Mejorando el Rendimiento de la Memoria Transaccional para Aplicaciones Irregulares

Manuel Pedrero [1] Eladio Gutiérrez [2] Sergio Romero [3] Óscar Plata [4]

Resumen— **La Memoria Transaccional (TM) ofrece un modelo de ejecución concurrente optimista en arquitecturas multinúcleo, permitiendo a los programadores extraer paralelismo cuando la información de las dependencias de datos no está disponible hasta la ejecución del programa. Existe investigación reciente enfocada a explotar paralelismo a nivel de hilo usando TM. Sin embargo estas propuestas son de uso general, válidas para cualquier tipo de aplicación.**

Este trabajo presenta ReduxSTM, un TM *software* especialmente diseñado para mejorar la extracción de paralelismo en aplicaciones irregulares. La gestión de las versiones y la detección de conflictos se han diseñado para aprovechar tanto la ordenación secuencial de las transacciones, necesaria para asegurar la corrección de los resultados, como la privatización de patrones de reducción, un patrón de acceso a memoria muy frecuente en aplicaciones irregulares. La información adicional que proporcionan estas propiedades en tiempo de ejecución se utiliza para evitar abortos transaccionales innecesarios.

Se ha elegido una función del *benchmark* 300.twolf de la suite SPEC CPU2000 como ejemplo de programa irregular con reducciones. Este código se ha paralelizado mediante TM utilizando ReduxSTM y una versión ordenada de TinySTM. Los resultados muestran que ReduxSTM es capaz de explotar más paralelismo.

Palabras clave— **Aplicaciones irregulares, Memoria transaccional, thread-level speculation, patrones de reducción**

I. Introducción

La disponibilidad de múltiples núcleos que comparten una memoria global en los procesadores de uso común está teniendo una fuerte influencia en cómo las aplicaciones han de ser diseñadas para que puedan beneficiarse de toda la potencia computacional disponible. Cuando se descompone un problema en varias tareas concurrentes, el rendimiento obtenible está sujeto a la correcta resolución de las dependencias de datos y de control en dichas tareas. En general, las dependencias se suelen resolver de modo conservativo, especialmente cuando se resuelven en tiempo de compilación. Tal es el caso de las aplicaciones que exhiben patrones de acceso irregulares, cuyas dependencias no pueden determinarse hasta la ejecución. En este contexto, la memoria transaccional (TM) [1] proporciona un soporte optimista para la concurrencia en arquitecturas multinúcleo, ayudando a los programadores a explotar paralelismo en aplicaciones irregulares cuando la información de las dependencias de datos no se puede analizar fácilmente (o no está disponible) antes de la ejecución.

La memoria transaccional aparece como un paradigma de coordinación de hilos concurrentes. Para ello proporciona el concepto de *transacción*, un constructor que asegura la atomicidad, consistencia y aislamiento en la ejecución de las instrucciones envueltas en la misma. Las transacciones pueden ejecutarse de manera concurrente, pero siempre asegurando que los resultados sean los mismos que se obtendrían si se ejecutasen en serie.

En un sistema TM, las transacciones se ejecutan de forma especulativa y sus cambios son monitorizados por un gestor de versiones de datos. Si dos transacciones concurrentes conflictúan (realizan una escritura/escritura, escritura/lectura en una misma dirección compartida), una de ellas debe abortar. Tras restaurar su estado inicial, la transacción abortada reinicia su ejecución. Cuando una transacción finaliza su ejecución, entra en una fase de confirmación o *commit*, donde los cambios en memoria se hacen definitivos. El diseño del gestor de versiones puede ser *eager* si los cambios se trasladan a memoria inmediatamente, y se usa un *undo-log* para almacenar los valores previos (para ser restaurados en caso de aborto). Por el contrario, un gestor de versiones *lazy*, almacena las operaciones de escritura en un *write-buffer*, y retrasa su volcado a memoria compartida hasta la fase de *commit*. De una forma similar, el gestor de conflictos puede detectarlos en el instante en que ocurren (*eager*), o puede posponer su comprobación a un momento posterior, generalmente la fase de *commit* (*lazy*). En las últimas dos décadas se han propuesto muchos sistemas TM, implementados tanto en *software* (STM) como en *hardware* (HTM), o combinando ambos enfoques (HyTM) [2].

Animados por los beneficios de los sistemas TM, se han realizado esfuerzos para aprovechar la TM para extraer paralelismo de aplicaciones secuenciales. De hecho, muchas operaciones básicas de un sistema TM, como la detección y resolución de conflictos de memoria, el almacenamiento de las escrituras a memoria compartida y los reinicios de transacciones se requieren también para especulación multihilo (SpMT) o *thread-level speculation* (TLS) [3]. Estas técnicas han demostrado ser útiles a la hora de extraer paralelismo de programas secuenciales.

En general, extraer paralelismo multihilo a partir de un programa secuencial requiere descomponer dicho programa en tareas y resolver correctamente las dependencias entre las mismas. Sin embargo no siempre es posible detectar y resolver estas dependencias

[1]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: mpedrero@uma.es

[2]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: eladio@uma.es

[3]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: sromero@uma.es

[4]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: oplata@uma.es

de forma estática (en tiempo de compilación), sobre todo en aplicaciones complejas. En estos casos las técnicas SpMT/TLS pueden ser de utilidad a la hora de encontrar paralelismo, ya que no requieren un análisis estático de dependencias de datos. De este modo, la concurrencia optimista que aportan los sistemas TM puede ayudar a paralelizar aplicaciones irregulares. Las tareas, que podemos definir como secciones de código, se pueden ejecutar como transacciones concurrentes de manera que sea el sistema TM quien se encargue de monitorizar los accesos a memoria durante la ejecución de las mismas, detectando y resoliviendo posibles conflictos (dependencias de datos) entre transacciones. Hay que tener en cuenta, sin embargo, que se deben cumplir algunas restricciones de orden entre transacciones para evitar violaciones de las dependencias de datos y asegurar resultados correctos. En general, las transacciones deben finalizar en un orden que preserve la semántica secuencial.

Se pueden encontrar trabajos de investigación que proponen explotar TLS usando técnicas TM [4] [5] [6] [7] [8] [9]. La mayoría de estas técnicas son de uso general, sin centrarse en aplicaciones específicas. Sin embargo, las técnicas especulativas son especialmente útiles para aplicaciones irregulares, como se ha explicado anteriormente. En este trabajo presentamos ReduxSTM, un sistema TM específicamente desarrollado para extraer paralelismo de aplicaciones irregulares, considerando que no se tiene a priori información alguna sobre dependencias de datos. Para explotar el paralelismo entre tareas, el sistema está diseñado para aprovechar tanto el orden secuencial de ejecución de transacciones como la privatización de patrones irregulares de reducción, evitando así abortos y reejecuciones innecesarias (sólo las dependencias reales implicarán algún aborto por parte de las transacciones).

Como aplicación motivadora se ha seleccionado un *benchmark* de la suite SPEC CPU2000. Este código contiene un comportamiento irregular en cuanto a accesos a memoria que lo hace un buen candidato para ser paralelizado usando TLS, ya que sus dependencias no son fácilmente analizables en tiempo de compilación. La paralelización de este tipo de aplicaciones requiere un esfuerzo importante. El objetivo de este artículo es mostrar cómo ReduxSTM puede ayudar a simplificar significativamente el trabajo del programador a la hora de paralelizar el código, extrayendo un mayor nivel de paralelismo que otros sistemas TM ordenados.

II. Aplicación Irregular Motivadora

Como caso de estudio y aplicación motivadora se ha seleccionado el benchmark de enrutamiento y posicionamiento de componentes TimberWolfSC (300.twolf) de la suite SPEC CPU2000. Este programa determina la posición y conexiones globales de componentes estándar en un microchip.

Nos hemos centrado en la función `new_dbox_a()` incluida en el fichero `dimbox.c`, en el *benchmark* 300.twolf. Esta función se muestra en la figura 1. El código tiene un bucle externo que incluye dos sentencias con patrones de reducción, las cuales usan `costptr` y `delta_vert_cost` como variables de reducción. Sin embargo, como `costptr` es un puntero, su dirección final puede coincidir con otra variable de reducción o con otras variables globales utilizadas dentro del bucle, como `tmp_num_feeds[]` o `tmp_missing_rows[]`. Debido a esto, el compilador no es capaz de analizar las dependencias ni por tanto de extraer paralelismo de este bucle.

Aunque no se pueden aplicar técnicas convencionales de paralelización, la condición de reducción puede ser válida para algunos de los accesos a la variable de reducción (este fenómeno se denomina *reducción parcial* [10] [11]). Si la fracción de accesos que cumplen dicha condición es suficientemente alto, se puede extraer un buen porcentaje de paralelismo del bucle.

III. Diseño de ReduxSTM

El principal objetivo en el diseño de ReduxSTM es aprovechar los mecanismos básicos de TM para explotar paralelismo especulativo a partir de aplicaciones secuenciales (especialmente las que presentan patrones irregulares en su acceso a memoria), considerando que no hay disponible ninguna información sobre sus dependencias de datos.

Transacciones ordenadas. Una característica necesaria en ReduxSTM es mantener las restricciones de orden secuencial entre transacciones. La ejecución concurrente de las tareas especulativas en las que se ha descompuesto la aplicación debe preservar la semántica secuencial del programa para asegurar la corrección del resultado final. Por ello, ReduxSTM debe finalizar las transacciones de acuerdo con el orden de ejecución secuencial. Esta restricción de orden se puede añadir al gestor de conflictos pero conlleva una pérdida de rendimiento debido a los retrasos que experimentan las transacciones que finalizan fuera de orden. Para hacer frente a este problema, el gestor de conflictos utiliza la información adicional del orden de cada transacción para filtrar algunos conflictos y evitar así abortos y reinicios innecesarios. [3] [12]

Patrones de reducción. Una segunda característica de ReduxSTM es un mecanismo de privatización. Esto permite filtrar conflictos innecesarios entre trasancciones, mejorando la explotación del paralelismo. De hecho, la privatización ha demostrado ser una técnica clave a la hora de conseguir paralelismo. [13] [14]

En particular, ReduxSTM utiliza el mecanismo de privatización implícito de TM para privatizar selectivamente los patrones acceso a memoria en reducciones. Las aplicaciones científicas presentan a menudo estos patrones y, en el caso de problemas irregulares, puede explotarse una buena cantidad de paralelismo mediante técnicas de privatización. Un patrón de reducción se caracteriza por una sentencia de reducción en el programa de la forma `A[] = A[]` $\otimes$ ξ, donde `A[]` es la variable de reducción (un array en general), ξ es una expresión que no incluye `A[]` y $\otimes$ es un operador asociativo y conmutativo. Esta sentencia se

```
1  new_dbox_a(...) {
2    ...
3    for ( termptr = antrmptr; termptr; termptr = termptr->nextterm ) {
4      ...
5      new_mean = ...; old_mean = ...;
6      for ( netptr = dimptr->netptr; netptr; netptr = netptr->nterm ) {
7        oldx = netptr->xpos;
8        if( netptr->flag == 1 ) { newx = netptr->newx; netptr->flag = 0; }
9        else { newx = oldx; }
10
11       // (1) Sentencia potencial de reducción
12       *costptr += ABS(newx - new_mean) - ABS(oldx - old_mean);
13     }
14     ...
15     tmp_num_feeds[net] = f;     // Alias potencial entre *costptr y delta_vert_cost
16     ...
17     tmp_missing_rows[net] = -m; // Alias potencial entre *costptr y delta_vert_cost
18
19     // (2) Sentencia potencial de reducción
20     delta_vert_cost +=
21         ((tmp_num_feeds[net]-num_feeds[net]) +
22         (tmp_missing_rows[net]-missing_rows[net])) *
23         2 * rowHeight;
24   }
25   return;
26 }
```

Fig. 1. Función `new_dbox_a()` del código 300.twolf de la *suite* SPEC CPU2000

encuentra en el cuerpo de un bucle.

En códigos irregulares los accesos a las variables de reducción suelen ir ligados al uso de indirecciones o punteros, lo que dificulta su paralelización. Si la variable de reducción es leída o escrita fuera de la sentencia de reducción puede que estemos ante una reducción parcial, donde las condiciones de reducción sólo son válidas en una porción de la variable de reducción.

Para habilitar la privatización de reducciones en ReduxSTM, se ha definido una primitiva explícita para reducciones. Esta primitiva es controlada por los gestores de conflictos, versión y *commit* como una tercera operación transaccional: lectura (R), escritura (W) y reducción (Rdx). Sin soporte específico de reducciones, Rdx estaría representada como una combinación de dos operaciones; una lectura seguida de una escritura en la misma posición de memoria de la variable de reducción:

$Rdx(addr, val, \otimes) := W(addr, R(addr) \otimes val)$

Gestión de versiones. Para poder soportar la privatización de operaciones de reducción la gestión de versiones de ReduxSTM es *lazy*, es decir, las actualizaciones a memoria compartida se mantienen en un *buffer* privado durante la ejecución de la transacción. El *buffer* de escritura, utilizado normalmente para mantener los nuevos valores de las escrituras transaccionales, se ha extendido para privatizar los valores de las reducciones transaccionales. La extensión consiste en una etiqueta de estado asociada a cada entrada del *buffer*, que especifica si la entrada correspondiente almacena un valor a escribir o a reducir (junto con la operación de reducción correspondiente). Cada vez que se lanza una reducción transaccional, se realiza una consulta en este nuevo *buffer* de la dirección a reducir. Si dicha dirección se encuentra en él y la etiqueta de estado corresponde a una escritura, la expresión de la operación de reducción se reduce con el valor almacenado en el *buffer*. Sin embargo, la etiqueta de estado se mantiene como escritura, ya que la condición de reducción no se ha cumplido. Por el contrario, si la etiqueta de estado corresponde a una reducción, se realiza un proceso similar, pero la etiqueta de estado se mantiene como una reducción. De un modo parecido, una escritura transaccional requiere también de una búsqueda de la dirección en el *buffer*. Si se encuentra, la entrada es actualizada y la etiqueta de estado se actualiza a escritura, independientemente del estado original. El nuevo *buffer* se denomina *buffer* de actualización.

Gestión de conflictos. La detección de conflictos en ReduxSTM es también *lazy*, es decir, se realiza durante la fase de *commit*, permitiendo filtrar la mayoría de los conflictos y mejorando así la concurrencia entre transacciones. La tabla I muestra qué conflictos se evitan gracias a la información de orden de las transacciones y la privatización de las operaciones de reducción. La primera fila especifica dos operaciones transaccionales ejecutadas por transacciones diferentes donde la primera debe finalizar antes de la segunda (para mantener el orden secuencial). Por ejemplo, R–W representa una antidependencia. La segunda fila corresponde al comportamiento de un TM clásico, mientras que las últimas dos filas consideran el soporte adicional de orden y de reducciones. El gestor de conflictos utiliza conjuntos de lectura y de escritura para la detección de conflictos. Dichos conjuntos han sido implementados como signaturas basadas en filtros de Bloom.

Gestión de *commit*. Para minimizar el sobrecoste de las operaciones transaccionales y dar soporte al sistema *lazy-lazy*, ReduxSTM se basa en un algoritmo *full-commit-invalidation* [15], donde los conflictos son detectados y resueltos durante el *commit*, invalidando ante un conflicto la transacción activa, que será siempre menos prioritaria al encontrarse fuera de su fase de *commit*.

La fase de *commit* se encarga de las siguientes cuatro tareas: (1) Comprobar si las restricciones de orden se cumplen (en caso contrario, esperar a que se

TABLA I
CONFLICTOS CON TRANSACCIONES ORDENADAS EN DIFERENTES ESCENARIOS

	R−W	W−R	W−W	Rdx−R	R−Rdx	Rdx−W	W−Rdx	Rdx−Rdx
TM	aborto	aborto	aborto	-	-	-	-	-
TM + Order	s/c	aborto	s/c	-	-	-	-	-
TM + Order + Rdx	s/c	aborto	s/c	aborto	s/c	s/c	s/c	s/c

n/c = sin conflicto

cumplan); (2) comprobar si la transacción ha sido invalidada por una transacción previa ya finalizada; (3) actualizar la memoria principal con los valores almacenados en el *buffer* de actualización, reduciendo en memoria las operaciones correspondientes; (4) comparar el filtro de Bloom de actualización con los filtros de Bloom de lectura de todas las transacciones activas subsiguientes (en el orden secuencial establecido), invalidando aquellas en las que se detecte un conflicto.

IV. EVALUACIÓN EXPERIMENTAL

En esta sección se evalúa el rendimiento de ReduxSTM aplicado a una versión paralela de la función `new_dbox_a()` del *benchmark* 300.twolf. El objetivo es mostrar cómo ReduxSTM permite extraer más paralelismo del código secuencial que otros sistemas TM ordenados actuales, gracias al filtrado de conflictos y a la privatización selectiva de reducciones.

A. Configuración experimental

Los experimentos se han realizado en un servidor con 8 GB RAM y un procesador quad-core Intel Core i7-3770 a 3.4GHz que soporta 8 hilos concurrentes. El servidor usa Linux (kernel 3.2.0-75 x86_64) y todos los programas se han compilado usando GNU GCC (con optimización `-O2`). Como TM base para la comparación se ha utilizado la última versión de TinySTM (v1.0.5), que permite la ejecución ordenada de transacciones. [16] [17]

Para la evaluación se ha utilizado la siguiente metodología: primero se ha instrumentado el programa original para obtener una traza de los accesos a memoria relevantes, es decir, los accesos a memoria compartida que pueden por tanto ocasionar conflictos (acceso compartido). La figura 2 muestra la versión instrumentada de la función `new_dbox_a()`. Las macros `MARKREAD()`, `MARKWRITE()` y `MARKREDUX()` especifican qué accesos a memoria deben aparecer en la traza y qué tipo de operación se ha de realizar (lectura, escritura o reducción). `MARKREDUX()` marca una posición de memoria sujeta a un patrón potencial de reducción. Esta reducción es *potencial* porque será el gestor de conflictos de ReduxSTM el encargado de determinar en tiempo de ejecución si dicho patrón es realmente una reducción o no. En un segundo paso, un simulador procesa la traza de accesos a memoria que simula la función original `new_dbox_a()` recreando tanto el patrón de accesos a memoria como los valores reales utilizados. Esta simulación se realiza en paralelo y utilizando memoria transaccional. Para ello se ha particionado el bucle externo de la función simulada en bloques de iteraciones consecutivas (*chunks*), cada uno de los cuales es ejecutado como una transacción. Las transacciones se asignan a los diferentes hilos de ejecución siguiendo un esquema *round-robin*.

Los experimentos se han ejecutado con una carga de trabajo media (correspondiente al perfil *training* del *benchmark* 300.twolf). El tiempo de ejecución de esta configuración es largo; la función `new_dbox_a()`, en concreto, ejecuta casi 12 millones de iteraciones. La traza de memoria resultante contiene aproximadamente 550M de lecturas, 46M de escrituras y 70M de reducciones. Los resultados que se van a mostrar en el artículo se han obtenido a partir de una sección de 500K iteraciones obtenidas de la parte media de la traza de ejecución. Esta sección representa alrededor de 24M de lecturas, 2M de escrituras y 3M de reducciones de la traza de memoria. Un dato interesante es que los 29M de accesos a memoria totales contienen sólo 43K de posiciones de memoria diferentes (menos del 0.15 % de las referencias de memoria son diferentes). Esto conlleva unos niveles de contención de memoria altos en la ejecución. Además, el código del *benchmark* es *memory-bounded*, lo que limita la cantidad de paralelismo que se puede explotar.

Los experimentos incluyen las siguientes versiones de la aplicación: (1) La versión original secuencial (ejecutada en un hilo); (2) una versión paralelizada usando el TM TinySTM en su versión ordenada, donde el orden de finalización viene dado por el orden de inicio de las transacciones (primera en comenzar, primera en finalizar); (3) una versión paralelizada basada en ReduxSTM con ordenación implícita basada en el momento de entrada a la fase de *commit* (primera en terminar su ejecución, primera en finalizar); (4) una versión paralelizada basada en ReduxSTM con ordenación explícitamente especificada por el programador (de modo que el orden de las transacciones sea el mismo que el orden secuencial). Esta última versión (4) es la única que asegura la corrección de los resultados (ya que el orden de finalización de las transacciones equivale al secuencial). Las otras dos versiones TM, cuyas restricciones de orden son más relajadas, se usan como referencia a efectos de comparación.

B. Rendimiento

La figura 3 (izquierda) muestra la aceleración obtenida con ambas versiones de ReduxSTM (con orden implícito y explícito) comparada con la obtenida usando TinySTM con orden y con el rendimiento de la versión secuencial. Los valores mostrados corresponden al óptimo de los experimentos realizados con

```
1  new_dbox_a(...) {
2    ...
3    for ( termptr=antrmptr; termptr; termptr=termptr->nextterm ) {
4      ...
5      MARKREAD(&dimptr->new_total);
6      MARKREAD(&dimptr->numpins);
7      MARKREAD(&dimptr->old_total);
8      new_mean = dimptr->new_total / dimptr->numpins;
9      old_mean = dimptr->old_total / dimptr->numpins;
10     MARKREAD(&dimptr->netptr);
11     for ( netptr=dimptr->netptr; netptr; netptr=netptr->nterm ) {
12       MARKREAD(&netptr->xpos);
13       oldx = netptr->xpos;
14       MARKREAD(&netptr->flag);
15       if( netptr->flag == 1 ) {
16         MARKREAD(&netptr->newx);
17         newx = netptr->newx;
18         netptr->flag = 0;
19         MARKWRITE(&netptr->flag,0);
20       }
21       else { newx = oldx; }
22
23       // (1) Sentencia potencial de reducción
24       *costptr += ABS(newx - new_mean) - ABS(oldx - old_mean);
25       MARKREDUX(&(*costptr),ABS(newx - new_mean) - ABS(oldx - old_mean));
26       MARKREAD(&netptr->nterm);
27     }
28     ...
29     tmp_num_feeds[net] = f; // Alias potencial entre *costptr y delta_vert_cost
30     MARKWRITE(&tmp_num_feeds[net],f);
31     ...
32     tmp_missing_rows[net] = -m; // Alias potencial entre *costptr y delta_vert_cost
33     MARKWRITE(&tmp_missing_rows[net],-m);
34     MARKREAD(&tmp_num_feeds[net]);
35     MARKREAD(&num_feeds[net]);
36     MARKREAD(&tmp_missing_rows[net]);
37     MARKREAD(&missing_rows[net]);
38
39     // (2) Sentencia potencial de reducción
40     delta_vert_cost += ((tmp_num_feeds[net] - num_feeds[net]) +
41                         (tmp_missing_rows[net] - missing_rows[net])) *
42                         2 * rowHeight;
43     MARKREDUX(&(delta_vert_cost), ((tmp_num_feeds[net] - num_feeds[net]) + ...));
44     MARKREAD(&termptr->nextterm);
45   }
46   return;
47 }
```

Fig. 2. Instrumented `new_dbox_a()` function

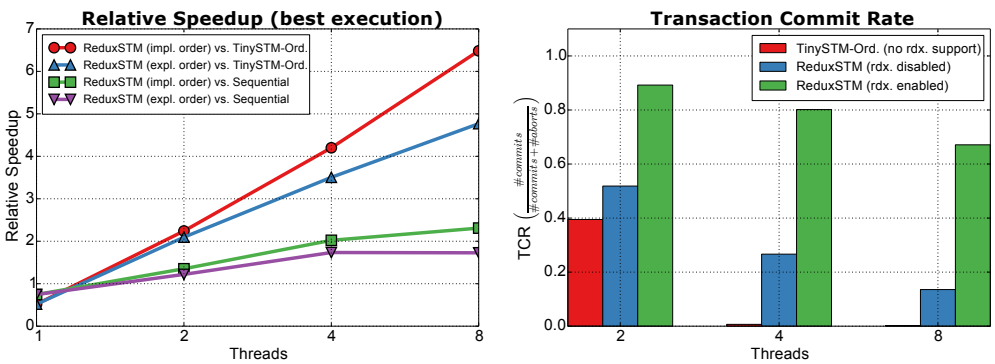

Fig. 3. Aceleración de ReduxSTM sobre TinySTM y versión secuencial (izda.), TCR de ReduxSTM y TinySTM (dcha.)

distintos tamaños de *chunk* en cada STM (de 1 a 8 iteraciones de bucle por transacción) dado que cada TM obtiene su mejor rendimiento con un tamaño de *chunk* diferente.

La aceleración obtenida con ReduxSTM es significativamente mayor que la obtenida usando TinySTM gracias a la habilidad del primero de evitar abortos y reinicios innecesarios. En estas pruebas se muestra cómo ReduxSTM es capaz de explotar el paralelismo a partir de un código secuencial, a pesar de ser *memory-bounded*. En códigos más intensivos computacionalmente, la mejora esperada es superior.

En general se espera que el número de conflictos entre transacciones aumente con el número de hilos. Sin embargo, ReduxSTM consigue una mejora de rendimiento superior a TinySTM gracias a que su diseño le permite filtrar gran parte de esos conflictos. Con 8 hilos, ReduxSTM es 6.5 veces (con orden implícito) o 4.8 veces (con orden explícito) mejor que TinySTM con orden.

C. Transaction Commit Rate

En la figura 3 (derecha) puede verse una comparativa del *Transaction Commit Rate* (TCR) de ambos sistemas TM. El TCR se define como el porcentaje de transacciones finalizadas respecto del total de transacciones ejecutadas, y es un parámetro usado a menudo para medir el nivel de concurrencia explotada. En la figura se muestran dos versiones de ReduxSTM; una con el soporte de reducciones activo y otra con esta característica desactivada. Se puede observar como para esta aplicación, este soporte de reducciones permite explotar mucho más paralelismo, especialmente al aumentar el número de hilos. Por ejemplo, para 8 hilos, la versión de ReduxSTM con soporte de privatización de reducciones obtiene un TCR de casi el 70 % (es decir, casi el 70 % de transacciones se ejecutaron y finalizaron sin ningún reinicio debido a conflictos). Sin este soporte de reducciones, sin embargo, el TCR baja a menos de un 20 %. Por su parte, el TCR de TinySTM ordenado es prácticamente cero (por lo que básicamente, las transacciones se ejecutan en serie, una detrás de otra).

De estos resultados se puede concluir que el filtrado adicional de conflictos que permite el reconocimiento y tratamiento de los patrones de reducción es muy relevante, ya que permite mantener un TCR elevado al aumentar el número de hilos de ejecución. Del mismo modo, en un contexto de ejecución especulativa, donde es necesario establecer un orden entre las transacciones, poder filtrar conflictos gracias a la información de orden también resulta beneficioso. Conviene mencionar que, gracias a su diseño, ReduxSTM limita el número de reinicios máximo que una transacción puede sufrir debido a conflictos al número de hilos utilizados menos uno.

D. Sensibilidad al tamaño de transacción

El impacto del tamaño de la transacción (en términos del número de iteraciones del bucle externo de la función `new_dbox_a()`) en el rendimiento de ReduxSTM se muestra en la figura 4 (izquierda). El gráfico muestra la aceleración obtenida usando ReduxSTM respecto a la ejecución con 1 hilo y transacciones de una iteración.

Con un hilo, el rendimiento crece con el tamaño de la transacción, ya que el número total de transacciones se reduce y por tanto el sobrecoste debido a su instrumentación. Cuando el número de hilos aumenta, la probabilidad de conflictos crece con el número de transacciones y decrementa el rendimiento obtenido. La figura muestra unas aceleraciones modestas debido a la naturaleza *memory-bounded* de la aplicación.

E. Sensibilidad a la carga computacional

Para poder comprobar el rendimiento de ReduxSTM en situaciones donde hay una combinación más balanceada de accesos a memoria y computación, se han realizado una serie de experimentos con diferentes cargas computacionales añadidas a las transacciones. Esta carga adicional se ha especificado en términos de un número de operaciones en punto flotante por acceso a memoria. La figura 4 (derecha) muestra los resultados obtenidos, medidos como aceleración respecto a la ejecución usando un hilo.

Como es de esperar, el rendimiento de ReduxSTM con 4 y 8 hilos mejora significativamente al aumentar la carga computacional, ya que el impacto relativo en el rendimiento de la gestión de transacciones disminuye.

F. Sensibilidad al tamaño del filtro de Bloom

ReduxSTM utiliza filtros de Bloom (signaturas) como estructuras de datos para almacenar los conjuntos de lectura y actualización (escrituras y reducciones) de las transacciones. Estas estructuras basadas en funciones *hash* permiten realizar operaciones rápidas de inserción y comprobación para un conjunto no limitado de direcciones de memoria, pero tienen la desventaja de que existe una cierta probabilidad de obtener falsos positivos (debido a posibles alias de la función de *hash*). Aumentar el tamaño de estos filtros reduce esta probabilidad a expensas de ocupar más espacio y de incrementar su tiempo de reinicio (que es necesario realizar después de un aborto o de un *commit*). Se hace necesario por tanto encontrar un equilibrio entre el tamaño de los filtros y la probabilidad de falsos positivos que maximice el rendimiento final.

La figura 5 muestra el rendimiento de ReduxSTM para diferentes tamaños de filtros de Bloom (tanto el filtro de lectura como el de actualización tienen el mismo tamaño). El rendimiento se mide como el ratio entre el TCR y el tiempo de ejecución obtenido, ya que maximizar dicho ratio significa obtener una máxima explotación de la concurrencia con un tiempo mínimo de ejecución.

A partir de la figura podemos concluir que el mejor tamaño de filtro está en el rango entre 2^{14} y 2^{18} bits para los diferentes tamaños de transacción y de hilos simultáneos. Con tamaños menores en los filtros se producen más abortos debido a falsos positivos, aunque el rendimiento permanece constante ya que el número máximo de abortos para una transacción está limitado en el diseño de ReduxSTM por el número de hilos en ejecución. Para filtros demasiado grandes, el rendimiento cae rápidamente debido al sobrecoste de almacenar los filtros en los primeros niveles de la jerarquía de la caché y el coste de reiniciarlos tras un aborto o una nueva transacción.

G. Conclusiones

Es un hecho que la mayoría de aplicaciones que exhiben patrones de acceso irregulares son complejas de paralelizar. En estas aplicaciones es común que no exista información disponible sobre las dependencias de datos en tiempo de compilación. En este contexto, el soporte de concurrencia optimista que proporciona la memoria transaccional (TM) puede ser útil para extraer paralelismo de estas aplicaciones. En este trabajo se ha presentado ReduxSTM, un

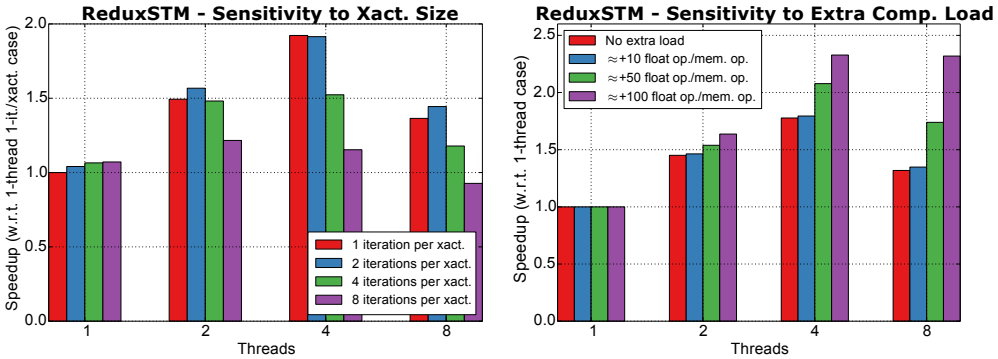

Fig. 4. Sensibilidad al tamaño de transacción (izda.), y a carga computacional (dcha.)

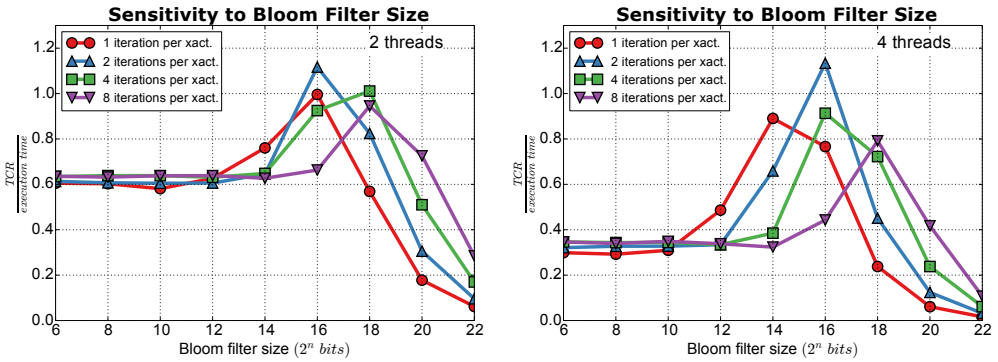

Fig. 5. Sensibilidad al tamaño del filtro de Bloom

TM *software* (STM) específicamente diseñado para extraer paralelismo de aplicaciones irregulares. Para ello se han adaptado el gestor de conflictos y de versiones para privatizar selectivamente variables de reducción y para aprovechar el orden secuencial, necesario para obtener un resultado correcto del programa. Ambas características han permitido evitar reinicios innecesarios de las transacciones filtrando conflictos potenciales. Los resultados experimentales utilizando una aplicación del *benchmark* SPEC CPU2000 muestran que ReduxSTM mejora sustancialmente el rendimiento de un sistema STM convencional.

Referencias

[1] M. Herlihy and J. Moss, "Transactional Memory: Architectural support for lock-free data structures," in *20th Ann. ISCA*, 1993.

[2] Tim Harris, James Larus, and Ravi Rajwar, *Transactional Memory, 2nd*, Morgan & Claypool Publishers, USA, 2010.

[3] L. Porter, B. Choi, and D.M. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *18th Int'l. Conf. on PACT*, 2009.

[4] C. von Praun, C. Ceze, and C. Cascaval, "Implicit parallelism with ordered transactions," in *12th ACM Symp. on Principles and Practice of Parallel Programming*, 2007, pp. 79–89.

[5] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup Dijkstra's algorithm," in *38th Int'l. Conf. on Parallel Processing (ICPP'2009)*, 2009, pp. 388–395.

[6] M. Mehrara, J. Hao, P-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *30th ACM Conf. on PLDI*, 2009.

[7] M. DeVuyst, D.M. Tullsen, and S.W. Kim, "Runtime parallelization of legacy code on a transactional memory system," in *6th HiPEAC Conf.*, 2011.

[8] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *ACM/IFIP/USENIX 13th Int'l. Middleware Conf.*, 2012, pp. 187–207.

[9] 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, pp. 1–7.

[10] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 2, 1999.

[11] L. Han, W. Liu, and J.M. Tuck, "Speculative parallelization of partial reduction variables," in *8th Ann. IEEE/ACM Int'l. Symp. on CGO*, 2010.

[12] M.A. Gonzalez-Mesa, E. Gutierrez, E.L. Zapata, and O. Plata, "Effective transactional memory execution management for improved concurrency," *ACM Trans. on Architecture and Code Optimization*, vol. 11, no. 3, 2014.

[13] P. Tu and D. Padua, "Automatic array privatization," in *6th Int'l. Workshop on Languages and Compilers for Parallel Computing*, 1994, pp. 500–521.

[14] N.P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D.I. August, "Speculative separation for privatization and reductions," in *33rd ACM Conf. on PLDI*, 2012.

[15] Justin E Gottschlich, Manish Vachharajani, and Jeremy G Siek, "An Efficient Software Transactional Memory Using Commit-Time Invalidation," pp. 101–110, 2010.

[16] 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)*, 2008, pp. 237–246.

[17] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.