Análisis comparativo del uso de STMs en códigos de reducción irregulares

Manuel Pedrero, Eladio Gutiérrez, Sergio Romero y Óscar Plata[1]

Resumen— **La memoria transaccional (TM) constituye un paradigma de concurrencia optimista en arquitecturas multinúcleo que puede ser de utilidad en la explotación de paralelismo en aplicaciones irregulares, en las que la información sobre las dependencias de datos no está disponible hasta la ejecución. Este trabajo presenta y discute cómo aprovechar las características de un sistema STM (software transactional memory) en patrones de computación que involucren operaciones de reducción, ligadas frecuentemente a aplicaciones irregulares. Con el fin de comparar el uso de enfoques STM en esta clase de patrones con otras soluciones más clásicas, se ha implementado como prueba de concepto un sistema STM, que denominaremos ReduxSTM, que combina dos ideas: una consolidación (commit) ordenada de las transacciones, que asegura una equivalencia con la ejecución secuencial del código; y una extensión del mecanismo de privatización subyacente al sistema STM que contempla las operaciones de reducción.**

Palabras clave— **Software Transactional Memory (STM), operaciones de reducción, aplicaciones irregulares.**

I. Introducción

La disponibilidad de múltiples núcleos compartiendo una memoria global en los computadores actuales está teniendo una gran influencia en cómo se diseñan las aplicaciones. Al descomponer un problema en tareas concurrentes, el rendimiento final está determinado en buena parte por cómo se gestionan las posibles dependencias de datos y de control. En general las dependencias se gestionan de una manera conservadora, especialmente cuando se resuelven en tiempo de compilación. En el caso de aplicaciones que presentan un patrón irregular de referencias de memoria ésta gestión puede resultar muy limitada, ya que muchas dependencias no se conocen completamente hasta que la aplicación no se ejecuta. En este contexto, la memoria transaccional (TM) [1] proporciona un modelo de concurrencia optimista en arquitecturas multinúcleo, facilitando al programador la explotación de paralelismo en códigos –como los de las aplicaciones irregulares– que no son fácilmente analizables de forma estática.

TM ha surgido como una alternativa que facilita la coordinación de threads concurrentes. TM proporciona el concepto de transacción: una estructura de programación que garantiza atomicidad, consistencia y aislamiento en bloque de las instrucciones que la componen. Las transacciones pueden ejecutarse concurrentemente, pero el sistema garantiza que los resultados de la ejecución sean los mismos que en una ejecución secuencial.

[1]Dpto. de Arquitectura de Computadores, Universidad de Málaga, 29071 Málaga, e-mail: mpedrero@uma.es, eladio@uma.es, sromero@uma.es, oplata@uma.es

En un sistema TM, las transacciones se ejecutan de manera especulativa, de forma que las modificaciones en las posiciones de memoria quedan registradas por un gestor de versiones. Si dos transacciones concurrentes entran en conflicto (escritura/escritura, lectura/escritura en la misma posición de memoria), una de ellas debe abortar. Tras restaurar su estado inicial (rollback), la transacción que aborta reintentará la ejecución especulativa. Cuando una transacción termina su ejecución libre de conflictos, consolida sus datos modificados (commit) haciéndolos visibles al resto del sistema. Se dice que la gestión de versiones en un sistema TM es *eager* si los cambios se trasladan a memoria inmediatamente, manteniendo un buffer con los valores antiguos (*undo log*) que se usará para restaurar el estado inicial en caso de aborto. Por el contrario, en una gestión de versiones *lazy* las modificaciones se mantienen en un buffer de escritura (*redo log*), consolidándose al final de la transacción, durante la fase de commit. Con una terminología similar, la detección de conflictos se puede realizar inmediatamente (*eager*) o bien posponerse hasta el comienzo de la fase de commit (*lazy*). Encontramos en la literatura un buen número de propuestas TM, implementadas bien en software (STM), en hardware (HTM) o con enfoques híbridos (HyTM) [2].

Dadas las ventajas de TM, se han realizado esfuerzos encaminados a extraer paralelismo de aplicaciones secuenciales ya existentes mediante este paradigma. De hecho, muchas de las características básicas de TM, como la detección de conflictos, la privatización especulativa de posiciones de memoria y el aborto/reintento de la ejecución se pueden encontrar en otras técnicas como el multithreading especulativo (SpMT), o la especulación a nivel de thread (thread-level speculation ó TLS) [3], las cuales se han probado eficaces en la paralelización de programas con threads.

En general paralelizar un programa secuencial implica descomponer el programa en tareas y resolver correctamente las dependencias de datos entre las mismas. De este modo, la concurrencia optimista que ofrece TM puede ayudar en la paralelización de aplicaciones irregulares, donde un análisis estático no es suficiente para determinar la mayor parte de las dependencias. Las transacciones, definidas como secciones del programa secuencial, pueden ejecutarse concurrentemente, dejando a cargo del sistema TM la detección y resolución de los conflictos entre transacciones en tiempo de ejecución. Nótese sin embargo que, además de los conflictos de datos, podría requerirse una ordenación de las transacciones para

asegurar resultados correctos. Los sistemas TM convencionales no garantizan ningún orden de commit entre las transacciones.

En este trabajo se discute cómo utilizar un sistema TM para explotar patrones de reducción (operaciones asociativas y conmutativas) con accesos de memoria irregulares (no conocidos en compilación) y se compara esta alternativa con otras soluciones clásicas. Para ello se ha desarrollado ReduxSTM; un sistema STM que añade características específicas para este tipo de operaciones. ReduxSTM garantiza el commit de las transacciones en un orden equivalente al secuencial, y proporciona un tratamiento específico de las reducciones aprovechando la privatización subyacente al sistema STM. Con ello se pretende reducir el número de abortos derivados de las operaciones de reducción esperando trasladar este hecho a una mejora del rendimiento.

II. Patrones de reducción en aplicaciones irregulares

Una sentencia de reducción es un patrón de la forma $O = O \oplus \xi$, donde $\oplus$ es un operador asociativo y conmutativo aplicado a un objeto de memoria O (objeto o variable de reducción), y ξ es una expresión computada con objetos privados que no dependen de O. Se dice que un bucle es *completamente* de reducción (o reducción de histograma) si contiene sentencias de reducción y las únicas dependencias entre iteraciones son causadas por el objeto de reducción. Así mismo, el objeto de reducción no debe ser accedido por ninguna otra sentencia que no sea de reducción [4].

Las operaciones de reducción son parte habitual del núcleo de muchas aplicaciones computacionales como el álgebra de matrices dispersas, resolutores de ecuaciones diferenciales, etc. En estos casos el objeto de reducción (comúnmente un vector multidimensional) es accedido a través de índices indirectos, lo que confiere una naturaleza irregular a la aplicación.

Desde el punto de vista de las dependencias de datos, las sentencias de reducción causan dependencias entre iteraciones, ya que la variable de reducción es leída y a su vez escrita. No obstante, en los bucles completamente de reducción, las iteraciones se pueden reordenar sin problemas siempre que todas las reducciones sobre el objeto de reducción tengan el mismo operador, al ser éste conmutativo y asociativo.

No obstante puede haber situaciones donde la condición de bucle de reducción no se verifique completamente. Por ejemplo si se accede al objeto de reducción fuera de las sentencias de reducción, si se combinan varios operadores diferentes (aunque sean de reducción), o si ocurren otras dependencias entre las iteraciones debidas a otras variables no reductivas [5], [6]. A pesar de ello, puede que la condición de reducción se siga verificando para un subconjunto de iteraciones. Es lo que se conoce como reducciones parciales. Un ejemplo de esto se muestra en la figura 3.

A la hora de paralelizar bucles de reducción, podemos clasificar los métodos propuestos en la literatura en dos grandes grupos: (a) métodos que garantizan la exclusión mutua entre los accesos a los objetos de reducción [7], y (b) métodos que acumulan parcialmente el resultado en copias privadas de los objetos de reducción [8], [9], [10] y luego realizan una reducción global en el objeto de reducción original.

A. Exclusión mutua

Una forma obvia de resolver los conflictos causados por las sentencias de reducción es convertir el bucle secuencial completamente de reducción en un DOALL, encerrando las sentencias de reducción (o un grupo de ellas) en una sección crítica, de manera que sólo un thread accederá a las variables de reducción a la vez. El principal inconveniente de este enfoque es el alto grado de serialización, aunque puede mejorarse empleando locks de grado fino (*fine-grained locks*). Si el objeto de reducción es un vector, asociaríamos un lock por cada elemento del mismo, de modo que dos threads puedan acceden a elementos diferentes en paralelo, y sólo haya serialización en caso de producirse un conflicto real. Un enfoque similar se puede conseguir con operaciones atómicas, aunque su disponibilidad depende en gran medida de la arquitectura hardware.

En este grupo de técnicas podemos incluir modelos basados en tareas, como la cláusula *omp task depend* incluida recientemente en el estándar OpenMP u otras soluciones similares como las de [11] y [12]. En estos casos las variables de reducción se marcarían como una dependencia de entrada-salida de la tarea. La principal limitación de estos enfoques en códigos irregulares es su capacidad para expresar dependencias complejas, como las indirecciones, y también para expresar las dependencias cuando éstas se computan dentro de la tarea.

B. Privatización

Privatizar los objetos de reducción es una solución bastante eficaz y extendida a la hora de paralelizar bucles de reducción. En este caso se distribuye el espacio de reducción entre los threads, cada uno de los cuales opera sobre copias privadas de las variables de reducción. Estas copias deben inicializarse al elemento neutro del operador. Tras finalizar su trabajo, las reducciones parciales realizadas en las copias privadas han de reducirse de forma segura sobre los objetos de reducción originales. Dos técnicas representativas de este grupo son *Replicated Buffer* y *Array Expansion*, que se diferencian en cómo resuelven los conflictos en la reducción final. El principal inconveniente de la privatización es su gasto de memoria extra, ya que multiplicamos el tamaño de los objetos de reducción por el número de threads.

III. Enfoques TM en patrones de reducción

Al considerar la paralelización de un bucle de reducción completo, un enfoque directo mediante TM consiste en sustituir la sección crítica que abarca las

```
Read subscript arrays:
    edge(1,*), edge(2,*)

do itime=1,nTimes
 do i=1,nEdges
   n1 = edge(1,i)
   n2 = edge(2,i)

   Compute ζ1, ζ2, ζ3

   vel(1,n1)=vel(1,n1)+ζ1
   vel(2,n1)=vel(2,n1)+ζ2
   vel(3,n1)=vel(3,n1)+ζ3

   vel(1,n2)=vel(1,n2)-ζ1
   vel(2,n2)=vel(2,n2)-ζ2
   vel(3,n2)=vel(3,n2)-ζ3
 enddo
   .....
enddo
```

(a)

```
Compute subscript arrays:
     m(*),
     mbeg(*),
     mend(*)

do irow=1, nRows
  do i=1,jdt+1
   im =m(i)
   imb=mbeg(i)
   ime=mend(i)
   do is=imb,ime,2
     Compute ζ1, ζ2...
      do ilev=1,2*jdlev
       f1(ilev,im)=f1(ilev,im)+ζ1
       f2(ilev,im)=f2(ilev,im)+ζ2
       .....
      enddo
   enddo
   .....
enddo
```

(b)

```
do ihop=1, nHops
  Update subscripts: B1(*),B2(*)
  do itime=1,nTimes
    do ih=1,nParticles
      i=B1(ih)
      j=B2(ih)
      Compute r(i,j), ζ(i,j), η(i,j), θ(i,j)

      if (r .lt. CutOff) then
       AX(i)=AX(i) + ζ
       AX(j)=AX(j) - ζ
       AY(i)=AY(i) + ζ
       AY(j)=AY(j) - ζ
       U = U + η
       P = P + θ
      endif
    enddo
    .....
  enddo
enddo
```

(c)

Fig. 1

Algunos códigos con bucles de reducción: (a) Unstructured, (b) transformada de Legendre y (c) dinámica molecular 2D.

```
for (i=0; i<NInd; i++){
  Compute ξ1, ξ2
  #pragma omp critical{
    ...
    A[idx1[i]] ⊕= ξ1
    A[idx2[i]] ⊕= ξ2
    ...
  }
}
```

(a)

```
for (i=0; i<NInd; i++){
  Compute ξ1, ξ2
  BEGIN_XACT()
     ...
     TM_WRITE(A[idx1[i]]],
          TM_READ(A[idx1[i]] ⊕ ξ1))
     TM_WRITE(A[idx2[i]]],
          TM_READ(A[idx2[i]] ⊕ ξ1))
     ...
  END_XACT()
}
```

(b)

Fig. 2

Uso de transacciones (b) como reemplazo de una sección crítica (a).

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

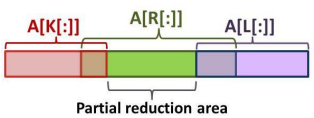

Fig. 3

Ejemplo de bucle con una zona de reducción parcial. Los subíndices, K, L and R, restringen el acceso a A tal como se muestra. Los accesos A[K[:]] and A[L[:]] no se solapan.

sentencias de reducción por una transacción, tal como se muestra en la figura 2. Esta solución es simple desde el punto de vista de la programabilidad, y los potenciales conflictos que surgen debido a las indirecciones son gestionados por el sistema TM. En caso de escenarios de baja contención, el sistema TM podrá mantener un buen nivel de paralelismo frente a la fuerte serialización que se produce en el caso de emplear secciones críticas (mutex, spinlocks, atomics, etc).

Si además imponemos cierto orden en el TM de forma que los commits de las transacciones tengan lugar de forma equivalente al orden de la ejecución secuencial, entonces podemos pensar en el sistema TM como una herramienta de apoyo a un enfoque TLS. Esto es especialmente interesante para aquellos casos en los que la paralelización de las reducciones no se puede abordar con soluciones clásicas porque las condiciones de bucle de reducción no se verifican completamente.

Un ejemplo de esta situación se muestra en la figura 3, donde hay conflictos potenciales entre accesos a un vector en una sentencia de reducción y accesos fuera de ella. Sin embargo puede existir un subconjunto de iteraciones que cumplan la condición de reducción si existen restricciones en los subíndices como las mostradas en la figura [5]. En la figura 4 podemos observar otro ejemplo donde existen lecturas y escrituras a través de punteros que pueden ser alias de las variables de reducción. Este hecho impide saber en tiempo de compilación si se trata de un bucle completamente de reducción y si, consecuentemente, las iteraciones pueden o no reordenarse con seguridad [6]. Estos patrones, a los que hemos denominado *reducciones parciales*, han sido tratados en la literatura por medio de enfoques especulativos, como se discute en la siguiente sección.

IV. Trabajos relacionados

La idea de resolver la paralelización de bucles de reducción especulativamente no es nueva. En [13] se propone el test LRPD que, tras una fase de inspección, es capaz de seleccionar aquellas iteraciones que se pueden lanzar especulativamente en paralelo mediante una estructura DOALL. Esta idea ha sido reformulada recientemente con Privateer [4]. Otro enfoque especulativo reciente para bucles de reducción lo encontramos en [6], centrado en la detección y ejecución especulativa de variables de reducción parcial (PRV). En [14] y [15] se explora la posibilidad de utilizar un sistema TM en bucles de reducción, deshabilitando la detección de conflictos y extendiendo el buffer de escritura para tal fin. Este enfoque requiere el conocimiento previo de que el bucle sea completamente de reducción y no es aplicable en el caso de reducciones parciales. Otras propuestas más

```
for (termptr = ... ; termptr = termptr->nextterm) {
    ...
    for (netptr = ... ; netptr=netptr->nterm) {
        ...
        *costptr += ...; // Reduction sentence
    }
    ...
    rowsptr = tmp_rows[net] ;
    for (row = 0 ; rowsptr[row] == 0 ; row++ ){
        ...
    }
    ...
    tmp_num_feeds[net] = f ;
    ...
    tmp_missing_rows[net] = -m ;
    ...
    delta_vert_cost += ( ... ); // Reduction sentence
}
```

Fig. 4

Esquema de un bucle de la función *new_dbox_a()* del código *300.twolf* (benchmark SPEC CPU2000). Los alias entre punteros hacen imposible saber si se trata de un bucle de reducción aunque contiene sentencias de reducción.

generales basadas en TM y que son aplicables a reducciones irregulares las encontramos en [16], [17], [18]. IPOT [16] es capaz de lanzar bloques de instrucciones en paralelo contenidas en estructuras similares a transacciones ordenadas, y permite relajar las restricciones de consistencia para mejorar el rendimiento. ALTER [17] propone un esquema TLS al estilo transaccional en el que las variables pueden ser anotadas permitiéndoles un chequeo de consistencia más permisivo. Una de estas anotaciones está pensada precisamente para las variables de reducción. En [18] se propone una técnica de paralelización automática basada en transacciones hardware ordenadas. Por último cabe destacar RMW (Read-Modify-Write without aborts [19]), un enfoque STM reciente que proporciona soporte específico para los patrones de lectura-modificación-escritura de los cuales las reducciones son un caso particular. Este enfoque general está limitado, no obstante, por su limitada escalabilidad, no contempla transacciones ordenadas y está pensado fundamentalmente para variables escalares, excluyendo vectores y estructuras multidimensionales que aparecen con frecuencia en códigos con reducciones.

V. ReduxSTM

Además de ser un sustituto directo de una sección crítica, las transacciones realizan una privatización selectiva de aquellas variables marcadas como transaccionales. La idea que se plantea es aplicar este mecanismo de privatización subyacente a las variables de reducción, permitiendo evitar aquellos abortos derivados de las sentencias de reducción siempre que sea posible. Las transacciones realizarán reducciones parciales sobre su versión local de la variable, realizándose la reducción global en la variable compartida durante la fase de commit. La asociatividad y conmutatividad del operador garantiza que esto pueda hacerse de forma segura. De esta manera el programador sólo tiene que marcar como transaccionales las operaciones de reducción (sentencias) sin tener conocimiento de si se trata de un bucle completamente de reducción o de una reducción parcial. Aquellas sentencias de reducción que tengan conflictos con otros accesos a memoria (escritura o lectura) serán detectadas y corregidas por el STM de forma transparente al programador.

TABLA I

Diferentes técnicas de paralelización de bucles completamente de reducción.

	Requerimiento de memoria extra	**Paralelismo potencial**	**Sobrecarga de sincronismo**	**Reducción final**
Privatización	muy alto	muy alto	muy bajo	sí
Sección crítica	muy bajo	muy bajo	alto	ninguno
Lock de grano fino	alto	alto	alto	ninguno
Ops. atómicas	ninguno	muy alto	alto	ninguno
TM (directo)	bajo	alto/medio	bajo	sí
ReduxSTM	bajo	muy alto	bajo	sí

Nuestra propuesta, ReduxSTM, soporta por tanto patrones *potenciales* de reducción, donde resulta complicado determinar estáticamente si son reducciones parciales o totales. ReduxSTM ha sido planteado como una prueba de concepto para comprobar si un sistema TM se puede beneficiar de un tratamiento específico de los patrones de reducción. ReduxSTM ha sido construido como una librería desde cero. Una comparativa de los diferentes enfoques discutidos se incluye en la tabla I.

A. Características

Soporte de reducciones. Una de las características clave de ReduxSTM es la capacidad de explotar la privatización selectiva asociada al sistema transaccional. Al explotar esta privatización implícita se consiguen eliminar abortos innecesarios, mejorando el grado de concurrencia.

Con este propósito se introduce una nueva primitiva para las operaciones de reducción, que el programador podrá usar junto con las ya existentes de lectura y escritura. Esta primitiva es tratada por los gestores de conflictos y de versiones como una tercera operación básica: lectura (R), escritura (W) y reducción (Rdx). Su semántica es una lectura, seguida de una escritura en la misma posición de memoria tras haber realizado la operación de reducción. La nueva primitiva $Rdx(add, val, \oplus)$ es semánticamente equivalente a $W(add, R(add) \oplus val)$ pero permite una ejecución más eficiente.

Transacciones ordenadas. Una segunda característica importante de ReduxSTM es el mantenimiento de una restricción de orden entre los commits de las transacciones. De esta manera podemos garantizar que la ejecución especulativa lleva al mismo resultado que la versión secuencial. Obsérvese que, aunque los bucles completamente de reducción pueden ser reordenados sin problemas, esto no es así en el caso de las reducciones parciales, donde las reducciones coexisten con lecturas y escrituras sobre las variables de reducción fuera de las sentencias de reducción.

TABLA II
CONFLICTOS TRANSACCIONALES POTENCIALES.

	STM Estándar	**ReduxSTM (Orden + Rdx.)**
R−W	aborto	no hay conflicto
W−R	aborto	aborto
W−W	aborto	no hay conflicto
Rdx−R	como R-W-R	aborto
R−Rdx	como R-R-W	no hay conflicto
Rdx−W	como R-W-W	no hay conflicto
W−Rdx	como W-R-W	no hay conflicto
Rdx−Rdx	como R-W-R-W	no hay conflicto

Esta restricción de orden en la fase de commit se convierte en el mecanismo básico de atomicidad de los commits, pero a su vez tiene un coste de rendimiento, ya que supone esperas no deseables en el turno de commit. No obstante, la oportunidad de mejora radica en el ahorro de abortos y rollbacks generados por las falsas dependencias asociadas a las reducciones [3], [15] como se muestra en la tabla II. La primera columna especifica dos operaciones realizadas por dos transacciones diferentes, donde la primera debe realizar la fase de commit antes de la segunda. Por ejemplo, R−W, estaría asociado a una anti-dependencia. La segunda columna corresponde al comportamiento de un TM convencional y la tercera a nuestra propuesta.

Gestión de versiones. Soportar reducciones (que pueden entrar en conflicto con lecturas y escrituras) junto con las transacciones ordenadas conlleva una gestión de versiones *lazy*, donde la consolidación de las posiciones de memoria se realiza al finalizar la transacción en la fase de commit. Para ello se introducen dos buffers privados que almacenan información disjunta: el *write buffer* y el *reduction buffer*. El primero ya existe en los sistemas transaccionales convencionales y el segundo es específico para las operaciones de reducción. Obsérvese que es necesario un *reduction buffer* por cada operación de reducción soportada (suma, producto, etc).

Cada vez que se ejecuta una reducción transaccional sobre una posición de memoria, se busca dicha posición en el *write buffer*. Si dicha posición está ahí, se reduce el valor especificado en la reducción con el valor almacenado en el buffer y se mantiene en el *write buffer* ya que se viola la condición de reducción. En caso contrario, el valor de reducción se opera con el valor correspondiente del *reduction buffer* (o con el elemento neutro del operador si es la primera operación sobre esta posición), actualizando dicha posición en el *reduction buffer* con el resultado de la operación (reducción parcial). Si una posición almacenada en el *reduction buffer* es escrita con posterioridad en la misma transacción, debe ser eliminada de este buffer e insertada en el *write buffer* ya que deja de cumplirse en este momento la condición de reducción. Obsérvese que una lectura de una posición que está marcada como reducción en una transacción implica combinar el valor de memoria con el acumulado parcialmente en el buffer de reducción.

Detección de conflictos. En ReduxSTM la validación/invalidación de las transacciones se realiza durante la fase de commit, por lo que la detección de conflictos se considera *lazy* [20].

Gestión de commits. Puesto que sólo una transacción puede estar en fase de commit para garantizar el orden, tal transacción es la responsable de comprobar y resolver posibles conflictos con otras transacciones activas. El orden de finalización garantiza la naturaleza atómica de la fase de commit, y por tanto actúa como el principal mecanismo de sincronización. La fase de commit está sujeta a las características particulares de la estrategia de implementación, tal como se discute en la sección V-B. Independientemente de la implementación, la fase de commit de una transacción debe (1) Esperar su turno de commit; (2) Comprobar y resolver posibles conflictos con otras transacciones; y (3) Consolidar (actualizar) la memoria principal con los valores almacenados en los buffers de reducción y escritura (los valores de reducción necesitarán ser acumulados según su operador asociado).

B. Implementación

Hemos seleccionado dos algoritmos STM bien conocidos, *Commit Time Invalidation* y *Time-Based Validation*, como base de nuestras implementaciones de ReduxSTM. La elección de estos algoritmos radica en que son adecuados para implementar de forma efectiva las características descritas anteriormente. Ambas implementaciones fueron codificadas desde cero.

Commit Time Invalidation (CTI)

En esta implementación la transacción que realiza el commit marcará como invalidadas (a abortar) aquellas transacciones activas que tengan algún conflicto con ella [21]. Los conflictos de datos se detectan a partir de las direcciones de memoria. Al comenzar su fase de commit, y tras esperar su turno, cada transacción comprueba si es o no válida, abortando en caso negativo. Si es válida, consolida (commit) los datos transaccionales en la memoria, tras lo cual comprueba posibles conflictos con las transacciones en ejecución, invalidándolas si sus conjuntos de datos de lectura no son disjuntos con los conjuntos de escritura y reducción de la transacción en el commit (ver tabla II). Nuestra implementación usa filtros de Bloom para representar cada uno de los conjuntos de datos (lectura, escritura y reducción).

Time-based validation (TS)

A diferencia de la estrategia anterior, ésta emplea marcas de tiempo (timestamps) para registrar *cuándo* tienen lugar las lecturas y actualizaciones [22]. Nuestra implementación usa como reloj global de estas marcas el orden global de la última transacción finalizada. Cada transacción mantiene una tabla privada para las marcas de tiempo de sus lecturas. Así mismo se mantiene una tabla global de marcas de tiempo para las escrituras y reducciones consolidadas en memoria. En la fase de commit se comprueba si hay lecturas cuya marca de tiempo sea posterior a la de la escritura consolidada correspon-

TABLA III
CÓDIGOS TESTEADOS

Fluidanimate	Forma parte de la suite PARSEC [23]. Esta aplicación simula fluidos usados en animaciones de tiempo real. Se comprobaron dos configuraciones: una de 100K partículas durante 5 fotogramas y otra de 500K partículas durante 500 fotogramas.
MD2	Esta aplicación [24] corresponde a una simulación de dinámica molecular 2D en sistemas con un número elevado de partículas con interacciones de corto alcance. Para limitar la complejidad del problema las interacciones se acotan a partículas cercanas mediante una lista de partículas vecinas.
Unstructured	Este código [25] resuelve las ecuaciones de Euler en simulaciones físicas. En cada paso de tiempo la aplicación computa fuerzas y velocidades en los nodos de una malla.
Legendre	Corresponde al núcleo de la transformada de Legendre [26] usada en predicción meteorológica. En cada paso de tiempo se invocan la transformada directa e inversa que llevan asociadas reducciones irregulares debido a los accesos a través de indirecciones.
300.twolf	Es un simulador de *place and route* incluido en el benchmark SPEC 2000 [27]. Contiene patrones de reducción interesantes en la rutina `new_dbox_a()` (`dimbox.c`), donde pueden aparecer conflictos potenciales entre variables de reducción y de no reducción debido a los alias entre punteros.

diente, en cuyo caso la transacción debe abortar. Para reducir la memoria requerida por las marcas de tiempo, las tablas están limitadas en tamaño y son accedidas por un hash de la dirección de memoria, lo que implica cierta probabilidad de falsos positivos.

VI. Evaluación experimental

En esta sección se evalúa experimentalmente cómo se comportan las técnicas STM en códigos dominados por operaciones de reducción, en especial cuando se incluye soporte para reducciones. Los experimentos se realizaron en un servidor con 256 GB de RAM y 16 cores (32 threads) Intel Xeon E5-2698 a 2.3GHz, con sistema operativo Linux kernel 3.13 (64 bits). Los programas se compilaron con GNU GCC 4.8.2 con opción de optimización `-O2`.

Como sistema STM de referencia a efectos de comparación se ha empleado TinySTM (v.1.0.5) [22], un STM que puede considerarse representativo del estado del arte actual. Se emplearon tanto la versión base como la versión ordenada de TinySTM.

En la evaluación se han seleccionado varios códigos representativos que se describen brevemente en la tabla III. Fluidanimate, MD2, Unstructured y Legendre contienen bucles completamente de reducción. Por su parte, el código 300.twolf incluye patrones de reducción parcial.

Las técnicas de paralelización que se han comparado son las siguientes:

- *Locks de grano grueso* (CG Locks), que se corresponde con el uso de locks para proteger secciones críticas;
- *Locks de grano fino* (FG Locks), donde se usa un lock individual para proteger cada elemento del array de reducción compartido;
- Privatización completa de los arrays de reducción, implementada como *Array Expansion* [9];
- TinySTM (configuración por defecto) usado como referencia;
- TinySTM-ordered, la versión de TinySTM con commits ordenados;
- ReduxSTM en sus dos implementaciones: basada en marcas de tiempo (ReduxSTM-TS) y con invalidación en fase de commit (ReduxSTM-CTI).

En todos los sistemas STM la paralelización de los bucles de reducción se llevó a cabo descomponiendo los bucles en bloques de iteraciones consecutivas (chunks) y ejecutando cada bloque dentro de una transacción. Obsérvese que el número de iteraciones en cada bloque determina el tamaño de la transacción y es un parámetro relevante, que debe ser elegido cuidadosamente: transacciones muy grandes reducen la carga extra introducida por la instrumentación del sistema transaccional, pero implican mayores conjuntos de datos y por tanto mayor probabilidad de conflicto y de abortos. Las transacciones pequeñas tienen menor probabilidad de conflicto, pero implican un mayor coste de instrumentación debido al lanzamiento y cierre de las transacciones (por ejemplo de la inicialización de las estructuras de datos).

En la privatización y las técnicas basadas en locks, los bucles se han particionado equitativamente entre los threads, sin agrupar las iteraciones en chunks. Recuérdese que la privatización requiere una fase de inicialización y otra de reducción final.

Los locks se han implementado con *spinlocks* de POSIX. Adicionalmente, para los locks de grano fino, se ha optimizado el código mediante el uso de un mismo lock para aquellos bloques de sentencias de reducción cuyos accesos a los arrays de reducción están indexados por el mismo índice. Esto reduce el número de locks necesario y el número de pares lock/unlock ejecutados, lo que se traduce en una menor sobrecarga.

A. Comparación de rendimiento

A continuación se ofrece una comparativa de las diferentes técnicas en términos de rendimiento. El speedup observado ha sido calculado con respecto a la versión secuencial no instrumentada de los códigos, usando el menor tiempo de ejecución de entre al menos diez ejecuciones.

En los experimentos, las variables independientes consideradas son el número de threads y el tamaño de las transacciones (iteraciones por bloque) en los sistemas STM. En este caso, los speedups mostrados corresponden al mejor chunk, esto es, al tamaño de transacción que proporciona un speedup más alto. Las gráficas que muestran el speedup en función del tamaño de transacción muestran ejecuciones con 16 threads.

La figura 5 muestra los resultados para Fluidanimate usando dos conjuntos de datos de 100K y 500K partículas. Como se espera, los locks de grano grueso no aceleran en absoluto debido al uso de un sólo lock global. Este método podría tener sentido en aplicaciones en las que el tiempo de ejecución en sección

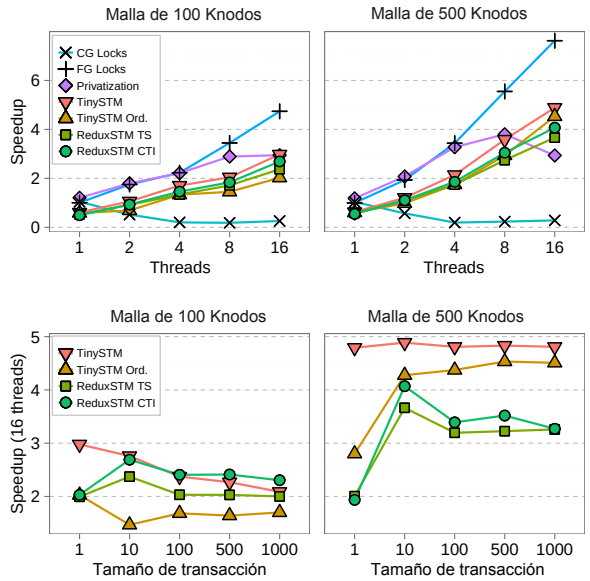

Fig. 5

FLUIDANIMATE: SPEEDUP PARA EL MEJOR TAMAÑO DE TRANSACCIÓN EN CADA CASO, E INFLUENCIA DEL TAMAÑO DE TRANSACCIÓN PARA 16 THREADS.

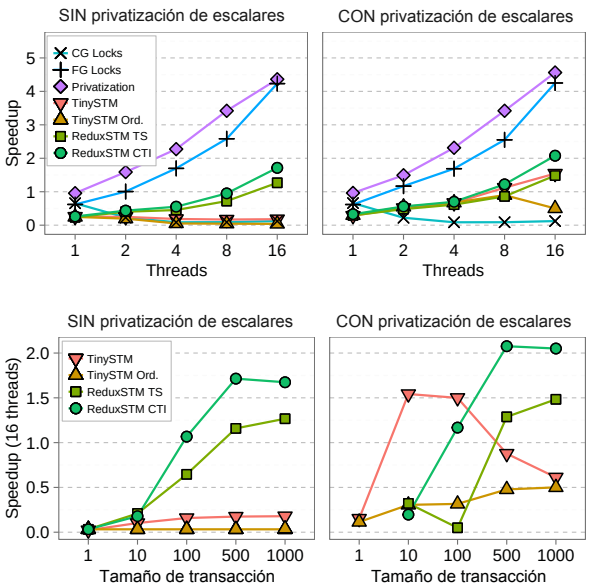

Fig. 6

MD2: SPEEDUP PARA EL MEJOR TAMAÑO DE TRANSACCIÓN EN CADA CASO, E INFLUENCIA DEL TAMAÑO DE TRANSACCIÓN PARA 16 THREADS.

crítica fuera despreciable con respecto al resto, pero no es el caso en este problema. Los mejores resultados se obtienen con locks de grano fino o privatización, debido principalmente a la baja contención del problema. Obsérvese que, para la malla más grande, el rendimiento de la privatización deja de escalar para un número de threads alto. Esto es consecuencia de las características NUMA de la arquitectura y la gran cantidad de memoria extra que necesita este método. Por su parte, todos los sistemas STM presentan un comportamiento similar, debido a que la tasa de abortos se mantiene baja. Esta situación no le da gran margen de ventaja a ReduxSTM. En este escenario los enfoques STM obtienen un speedup moderado sin grandes requerimientos de memoria. Con respecto la influencia del tamaño de transacción, el comportamiento de los sistemas STM es muy dependiente de las características de la entrada. De esta manera, para la malla más pequeña la penalización de las transacciones grandes es más significativa que en el caso de la malla mayor, para la cual las transacciones más pequeñas implican una alta penalización.

MD2 incluye reducciones sobre variables escalares y sobre vectores (ver figura 1(c)). En este código se han probado dos estrategias, que se muestran en la figura 6. En la primera, todas las variables de reducción (escalares y vectoriales) han sido tratadas transaccionalmente. En la segunda, las variables escalares fueron privatizadas, de manera que el número de sentencias de reducción sobre objetos compartidos se reduce de 6 a 4 por iteración. Esta optimización es bastante común en este tipo de códigos [28], pero requiere un conocimiento mayor por parte del programador/compilador. Sin la privatización de escalares, la elevada contención causada por los escalares hacen

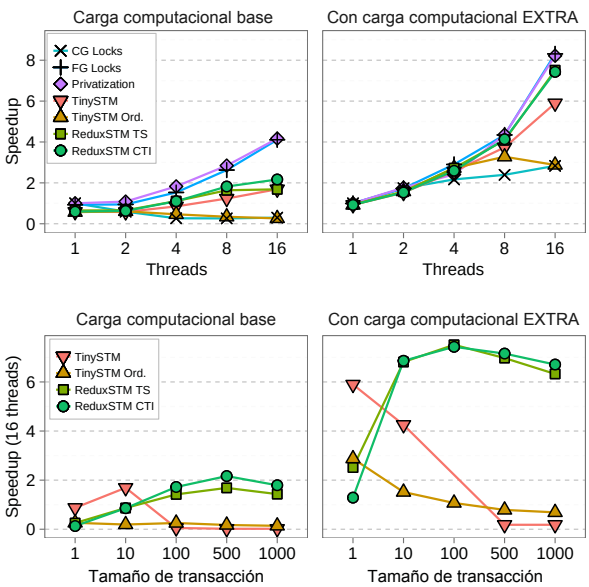

Fig. 7

UNSTRUCTURED: SPEEDUP PARA EL MEJOR TAMAÑO DE TRANSACCIÓN EN CADA CASO, E INFLUENCIA DEL TAMAÑO DE TRANSACCIÓN PARA 16 THREADS.

que TinySTM tenga una tasa de abortos muy alta, y por tanto un peor rendimiento. Como se observa, es precisamente en estos casos de alta contención en las variables de reducción donde ReduxSTM obtiene una ventaja en el rendimiento. En cualquier caso, el pequeño tamaño de los arrays de reducción hacen que la privatización obtenga los mejores resultados, sobrepasando a los locks de grado fino.

La figura 7 presenta los resultados de Unstructu-

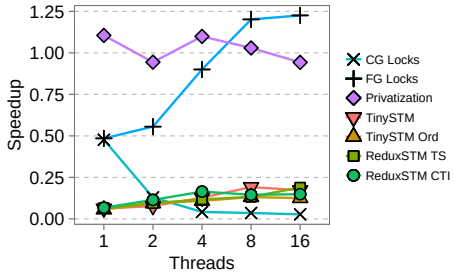

Fig. 8
Legendre: speedup (considerando una iteración por transacción).

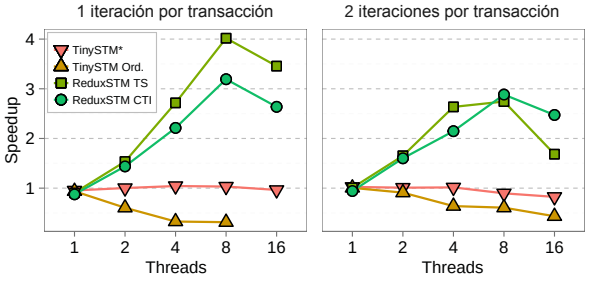

Fig. 9
300-Twolf: speedup para transacciones de una y dos iteraciones. Sólo los STM ordenados garantizan el resultado correcto. TinySTM (no ordenado) se muestra como una referncia optimista.

red. En estos experimentos, el soporte para reducciones de ReduxSTM le permite tener ventaja con respecto a TinySTM. El rendimiento de TinySTM se degrada rápidamente por los conflictos. Por contra, ReduxSTM se beneficia de su filtrado de conflictos para poder conseguir mejores resultados con transacciones más grandes. Como en los casos anteriores, los locks de grano fino y la privatización siguen teniendo los mejores speedups. No obstante también se ha testeado el código incorporando una carga computacional sintética adicional. En casos con mayor intensidad computacional, ReduxSTM es capaz de alcanzar el speedup de la privatización.

La figura 8 recoge los resultados para Legendre. En este caso no se han analizado diferentes tamaños de transacción, dado que el bucle exterior paralelizado no contiene un número elevado de iteraciones. En este código ninguna de las técnicas consigue un buen rendimiento. La baja intensidad computacional del mismo (es un problema *memory-bound*) hace que cualquier instrumentación adicional deteriore el rendimiento.

La figura 9 muestra los resultados obtenidos con 300.twolf. Los resultados están referidos a la rutina `new_dbox_a()`. Esta función presenta un patrón de acceso a memoria con alta contención, y la cantidad de paralelismo explotable está limitada por la baja intensidad computacional del código. Sólo se han considerado chunks de una y dos iteraciones por transacción, ya que transacciones mayores degradan el rendimiento. Es importante destacar que en este problema sólo son aplicables los métodos que garantizan el orden de la ejecución secuencial original, ya que las operaciones de reducción coexisten con otras operaciones de lectura y escritura potencialmente conflictivas. No obstante, aunque TinySTM no ordenado no cumple esta condición, se ha incluido a modo de referencia optimista. Nótese que ReduxSTM presenta un rendimiento significativamente mejor para todas las configuraciones. Cabe mencionar que, a pesar de que es difícil de explotar paralelismo en este benchmark, ReduxSTM es capaz de obtener aceleración hasta un número relativamente alto de threads aunque, para la carga computacional analizada, a partir de 8 threads los conflictos empeoran el rendimiento.

VII. Conclusiones

Es conocido que muchas aplicaciones con patrones irregulares de acceso de memoria son difíciles de paralelizar. En este contexto, la concurrencia optimista que proporcionan los sistemas de memoria transaccional (TM) puede resultar útil para extraer paralelismo. En este trabajo se ha presentado ReduxSTM, un sistema TM software con soporte específico para operaciones de reducción, un patrón común en muchas aplicaciones irregulares. Las características clave de ReduxSTM son que los commits de las transacciones se realizan en el orden equivalente al secuencial y que se utiliza la privatización subyacente al TM para filtrar los conflictos derivados de las operaciones de reducción cuando es posible, disminuyendo así el número de potenciales conflictos y, por tanto, de abortos. Comparado con técnicas clásicas de paralelización de bucles de reducción, se ha comprobado que los enfoques STM son un buen compromiso entre facilidad de programación, paralelismo explotado y sobrecarga de memoria extra necesaria; y que es posible mejorar el rendimiento de los STM si se añade soporte específico para reducciones, especialmente en patrones con una alta contención.

Agradecimientos

Este trabajo ha recibido soporte del Gobierno de España (proyecto TIN2013-42253-P), de la Junta de Andalucía (proyecto P12-TIC-1470) y de la Universidad de Málaga, campus de excelencia internacional Andalucía Tech.

Referencias

[1] 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'93)*, San Diego, CA, USA, 1993, pp. 289–300.

[2] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Ed*, 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 Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, NC, USA, 2009.

[4] N.P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D.I. August, "Speculative separation for privatization and reduc-

tions," in *33rd ACM Conf. on Programming Language Design and Implementation (PLDI'12)*, Beijing, China, 2012, pp. 359–370.

[5] Lawrence Rauchwerger, "Speculative parallelization of loops," in *Encyclopedia of Parallel Computing*, pp. 1901–1912. Springer, 2011.

[6] L. Han, W. Liu, and J.M. Tuck, "Speculative parallelization of partial reduction variables," in *8th Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10)*, Toronto, Canada, 2010, pp. 141–150.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence, "Parallel programming with Polaris," *IEEE Computer*, vol. 29, no. 12, pp. 78–82, 1996.

[8] 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*, vol. 29, no. 12, pp. 84–89, 1996.

[9] P. Feautrier, "Array expansion," in *2nd Int'l Conf. on Supercomputing (ICS'88)*, Saint Malo, France, 1988, pp. 429–441.

[10] H. Yu and L. Rauchwerger, "An adaptive algorithm selection framework for reduction parallelization," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1084–1096, 2006.

[11] Josep M Perez, Rosa M Badia, and Jesus Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *10th IEEE Int'l Conf. on Cluster Computing*, 2008, pp. 142–151.

[12] Carlos H González and Basilio B Fraguela, "A framework for argument-based task synchronization with automatic detection of dependencies," *Parallel Computing*, vol. 39, no. 9, pp. 475–489, 2013.

[13] Lawrence Rauchwerger and David A Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans, on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 160–180, 1999.

[14] M. Gonzalez-Mesa, R. Quislant, E. Gutierrez, and O. Plata, "Dealing with reduction operations using transactional memory," in *25th Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'13)*, Porto de Galinhas, Brasil, 2013, pp. 128–135.

[15] 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, pp. 24:1–24:27, 2014.

[16] C. von Praun, C. Ceze, and C. Cascaval, "Implicit parallelism with ordered transactions," in *12th ACM Symp. on Principles and Practice of Parallel Programming (PLDI'07)*, San Diego, CA, USA, 2007, pp. 79–89.

[17] A. Udupa, K. Rajan, and W. Thies, "ALTER: Exploiting breakable dependences for parallelization," *32nd ACM Conf. on Programming Language Design and Implementation (PLDI'11)*, pp. 480–491, 2011.

[18] M. DeVuyst, D.M. Tullsen, and S.W. 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)*, Heraklion, Greece, 2011, pp. 127–136.

[19] Wenjia Ruan, Yujie Liu, and Michael Spear, "Transactional Read-Modify-Write without aborts," *ACM Trans. on Architecture and Code Optimization*, vol. 11, no. 4, pp. 1–24, 2015.

[20] Ricardo Quislant, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata, "Conflict detection in hardware transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pp. 127–149. Springer, 2015.

[21] J.E. Gottschlich, M. Vachharajani, and J.G. Siek, "An efficient software transactional memory using commit-time invalidation," in *8th Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10)*, Toronto, Canada, 2010, pp. 101–110.

[22] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, "Time-based software transactional memory," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[23] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, Toronto, Canada, 2008, pp. 72–81.

[24] J.J. Morales and S. Toxvaerd, "The Cell-Neighbour table method in molecular dynamics simulations," *Computer Physics Communications*, vol. 71, pp. 71–76, 1992.

[25] Ian Foster, Rob Schreiber, and Paul Havlak, "HPF-2, scope of activities and motivating applications," Tech. Rep., Technical Report CRPC-TR94492, Rice University, 1994.

[26] N. Mukherjee and J.R. Gurd, "A comparative analysis of four parallelisation schemes," in *13th Int'l. Conf. on Supercomputing (ICS'99)*, Portland, OR, USA, 1999, pp. 278–285.

[27] Manohar K. Prabhu and Kunle Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *10th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'05)*, Chicago, IL, USA, 2005, p. 142.

[28] 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.