaccionales, detección de conflictos con granularidad de palabra y mejor soporte del anidamiento de transacciones. Por otra parte, la posibilidad de modificar la política de detección y resolución de conflictos del HTM permitiría filtrar algunos conflictos entre transacciones especulativas y no especulativas, así como priorizar las segundas sobre las primeras en caso de producirse un conflicto real.

Las limitaciones de TMbarrier son principalmente sus escenarios de aplicación. Dado que lo que se optimiza es el tiempo que los hilos pasan bloqueados en directivas de sincronización, el porcentaje de tiempo en barreras del algoritmo original determina el margen potencial de mejora. Por otra parte, al utilizar especulación, un algoritmo con un ratio de dependencias alto entre el código anterior y posterior a la sincronización en barreras apenas podrá especular con éxito. A pesar de estas restricciones, siguen existiendo escenarios donde el uso frecuente de barreras para sincronizar los distintos hilos constituye un porcentaje importante del tiempo de ejecución y donde las dependencias reales entre hilos son relativamente poco frecuentes [78]. En estas situaciones TMbarrier puede resultar útil a la hora de extraer rendimiento.

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Capítulo 5

Conclusiones y trabajos futuros

El paralelismo se ha convertido en un elemento clave para aumentar el rendimiento en los procesadores modernos. Tras su popularización en sistemas de alto rendimiento, hoy día se pueden encontrar chips multiprocesador en un rango de sistemas que se extiende desde servidores a *wearables* pasando por *smartphones* y otros dispositivos portátiles.

A diferencia de otros avances en la arquitectura de computadores, la explotación de la concurrencia en estos procesadores requiere de un esfuerzo adicional por parte del programador: no es fácil programar en paralelo, y menos aún hacerlo de forma eficiente. A la dificultad adicional que conlleva el uso de varios hilos concurrentes de ejecución y su interacción con los modelos relajados de consistencia de memoria presentes en estos procesadores hay que añadir la incompatibilidad de patrones ya establecidos en la programación secuencial, la necesidad de utilizar directivas de bajo nivel, la interacción de las mismas a la hora de manejar la abstracción y la dificultad en la depuración de posibles errores. Los modelos de programación deben proporcionar un equilibrio entre la eficiencia de las implementaciones y la productividad.

La memoria transaccional es un paradigma que surge como respuesta a la necesidad de herramientas que permitan elevar el nivel de abstracción a la hora de explotar el paralelismo en aplicaciones. Propuesta en la década de los 90, es en los últimos años cuando ha ganado gran popularidad, con numerosas propuestas en la literatura tanto para su implementación con soporte hardware como

UNIVERSIDAD DE MÁLAGA

mediante el uso de librerías software. Este interés creciente se ha materializado en la implementación de soporte transaccional hardware básico por parte de Intel e IBM en sus procesadores más recientes. Aunque estos primeros diseños presentan algunas limitaciones importantes, es esperable que este soporte mejore en microarquitecturas posteriores y que se extienda a otros fabricantes.

5.1. Conclusiones

En esta tesis hemos abordado la posibilidad de aprovechar la memoria transaccional como base para el desarrollo de modelos y técnicas de paralelización de aplicaciones que requieran de un menor esfuerzo por parte del programador. Esta motivación ha condicionado algunos aspectos del desarrollo de la investigación, donde hemos optado por el diseño de sistemas con garantías suficientes para poder ser aplicados en la paralelización de aplicaciones en las que el programador no tenga un conocimiento exhaustivo acerca de sus posibles dependencias o interacciones.

En el capítulo 3 hemos estudiado cómo el soporte explícito de operaciones de reducción en sistemas transaccionales permite mejorar el rendimiento a la hora de paralelizar aplicaciones que presenten este tipo de patrones, y cómo su combinación con restricciones de orden permite la aplicación de estos sistemas a códigos donde el programador no tiene garantías de que las reducciones puedan interaccionar con otros accesos de lectura o escritura.

En este sentido cabe destacar que el soporte explícito de operaciones de reducción resulta atractivo desde el punto de vista del rendimiento. Tras el análisis realizado en diversas aplicaciones de ámbito científico hemos comprobado que las operaciones de reducción no sólo aparecen con relativa frecuencia en estas aplicaciones, si no que a menudo son las responsables de buena parte de los conflictos de datos de la aplicación. El tratamiento explícito de estas operaciones en un sistema TM permite prevenir conflictos entre transacciones, lo cual tiene un impacto positivo en el rendimiento de la aplicación.

Hemos propuesto un modelo que combina soporte explícito de operaciones de reducción y restricciones de orden entre transacciones con el doble propósito de mejorar el rendimiento de aplicaciones que presenten accesos de reducción y de ofrecer un soporte que permita paralelizar códigos iterativos complejos sin necesidad de disponer de información detallada sobre sus dependencias. Esta propuesta se ha materializado en ReduxSTM, un sistema TM software que combina soporte específico para operadores conmutativos y asociativos con un

UNIVERSIDAD DE MÁLAGA

mecanismo de garantía de orden secuencial, necesario para garantizar resultados correctos en situaciones en las que los patrones de reducción pueden coexistir con otros accesos a memoria.

La evaluación llevada a cabo muestra que el soporte propuesto puede mejorar el rendimiento de sistemas TM en varios escenarios. ReduxSTM se ha comparado con otros sistemas TM software del estado del arte y, en su caso, con otras técnicas clásicas para la paralelización de reducciones en un amplio conjunto de *benchmarks* que cubren varios escenarios de interés incluyendo reducciones irregulares, reducciones parciales y soporte para técnicas de especulación. Los resultados obtenidos son alentadores de cara a incluir este tipo de soporte en futuros sistemas transaccionales.

Las limitaciones de ReduxSTM están relacionadas principalmente con la penalización de rendimiento que introduce la instrumentación de los accesos a memoria para dar soporte al sistema transaccional, algo común en los sistemas TM software; y la penalización derivada del soporte de las restricciones estrictas de orden, que requiere serializar la fase de *commit* de las transacciones. En la práctica, el rendimiento obtenible con ReduxSTM depende del tamaño de las transacciones que requiera el problema, de la carga computacional del mismo y de la contención de los accesos transaccionales. El rendimiento en transacciones muy cortas se verá afectado por la serialización de la fase de *commit*. Por su parte las aplicaciones intensivas en memoria pueden sufrir una penalización del rendimiento explotable debido a la instrumentación de los accesos.

En el capítulo 4 hemos investigado la posibilidad de relajar las restricciones de orden que se suelen utilizar en técnicas de especulación a un modelo de orden parcial, menos restrictivo, que ofrezca más oportunidades para la explotación del paralelismo cuando no sea necesario un mantener un orden estricto entre las transacciones individuales. El trabajo realizado parte de la observación de que las restricciones de orden totales pueden resultar excesivas en algunos escenarios, donde sólo son necesarias entre determinadas fases de la aplicación.

En este contexto hemos propuesto TMbarrier, un modelo de orden parcial con soporte transaccional que permite a un sistema TM trabajar con restricciones de orden dinámicas que afecten a grupos completos de transacciones, en lugar de a transacciones individuales, permitiendo una mayor explotación del paralelismo disponible. Nuestra propuesta introduce una primitiva adicional en la interfaz transaccional que permite establecer barreras especulativas. Estos objetos, con una semántica similar a las barreras de sincronización tradicionales, establecen restricciones de orden parciales en tiempo de ejecución y permiten a los hilos seguir ejecutando código de forma especulativa sin esperar a la sincronización. El

UNIVERSIDAD DE MÁLAGA

objetivo de TMbarrier es aumentar el rendimiento en aplicaciones que empleen un porcentaje notable de su tiempo de ejecución sincronizando los distintos hilos de ejecución mediante barreras.

La implementación propuesta utiliza el soporte transaccional hardware del procesador IBM POWER8 y hace uso del *suspended mode*, una característica que permite ejecutar código no transaccional en el contexto de una transacción hardware, permitiendo establecer cierta comunicación entre transacciones activas sin producir abortos.

La evaluación llevada a cabo muestra que es posible realizar una implementación eficiente del modelo de orden parcial a pesar de las limitaciones del soporte transaccional actual. TMbarrier se ha comparado con el uso de barreras tradicionales en una serie de escenarios de interés, que presentan un uso frecuente de barreras para sincronización combinado con un ratio de dependencias limitado entre código ejecutado antes y después de las mismas. Los resultados muestran cómo es posible utilizar este tipo de técnicas especulativas para aumentar la explotación del paralelismo en estos escenarios.

Las publicaciones derivadas de esta tesis son las siguientes:

M. Pedrero, E. Gutiérrez, S. Romero, and O. Plata. Improving Transactional Memory Performance for Irregular Applications. Procedia Computer Science, Vol. 51 pp. 2714-2718, 2015. In *International Conference on Computational Science (ICCS'15)*, Reykjavík, June 2015.

M. Pedrero, E. Gutiérrez, S. Romero, and O. Plata. Mejorando el Rendimiento en Aplicaciones Irregulares con Memoria Transaccional. In *XXVI Jornadas de Paralelismo*, Córdoba, 2015.

M. Pedrero, E. Gutiérrez, S. Romero, and O. Plata. A Comparative Analysis of STM Approaches to Reduction Operations in Irregular Applications. In *Journal of Computational Science*, Vol. 17, pp. 630–638, 2016.

M. Pedrero, E. Gutiérrez, S. Romero, and O. Plata. Análisis Comparativo del Uso de STMs en Códigos de Reducción Irregulares In *XXVII Jornadas de Paralelismo*, Salamanca, 2016.

UNIVERSIDAD DE MÁLAGA

M. Pedrero, E. Gutiérrez, S. Romero, and O. Plata. ReduxSTM: Optimizing STM Designs for Irregular Applications. In *Journal of Parallel and Distributed Computing*, Vol. 107, pp. 114–133, 2017.

M. Pedrero, E. Gutiérrez, and O. Plata. TMbarrier: Speculative Barriers Using Hardware Transactional Memory. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Vol. 107, pp. 114–133. Cambridge, 2018.

5.2. Trabajos futuros

En el transcurso de esta investigación hemos podido comprobar la efectividad de la memoria transaccional como herramienta para la paralelización. Para concluir esta tesis presentamos a continuación una serie de líneas de investigación derivadas de este trabajo que creemos vale la pena explorar.

La adaptación de las ideas de ReduxSTM a un sistema HTM reduciría en gran medida la penalización asociada a la instrumentación software. Esta adaptación requiere resolver algunos aspectos clave que incluyen el uso de nuevas políticas de resolución de conflictos, el soporte de operaciones de reducción, ya sea mediante una implementación hardware o con el apoyo de rutinas software; y la adaptación de los sistemas de detección de conflictos a estas operaciones.

Las restricciones de orden de ReduxSTM permiten su aplicación a un gran número de situaciones, incluyendo aquellas donde el programador o el compilador no dispone de información suficiente sobre las posibles interacciones de las sentencias de reducción con el resto del código. Existen escenarios, sin embargo, donde las sentencias de reducción no interactúan con otros accesos, y pueden beneficiarse de soporte TM para su paralelización. Un modelo transaccional con soporte de reducciones sin restricciones de orden puede conseguir un mayor rendimiento en estos escenarios y resulta además más sencillo de adaptar a un sistema hardware al no requerir cambios sustanciales en las políticas de resolución de conflictos.

Por otra parte, el uso de TM para especulación es un campo investigado en la literatura por su importancia tanto a la hora de paralelizar código *legacy* como de extraer rendimiento de programas ya existentes en procesadores compatibles.

UNIVERSIDAD DE MÁLAGA

Durante nuestra investigación con HTM, uno de los principales problemas para la obtención de rendimiento ha sido la sensibilidad del sistema a falsos conflictos derivados de las interacciones de variables privadas y metadatos transaccionales con el código paralelo. Una línea de investigación interesante en este sentido es el estudio de posibles mejoras en el diseño hardware o en la compilación para reducir estos abortos en la medida de lo posible. El uso de información del ámbito de las variables o de *hints* para especificar variables de uso exclusivo del hilo puede ser interesante a la hora de filtrar conflictos.

En relación a nuestra propuesta de TMbarrier, las limitaciones de los HTM actuales pueden producir el aborto de transacciones no especulativas si se producen conflictos durante la especulación. Sería interesante una extensión del modelo propuesto en TMbarrier para evitar este fenómeno, junto con las estructuras hardware necesarias para llevarlo a cabo. En concreto, una gestión de conflictos que tenga en cuenta la coexistencia de transacciones especulativas permitiría evitar buena parte de estos abortos, filtrando dependencias *write-after-write* y *write-after-read* entre transacciones tradicionales y especulativas. De modo similar, una política de resolución de conflictos que priorice las transacciones tradicionales en detrimento de las especulativas evitará descartar trabajo útil, permitiendo usar nuevas estrategias para la especulación.

Por último, y dada la popularización de chips multiprocesador en dispositivos de bajo consumo, otra posible línea de investigación es el estudio de la eficiencia energética de los sistemas propuestos y del efecto del uso de sistemas TM hardware y software en términos de consumo de energía. Este estudio se puede ampliar incluyendo técnicas de paralelización tradicionales y técnicas de especulación con soporte transaccional.

UNIVERSIDAD DE MÁLAGA

Apéndice A

API transaccional y simulación de trazas

El presente apéndice describe la interfaz utilizada para implementar los sistemas transaccionales propuestos en los capítulos 3 y 4. Por su parte, la sección A.2 describe la metodología utilizada en la evaluación de los *benchmarks* de las secciones 3.4.3 y 3.4.5.

A.1. Una interfaz transaccional unificada

A la hora de analizar las distintas propuestas hardware y software de esta tesis conviene disponer de un modelo común. En este trabajo se ha utilizado un subconjunto de la API (*Application User Interface*) propuesta por los autores de la suite STAMP [14]. Esta interfaz define una serie de primitivas que se pueden asociar durante la compilación a diferentes funciones de una librería transaccional. La interfaz distingue dos tipos de funciones: por una parte contiene métodos para iniciar y finalizar transacciones, así como para realizar operaciones transaccionales de lectura y escritura en TM explícitos. Por otra parte incluye un conjunto de funciones asociadas a determinados eventos que se producen durante la ejecución del código para realizar tareas de inicio y finalización del sistema transaccional. La tabla A.1 detalla el subconjunto de la API utilizado.

Resultado de las propuestas de este trabajo se han añadido algunas primitivas adicionales a la API anterior para soportar nuevas funcionalidades en el sistema

UNIVERSIDAD DE MÁLAGA

Función	Argumentos	Descripción
TM_STARTUP	Ninguno	Debe llamarse al inicio de la aplicación. Inicializa el sistema transaccional.
TM_SHUTDOWN	Ninguno	Debe llamarse antes de finalizar la aplicación. Libera los recursos del sistema transaccional.
TM_THREAD_ENTER	Ninguno	Debe llamarse al inicio de cada nuevo hilo de ejecución. Inicializa los metadatos transaccionales locales al hilo.
TM_THREAD_EXIT	Ninguno	Debe llamarse antes de finalizar cada hilo de ejecución. Libera los metadatos transaccionales locales al hilo.
TM_BEGIN	readonly	Inicia una nueva transacción. Si se produce un aborto, la ejecución debe volver a este punto. El parámetro readonly es opcional y garantiza que la transacción no contiene operaciones de escritura transaccional, permitiendo al sistema habilitar posibles optimizaciones.
TM_END	Ninguno	Finaliza una transacción en curso.
TM_RESTART	Ninguno	Aborta incondicionalmente una transacción en curso.
TM_SHARED_READ	address	Operación de lectura transaccional sobre la dirección address. Por defecto se considera un dato unsigned int de 8 bytes. Los sufijos _F o _P permiten especificar si el dato es de tipo float de 4 bytes o un puntero.
TM_SHARED_WRITE	address, value	Operación de escritura transaccional del dato value sobre la dirección address. Por defecto se considera un dato unsigned int de 8 bytes. Los sufijos _F o _P permiten especificar si el dato es un float de 4 bytes o un puntero.

Tabla A.1: Primitivas básicas de la API transaccional de STAMP.

transaccional. Estas primitivas se describen brevemente en la tabla A.2 y permiten el uso de transacciones ordenadas y de operaciones de reducción transaccionales (capítulo 3), así como dar soporte a barreras especulativas (capítulo 4).

La figura A.1 ilustra el uso de esta API en un código paralelo sencillo. Para mayor claridad las funciones auxiliares del sistema transaccional se muestran en color azul y las funciones transaccionales básicas se muestran en color rojo. La línea 8 inicializa el sistema transaccional en el hilo de ejecución principal, mientras que la línea 11 inicializa los metadatos locales a cada hilo. A partir de esta línea el programador puede ejecutar transacciones encapsulando el cuerpo de las mismas con las instrucciones correspondientes (líneas 17 y 21). En sistemas transaccionales implícitos todos los accesos a memoria dentro de una transacción son monitorizados por el sistema. Los sistemas explícitos, sin embargo, sólo monitorizarán los accesos instrumentados con las funciones transaccionales correspondientes (línea 19), y no garantizarán el aislamiento ni la atomicidad del resto de accesos. Aunque se detallará en los capítulos correspondientes, en general los sistemas TM software son explícitos con el objetivo de minimizar la

UNIVERSIDAD DE MÁLAGA

Función	Argumentos	Descripción
`TM_BEGIN_ORD`	`order`, `readonly`	Inicia una nueva transacción ordenada. Si se produce un aborto, la ejecución debe volver a este punto. El parámetro `readonly` es opcional y garantiza que la transacción no contiene operaciones de escritura transaccional, permitiendo al sistema habilitar algunas optimizaciones. El parámetro `order` es un entero que indica el número de orden de la transacción, garantizando que ésta no haga visibles sus cambios al resto del sistema hasta que todas las transacciones con orden menor hayan finalizado.
`TM_SHARED_RDX`	`address`, `value`, `operation`	Operación de reducción transaccional del valor `value`, con la operación asociativa y conmutativa `operation`, sobre la dirección `address`. Por defecto se considera un dato unsigned int de 8 bytes. Los sufijos _F o _P permiten especificar si el dato es de tipo float de 4 bytes o un puntero.
`TM_BARRIER_INIT`	`nthreads`	Inicializa una barrera transaccional para sincronizar `nthreads` hilos de ejecución.
`TM_BARRIER`	Ninguno	Sincroniza un conjunto de hilos de ejecución.

Tabla A.2: Primitivas añadidas a la API transaccional de STAMP.

instrumentación adicional requerida, mientras que los sistemas TM hardware son implícitos, ya que la detección de conflictos se lleva a cabo modificando el protocolo de coherencia de caché. Una vez finalizada la ejecución transaccional es necesario llamar a las funciones de cierre correspondientes (líneas 24 y 26) para liberar los descriptores y finalizar la librería. Los sistemas transaccionales existentes o propuestos deben soportar esta API, que será enlazada en el momento de compilación al sistema a analizar. De este modo no es necesario adaptar un código transaccional a un sistema TM concreto. Los sistemas que no necesiten utilizar alguna de las funciones auxiliares pueden optar por no instrumentar la primitiva correspondiente. Por su parte, los sistemas implícitos deben implementar las macros para operaciones transaccionales básicas como accesos locales.

A.2. Simulación de trazas

Los experimentos de la sección 3.4.3 y 3.4.5 se han realizado siguiendo un esquema traceador/ejecutor que se describe a continuación.

En un primer paso, el código secuencial original es instrumentado para generar una traza de sus accesos a memoria. Para ello se marcan las operaciones de lectura, escritura y reducción mediante una serie de macros añadidas al código original. Además de los accesos, también se indican los puntos de inicio y fin de bloques de instrucciones, como las iteraciones del bucle.

UNIVERSIDAD DE MÁLAGA

```
1  int main(int argc, char** argv){
2      int i, c;
3      type_t* I = readImg();
4      type_t* H = newHist();
5      params_t p = parseParams(argc, argv);
6      omp_set_num_threads(p.nthreads);
7
8      TM_STARTUP();
9      #pragma omp parallel private(i, c)
10     {
11         TM_THREAD_ENTER();
12         int tid = omp_get_thread_num();
13         int start = tid * p.limit;
14         int stop = start + p.limit;
15
16         for(i=start; i<stop; ){
17             TM_BEGIN();
18                 for(c=i; (c<(i+p.chunk)) & (c<stop); c++){
19                     TM_SHARED_RDX(B[A[c]].data, 1, '+');
20                 }
21             TM_END();
22             i += p.chunk;
23         }
24         TM_THREAD_EXIT();
25     }
26     TM_SHUTDOWN();
27 }
```

Figura A.1: Ejemplo del uso de la API de STAMP en un código en C con OpenMP. Cálculo de un histograma.

Una vez instrumentado el código, la ejecución del programa original genera una traza de los accesos a memoria realizados durante la ejecución similar al mostrado en la figura A.2. Normalmente la instrumentación afecta a un determinado bucle de interés, como muestra la tabla 3.8, por lo que se añaden un par de macros adicionales que permiten habilitar y deshabilitar la generación de la traza. Para recrear una carga computacional equivalente es necesario analizar manualmente las operaciones realizadas en el cuerpo del bucle en el contexto del cuerpo de las transacciones.

Un programa ejecutor recibe como entrada la traza generada en el paso anterior, la librería transaccional a utilizar, la carga computacional a recrear por transacción, el número de iteraciones a ejecutar en cada transacción y el número de hilos de ejecución. El ejecutor distribuye las iteraciones descritas por la traza entre los hilos y genera un *pool* de páginas de memoria virtual con un número

UNIVERSIDAD DE MÁLAGA

```
# acceso, dirección, valor, (op)
LOOP START
ITERATION START
WR,       0x2333,    1
RDX,      0x1234,    42,    sum
RDX,      0x2333,    21,    sum
RD,       0x2333,    22
WR,       0x1234,    22
ITERATION END
ITERATION START
...
RD,       0x123c,    26
ITERATION END
...
LOOP END
```

Figura A.2: Ejemplo de traza de ejecución.

de páginas equivalente al utilizado por el conjunto de los accesos de la traza.

Las direcciones de memoria de los accesos originales son mapeadas al conjunto de páginas del *pool*. El objetivo de este mapeo es mantener los requerimientos reales de memoria del programa independientemente del rango de direcciones lógicas asignadas durante la generación de la traza. Cada una de las páginas del espacio de memoria original se mapea en una página exclusiva del *pool*, conservando el *offset* relativo entre las direcciones de los accesos de la traza. De este modo, accesos a la misma página en el código original se mantienen en una misma página del *pool*.

A continuación cada hilo recorre su parte asignada de la traza, agrupando las iteraciones en transacciones según los parámetros de entrada. Cada acceso de la traza es transformado en una operación de memoria (`TmRead`, `TmWrite`, `TmRdx`) sobre las páginas del *pool*. Asimismo se recrea la carga computacional original de cada iteración.

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

Bibliografía

[1] Sarita V. Adve and Hans-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90, aug 2010. (Citado en página 89)

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, mar 1995. (Citado en página 26)

[3] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Euro-Par*, 2008. (Citado en página 172)

[4] W. Baek, C.C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *16th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, pages 376–387, 2007. (Citado en página 179)

[5] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986. (Citado en página 144)

[6] J. Barreto, P. Ferreira Dragojevic, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *ACM/IFIP/USENIX 13th Int'l. Middleware Conf. (Middleware'12)*, pages 187–207, 2012. (Citado en página 143)

[7] Barna L Bihari. Transactional memory for unstructured mesh simulations. *Journal of Scientific Computing*, 54(2-3):311–332, 2013. (Citado en página 138)

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (Citado en páginas 40, 42 y 96)

UNIVERSIDAD
DE MÁLAGA

[9] C. Blundell, E.C. Lewis, and M.M.K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006. (Citado en páginas 20, 21 y 22)

[10] 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. *IEEE Micro*, 28(1):32–41, 2008. (Citado en página 17)

[11] Lars Bonnichsen and Artur Podobas. Using transactional memory to avoid blocking in OpenMP synchronization directives. In *IWOMP*, 2015. (Citado en páginas 165, 167, 180 y 181)

[12] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell : A hybrid transactional memory for Haswell's restricted transactional memory. *PACT*, pages 187–200, 2014. (Citado en páginas 44 y 54)

[13] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop. ACM*, 2014. (Citado en página 54)

[14] 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)*, pages 35–46, 2008. (Citado en páginas 107, 131, 132 y 189)

[15] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 2008. (Citado en página 33)

[16] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News*, 34(2):227–238, 2006. (Citado en página 96)

[17] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. 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 página 6)

[18] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 extension

UNIVERSIDAD DE MÁLAGA

for lock-free data structures and transactional memory. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 39–50, 2010. (Citado en página 146)

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2016. (Citado en página 45)

[20] M. Dai Wang, M. Burcea, L. Li, et al. Exploring the performance and programmability design space of hardware transactional memory. In *ACM Workshop on Transactional Computing (TRANSACT'14)*, Salt Lake City, UT, USA, 2014. (Citado en página 124)

[21] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ASPLOS*, 2011. (Citado en página 53)

[22] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par, pages 2–13. Springer-Verlag, 2010. (Citado en página 34)

[23] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'10)*, pages 67–78, 2010. (Citado en páginas 37, 98, 104 y 138)

[24] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, et al. Hybrid transactional memory. *ACM SIGPLAN Notices*, 41(11):336, oct 2006. (Citado en página 51)

[25] Robert H. Dennard, Fritz H. Gaensslen, Hwa Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, oct 1974. (Citado en página 1)

[26] Matthew DeVuyst, Dean M Tullsen, and Seon Wook Kim. Runtime parallelization of legacy code on a transactional memory system. In *6th Int'l. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, pages 127–136, 2011. (Citado en páginas 31, 139 y 143)

[27] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, et al. Applications of the adaptive transactional memory test platform. In *TRANSACT*, 2008. (Citado en página 149)

UNIVERSIDAD DE MÁLAGA

[28] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208. Springer Berlin Heidelberg, 2006. (Citado en páginas 35 y 138)

[29] K. R. Duffy, N. O'Connell, and A. Sapozhnikov. Complexity analysis of a decentralised graph colouring algorithm. *Information Processing Letters*, 107(2):60–63, 2008. (Citado en páginas 165, 174, 175 y 181)

[30] Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. Consistency for transactional memory computing. *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 3–31, 2015. (Citado en página 22)

[31] P. Feautrier. Array expansion. In *2nd Int'l Conf. on Supercomputing (ICS'88)*, pages 429–441, 1988. (Citado en páginas 8 y 64)

[32] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Página principal de TinySTM. URL: http://tmware.org/tinystm.html. Último acceso: 7 de noviembre de 2018. (Citado en página 105)

[33] 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 página 138)

[34] 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 36, 138 y 144)

[35] John T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing*, 7(2):163–185, 1988. (Citado en páginas 150 y 170)

[36] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *Lecture Notes in Computer Science*, 167 LNCS(3):125–132, 1984. (Citado en página 59)

[37] C. Ferri, R.I. Bahar, A. Marongiu, L. Benini, M. Herlihy, B. Lipton, and T. Moreshet. SoC-TM: Integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *9th Int'l. Conf. on Hardware/Software Codesign and System Synthesis (CODES'11)*, pages 39–48, 2011. (Citado en página 144)

UNIVERSIDAD DE MÁLAGA

[38] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, jul 1981. (Citado en página 49)

[39] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. (Citado en página 3)

[40] Agner Fog. *Instructions for ASMlib: a multiplatform library of highly optimized functions for C and C++*. Technical University of Denmark, 2013. version 2.34. (Citado en página 104)

[41] Carlos H. Gonzalez and Basilio B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475–489, 2013. (Citado en páginas 8 y 64)

[42] Miguel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Dealing with reduction operations using transactional memory. In *25th Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'13)*, pages 128–135, 2013. (Citado en páginas 67 y 140)

[43] Miguel Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Exploring irregular reduction support in transactional memory. In *13th Int'l. Conf. on Algorithms and Architectures for Parallel Processing*, volume 8285 LNCS, Part 1, pages 257–266. Springer, Cham, dec 2013. (Citado en página 8)

[44] Miguel Angel Gonzalez-Mesa, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Effective transactional memory execution management for improved concurrency. *ACM Transactions on Architecture and Code Optimization*, 11(3):24, 2014. (Citado en páginas 59 y 140)

[45] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. *CGO*, pages 101–110, 2010. (Citado en páginas 39, 98 y 138)

[46] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. In *Lecture Notes in Computer Science*, volume 1697, pages 11–18, 1999. (Citado en páginas 106 y 166)

[47] Open Systems Group. Systems performance evaluation cooperation. SPEC benchmarks. http://www.spec.org. Retrieved: 2015-07-19. (Citado en página 107)

UNIVERSIDAD DE MÁLAGA

[48] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 175–184, 2008. (Citado en página 27)

[49] 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 - ICS '00*, pages 78–87, 2000. (Citado en páginas 59 y 65)

[50] 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. *IEEE Computer*, 29(12):84–89, 1996. (Citado en páginas 8 y 64)

[51] P. Hammarlund and A. J. Martinez et. al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014. (Citado en página 44)

[52] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, et al. Transactional memory coherence and consistency. In *ISCA*, 2004. (Citado en páginas 43 y 144)

[53] H. Han and C.W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. on Parallel and Distributed Systems*, 17(7):606–618, 2006. (Citado en página 65)

[54] Liang Han, Wei Liu, and James M. Tuck. Speculative parallelization of partial reduction variables. In *8th Annual IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10)*, pages 141–150, 2010. (Citado en páginas 59, 62, 68, 129 y 139)

[55] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Ed.* Morgan & Claypool Publishers, USA, 2nd edition, 2010. (Citado en páginas 7, 13, 30, 31 y 88)

[56] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA)*, pages 289–300, 1993. (Citado en páginas 5, 7, 15 y 41)

[57] IBM. *Power ISA Version 2.07, Book II*. IBM, 2013. (Citado en páginas 147 y 163)

[58] Damien Imbs and Michel Raynal. Virtual World Consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, 2012. (Citado en página 28)

UNIVERSIDAD DE MÁLAGA

[59] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System Z. *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36, 2012. (Citado en páginas 6 y 47)

[60] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. In *33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'12)*, pages 359–370, 2012. (Citado en páginas 59 y 139)

[61] Rajesh Kumar Karmani, Nicholas Chen, Bor-Yiing Su, Amin Shali, and Ralph Johnson. Barrier synchronization pattern. In *ParaPLOP*, 2009. (Citado en página 144)

[62] C. Kevin Shum, Fadi Busaba, and Christian Jacobi. IBM zEC12: The third-generation high-frequency mainframe microprocessor. *IEEE Micro*, 33(2):38–47, 2013. (Citado en página 47)

[63] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the ACID properties for transactional objects. *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 202–213, 2006. (Citado en página 13)

[64] 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 66)

[65] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005. (Citado en páginas 39 y 98)

[66] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015. (Citado en páginas 6, 48 y 148)

[67] D.J. Leith and P. Clifford. Convergence of distributed learning algorithms for optimal wireless channel allocation. *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2980–2985, 2006. (Citado en página 174)

[68] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. *Transact*, 2007. (Citado en página 52)

[69] Yuan Lin and David Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Scientific Programming*, volume 7, pages 231–246, 1999. (Citado en página 59)

UNIVERSIDAD DE MÁLAGA

[70] Karl Ljungkvist, Martin Tillenius, David Black-Schaffer, Sverker Holmgren, Martin Karlsson, and Elisabeth Larsson. Using hardware transactional memory for high-performance computing. In *IEEE Int'l. Symp. on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11)*, pages 1660–1667, 2011. (Citado en página 138)

[71] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, volume 12, pages 128–137. ACM Press, 1977. (Citado en página 5)

[72] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998*, March:42–51, 1998. (Citado en página 63)

[73] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, 2002. (Citado en páginas 133 y 179)

[74] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM SIGPLAN Notices*, 44(6):166, may 2009. (Citado en páginas 58 y 144)

[75] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. (Citado en página 1)

[76] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA*, pages 258–269, 2006. (Citado en página 43)

[77] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, et al. Supporting nested transactional memory in logTM. In *ASPLOS*, 2006. (Citado en páginas 155 y 164)

[78] Vijay Nagarajan and Rajiv Gupta. Speculative optimizations for parallel programs on multicores. In *Languages and Compilers for Parallel Computing*, pages 323–337. Springer Berlin Heidelberg, 2010. (Citado en páginas 145, 151, 165, 180 y 181)

[79] Konstantinos Nikas, Nikos Anastopoulos, Georgios Goumas, and Nectarios Koziris. Employing transactional memory and helper threads to speedup

UNIVERSIDAD DE MÁLAGA

Dijkstra's algorithm. In *38th Int'l Conf. on Parallel Processing (ICPP'09)*, pages 388–395, September 2009. (Citado en página 138)

[80] R. Odaira and T. Nakaike. Thread-level speculation on off-the-self hardware transactional memory. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'14)*, pages 212–221, 2014. (Citado en páginas 7, 44, 107, 135, 136, 138, 143, 152 y 179)

[81] Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'10)*, pages 1–11, December 2010. (Citado en páginas 107 y 115)

[82] Venkatesan Packirisamy, Antonia Zhai, and et. al. Exploring speculative parallelism in SPEC2006. In *ISPASS*, 2009. (Citado en páginas 135 y 179)

[83] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, oct 1979. (Citado en páginas 14 y 26)

[84] Manuel Pedrero, Eladio Gutierrez, Sergio Romero, and Oscar Plata. A comparative analysis of STM approaches to reduction operations in irregular applications. *Journal of Computational Science*, 17:630–638, 2016. (Citado en página 144)

[85] Manuel Pedrero, Eladio Gutierrez, Sergio Romero, and Oscar G. Plata. ReduxSTM: Optimizing STM designs for irregular applications. *Journal of Parallel and Distributed Computing*, 107:114–133, 2017. (Citado en página 144)

[86] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 142–151, 2008. (Citado en páginas 8 y 64)

[87] Leo Porter, Bumyong Choi, and Dean M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 313–324, 2009. (Citado en páginas 7, 58 y 179)

[88] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Hardware signature designs to deal with asymmetry in transactional data sets. *IEEE Transactions on Parallel and Distributed Systems*, 24(3):506–519, March 2013. (Citado en página 96)

UNIVERSIDAD DE MÁLAGA

[89] Ricardo Quislant, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata. Lazy irrevocability for best-effort transactional memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1919–1932, 2017. (Citado en página 149)

[90] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505, 2005. (Citado en página 42)

[91] L. Rauchwerger and D.A. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999. (Citado en páginas 68 y 139)

[92] Lawrence Rauchwerger. Speculative parallelization of loops. In *Encyclopedia of Parallel Computing*, pages 1901–1912. Springer, 2011. (Citado en páginas 62 y 68)

[93] VT Ravi and G Agrawal. Integrating and optimizing transactional memory in a data mining middleware. In *Int'l. Conf. on High Performance Computing (HiPC'09)*, pages 215–224, 2009. (Citado en página 66)

[94] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, volume 4167, pages 284–298. Springer, 2006. (Citado en página 36)

[95] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP need not be considered harmful. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'14)*, 2014. (Citado en página 133)

[96] Wenjia Ruan, Yujie Liu, and Michael Spear. Transactional read-modify-write without aborts. *ACM Transactions on Architecture and Code Optimization*, 11(4):1–24, January 2015. (Citado en página 140)

[97] Karl Rupp. Microprocessor trend data. URL: https://github.com/karlrupp/microprocessor-trend-data.
Último acceso: 7 de noviembre de 2018. (Citado en página 2)

[98] M.M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: extracting parallelism from legacy sequential code using STM. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12)*, 2012. (Citado en páginas 31 y 139)

[99] Juan Salamanca, Jose Nelson Amaral, and Guido Araujo. Evaluating and improving thread-level speculation in hardware transactional memories. In *IPDPS*, 2016. (Citado en páginas 143, 146 y 147)

UNIVERSIDAD DE MÁLAGA

[100] Jack Sampson, Ruben Gonzalez, Jean Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. *IEEE Micro*, pages 235–246, 2006. (Citado en páginas 150, 165, 170, 172 y 181)

[101] 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 páginas 59 y 66)

[102] Nir Shavit and Dan Touitou. Software transactional memory. *Proceedings of the Symposium on Principles of Distributed Computing*, 10(2):204–213, 1995. (Citado en página 33)

[103] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*, pages 141–150, 2009. (Citado en página 89)

[104] Michael F Spear, Kirk Kelsey, Tongxin Bai, Luke Dalessandro, Michael L Scott, Chen Ding, and Peng Wu. Fastpath speculative parallelization. In *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*, pages 338–352, 2009. (Citado en página 96)

[105] Michael F Spear, Virendra J Marathe, William N Scherer III, and Michael L Scott. Conflict detection and validation strategies for software transactional memory. In *20th Int'l. Symp. on Distributed Computing (DISC'06)*, pages 179–193, 2006. (Citado en página 27)

[106] Michael F. Spear, Arrvindh Shriraman, Luke Dalessandro, and Michael L. Scott. Transactional mutex locks. In *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'14)*, 2009. (Citado en página 138)

[107] Jaswanth Sreeram and Santosh Pande. Parallelizing a real-time physics engine using transactional memory. In *17th Int'l. Euro-Par Conference (Euro-Par'11)*, pages 206–223, 2011. (Citado en página 138)

[108] Abhishek Udupa, Kaushik Rajan, and William Thies. ALTER: Exploiting breakable dependences for parallelization. In *32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'11)*, pages 480–491, 2011. (Citado en página 139)

UNIVERSIDAD DE MÁLAGA

[109] C. von Praun, C. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *12th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 79–89, 2007. (Citado en páginas 7, 32, 58, 139 y 179)

[110] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, page 127, 2012. (Citado en páginas 6 y 45)

[111] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–23, jan 2012. (Citado en páginas 46 y 138)

[112] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 1–11. ACM Press, 2013. (Citado en página 6)

[113] H. Yu and L. Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 17(10):1084–1096, 2006. (Citado en páginas 59 y 65)

[114] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. WormBench - a configurable workload for evaluating transactional memory systems. In *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 61–68, 2008. (Citado en páginas 107 y 117)

UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD
DE MÁLAGA