Barreras Especulativas con Memoria Transaccional

Manuel Pedrero, Ricardo Quislant, Eladio Gutiérrez, Emilio L. Zapata, and Óscar Plata [1]

Resumen— **La Memoria Transaccional (TM) es una alternativa al modelo de programación basado en locks que pretende simplificar la programación paralela. TM sustituye locks por transacciones para resolver el problema de la exclusión mutua. Las transacciones se ejecutan de manera optimista, en paralelo, mientras el sistema transaccional comprueba si hay conflictos entre ellas. En este trabajo proponemos el uso de transacciones para implementar una barrera especulativa (SB) optimista que reemplace las barreras pesimistas basadas en locks. SBs aprovecha el soporte TM hardware para permitir que los hilos salten la barrera y ejecuten especulativamente. Cuando un hilo llega a una barrera abre una transacción para proteger la ejecución y poder volver a la barrera en caso de conflicto. Cuando el último hilo alcanza la barrera los hilos especulativos pueden acometer los cambios.**

Las contribuciones de este trabajo son: una API para las SBs implementada con extensiones TM de un sistema real (IBM Power8); un procedimiento para comprobar el estado especulativo de los hilos entre barreras para usar en códigos no transaccionales; un conjunto de directrices para el uso de las SBs. Los resultados muestran un incremento en el rendimiento de hasta 6× sobre las barreras tradicionales para algunas configuraciones de las aplicaciones evaluadas.

Palabras clave— **Barreras Especulativas, Memoria Transaccional, Memoria Compartida, IBM Power8.**

I. Introducción

La Memoria Transaccional (TM) [1] trata de simplificar la programación paralela con el uso de transacciones en lugar de locks. Con locks, una sección crítica se resuelve haciendo esperar a los hilos mientras sólo uno de ellos la ejecuta. Con TM se apuesta por una solución optimista en la que todos los hilos ejecutan la sección crítica en paralelo y sólo se serializa la ejecución si hay algún conflicto. Las transacciones que delimitan un código proporcionan atomicidad y aislamiento en la ejecución de ese código. El sistema transaccional ejecuta las transacciones especulativamente y mantiene un registro de las posiciones de memoria accedidas para detectar y resolver conflictos. En los último años se han propuesto varios sistemas TM hardware (HTM) [2], [3], [4] y los principales fabricantes de chips han añadido extensiones HTM a sus arquitecturas [5], [6], [7]. Estas extensiones se llaman *best-effort* ya que no ofrecen garantías para la finalización de las transacciones, proporcionando un mecanismo de *fallback* [8] para este caso.

En este trabajo se proponen barreras especulativas (SB) optimistas basadas en transacciones para sustituir el sistema pesimista tradicional basado en locks. Cuando un hilo llega a una SB se inicia una transacción que mantiene la ejecución aislada del sistema mientras este comprueba si hay conflictos con otros hilos. Una vez que el último hilo ha llegado a la SB, los hilos especulativos tratan de acometer los cambios realizados. Las SBs implementan un orden parcial entre las transacciones previas a la SB y las transacciones especulativas donde las últimas tienen que esperar a que finalicen las primeras para realizar el commit. Aquí se propone una implementación de las SBs con HTM. Para ello se requiere una característica indispensable del sistema HTM, que es el soporte para acciones escapadas [9]. Las acciones escapadas permiten el acceso no transaccional a memoria desde dentro de una transacción rompiendo así el aislamiento y la atomicidad de la transacción. Su utilización permite mantener la comunicación entre transacciones SB sin que se produzcan abortos.

Este trabajo hace las siguientes contribuciones sobre un trabajo preliminar [10]:

- Proporcionamos un API para las SBs implementado con extensiones HTM. El API incluye wrappers para las instrucciones de comienzo y commit de transacción, así como para el reemplazo de las barreras tradicionales por SBs. Se proporciona un procedimiento para la última barrera que termina la especulación de barreras previas y no empieza nuevas transacciones SB.
- Proponemos un procedimiento para comprobar el estado especulativo entre barreras para ser usado en códigos sin transacciones. Este procedimiento también se puede utilizar para incrementar el número de comprobaciones en códigos transaccionales.
- Proporcionamos un conjunto de directrices para el uso de SBs derivadas de nuestra experiencia usando SBs en una variedad de aplicaciones intensivas en el uso de barreras.
- Evaluamos nuestra propuesta en una máquina real IBM Power8 server.

Los resultados muestran una mejora general en el rendimiento de las SBs sobre las barreras tradicionales. La aceleración de los códigos crece con el número de hilos, especialmente debido a los dos sockets de la máquina Power8 cuando los hilos de las aplicaciones se comunican entre sockets. Las SBs con el procedimiento de comprobación de especulación mejoran el rendimiento sobre la aplicación paralela desde 2× a 6× a partir de 16 hilos y hasta 128.

II. Barreras Especulativas

A. Motivación

Las aplicaciones que hacen un uso intensivo de barreras [11] pueden mostrar un bajo rendimiento de-

[1]Departamento de Arquitectura de Computadores, Universidad de Málaga, España, 29071. E-mail: {mpedrero,quislant,eladio,zapata,oplata}@uma.es

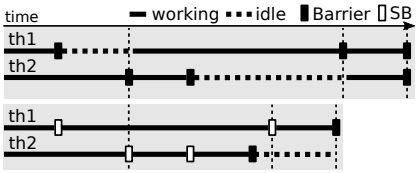

Fig. 1. Barreras tradicionales (arriba) versus SBs (abajo).

bido a (i) el desbalanceo de barrera, donde los hilos más rápidos tienen que esperar a los más lentos para continuar su ejecución; y (ii) la contención en la comunicación de barrera, especialmente en procesadores multi-chip. Las SBs están pensadas para que los hilos más rápidos especulen al llegar a la barrera en lugar de esperar. De este modo, el desbalanceo de barrera se puede utilizar para ocultar la contención de barrera. La Fig. 1 muestra un escenario donde el desbalanceo causa un retraso con las barreras tradicionales (arriba) mientras que las SBs (abajo) dan un rendimiento mejor incluso cuando el hilo más rápido tiene que esperar en la última barrera. La Fig. 1 asume que no hay conflictos de memoria entre los dos hilos. En presencia de conflictos los hilos deben sincronizarse para solucionarlos. SBs utiliza HTM para este propósito.

B. Descripción

Las SBs están ideadas para su uso con códigos transaccional donde una transacción se puede encontrar entre barreras. En esencia, las SBs hacen que la barrera se elimine y en su lugar se abra una transacción (la transacción SB). A continuación, si se encuentra una transacción en el código después de la barrera esta se ignora. La instrucción de comienzo de transacción no se ejecuta y la instrucción de commit se sustituye por una comprobación de la especulación. La especulación continuará hasta que (i) se detecta un conflicto, en cuyo caso se abortará la transacción SB, (ii) se llegue al límite de especulación, que se establece debido al número limitado de recursos del sistema HTM, y por lo tanto el hilo ha de esperar a los demás hilos, o (iii) todos los hilos hayan cruzado la SB, y la transacción SB acometerá los cambios realizados.

El API que implementa las SBs se muestra en la Fig. 2. El procedimiento SPEC_BARRIER está indicado para sustituir a las barreras tradicionales; LAST_BARRIER es similar a SPEC_BARRIER pero bloquea cualquier especulación subsiguiente. Se usa en casos para los que estamos seguros de que la especulación no tendrá éxito. TX_START y TX_STOP reemplazan a las llamadas de HTM para empezar y terminar una transacción, e incluyen código para integrarlas con las SBs. El orden parcial entre transacciones se implementa por medio de la variable *sbOrder* en la línea 2, que mantiene el orden global de la ejecución. Dicho orden se incrementa siempre que el último hilo alcanza una SB (línea 68), permitiendo que los hilos especulativos puedan acometer sus cambios. Cada hilo mantiene un orden local en la variable *order* (línea 4) que se incrementa siempre que el hilo pasa una SB. De esta manera el hilo se considera especulativo si su orden local es mayor que *sbOrder*, y puede hacer commit en caso contrario. La bandera local *spec* (línea 6) representa si el hilo está en estado especulativo.

SPEC_BARRIER: Este procedimiento implementa una SB (línea 54). Cualquier hilo no especulativo que llega a una SB incrementa su *order* local (línea 65). El último hilo incrementa *sbOrder* para generar un nuevo orden global y no inicia una transacción SB. Sin embargo, los otros hilos, que quedarían bloqueados en una barrera tradicional, continúan la ejecución abriendo una transacción SB (línea 83). Un aborto de esta transacción vuelve al hilo a la línea 70, donde el hilo comprueba si la especulación debe terminar (líneas 71 a 74), si no, el nivel de especulación se reajusta (líneas 76 to 81). especulación se limita a una SB por hilo, por lo que si se llega a otra SB el hilo espera a que su orden local sea igual a *sbOrder*. Esto se hace en una espera activa escapada para evitar abortos debido al acceso a *sbOrder* (líneas 56 to 59). Tras esto el hilo puede acometer sus cambios.

LAST_BARRIER: este procedimiento es similar a SPEC_BARRIER, pero no permite especulación tras su ejecución. Se debe usar como última barrera del hilo o cuando lo que hay después es alguna instrucción no compatible con transacción como reserva de memoria, llamadas al sistema, etc.

TX_START: este procedimiento reemplaza a la primitiva HTM que inicia una transacción (*xBegin()*), y añade soporte para SBs. Primero comprueba si el hilo ya está especulando, en cuyo caso no se hace nada puesto que ya está dentro de transacción. Si no está especulando, se sigue el procedimiento normal para abrir una transacción. Aquí se ha utilizado un fallback con eager subscription [8] que consta de un lock global que se ejecuta tras MAX_RETRIES abortos.

TX_STOP: este procedimiento reemplaza a la primitiva HTM que hace el commit de una transacción (*xCommit()*). Si el hilo no estaba especulando (líneas 22 a 30) se ejecuta el procedimiento estándar para hacer commit y se resetean las variables del descriptor. En el caso de estar especulando se comprueba si el hilo puede hacer commit (línea 32). La comprobación del orden (línea 31) debe estar escapada para que no aborte la transacción por una actualización de *sbOrder*. En el caso de que no se pueda hacer commit, la transacción SB continúa hasta que se llegue al máximo nivel de especulación, en cuyo caso queda bloqueada.

La Fig. 3 muestra un escenario con SBs y dos hilos. Las líneas punteadas representan el orden global. Primero, th1 y th2 ejecutan transacciones (en tx #, # es el orden local) antes de la primera barrera. La bandera *spec* está a 0 por lo que las transacciones empiezan y acaban con normalidad. A continuación th1 alcanza una SB, incrementa su orden local, decrementa *sbBarrier* y comprueba que no es el último hilo en cruzar la barrera. Por lo tanto, th1 empieza una transacción SB. Para cuando th2 alcanza la barrera, th1 ya ha ejecutado una transacción, que no se

```
1:  global sbBarrier ← #TH                          ▷ Barrier counter
2:  global sbOrder ← 1                              ▷ Global order for barriers
3:  thread local tx {                               ▷ Transaction descriptor
4:    order ← 1                                     ▷ Local order for the transaction
5:    retries ← 0                                   ▷ Retry count for the transaction
6:    spec ← 0                                      ▷ Flag to signal SB mode
7:    specMax ← MAX_SPEC                            ▷ Maximum spec level
8:    specLevel ← MAX_SPEC                          ▷ Current spec level
9:  }
10:
11: procedure TX_START(tx)
12:   if (tx.spec = 0) then                         ▷ Not executing a SB
13:     tx.retries ← tx.retries + 1
14:     if (tx.retries > MAX_RETRIES) then
15:       fallbackBegin()                           ▷ Acquire fallback lock
16:     else
17:       xBegin()                                  ▷ Go to line 13 on abort
18:     end if
19:   end if      ▷ If the thread is in SB mode, do nothing
20: end procedure
21: procedure TX_STOP(tx)
22:   if (tx.spec = 0) then                         ▷ Not in SB mode
23:     if (tx.retries ≤ MAX_RETRIES) then
24:       xCommit()
25:     else
26:       fallbackEnd()                             ▷ Release fallback lock
27:     end if
28:     tx.retries ← 0                              ▷ Reset tx descriptor
29:     tx.specLevel ← tx.specMax
30:   else                                          ▷ In SB mode
31:     beginEscape()
32:     if (tx.order ≤ sbOrder) then
33:       escapeEnd()
34:       xCommit()
35:       tx.retries ← 0                            ▷ Reset tx descriptor
36:       tx.specLevel ← tx.specMax
37:       tx.spec ← 0
38:     else
39:       escapeEnd()
40:       tx.specLevel ← tx.specLevel - 1;
41:       if (tx.specLevel = 0) then
42:         escapeBegin()
43:         while (tx.order > sbOrder) do
44:         end while                               ▷ Busy waiting
45:         escapeEnd()
46:         xCommit()
47:         tx.retries ← 0                          ▷ Reset tx descriptor
48:         tx.specLevel ← tx.specMax
49:         tx.spec ← 0
50:       end if
51:     end if
52:   end if
53: end procedure
54: procedure SPEC_BARRIER(tx)
55:   if (tx.spec = 1) then                         ▷ Executing a SB
56:     escapeBegin()
57:     while (tx.order > sbOrder) do
58:     end while                                   ▷ Busy waiting
59:     escapeEnd()
60:     xCommit()
61:     tx.retries ← 0                              ▷ Reset tx descriptor
62:     tx.specLevel ← tx.specMax
63:     tx.spec ← 0
64:   end if
65:   tx.order ← tx.order + 1;
66:   if ((sbBarrier ← sbBarrier - 1) = 0) then     ▷ Atomic
67:     sbBarrier ← #TH
68:     sbOrder ← sbOrder + 1                       ▷ Atomic
69:   else
70:     tx.retries ← tx.retries + 1
71:     if (tx.order ≤ sbOrder) then ▷ Do not begin a SB
72:       tx.retries ← 0                            ▷ Reset tx descriptor
73:       tx.specLevel ← tx.specMax
74:       tx.spec ← 0
75:     else
76:       if (tx.retries > MAX_RETRIES) then
77:         if tx.specMax > 1 then
78:           tx.specMax ← tx.specMax - 1
79:         end if
80:         tx.specLevel ← tx.specMax
81:       end if
82:       tx.spec ← 1
83:       xBegin()                                  ▷ Go to line 70 on abort
84:     end if
85:   end if
86: end procedure
87: procedure LAST_BARRIER(tx)
88:   if (tx.spec = 1) then                         ▷ Executing a SB
89:     escapeBegin()
90:     while (tx.order > sbOrder) do
91:     end while                                   ▷ Busy waiting
92:     escapeEnd()
93:     xCommit()
94:     tx.retries ← 0                              ▷ Reset tx descriptor
95:     tx.specLevel ← tx.specMax
96:     tx.spec ← 0
97:   end if
98:   tx.order ← tx.order + 1;
99:   if ((sbBarrier ← sbBarrier - 1) = 0) then     ▷ Atomic
100:    sbBarrier ← #TH
101:    sbOrder ← sbOrder + 1                       ▷ Atomic
102:  else
103:    while (tx.order > sbOrder) do
104:    end while                                   ▷ Busy waiting
105:  end if
106: end procedure
```

Fig. 2. Implementación de la API para las barreras especulativas (SBs)

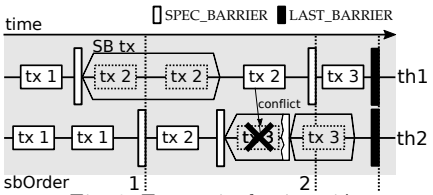

Fig. 3. Escenario de ejecución.

ha abierto realmente ya que está dentro de la transacción SB, de ahí la línea punteada. Sin embargo, th1 no puede acometer los cambios hasta el final de su segunda transacción, puesto que la comprobación del orden se realiza en el procedimiento de commit y para el commit de su primera transacción el orden local era mayor que el global. Luego, th2 ejecuta una transacción y llega a otra barrera donde abre una transacción SB. Dicha transacción es abortada debido a un conflicto con una transacción normal de th1 y es reiniciada. Se puede ver cómo cada hilo cruza las SBs sin esperar al otro hilo, excepto cuando llegan a LAST_BARRIER.

C. SBs y código no transaccional

La API que proponemos se puede usar con códigos no transaccionales. Sin embargo, las transacciones SB pueden acabar siendo muy largas para un sistema HTM best-effort si las barreras están muy separadas. Cuando hay transacciones entre las barreras el procedimiento TX_STOP comprueba el orden global y termina la transacción SB si procede. En ausencia de transacciones carecemos de este punto de comprobación intermedio. Por ello, proponemos el procedimiento CHECK_SPEC (Fig. 4).

CHECK_SPEC es básicamente el *else* de TX_STOP. La línea 2 comprueba si el hilo está ejecutando una transacción SB, e intenta terminarla si así es. Para este fin, se comprueba que el orden local es menor o igual que el global. Si no, la transacción SB continúa.

```
1:  procedure CHECK_SPEC(tx)
2:    if (tx.spec = 1) then                          ▷ In SB mode
3:      beginEscape()
4:      if (tx.order ≤ sbOrder) then
5:        endEscape()
6:        xCommit()
7:        tx.retries ← 0                  ▷ Reset tx descriptor
8:        tx.specLevel ← tx.specMax
9:        tx.spec ← 0
10:     else
11:       endEscape()
12:       tx.specLevel ← tx.specLevel - 1;
13:       if (tx.specLevel = 0) then
14:         beginEscape()
15:         while (tx.order > sbOrder) do
16:         end while                          ▷ Busy waiting
17:         endEscape()
18:         xCommit()
19:         tx.retries ← 0                  ▷ Reset tx descriptor
20:         tx.specLevel ← tx.specMax
21:         tx.spec ← 0
22:       end if
23:     end if
24:   end if
25: end procedure
```

Fig. 4. Procedimiento para comprobar la especulación.

```
1: for (i ← 0; i < N; i ← i+1) do
2:   TX_START(tx)   ▷ Not necessary when using check_spec
3:   for (j ← 0; j ≠ L*((tid+1) %2); j ← j+1) do
4:     dummy ← dummy+1
5:   end for
6:   TX_STOP(tx) / CHECK_SPEC(tx)
7:   SPEC_BARRIER(tx)
8: end for
9: LAST_BARRIER(tx)
```

Fig. 5. Microbenchmark Barrier.

Nótese que se mantiene la comprobación de *specLevel* en la línea 13 aún en ausencia de transacciones entre barreras. De esta manera podemos controlar la longitud de la especulación. CHECK_SPEC se puede usar también en códigos con transacciones para aquellos casos en que el código entre transacciones sea muy largo.

D. Uso de las Barreras Especulativas

El microbenchmark Barrier de la Fig. 5 representa un escenario en el que no hay conflictos entre barreras [12]. Lo usamos para comprobar que la API propuesta no introduce conflictos adicionales relacionados con la implementación. Los hilos ejecutan cómputo local (líneas 3 a 5) y se sincronizan en una barrera (línea 7). La condición en la línea 3 implica que sólo la mitad de los hilos ejecuta el cómputo en cada iteración creando así un desbalanceo. En este escenario el uso de SPEC_BARRIER permite que los hilos ociosos continúen reduciendo el desbalanceo de barrera.

Cholesky y Recurrence se corresponden con el segundo y el sexto kernel de los Livermore Loops (LFK) [13]. Estos kernels explotan el paralelismo de grano fino con alta frecuencia de barreras [11].

La Fig. 6 muestra el código de Recurrence. Nosotros hemos añadido un chunk interno (líneas 7 a 12) para soportar diferentes tamaños de transacción. Primero, el kernel distribuye el cómputo entre hilos (líneas 1 a 3). El bucle de las líneas 5 a 14 actualiza un sólo elemento del array w accediendo a varios

```
1:  ochunk ← N/#TH
2:  start ← tid*ochunk
3:  stop ← start + ochunk
4:  for (t ← 0; t ≤ N-2; t ← t+1) do
5:    for (k ← start; k < stop; ) do
6:      TX_START(tx) ▷ Not necessary when using check_spec
7:      for (c ← 0; c < ichunk; c ← c+1) do
8:        if (k < (N-t-1)) then
9:          w[t+k+1] ← w[t+k+1] + b[k][t+k+1]*w[t]
10:       end if
11:       k ← k + 1
12:     end for
13:     TX_STOP(tx) / CHECK_SPEC(tx)
14:   end for
15:   SPEC_BARRIER(tx)
16: end for
17: LAST_BARRIER(tx)
```

Fig. 6. Livermore loop 6 (Recurrence).

```
1:  ii ← N
2:  ipntp ← 0
3:  repeat
4:    ipnt ← ipntp
5:    ipntp ← ipntp+ii
6:    ii ← ii/2
7:    i ← ipntp-1
8:    ochunk ← (ipntp-ipnt)/2 + (ipntp-ipnt) %2
9:    ochunk ← ochunk/#TH + ((ochunk %#TH)?1:0)
10:   i ← tid*ochunk
11:   end ← (ochunk*2*(tid+1)) + ipntp + 1
12:   start ← ipnt + 1 + (tid*2*ochunk)
13:   for (k←start; k < end; ) do
14:     TX_START(tx) ▷ Not necessary when using check_spec
15:     for (c ← 0; c < ichunk; c ← c+1) do
16:       i ← i+1
17:       x[i] ← x[k] - v[k]*x[k-1] - v[k+1]*x[k+1]
18:       k ← k + 2
19:     end for
20:     TX_STOP(tx) / CHECK_SPEC(tx)
21:   end for
22:   SPEC_BARRIER(tx)
23: until (ii > 1)
24: LAST_BARRIER(tx)
```

Fig. 7. Livermore loop 2 (Cholesky).

elementos de w y de la matriz b. Esto crea una recurrencia debido al patrón RAW en w, por lo que la barrera en la línea 15 es necesaria al final de cada iteración del bucle externo (línea 4 a 16). Sin embargo, las dependencias en w no se producen siempre, resultando un escenario prometedor para el uso de SBs. La LAST_BARRIER en la línea 17 asegura que no se libera memoria hasta que los hilos especulativos han acabado.

La Fig. 7 muestra la factorización incompleta de Cholesky basada en [11]. Se ha añadido un chunk interno (líneas 15 a 19) para soportar diferentes tamaños de transacción. El bucle interno (líneas 13 a 21) actualiza elementos de la mitad superior del array x (línea 17) con elementos de x y v. Una barrera en la línea 22 asegura que cada iteración del bucle externo (líneas 3 a 23) acceden a elementos actualizados de x. Sólo una fracción de elementos de x se actualiza en cada paso, siendo una aplicación adecuada para el uso de SBs. CHECK_SPEC se puede usar si estamos seguros de que no hay dependencias en el bucle interno, lo que no es trivial, aunque se dice que no hay dependencias en el código original [11].

DGCA [14] es un algoritmo de coloreado de grafos que no requiere información completa de todo el grafo para computar el color de cada nodo. Se mantiene

```
1:  p[N][#colors]                          ▷ Global color probability array
2:  start ← tid*N/#TH
3:  stop ← start + N/#TH
4:  for (m ← 0; t < M; m ← m+1) do
5:    for (i ← start; i < stop; i ← i + 1) do
6:      TX_START(tx) ▷ Not necessary when using check_spec
7:      collision = 0
8:      for (j ← 0; j < #adjacentNodes(i); j ← j+1) do
9:        if (nodeColor(i) = nodeColor(j)) then
10:         collision ← 1
11:       end if
12:     end for
13:     if collision then
14:       for (k ← 0; k < #colors; k ← k + 1) do
15:         p[i] ← updateNodeColorProbCollision(i,k)
16:       end for
17:     else
18:       for (k ← 0; k < #colors; k ← k + 1) do
19:         p[i] ← updateNodeColorProbNoCollision(i,k)
20:       end for
21:     end if
22:     updateNodeColor(i,p)      ▷ Potentially updates node
    color
23:     TX_STOP(tx) / CHECK_SPEC(tx)
24:   end for
25:   SPEC_BARRIER(tx)
26: end for
27: LAST_BARRIER(tx)
```

Fig. 8. Decentralised graph colouring algorithm (DGCA).

```
1:  [start,stop] ← createPartition(#edges, tid)
2:  #vert ← #vertices(startVertex,start,stop)
3:  xBegin()                                  ▷ Standard transaction
4:  globMax#Vert ← max(globMax#Vert,#vert)
5:  xCommit()
6:  barrier()                                 ▷ Standard barrier
7:  if (tid = 0) then
8:    outDegree ← dynamicAllocation(globMax#Vert)
9:    outVertexIndex ← dynamicAllocation(globMax#Vert)
10: end if
11: barrier()
12: [start,stop] ← createPartition(globMax#Vert,tid)
13: [outDegree,outVertexIndex] ← initialization(start,stop)
14: SPEC_BARRIER(tx)
15: outVertListSize ← fillArrays(outDegree,outVertexIndex)
16: LAST_BARRIER(tx)
17: prefixSums(outVertexIndex,outDegree,globMax#Vert) ▷
    Dynamic allocation and barriers inside
18: SPEC_BARRIER(tx)
19: TX_START(tx)
20: globOutVertListSize ← globOutVertListSize + out-
    VertListSize
21: TX_STOP(tx)
22: LAST_BARRIER(tx)
23: ...
```

Fig. 9. SSCA2 (Kernel 1).

un array global que mantiene la función de probabilidad de cada color para cada nodo y que se actualiza usando el color de los nodos vecinos. El código se muestra en la Fig 8. Las líneas 4 a 26 comprenden un paso de la ejecución donde se comprueba que un nodo no coincida en color con sus vecinos (líneas 7 a 12). Con esta información se actualiza p en las líneas 13 a 21, asignando una probabilidad máxima al color del nodo si no hay coincidencias, y actualizando la probabilidad del color de otra manera. Las dependencias se generan en la línea 22, donde el color del nodo se actualiza o no dependiendo de p. Puesto que sólo una fracción decreciente de nodos se cambia en cada paso, es un buen escenario para el uso de SBs.

SSCA2 [15] es una aplicación de teoría de grafos que consiste en cuatro kernels que muestran un acceso irregular a un multi-grafo dirigido. STAMP [16] incluye una versión paralela transaccional de SSCA2 y se centra en el kernel 1 debido a su adecuación a TM. Nosotros hemos usado SBs en el kernel 1 y el 4 que muestran un número considerable de barreras.

El kernel 1 construye un grafo a partir de una lista computando los arrays de adyacencia para los vértices de un grafo denso. La Fig. 9 muestra el código con barreras en las líneas 6 y 11 que no deberían reemplazarse por SBs ya que se utilizan instrucciones para la reserva de memoria de arrays que podrían ser usados por transacciones SB sin ser reservados. Hemos usado SBs en la línea 18 y dentro de la función *prefixSums()*. Este kernel no ofrece demasiadas oportunidades para el uso de SBs.

El kernel 4 de SSCA2 (Fig. 10) divide el grafo en clusters de nodos con alta conectividad y minimiza los enlaces entre ellos. El bucle en la línea 4 computa los clusters en paralelo. La barrera de la línea 8 podría ser tradicional ya que el código de después trabaja con los arrays que se computan antes, pero hemos usado una SB en su lugar. La barrera siguiente es adecuada para el uso de SBs ya que la única dependencia es con la inicialización de *globCliqueSize* por el hilo 0 en la línea 10. La siguiente barrera debería ser una LAST_BARRIER puesto que hay dependencias con *globCliqueSize* y *globIter*. Después del bucle, las listas parciales de cortes del grafo se unen para proceder con la minimización de enlaces entre clusters. Hemos dejado sin tocar la barrera de la línea 30 debido a la reserva de memoria, mientras que la barrera en la línea 33 se puede sustituir por una SB.

E. Directrices de uso de las SBs

Enumeramos aquí una serie de directrices para facilitar el uso eficiente de las SBs. Debemos examinar cada barrera del código y no reemplazarla por una SB si: (i) El código previo a la barrera reserva memoria que se usa después de la barrera. Acceder memoria sin reservar por medio de una transacción SB podría resultar en una violación de segmento o un aborto dependiendo del sistema HTM. (ii) El código posterior a la barrera ejecuta instrucciones que siempre abortarán una transacción, tales como llamadas al sistema, interrupciones,... Si colocamos una SB este caso sólo nos arriesgamos a que haya una disminución del rendimiento; (iii) Hay dependencias fáciles de detectar que evitarían que la especulación terminara.

Debemos reemplazar una barrera con una SB si: (i) Las dependencias no son evidentes y la barrera se ha colocado por precaución; (ii) Las dependencias varían dependiendo de los datos, e.g. algoritmos con dependencias sólo en los límites de las particiones de los datos.

En caso de que las barreras estén muy alejadas unas de otras se pueden colocar CHECK_SPECs para tener más puntos de comprobación de la especulación. Una buena práctica es medir el tiempo de la ejecución y utilizarlo como información para la colocación de SBs.

```
1:  ...
2:  vertVisited ← 0
3:  iter ← 0
4:  while (vertVisited < #vert ∨ iter < #vert/2) do
5:    if (tid = 0) then
6:      chooseVertToStart()
7:    end if
8:    SPEC_BARRIER(tx)
9:    if (tid = 0) then
10:     globCliqueSize ← 0
11:   end if
12:   [cliqueSize,cutSetIndex] ← determineClusters(iter)
13:   SPEC_BARRIER(tx)
14:   if (tid = 0) then
15:     iter ← iter + 1
16:     globIter ← iter
17:   end if
18:   TX_START(tx)
19:   globCliqueSize ← globCliqueSize + cliqueSize
20:   TX_STOP(tx)
21:   LAST_BARRIER(tx)
22:   iter ← globalIter
23:   vertVisited ← vertVisited + globCliqueSize
24: end while
25: barrier()                          ▷ Merge partial cutset lists
26: if (tid = 0) then
27:   edgeStartCount ← dynamicAllocation(#THs)
28:   edgeEndCount ← dynamicAllocation(#THs)
29: end if
30: barrier()
31: edgeEndCount[tid] ← cutSetIndex
32: edgeStartCount[tid] ← 0
33: SPEC_BARRIER(tx)
34: if (tid = 0) then
35:   for (t ← 1; t < #TH: t ← t + 1) do
36:     edgeEndCount[t] ← edgeEndCount[t] + edgeEndCount[t-1]
37:     edgeStartCount[t] ← edgeEndCount[t-1]
38:   end for
39: end if
40: TX_START(tx)
41: globCutSetIndex ← globCutSetIndex + cutSetIndex
42: TX_STOP(tx)
43: LAST_BARRIER(tx)
44: ...
```

Fig. 10. SSCA2 (Kernel 4).

III. Evaluación Experimental

Hemos utilizado un procesador IBM Power8 que incluye extensiones HTM best-effort [7] que proporcionan aislamiento fuerte, manejo de versiones lazy, detección de conflictos eager con granularidad de línea de caché, y una política de resolución de conflictos mixta donde las lecturas se resuelven con una política requester-loses y las escrituras con requester-wins. El procesador admite hasta 8KB de accesos transaccionales por núcleo. Las acciones escapadas se implementan por medio del *suspended-mode* que se puede activar con las instrucciones *tsuspend/tresume*. La evaluación se realizó en un servidor S822LC-8335 con 2 sockets y procesadores de 10 núcleos. Cada núcleo puede ejecutar hasta 8 hilos SMT resultando un total de 160 hilos de ejecución.

A. Metodología

Las métricas analizadas incluyen el *speedup* con respecto a la ejecución secuencial no transaccional, el *transaction commit ratio* (TCR), que es el porcentaje de las transacciones que hacen commit con respecto al número total de transacciones iniciadas, y el *speculative commit ratio* (SCR), que es el porcentaje de transacciones SB que hacen commit con respecto a todas las SB iniciadas. Ejecutamos 20 veces cada experimento y obtenemos la media del tiempo de ejecución. Para reducir la variabilidad atamos cada hilo a un núcleo para prevenir migraciones. Usamos 128 núcleos del servidor con la afinidad hilo/núcleo distribuida con un esquema round-robin evitando el SMT en lo posible, lo que da lugar a tres escenarios diferentes: (i) de 1 a 8 hilos sólo se usa un socket por lo que se reduce el impacto de la comunicación entre sockets. En este caso sólo se usa un núcleo por hilo por lo que se aprovechan todos los recursos transaccionales; (ii) los experimentos con 16 hilos utilizan 2 sockets pero sin SMT; (iii) de 32 hilos en adelante se utilizan los dos sockets con SMT por lo que tenemos la latencia de la comunicación entre sockets y los recursos transaccionales repartidos entre hilos SMT. Usamos *padding* para fijar las variables de las líneas 4 y 5 de la Fig. 2 en bloques de caché diferentes y así prevenir conflictos.

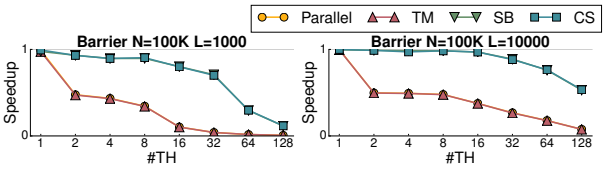

Fig. 11. Resultados del microbenchmark Barrier.

Hemos considerado 5 perfiles diferentes para la evaluación de cada aplicación:

- *Unprotected*: no usa ni transacciones ni barreras, por lo que los resultados pueden ser incorrectos. Este perfil representa un limite superior del rendimiento que se puede llegar a conseguir.
- *Parallel*: con las barreras tradicionales originales de los códigos y sin transacciones ni SBs.
- *CS*: similar a Parallel, pero con SBs en lugar de las tradicionales y con la primitiva CHECK_SPEC para añadir comprobaciones de la especulación.
- *TM*: con transacciones y las barreras tradicionales originales de los benchmarks.
- *SB*: similar a TM, pero con SBs. En este caso las transacciones entre barreras ya proporcionan las comprobaciones de la especulación, por lo que no es necesario el uso de CHECK_SPEC.

B. Resultados

Microbenchmark Barrier

La Fig. 11 muestra los resultados obtenidos para el microbenchmark Barrier de la Fig. 5. El eje y es el speedup sobre la aplicación secuencial, que en este caso particular representa la eficiencia, ya que el benchmark está implementado de manera que la carga no se reparte entre los hilos sino que se replica. De esta manera se puede medir la perdida de rendimiento en barreras y transacciones.

La espera en barrera producida por el desbalanceo de la carga que tiene el benchmark (líneas 3 a 8 en Fig. 5) se puede notar en los perfiles sin SBs (Parallel y TM) donde la eficiencia cae a la mitad. Sin embargo, SB y CS pueden especular más allá de la barrera de manera que los hilos ociosos en una

iteración pueden avanzar y empezar el trabajo de la siguiente. En la gráfica de la izquierda los hilos realizan menos trabajo (L) y se puede ver cómo afecta la contención de barrera, sobre todo cuando se usan los 2 sockets, de 16 hilos en adelante. A los perfiles Parallel y TM les afecta tanto el desbalanceo como la contención de barrera, y los SB y CS también empeoran su rendimiento pero debido sólo a la contención a partir de 16 hilos. Todos los experimentos tienen un 100 % de SCR con SB y CS, como se muestra en la Tabla I, confirmando que el API no introduce falsos conflictos.

TABLA I
TCR y SCR para 16 hilos (%).

Bench		TCR/SCR		
		TM	SB	CS
Barr	L1K	99.99/-	99.99/99.99	-/99.98
	L10K	99.96/-	99.75/99.97	-/99.95
Recurren	C1	98.63/-	99.30/6.37	-/34.60
	C5	95.56/-	98.64/8.31	-/33.09
	C10	92.18/-	97.91/8.42	-/26.68
	C15	88.64/-	96.89/7.91	-/29.20
Cholesky	C1	100.00/-	100.00/38.04	-/60.87
	C5	99.98/-	100.00/43.82	-/58.83
	C10	99.97/-	100.00/45.14	-/59.25
	C15	99.95/-	100.00/45.71	-/57.08
DGCA	N2K A2	99.78/-	99.92/99.60	-/99.00
	N8K A2	99.95/-	99.94/99.36	-/99.76
	N2K A8	94.24/-	93.29/42.04	-/48.59
	N8K A8	98.13/-	97.79/29.59	-/40.31
SSCA2	k1-s14	98.64/-	98.54/3.04	-
	k1-s20	99.98/-	99.97/0.18	-
	k4-s14	71.77/-	30.07/0.51	-
	k4-s20	62.61/-	30.47/0.05	-

Recurrence

La Fig. 12 muestra los resultados para Recurrence. Hemos variado el tamaño del chunk (C) de 1 a 15 iteraciones. Se pueden observar dos comportamientos: los perfiles Unprotected, Parallel y CS no muestran la sobrecarga debido a transacciones, mientras que TM y SB sí. Esto se puede apreciar cuando se usan de 1 a 8 hilos, donde los primeros obtienen speedups similares y los segundos también pero más bajos. A partir de 16 hilos se aprecia un comportamiento superlineal con Unprotected y CS debido a la caché. Recurrence tiene poco reuso de datos en su ejecución secuencial debido al patrón irregular de accesos a la matriz b (Fig. 6, línea 9), y al usar más hilos se disminuye el número de fallos de caché hasta 10× por el uso de las cachés privadas de los núcleos. Por otro lado, se puede observar que el perfil Parallel no escala a partir de 16 hilos debido a la contención de barrera cuando se usan 2 sockets. Esto se puede ver en el perfil TM también, mientras que la especulación de barrera en el perfil SB reduce esta sobrecarga. A partir de 32 hilos se nota la sobrecarga sobre los recursos transaccionales debido al SMT en los perfiles que usan transacciones, sobre todo con chunks más grandes que implican transacciones mayores.

La Tabla I muestra el TCR y SCR para 16 hilos. Para Recurrence los resultados de TCR son buenos. El incremento del chunk produce ligeros decrementos en los porcentajes. Sin embargo, CS produce mejores porcentajes de SCR, lo que es lógico debido a que hay menos transacciones con las que entrar en conflicto (no están las transacciones entre barreras).

Cholesky

Los resultados para Cholesky se muestran en la segunda fila de la Fig. 12. Este benchmark presenta dos diferencias importantes con respecto a Recurrence. Primero, las dependencias son menos frecuentes, lo que se refleja en mejores valores de TCR y SCR, y la mejor escalabilidad independientemente del perfil. Y segundo, la superlinealidad no es tan evidente, debido a que los fallos de caché se reducen hasta 0,75× en versiones paralelas.

El rendimiento del perfil CS se encuentran entre el Unprotected y el Parallel independientemente del número de hilos. Nótese que la diferencia de rendimiento entre estos perfiles se incrementa a partir de 16 hilos debido a la contención en barreras cuando se usan 2 sockets. Este efecto también está presente cuando se comparan los perfiles TM y SB. En este caso, las diferencias son más evidentes a partir de 32 hilos. La sobrecarga causada por las transacciones enmascara las diferencias de rendimiento.

DGCA

Los resultados de DGCA se muestran en la tercera fila de la Fig. 12. Los gráficos muestran diferentes configuraciones variando el grado de adyacencia de los nodos (A) y el número de nodos (N) del grafo. Las conexiones individuales entre nodos se eligen aleatoriamente siguiendo una distribución uniforme. De esta manera, obtenemos una variedad de configuraciones con escenarios de baja contención con grafos pequeños y grandes en las dos primeras gráficas, y escenarios de alta contención en el resto.

En los escenarios de baja contención se obtiene una sobrecarga notable en los perfiles que usan transacciones hasta 8 hilos. En este caso, los perfiles Unprotected y Parallel tienen mejor rendimiento que CS, mientras que TM y SB son parecidos. Sin embargo, Parallel se ve afectado por la contención en barreras de 8 hilos en adelante, obteniendo speedups comparables a TM. La especulación funciona en estos escenarios con CS y SB escalando hasta los 32 hilos y obteniendo los valores más altos de SCR, como se puede ver en la Tabla I. Con el grafo más grande (N=8000), en la segunda gráfica podemos ver un comportamiento similar con ganancias en perfiles especulativos hasta los 64 hilos debido a la menor probabilidad de conflicto entre transacciones.

La escalabilidad en los escenarios de alta contención (las dos gráficas de la derecha) se ve limitada principalmente por las dependencias. En estos experimentos las diferencias entre CS, Parallel y Unprotected se reducen con 1-8 hilos debido a la ejecución más intensiva en cómputo asociada con una mayor conectividad. Nótese la diferencia de rendimiento entre el perfil CS y el SB a partir de 32 hilos en la gráfica de la derecha. En este caso SB baja mucho

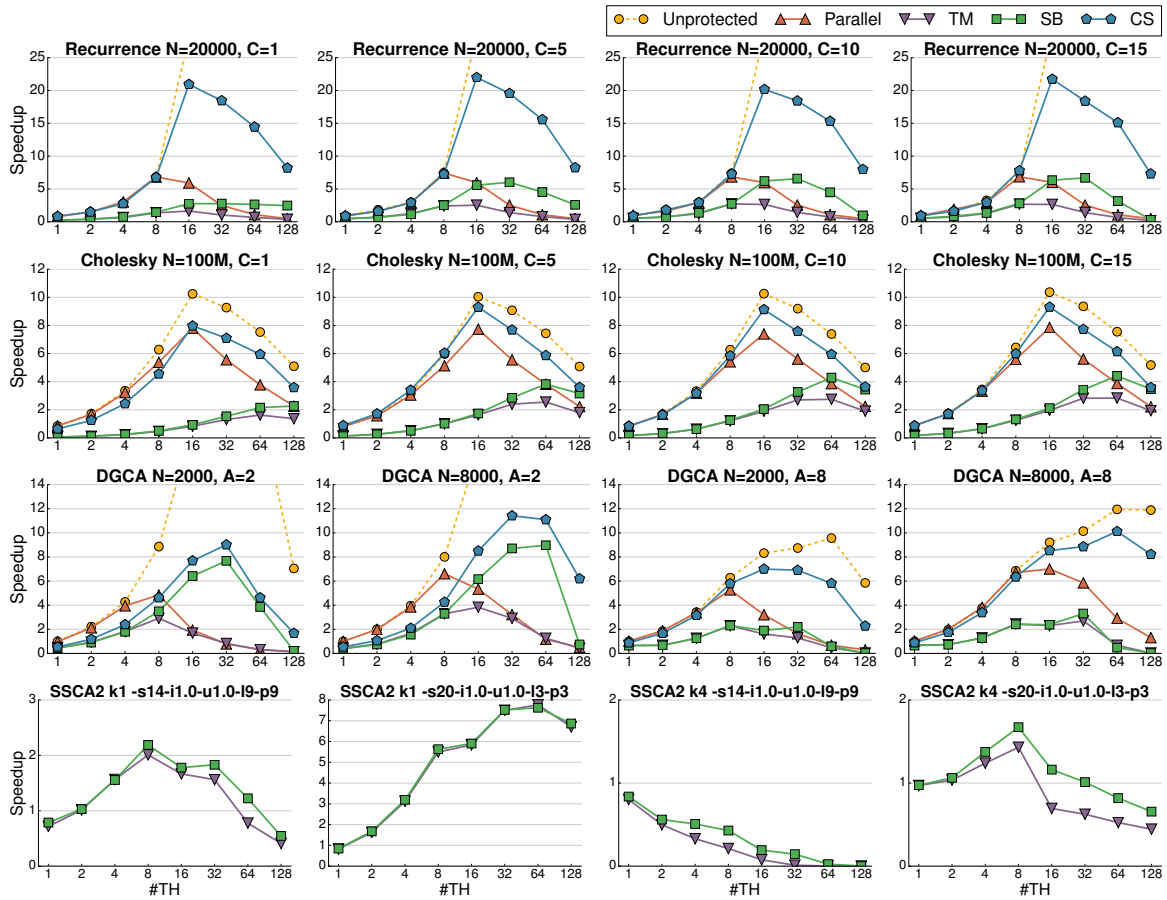

Fig. 12. Resultados de Power8 para Recurrence (Livermore loop 6), Cholesky (Livermore loop 2), DGCA and SSCA2.

su rendimiento ya que las transacciones normales y las SB comparten recursos transaccionales, y el uso de SMT causa abortos por capacidad. Esto no ocurre con CS porque no hay transacciones normales, ni con A=2, ya que las transacciones acceden a menos posiciones de memoria por la menor conectividad.

Las diferencias entra las distintas configuraciones son especialmente evidentes en el TCR y el SCR en la Tabla I. Se aprecia un alto TCR en todos los experimentos mientras que el SCR depende del tamaño del grafo y el número de conexiones, que determinan los potenciales conflictos.

SSCA2

Los resultados para SSCA2 se muestran al final de la Fig. 12. Como parte de STAMP estos benchmarks ya están paralelizados usando transacciones por lo que sólo se consideran los perfiles SB y TM.

El kernel 1 muestra ganancias limitadas de SB sobre TM en la primera gráfica y ninguna en la segunda que tiene una entrada más grande. La mayoría de las barreras del kernel protegen reserva dinámica de memoria e impiden el uso de SBs. Además, a diferencia de los benchmarks previos, las barreras no aparecen dentro de bucles, y como consecuencia sólo un centenar de barreras se llaman a lo largo del código reduciendo las oportunidades de especulación.

En el kernel 4 las principales oportunidades de especulación están en el bucle while de las líneas 4 a 24 de la Fig. 10, que contiene dos SBs. La primera, en la línea 8, especula sobre la función *determineClusters* (línea 12) que, al igual que *fillArrays* del kernel 1, contienen bucles que causan abortos por capacidad. Sin embargo, la computación de *determineClusters* se decrementa con las iteraciones del bucle, por lo que alguna transacción SB puede llegar a acometer sus cambios.

Desafortunadamente, los conflictos causados por la operación de reducción de la línea 19 y las frecuentes interacciones con variables globales en las líneas 10, 16, 22 y 23 producen abortos que limitan el rendimiento. Esto se traduce en los pequeños speedups de la Fig. 12. Este hecho se puede observar también en los bajos valores de TCR y SCR de la Tabla I.

IV. Trabajo Relacionado

La idea de usar técnicas de especulación con código heredado para facilitar la explotación de las arquitecturas multicore no es nueva. Algunas técnicas se centran en la especulación de códigos paralelos donde se consideran las iteraciones de bucles y las llamadas a funciones como objetivos principales de la especulación. En [17] se identifican las limitaciones de TM y encuentran que existe una falta de soporte para diferenciar hilos más o menos especulativos, la falta de forwarding o los conflictos por false sharing como puntos a mejorar del soporte especulativo.

Torrellas et. al. introducen *Speculative Synchronization* como una técnica que aplica Thread-Level Speculation (TLS) a código paralelo. En [18] proponen soporte hardware para permitir especulación en locks, flags y barreras. El hardware comprueba de-

pendencias entre hilos y descarta el trabajo especulativo en caso de conflicto. Esta propuesta no soporta restricciones de orden y no resuelve dependencias de nombre, lo que simplifica los requisitos del hardware.

Otras propuestas se orientan a implementar TLS con extensiones HTM. En [19] se analizan las limitaciones que muestra Intel TSX para la especulación comparado con otras propuestas que aplican hardware adicional no presente en procesadores reales. Se centran en bucles de aplicaciones SPEC CPU2006 y modifican el código manualmente para soportar la especulación. El rendimiento se degrada en la mayoría de los casos (TSX no ofrece acciones escapadas).

En cuanto a optimización de barreras, en [20] se analizan los efectos de la sincronización con barreras de grano fino. Los autores encuentran una cifra relativamente baja de dependencias y proponen un esquema de optimización de código para asumir que no hay dependencias entre dos barreras consecutivas. Usan la Advance Load Address Table (ALAT) presente en los procesadores Itanium para detectar posibles fallos en la especulación en tiempo real.

En [12] se usa soporte HTM comercial para especular en barreras. Esta propuesta requiere especificar manualmente un punto de sincronización para los hilos especulativos después de la barrera. No se proporciona soporte para poner restricciones de orden a las transacciones ni se utilizan acciones de escape, por lo que no hay mejora de rendimiento. Tampoco tienen en cuenta las limitaciones del sistema HTM al no proporcionar un control de la longitud de la especulación.

V. Conclusiones

En este artículo se propone una barrera especulativa (SB) optimista como alternativa a las barreras tradicionales basadas en locks. Las SBs permiten que los hilos salten la barrera usando ejecución especulativa con HTM. Proporcionamos un API para las SBs implementado con extensiones transaccionales para su uso en aplicaciones con y sin transacciones.

Evaluamos nuestras propuestas usando un servidor IBM Power8 que proporciona las extensiones transaccionales necesarias para nuestra propuesta ya que ofrece la posibilidad de ejecutar acciones escapadas dentro de transacción. Los resultados muestran una mejora general del rendimiento sobre las barreras tradicionales con speedups de hasta 6× en media sobre la aplicación paralela para ciertas configuraciones. La configuración en 2 sockets del servidor IBM penaliza la comunicación de las barreras tradicionales mientras que las SBs ocultan la latencia de la contención de barrera. La sobrecarga de las SBs no parece afectar el rendimiento negativamente en la mayoría de benchmarks evaluados. En general se recomienda el uso de SBs incluso cuando hay dependencias obvias entre barreras.

Agradecimientos

Este trabajo ha sido posible gracias a los proyectos TIN2016-80920-R y P12-TIC-1470.

Referencias

[1] M. Herlihy and J.E.B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Int'l. Symp. on Computer Architecture (ISCA'93)*, 1993, pp. 289–300.

[2] L. Hammond, V. Wong, M. Chen, et al., "Transactional memory coherence and consistency," in *Int'l. Symp. on Computer Architecture (ISCA'04)*, 2004, pp. 102–113.

[3] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Int'l. Symp. on Computer Architecture (ISCA'05)*, 2005, pp. 494–505.

[4] K.E. Moore, J. Bobba, Moravan, et al., "LogTM: Log-based transactional memory," in *Int'l. Symp. on High-Performance Computer Architecture (HPCA'06)*, 2006, pp. 254–265.

[5] C. Kevin Shum, Fadi Busaba, and Christian Jacobi, "IBM zEC12: The third-generation high-frequency mainframe microprocessor," *IEEE Micro*, vol. 33, no. 2, pp. 38–47, 2013.

[6] P. Hammarlund and A. J. Martinez et. al., "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[7] H. Q. Le, G. L. Guthrie, D. E. Williams, et al., "Transactional memory support in the IBM POWER8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 8:1–8:14, Jan 2015.

[8] Ricardo Quislant, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata, "Insights into the Fallback Path of Best-Effort Hardware Transactional Memory Systems," in *Int'l. Conf. on Parallel Processing (Euro-Par'16)*, 2016, pp. 251–263.

[9] Michelle J Moravan, Jayaram Bobba, Kevin E Moore, et al., "Supporting Nested Transactional Memory in logTM," in *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, 2006, pp. 359–370.

[10] Manuel Pedrero, Eladio Gutierrez, and Oscar Plata, "TMbarrier: Speculative Barriers Using Hardware Transactional Memory," in *Euromicro Int'l. Conf. on Parallel, Distributed and Network-Based Processing (PDP'18)*, 2018, pp. 214–221.

[11] J Sampson, R Gonzalez, J Collard, et al., "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *Int'l. Symp. on Microarchitecture (MICRO'06)*, 2006, pp. 235–246.

[12] Lars Bonnichsen and Artur Podobas, "Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives," in *Int'l. Workshop on OpenMP (IWOMP'15)*, 2015, pp. 149–161.

[13] John T. Feo, "An analysis of the computational and parallel complexity of the Livermore Loops," *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988.

[14] K R Duffy, N O'Connell, and A Sapozhnikov, "Complexity analysis of a decentralised graph colouring algorithm," *Information Processing Letters*, vol. 107, no. 2, pp. 60–63, 2008.

[15] David A Bader and Kamesh Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *Int'l. Conf. on High Performance Computing (HiPC'05)*, 2005, pp. 465–476.

[16] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Int'l. Symp. on Workload Characterization (IISWC'08)*, 2008, pp. 35–46.

[17] Leo Porter, Bumyong Choi, and Dean M. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009, pp. 313–324.

[18] José F. Martínez and Josep Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 18–29, 2002.

[19] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-self hardware transactional memory," in *Int'l. Symp. on Workload Characterization (IISWC'14)*, 2014, pp. 212–221.

[20] Vijay Nagarajan and Rajiv Gupta, "Speculative optimizations for parallel programs on multicores," in *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*, 2009, pp. 323–337.