Universidad de Valladolid

E.T.S. Ingeniería Informática
Trabajo Fin de Grado
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DE SOFTWARE

Caracterización de memoria transaccional hardware sobre procesadores Intel

---

Alumno: Adrián de la Rosa Martín

Tutor: Benjamín Sahelices Fernández

A la familia, que vive este proceso con más emoción que yo mismo.

Agradecimientos

Me gustaría dar las gracias a la Escuela, pero prefiero concretar el agradecimiento en aquellas personas que han hecho mi paso por estos muros todo un placer. Al final, cualquier institución no es nada sin personas.

Ha sido toda una experiencia trabajar con esos profesores que tienen la suficiente seguridad como para tratarte de tú a tú; sabiendo que los alumnos hemos venido a aprender de ellos, pero sin descartar la posibilidad de que, algunas veces, sean los profesores los que también aprendan de nosotros. Para eso es necesaria una buena dosis de humildad y de ganas de mejorar que no todo el mundo posee. Trabajar con algunos de estos profesionales de la academia fuera del contexto estricto de las clases magistrales también ha sido un verdadero placer. Gracias.

También ha sido maravilloso conocer a compañeros tan dispares. El cambio desde el instituto a la universidad, desde el pueblo a la ciudad, ha sido notorio pero agradable gracias a ellos. Es muy enriquecedor ver distintas personas, ver qué piensan, cómo se desenvuelven, conocer sus contextos y compartir experiencias con ellos. Hago extensivo este agradecimiento al "núcleo duro" del GUI, que tantas horas hemos echado y tantas cosas hemos sacado adelante de la nada juntos.

Resumen

La memoria transaccional es un tema recurrente en la investigación, visto como posible solución simple al problema de la concurrencia. Esta técnica promete hacer los accesos concurrentes a recursos compartidos lo más rápido posibles sin perder la corrección de los programas que los usan, evitando bloqueos en el acceso hasta que realmente se produzca un conflicto, momento en el que la transacción será abortada para evitar errores, dándonos la opción de recuperarnos.

En este trabajo nos centramos en la implementación que por primera vez Intel hace disponible en sus procesadores de última generación dirigidos a consumidores domésticos. Las instrucciones TSX permiten acceder a memoria transaccional implementada directamente en el hardware, con considerables mejoras del rendimiento y sin aumentar la dificultad a la hora del desarrollo.

Para soportar estas afirmaciones, en este trabajo compararemos las características, implemetación, rendimiento y comportamiento de la memoria transaccional frente a una ejecución más tradicional con cierres de exclusión mutua. Usaremos para ello un benchmark sintético que simula una base de datos de un servidor web concurrido, situación que pensamos que podría beneficiarse de una implementación con memoria transaccional.

Esta experimentación estará precedida por una introducción al contexto del problema y la historia de la memoria transaccional hardware.

Abstract

Transactional memory is a recurrent topic in investigation, seen as a simple, plausible solution to the concurrency problem. This technique promises to make concurrent accesses to shared resources as fast as possible without losing correctness in software that uses it, avoiding blocking when accessing shared resources until a real conflict is produced, time when the transaction will be aborted to avoid errors, giving us the option to recover from that error.

In this work we will focus on the implementation that, for the first time, Intel made available in their last generation processors directed towards domestic consumers. TSX instructions allow to access to transactional memory implemented right into harware, with drastic performance improvements and without making development more difficult.

To support this affirmations, in this work we will compare characteristics, implementation, performance and behaviour of hardware-implemented transactional memory against a more traditional execution with mutual exclusion locks. We will use to do so a synthetic benchmark that simulates a crowded web server's database, situation that we think would benefit from a transactional memory implementation.

This experimentation will be preceded by an introduction to the context of the problem and the history of hardware transactional memory.

Índice general

Índice de figuras

Capítulo 1

Introducción y objetivos

En el mundo de la informática es un tema recurrente que, pese a que los ordenadores continúan duplicando el número de transistores cada dos años [27], cada vez encontramos más posibilidades a las plataformas de computación que requieren rendimientos exponencialmente mayores.

1.1. El cuello de botella en servidores web

Un claro ejemplo es la evolución de la población en Internet. Se estima que en el año 2000 existían un total de 400 millones de usuarios en Internet, y en 2015 esa cifra ha subido hasta los 3200 millones, multiplicándose el número de usuarios por 8 en estos 15 años [12]. Esta inmensa cantidad de usuarios provoca grandes problemas a los individuales y empresas que proveen los servicios en red.

Es común hablar de los ataques de denegación de servicio distribuidos (DDoS) como uno de los problemas más frecuentes para el buen funcionamiento de Internet a día de hoy, y es un problema real debido a que el rendimiento de los servidores no puede crecer suficientemente rápido como para acomodar los inmensos picos que se dan, ya sea voluntariamente y con buena intención [24], o todo lo contrario [2].

Uno de los cuellos de botella de los servidores web suelen ser las bases de datos. Además de las restricciones de rendimiento que puedan sufrir por los accesos a disco, se espera que las bases de datos garanticen las propiedades ACID (atomicidad, consistencia, aislamiento, durabilidad) [9] [10] lo que complica el control de la concurrencia.

La solución tradicional a este problema suele ser usar algún tipo de cierre de exclusión mutua por software. Esta decisión afecta al rendimiento en general provocando la serialización de muchas operaciones que de otra manera podrían ser paralelas y con el *overhead* que conlleva la implementación de estos sistemas; todo a cambio de mantener las propiedades ACID.

1.2. La perspectiva de memoria transccional

Otro método más moderno para solucionar este problema es la memoria transaccional. En este caso, es el propio sistema el que se encarga de detectar estas violaciones y repetir las operaciones invalidadas sin afectar a la validez de las transacciones. Sin embargo, las implementaciones accesibles a día de hoy son sobre software y el resultado es que el *overhead* que implican no compensa las ventajas a la hora de programar.

Puede que este panorama comience a cambiar con la popularización de la memoria transaccional sobre hardware. Los procesadores de última generación de Intel (sexta generación, denominada *Skylake*) destinados a consumidores generales ya soportan una implementación de memoria transaccional sobre hardware que promete proporcionar las ventajas de la memoria transaccional sin las graves penalizaciones en el rendimiento de las implementaciones por software.

1.3. Objetivos

En este trabajo quiero explorar las posibilidades de las extensiones TSX que acercan la memoria transaccional sobre hardware a todos los consumidores. A través de una serie de pruebas de rendimiento investigaré bajo qué condiciones la solución de memoria transaccional por hardware que implementa Intel a partir de su sexta generación es más eficiente que las implementaciones habituales con cierres de exclusión mutua.

Una de las conclusiones que se podrán extraer es el estado de esta implementación y si merece la pena hacer soluciones a medida que la aprovechen a día de hoy; y qué problemas implica y cómo debería mejorar esta tecnología para generalizar su uso y ser comparativamente mejor que las alternativas sobre software.

Esta línea de investigación puede ser útil pues si se demuestra que el rendimiento es superior al de un cerrojo de exclusión mutua, y que la implementación es tan o más sencilla, se podrá diseñar sofware que aproveche estas características llevando el paralelismo real a aplicaciones en las que a día de hoy ni siquiera se considera.

Un objetivo secundario de este trabajo es ver cómo se puede generalizar la paralelización de forma que se pueda aplicar a diversas estructuras y algoritmos. Por ello, el último capítulo reune algunas pinceladas sobre el paralelismo en estructuras irregulares y concretamente en un framework denominado Galois que persigue este objetivo.

Capítulo 2

Introducción a la memoria transaccional

2.1. El problema de la concurrencia

Desde los primeros ordenadores multiproceso y multiusuario, la concurrencia ha sido uno de los mayores problemas de la gestión a bajo nivel. Una situación común generada por la concurrencia son las condiciones de carrera, en las que se realizan operaciones simultáneas sobre recursos compartidos que pueden llegar a dejar éstos en un estado inconsistente.

Para solucionar estos peligrosos conflictos, han surgido múltiples propuestas a lo largo de la historia de la informática. Inicialmente estas soluciones han partido del software, con la consecuente pérdida de rendimiento a cambio de fiabilidad. Sin embargo, la tendencia en los últimos años es mover este problema al hardware, de forma que podemos asegurar la fiabilidad minimizando la pérdida de rendimiento.

La solución que suele venir a la mente es la exclusión mutua. Con *locks* o *semáforos* se definen secciones de código a las que denominamos **sección crítica** que solo se ejecutarán si no se están ejecutando ya en otro núcleo del procesador. De estarlo, se esperará hasta que se termine de ejecutar en otros núcleos para entrar en la sección crítica.

Aunque esta estrategia es cada vez más eficiente por la inclusión de nuevas instrucciones de hardware que simplifican y aceleran la implementación de este tipo de estructuras, la exclusión mutua provoca una pérdida considerable de rendimiento al evitar que ciertas secciones de código se puedan ejecutar paralelamente. Por tanto, es trabajo del programador reducir al máximo el tamaño de las secciones críticas para afectar lo menos posible al rendimiento.

2.2. ¿Qué es la memoria transaccional?

El enfoque de memoria transaccional parte de la premisa contraria: en lugar de partir de la existencia de posibles conflictos en las secciones críticas y por tanto ejecutarlas siempre de forma secuencial, de forma que sea imposible que se acceda a la vez a un mismo recurso compartido; en el caso de la memoria transaccional se supone de forma optimista que la mayor parte de veces no habrá conflictos, pensando en que dada la velocidad a la que se ejecutan las instrucciones en los procesadores, el tiempo durante el cual se está accediendo a un recurso compartido es mínimo.

El reto que propone esta solución es detectar si ha ocurrido un conflicto y, en caso de que se detecte, revertir los cambios realizados para pasar a ejecutar las secciones críticas involucradas de forma segura. De esta forma, el coste inicial de usar de forma segura un recurso compartido es menor, aunque potencialmente se traslada ese coste al proceso de recuperación en caso de conflicto detectado.

La solución de memoria transaccional es conceptualmente simple pero implica grandes problemas de rendimiento. Las primeras implementaciones de este modelo fueron sobre software, y aunque el funcionamiento era satisfactorio, cumpliendo los requisitos de fiabilidad, el rendimiento era muy pobre. . Sin embargo, ya se puede probar este concepto sobre hardware en los procesadores Intel más recientes, con la promesa de sortear los problemas de rendimiento.

2.3. Ventajas de la memoria transaccional

Este concepto permite mucha más flexibilidad para el programador. El ejemplo más básico es el que usaremos en este trabajo: si es necesario tener una tabla hash como recurso compartido, hasta ahora existía la opción de poner un cerrojo a toda la tabla o poner un cerrojo a cada entrada de la tabla. La primera opción facilita la programación a costa de empeorar el rendimiento, pues cualquier operación sobre la tabla bloqueará el acceso a toda la tabla por parte de cualquier otro hilo de ejecución, así que los accesos a cualquier entrada del recurso compartido serán secuenciales. La segunda opción hace que los accesos a entradas distintas de la tabla sean mucho más eficientes mientras no haya conflictos, caso en que habrá que esperar a que termine la operación en esa entrada. Sin embargo, si usamos cerrojos, es necesario complicar el código añadiendo gestión granular de cada entrada.

La promesa de la memoria transaccional une las ventajas de ambas opciones: por un lado, la implementación de la zona crítica es tan sencilla como poner un cerrojo global a toda la tabla, sin tener que gestionar cada entrada de la tabla con distintos cerrojos En memoria transaccional sería tan sencillo como indicar toda la zona en la que puede haber conflictos y el código se ejecutará concurrentemente mientras no los haya.

Con este trabajo quiero ver si, con el hardware actual a disposición de los consumidores, concretamente un procesador Intel de última generación destinado a ordenadores domésticos, es factible crear programas con un código sencillo en cuanto a la gestión de concurrencia

pero con el rendimiento de la gestión granular de locks. Quizás, la memoria transaccional por hardware sea la manera de obtener lo mejor de los dos mundos.

2.4. Historia de la memoria transaccional en hardware

Dado que las soluciones de memoria transaccional en software incurren en una grave penalización de rendimiento para hacer el sistema seguro y reducir los falsos positivos y falsos negativos, los fabricantes de procesadores han tratado de implementar este paradigma directamente en el hardware.

Hasta recientemente, la solución por hardware no era viable para un uso generalizado. Aunque aún no sabemos nada de cuándo será puesto a disposición del público, en 2009 AMD anunció que estaban trabajando en unas extensiones para sus procesadores denominadas *Advanced Synchronization Facility* que prometían llevar la memoria transaccional a los procesadores dirigidos a consumidores. Este fue el primer anuncio de un proyecto de este tipo destinado a procesadores de uso común. Sin embargo, siete años después seguimos sin oir noticias sobre este desarrollo. Por tanto, no podemos confirmar ni desmentir que el proyecto esté acabado. De hecho, podriamos pensar que viendo que Intel apuesta por la memoria transaccional sobre hardware, AMD se vea obligada a despertar el proyecto ASF para seguir compitiendo.

2.4.1. Primeras apariciones de la memoria transaccional en hardware

Antes de AMD, Sun fue la primera empresa en prometer llevar la memoria transaccional a un procesador de uso general. Durante el desarrollo de su procesador llamado *Rock*, la comunidad estaba expectante ante las características que Sun iba desgranando de su futurible producto. Además de ser el primer procesador comercial con soporte para memoria transaccional, contaba con otras características innovadoras como precarga del caché con los núcleos en desuso durante los fallos de caché o ramas especulativas a nivel de hilo [23].

Como parte del desarrollo de este procesador, se puso a disposición de desarrolladores externos kits de desarrollo donde se pudo probar sobre hardware real la memoria transaccional en la comunidad académica. También, Sun puso a disposición de los desarrolladores un simulador de memoria transaccional por software para poder hacer pruebas con las extensiones a la arquitectura SPARC que desarrollaron para su procesador. Desafortunadamente, tras la compra de Sun por parte de Oracle en 2010, Sun confirmó que los planes para el desarrollo del procesador habían sido cancelados.

Otros procesadores que soportan instrucciones de memoria transaccional desde hace años, aunque con ciertas limitaciones, son el Blue Gene/Q de IBM o los procesadores RISC en algunas de sus instrucciones. Sin embargo, estos procesadores no suelen estar dirigidos al público general por su incompatibilidad con las instrucciones x86, sino que se enfocan a usos empresariales, sistemas embebidos u otras aplicaciones más especializadas.

2.4.2. Memoria transaccional en hardware con Intel

Tenemos que esperar hasta la cuarta generación de procesadores Intel Core, denominados Haswell, para encontrar la primera comercialización de una implementación de instrucciones de memoria transaccional sobre hardware destinado a usuarios domésticos, que ocurrió en junio de 2013 por primera vez. Estos procesadores incluían el conjunto de instrucciones TSX (Transactional Synchronization Extensions) que permiten acceder a las capacidades de memoria transaccional del procesador de dos maneras: de una forma más restringida pero retrocompatible, con la interfaz HLE (Hardware Lock Ellision); o de una forma más flexible pero incompatible con procesadores que no lo soporten con la interfaz RTM (Restricted Transactional Memory). Más adelante elaboraremos sobre las diferencias en estas dos interfaces.

Sin embargo, en agosto de 2014, Intel se vio obligado a desactivar las funcionalidades de TSX [25] con una actualización de microcódigo en los procesadores Haswell debido a un error que, bajo ciertas condiciones, provocaba un comportamiento impredecible en el procesador. Este desafortunado evento provocó que hubiera que esperar hasta la sexta generación de procesadores Intel (Skylake) para poder usar estas funcionalidades con normalidad.

En la sexta generación de procesadores, la última lanzada por Intel, encontramos las mismas extensiones que en la cuarta, cuando se hizo el primer lanzamiento. La única diferencia es que a priori los errores encontrados en aquella han sido resueltos. Las pruebas de este trabajo están realizadas sobre un procesador de esta generación. A día de hoy no se han reportado errores en la implementación de memoria transaccional de los procesadores de sexta generacion.

2.5. Funcionamiento de la memoria transaccional en hardware

Como ya adelantamos en la sección anterior, Intel proporciona un conjunto de instrucciones denominado TSX que nos permite hacer uso de las capacidades de memoria transaccional en sus procesadores de última generación; pero estas instrucciones pueden ser aprovechadas de dos maneras. Estas dos formas coinciden con las dos implementaciones habituales en las que se suele dar la memoria transaccional. En los siguientes apartados describiremos en más detalle cada uno de estos acercamientos, acompañándolo de algunos ejemplos [22].

2.5.1. *Hardware Lock Ellision* (HLE)

La más simple entre ambas interfaces para acceder a las capacidades de memoria transaccional del procesador es la denominada elipsis de cierre de exclusión mutua (*lock ellision*). Mientras que la programación con cierres de exclusión mutua se considera pesimista al bloquear todos los accesos a la sección crítica para evitar cualquier tipo de conflicto, este enfoque se basa en convertir la programación con cierres de exclusión mutua en optimista, sin alterar

sustancialmente el código ya escrito con cerrojos de exclusion mutua. Para ello, el procesador ejecuta inicialmente todas las secciones críticas como si no fueran tales, de forma paralela transformándolas en transacciones; y en caso de que se produzca algún conflicto, el procesador lo detectará y dará marcha atrás a las transacciones involucradas para acto seguido volver a ejecutarlas secuencialmente pasando por los cierres de exclusión mutua que ignoró en un principio.

En la implementación de Intel, las instrucciones del conjunto denominado HLE (*hardware lock ellision*), que son las que implementan la elipsis de cierre de exclusión mutua, son ignoradas sin provocar errores en procesadores que no las soportan, por lo que automáticamente el código se ejecuta con todos los cierres de exclusión mutua con seguridad y sin necesitar cambios para ser retrocompatible. Esta característica permite actualizar fácilmente el código ya escrito con cerrojos, o escribir código que aproveche las bondades de la memoria transaccional sin mucho esfuerzo y sin perder la compatibilidad con procesadores anteriores a nivel de binario.

Una implementacion sencilla a nivel de código ensamblador sería la siguiente. En primer lugar, el código que obtiene el cerrojo para comenzar la sección crítica:

```
        mov eax, 1
Try:    xacquire lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

Vemos que es un código bastante tradicional para adquirir un cerrojo. Algo que destaca es la instruccion `xacquire`, que determina el comienzo de la zona transaccional con HLE. Como decíamos, esta instruccion es en realidad una anotación que será ignorada por aquellos procesadores anteriores que no la reconozcan, asegurando la retrocompatibilidad y la corrección del programa en procesadores antiguos pues en dicho caso, ignorando la etiqueta se interpretará el cerrojo de forma tradicional.

Este es el claro ejemplo de que realmente el coste de implementar un cerrojo normal o implementarlo haciendo uso de HLE es el mismo, tanto en términos de productividad del programador como de compatibilidad con procesadores.

Tras la sección crítica, el código para liberar el cerrojo sería el siguiente:

```
        xrelease mov mutex, 0
```

Siguiendo con la tendencia anterior, vemos que el único cambio es la anotación `xrelease` que, como podríamos esperar, marca el fin de la zona transaccional. Con estos dos pequeños cambios tendremos un programa que funciona correctamente y sin cambios en la práctica en

un procesador no compatible con TSX, pero que se ejecuta transaccionalmente a través de la técnica de la elisión de cierre de exclusión mutua sin ningún esfuerzo para el programador. Esto implica que el código de la sección crítica se ejecutará en paralelo mientras no haya conflictos, y en el caso de que los haya (u ocurra cualquier otro problema que aborte la transacción) será el propio procesador quien deshaga los cambios realizados hasta el momento y repetirá la ejecución usando el cerrojo tradicional. Todo esto, sin ningún tipo de runtime y aprovechando la eficiencia de la implementación en hardware.

2.5.2. *Restricted Transactional Memory* (RTM)

Frente a la propuesta más conservadora que plantean las instrucciones HLE, Intel también proporciona dentro del conjunto TSX una segunda interfaz que pierde la retrocompatibilidad con anteriores procesadores a cambio de aportar una mayor flexibilidad a la resolucion de los conflictos y otros tipos de errores que se pueden dar en las transacciones, obteniendo a cambio un rendimiento potencialmente mejor al permitir al programador la posibilidad de adaptar la estrategia de respuesta frente a errores a cada carga de trabajo concreta.

El conjunto de instrucciones RTM (*Restricted Transactional Memory*) permite designar una parte del código como transaccional con las instrucciones `XBEGIN` y `XEND`, incluso de forma anidada; y permite indicar un desplazamiento hasta una instrucción en la que se implemente un camino seguro alternativo, o incluso una lógica distinta que, por ejemplo, decida si reintentar la transacción en función de la información proporcionada por el procesador.

Un ejemplo útil de implementacion de RTM es el benchmark que desarrollaremos para caracterizar el comportamiento de las instrucciones TSX en posteriores secciones. Pese a la sencillez de uso de HTM, esa estrategia no sirve para caracterizar el comportamiento pues no nos permite acceder a las razones del fallo de una transacción, a diferencia de las RTM.

Veremos también un ejemplo sencillo de implementación de RTM en ensamblador. Comenzamos con el código que realiza la adquisición del cerrojo:

```
Retry:  xbegin Abort
        cmp mutex, 0
        jz Success
        xabort $0xff
Abort:
        ; Comprobar el registro EAX para conocer la razón del fallo.
        ; Aquí podemos elegir si reintentar la transaccion, realizar
        ; la operación con un lock tradicional...
```

El código comienza con la instrucción `xbegin`, que recibe como parámetro la etiqueta del bloque de código que gestionará los fallos de la transacción, donde se podrá leer la razón del fallo y decidir cómo continuar: repitiendo la transacción, adquiriendo un cerrojo tradicional u otras alternativas más dependientes de cada problema que se puedan adaptar mejor. De aquí es de donde surge la flexibilidad de RTM: es nuestro propio código quien

gestiona los fallos en las transacciones, con lo que podemos resolverlos con total libertad. Por supuesto, esto conlleva algo más de trabajo por parte del programador si lo comparamos con la implementación de HLE sobre un programa ya existente, pero vemos que el codigo no es mucho más complejo que el de un cerrojo tradicional.

La adquisición del cerrojo salta a la zona crítica si el cerrojo no ha sido adquirido aún (podria haber hilos que lo cojan directamente sin transacciones, o transacciones fallidas que se hayan resuelto usando un lock tradicional), y de lo contrario vemos una instrucción nueva, `xabort`, que fuerza el fallo de la transacción. `xabort` recibe como parámetro un valor que se pasará como código de error al código que gestiona los fallos [13].

Tras la sección crítica, podríamos ver un código como el siguiente:

```
cmp mutex, 0
jnz release_lock ; Salto a "mov mutex, 0"
xend
```

Este código puede gestionar simplemente el fin de la transacción, que se marca con la instrucción `xend`, o bien la liberación del cerrojo de exclusión mutua.

2.6. Implementación en hardware de las instrucciones para memoria transaccional hardware de Intel (TSX)

Los dos sistemas anteriormente descritos (HLE y RTM) tienen interfaces distintas que habilitan posibilidades distintas. HLE permite convertir a código transaccional un código habitual con cierres de exclusión mutua, manteniendo la retrocompatibilidad; mientras que RTM permite soluciones de memoria transaccional más creativas y posiblemente más adaptadas a cada problema y por tanto más eficientes, a cambio de que el software desarrollado con RTM no pueda ser ejecutado en procesadores que no dispongan de las extensiones TSX.

A pesar de tener estas dos interfaces distintas, el funcionamiento interno de ambas es el mismo. Paso a describir su comportamiento desde diferentes propiedades.

2.6.1. Detección de colisiones

Intel se decantó por implantar memoria transaccional con aislamiento fuerte (*strong isolation*). Esta forma de memoria transaccional es más segura que otras alternativas como el aislamiento débil (*weak isolation*). Aunque no entraremos en el debate entre ambas propuestas, que cuentan con entusiastas defensores y detractores, sí describiremos en qué se traduce esta decisión a la hora de desarrollar código transaccional con TSX.

La principal diferencia entre ambos acercamientos es que un aislamiento fuerte no solo detectará colisiones entre secciones de código transaccional, sino que también detectará situaciones en las que código no transaccional afecta a la integridad de una transacción.

El desarrollo de sistemas de memoria transaccional ha estado limitado más allá de la investigación a la memoria transaccional por software que, como hemos comentado, incurre en unas graves penalizaciones de rendimiento. Por tanto, aunque algunos investigadores entienden que el aislamiento fuerte es una propiedad deseable, no se ha podido aplicar este método puesto que limita más el rendimiento que el aislamiento débil [20]. Por tanto, las soluciones de memoria transaccional software no se pueden permitir aislamiento fuerte. Al implementar la memoria transaccional en hardware, podemos disfrutar de la seguridad del aislamiento fuerte sin perder demasiado rendimiento.

Para implementar este aislamiento fuerte y detectar colisiones en los accesos y escrituras a la memoria que puedan invalidar la transacción, ya sean desde otros bloques transaccionales o desde código no transaccional, la implementación de Intel hace un seguimiento de las líneas de caché que se leen y se escriben en cada transacción (*read-set* y *write-set*). Actualmente, estas líneas de caché tienen un tamaño de 64B.

Al conocer las líneas de caché leídas y escritas, el procesador puede detectar cuándo se produce algún conflicto, ya sea entre secciones de código transaccional o no, que obligue a abortar la transacción en curso.

La elección conservadora de optar por el aislamiento fuerte evita posibles condiciones de carrera, conflictos y otros tipos de errores que podrían no ser detectados con un aislamiento débil, pero es un lujo que podemos permitirnos al estar el sistema implementado en silicio.

2.6.2. Anidamiento de secciones transaccionales

Tanto HLE como RTM permiten anidar transacciones, pero ambos métodos tratan estos anidamientos de la misma manera. El procesador tiene una constante cuyo valor depende de la implementacion y no es accesible al sofware que define el máximo número de transacciones anidadas permitidas para cada tipo de interfaz (HLE o RTM), así como un contador que tampoco es accesible dinámicamente que suma uno al abrir una zona transaccional y resta uno al cerrarla. Cuando el valor del contador alcanza el valor prefijado en la constante, todas las transacciones que estén anidadas en ese momento fallarán por exceder el límite de anidamiento.

Hay que tener en cuenta que el procesador tratará todas las transacciones anidadas como una gran transacción monolítica, lo que explica que todas las transacciones no solo serán abortadas si se supera el límite de anidamiento, sino que en el caso de que cualquiera de las transacciones internas falle por cualquier motivo, todas las transacciones anidadas fallarán también.

2.6.3. Instrucciones que invalidan las transacciones

La implementación de memoria transaccional de Intel hace que ciertas instrucciones del procesador dentro de una transacción provoquen que sea abortada. La especificación de las

instrucciones TSX no define qué instrucciones provocarán que la transacción se anule, puesto que el objetivo es que cada vez menos instrucciones provoquen estos fallos.

Algunas de las instrucciones que obligan a abortar una transacción son:

- Funciones que impriman a la salida estándar (`printf`).
- La instrucción `PAUSE`, usada para hacer eficientes los bucles de espera activa que se suelen usar con los cerrojos de bloqueo mutuo.
- La instrucción `CPUID`, que permite obtener información sobre la versión y capacidades del procesador en tiempo de ejecución.
- Operaciones del coprocesador de coma flotante, del subconjunto de instruccciones `x87`.
- Operaciones `MMX`, que operan sobre los registros del coprocesador de coma flotante.
- Operaciones simultáneas en una misma transacción sobre los registros `XMM`, que son usados para operaciones de coma flotante en las instrucciones `SSE` (*Streaming SIMD Extensions*), y los registros `YMM`, usados por las `AVX` (*Advanced Vector Extensions*) que expanden el tamaño de los registros SIMD hasta 256 bits.
- Cambios a cualquier campo que no sea de estado en los registros tipo `FLAGS`.
- Interrupciones.
- Eventos de E/S.

Otro evento habitual que cancela una transaccion es un cambio de contexto, puesto que en esa situación se anularán la mayoría de las líneas de caché del procesador. Esto hace que sea arriesgado realizar operaciones del sistema operativo, puesto que al ceder el control a éste, se producirá un cambio de contexto con las consecuencias anteriormente mencionadas.

Capítulo 3

Análisis y diseño de una aplicación sintética para caracterizar la memoria transaccional

3.1. Objetivos de la aplicacion sintética

Para estudiar el comportamiento de la memoria transaccional, caracterizarlo y ver para qué tipo de aplicaciones es éste acercamiento más apropiado, haremos uso de un benchmark sintético con la intención de simular a través de unos parámetros las diversas condiciones que se pueden dar en el acceso a una base de datos que se encuentre tras un servidor web, tratando de dar respuesta a las preguntas que nos hacíamos en la introducción.

En las siguientes subsecciones describiremos pormenorizadamente cada uno de los objetivos que nos proponemos para este benchmark sintético. Tras la descripción, analizaremos qué características impondrán al diseño final del benchmark para que el resultado final pueda cumplir estos objetivos.

3.1.1. Simular el modelo de una base de datos

Partiendo de la problemática que describimos en el trabajo, aspiramos a hacer un benchmark que, sabiendo que no puede equipararse al uso de una base de datos en producción, sí nos pueda dar nociones sobre en qué situaciones se puede comenzar a utilizar esta incipiente tecnología. Por tanto, deberemos intentar que el benchmark sea comparable al funcionamiento de una base de datos en memoria.

Para simular esta estructura de datos, usaremos una tabla hash. La misma estará compuesta con un vector en el que cada uno de sus elementos es la cabeza de una lista enlazada. Podríamos comparar esta estructura, de forma simplificada, a la de una tabla de una base de datos moderna cacheada en la memoria para un acceso rápido. La memoria transaccional o el cerrojo de exclusión mutua nos proporcionarán las garantías de que los datos se mantienen consistentes. La consistencia a través de la red en múltiples nodos queda fuera del ámbito de este trabajo.

Además, no solo la forma de la estructura de datos deberá asemejarse a la que podría usar una base de datos en memoria, sino que también los patrones de acceso deberán ser similares. Como hemos puesto el foco en simular una base de datos que da servicio a un concurrido servidor web, supondremos que se lanzan varios hilos de forma concurrente, con algunos de ellos de lectura y otros de escritura. Para simular la naturaleza estocástica de los accesos a un servidor web, tanto las lecturas como las escrituras se harán a posiciones aleatorias de la tabla. Además, la tabla será previamente rellenada con datos aleatorios.

3.1.2. Altamente parametrizable

Debemos hacer un análisis de los parámetros que pueden influir en el rendimiento de una ejecución con memoria transaccional de forma que podamos automatizar a través un *script* la ejecución de este benchmark con todas las posibilidades dentro de los rangos de parámetros que decidamos. Esto permitirá facilitar las pruebas, ya que además de permitir una automatización sencilla de la ejecución, evitamos nuevas compilaciones para cambiar los parámetros y agilizaremos la investigación al poder partir de una ejecución amplia para luego dirigir el foco a las partes que consideremos más interesantes.

Es importante en este punto decidir qué variables es interesante parametrizar para poder estudiar fielmente situaciones que puedan caracterizar el comportamiento de las extensiones TSX. Para ello no podemos olvidar el anterior capítulo de este trabajo en el que analizamos a nivel técnico esta tecnología. De ahí deducimos varios parámetros que pueden cambiar seriamente el comportamiento de las transacciones.

En el punto anterior ya hemos hablado de la estructura de datos elegida para representar la base de datos en memoria, con lo que de aquí en adelante hablaremos en función de dicha estructura.

Los primeros parámetros que necesitamos manipular serán las dimensiones de la tabla. Éstas afectarán a la probabilidad de que las transacciones aborten, y en función de qué dimensión se modifique, variará más notablemente un tipo concreto de fallos u otro. Extenderemos sobre este punto en el siguiente apartado.

Otro parámetro que deberíamos poder modificar para caracterizar el comportamiento de la memoria transaccional es el número de hilos que están trabajando simultáneamente sobre la estructura. Aquí es donde podremos ver si realmente la memoria transaccional puede solucionar el cuello de botella que impone un cerrojo de bloqueo mutuo a toda la estructura, si realmente se puede superar el mismo con la memoria transaccional por hardware con una dificultad de desarrollo similar, y en qué condiciones.

Por supuesto, también es necesario poder parametrizar la carga de trabajo de la aplicación. Deberíamos poder especificar el número de accesos, ya sean de lectura o escritura, que realiza cada hilo.

3.1.3. Caracterización en base a los tipos de fallos

Como nuestro objetivo es caracterizar el comportamiento de la memoria transaccional, nos es necesario conocer cómo afectan los cambios en los parámetros a dicho comportamiento. Nos valdremos para ello por una parte de la comparativa en el tiempo de ejecución frente a los cerrojos tradicionales, pero también nos será útil conocer la razón de los posibles fallos en las transacciones para entender cómo afectan al rendimiento.

En primer lugar, nos fijamos en los fallos por capacidad. Este tipo de fallos ocurren cuando antes de que termine una transacción, el procesador ha tenido que invalidar una línea de caché que estaba siendo editada en una zona transaccional debido a que no hay más espacio en la caché y, según la política de gestión de la caché del procesador, esa línea tenga que ser invalidada. En la implementación actual de las extensiones TSX no se da ninguna prioridad especial a las líneas de caché usadas en una transacción así que éstas pueden ser invalidadas con la misma facilidad que el resto de líneas.

El parámetro que más efecto tiene en los fallos por capacidad en nuestra simulación de base de datos es la longitud de la lista enlazada, pues cuanto más larga sea la misma, más se necesitará llenar la caché para alcanzar el dato deseado. Aunque el acceso es aleatorio, la probabilidad de navegar por un número mayor de nodos de la lista enlazada, llenando la caché, será mayor cuanto mayor sea la longitud de la lista enlazada.

Otro tipo de fallos que pueden afectar seriamente al comportamiento de la memoria transaccional son los fallos por conflicto. Estos ocurren cuando se detecta un conflicto, es decir, cuando se producen lecturas y escrituras, o varias escrituras, durante una transacción sobre una misma línea de caché. Recordemos que al ser éste un modelo de memoria transaccional con aislamiento fuerte, incluso un acceso desde fuera de cualquier código transaccional dispararía un fallo por conflicto abortando la transacción; sin embargo todos los accesos a la estructura compartida se harán dentro de transacciones en este caso.

Para ver el efecto de este tipo de fallos, usaremos como parámetro el tamaño del vector que contiene las cabezas de las listas enlazadas. Cuanto mayor sea este tamaño, más probable es que dos hilos caigan (como ya hemos dicho, aleatoriamente) en listas enlazadas distintas, y por tanto no compartan líneas de caché para acceder a los datos. A su vez, reduciendo el tamaño del vector podremos forzar conflictos.

3.1.4. Sencillo en el diseño y la implementación

Puesto que de un benchmark se trata, éste software no es el objetivo final del trabajo, sino los datos que podamos sacar de él y la información que podamos deducir de ellos. Sin

descuidar su diseño e implementación, debemos tener claro que debe primar la sencillez que nos permita iterar con rapidez y evitar posibles errores.

Una decisión que nos impone la técnica es el uso de un lenguaje de bajo nivel como ensamblador, C o C++. Para facilitar la tarea del desarrollo, evitaremos el lenguaje ensamblador. El desarrollo en C o C++ no dificulta la tarea puesto que a través del compilador GCC [28], tenemos un soporte asequible para usar las extensiones TSX sin tener que bajar al nivel del código ensamblador.

Para acelerar la implementación y la iteración, usaremos un paradigma de programación imperativa, usando funciones para facilitar la lectura y almacenando globalmente el estado de la aplicación cuando sea necesario. Intentaremos limitar el código tanto en número de ficheros como en tamaño de los mismos, aspirando a que el código sea lo suficientemente sencillo como para poder describirlo con claridad solamente con un fichero de código y otro de encabezado.

También, para simplificar la implementación de la memoria transaccional, usaremos las instrucciones RTM usando un cerrojo de exclusión mutua global como plan b. Como decíamos anteriormente, las instrucciones RTM permiten una mayor flexibilidad en cuanto al plan b en caso de fallo de la transacción, por lo que es probable que los resultados que obtengamos no sean óptimos comparándolos con la flexibilidad que ganaríamos aprovechándonos de la información sobre el fallo que nos da el procesador y de alguna solución adaptativa que optimice para cada situación. Esta investigación sobre los límites de RTM queda fuera del alcance de este trabajo.

3.1.5. Comparar con la ejecución tradicional con cerrojos

Para maximizar la utilidad de este estudio, buscaremos que el propio benchmark nos proporcione datos comparativos entre la ejecución usando TSX, y exactamente la misma ejecución pero usando un cerrojo tradicional. Para que la comparación sea justa, ambos métodos emplearán una restricción a la estructura de datos completa, no por líneas. De esta forma podremos comparar en qué casos es más eficiente el uso de memoria transaccional aún cuando la granularidad de las restricciones y el esfuerzo a la hora de programarlo es comparable. Es obvio que debemos garantizar que éstas ejecuciones no son lanzadas concurrentemente para que ambas puedan utilizar todo el potencial de la máquina en igualdad de condiciones.

3.2. Análisis y diseño de la aplicacion de benchmark sintético

En función de estos objetivos, plantearemos las consecuencias que tendrán éstos en el diseño final de la aplicación.

3.2.1. Interfaz

Tanto por simplicidad en el desarrollo como por simplicidad en el uso, la interfaz ideal para esta situación es la línea de comandos. Debería ser suficiente con llamar al ejecutable indicando como parámetros todos los valores variables que hemos indicado anteriormente. Una interfaz de línea de comandos podria tener la siguiente forma, suponiendo que los valores con la forma <valor> son requeridos y aquellos con la forma [valor] son opcionales.

```
./benchmark <Número de hilos> <Número de iteraciones>
        <Número de entradas en la lista enlazada> <Tamaño de la tabla>
        <Fichero de salida> [Verboso]
```

Para facilitar el posterior tratamiento de datos, sería bueno obtener éstos en un formato sencillo pero estándar de forma que pueda ser usado sin problema por sean las que sean las herramientas que se escojan para este fin. El formato CSV reune satisfactoriamente estas condiciones.

3.2.2. Estructura interna

Internamente, el programa constará de un fichero de encabezado `TablaHash.h` con las definiciones de todas las variables y funciones globales. Ese fichero será usado desde el fichero con el código del programa, `TablaHash.c`. En él tendremos una función `main()` que irá ejecutando secuencialmente las operaciones indicadas. Concretamente, esta función main seguiría el siguiente esquema:

- Leer y almacenar los parámetros.
- Inicializar la tabla según los parámetros dados, rellenándola con valores aleatorios.
- Hacer una ejecucion parametrizada lanzando los hilos indicados con la carga de trabajo indicada usando un cerrojo global de exclusión mutua. Tomar tiempos y almacenarlos.
- Repetir la misma ejecución pero usando memoria transaccional en lugar del cerrojo, midiendo de nuevo el tiempo.
- Obtener las estadísticas de fallos de transacciones.
- Con los tiempos y las estadísticas de fallos, escribir una línea en formato CSV al fichero que se indicó como parámetro indicando los parámetros de esta ejecución y todos los datos recogidos.
- Tras esto, el programa finaliza indicando al script que lo lanzará que ya puede ejecutarlo con la siguiente variación de los parámetros.

Las funcionalidades que se llaman desde la función `main()` se dividirán en funciones para aumentar la legibilidad. Detectamos las siguientes partes que se pueden dividir en funciones:

- La creación de la tabla hash global.

- El rellenado de la tabla con datos aleatorios.
- La adquisicion y liberación del cerrojo global o, en su caso, la apertura y cierre de la zona transaccional.
- La creación de los hilos de edición y búsqueda.
- Las funciones que ejecutarán dichos hilos, que repetirán su acción en bucle para simular una carga de trabajo.
- Las funciones que hacen la edicion y la búsqueda en sí, que serán ejecutadas por los hilos.
- La espera por todos los hilos, para saber cuándo todos ellos han terminado.
- La obtencion de las estadísticas de fallos de transacciones tras la ejecución.

La ejecución transaccional conlleva un problema adicional, y es que debemos compaginar el uso de la memoria transaccional con el del cerrojo global que se usará si la transacción aborta. La sección de código que abre la zona crítica será igual para todas las transacciones, pero la que la cierra deberá por un lado volver a ejecutar la transaccion con un cerrojo si ésta ha fallado, y por otro lado comprobar si se está usando o no el cerrojo para liberarlo o bien llamar a la función que cierra la sección transaccional. Para conseguir esto, nos vemos obligados a comprobar si tenemos el cerrojo y por tanto debemos liberarlo al terminar; podemos hacer uso de una comparacion como esta, siendo `mutex` el cerrojo global:

```
if ((*mutex).__data.__lock == 0) {
    // Estamos en transaccional
} else {
    // Tenemos el cerrojo y hay que liberarlo
}
```

Es probable que el compilador se queje del acceso a este campo privado, pero será un mal menor.

3.2.3. Camino transaccional y camino seguro

En la ejecución transaccional tenemos a priori dos opciones que nos proporciona el conjunto TSX. Bien podemos usar HLE, con la ventaja de la sencilla implementación, pues se basa en cerrojos tradicionales añadiendo una simple anotación en la adquisición y liberación del mismo para convertir la ejecución en transaccional; bien podemos usar RTM, que a cambio de una implementación algo más compleja, tenemos más flexibilidad sobre qué hacer tras el fallo de una transacción. Sin embargo, hay un factor a mayores que decanta la decisón por RTM, y es que éste es el único conjunto de instrucciones que permite recopilar estadísticas sobre las razones de fallo de las transacciones, un hecho vital para una aplicación como ésta que trata de caracterizar el comportamiento de la memoria transaccional, no limitándose únicamente al rendimiento.

Sin embargo, en nuestro caso nos sería suficiente con el comportamiento de HLE: ejecutar de forma transaccional y, en caso de fallo, deshacer los cambios para pasar a ejecutar con

un cerrojo tradicional que asegure la corrección de la ejecución. Con RTM podemos generar este comportamiento sin perder la posibilidad de recopilar las valiosas estadísticas sobre los motivos de fallo.

Por tanto, a la hora de desarrollar la aplicación deberemos crear un código de adquisición y liberación del cerrojo de exclusión mutua que compatibilice el uso de cerrojo tradicional con la gestión de las transacciones usando RTM por dos razones: por una parte, como hemos dicho, haremos que el código transaccional revierta a un cerrojo cuando la transacción falle, asegurando la corrección del programa; y por otra parte, la ejecución con la que compararemos la ejecución transaccional es una ejecución que usa cerrojos normales. Si nuestro programa tiene funciones para gestionar ambos casos, podemos manejar ambas ejecuciones (transaccional y con cerrojos tradicionales) usando el mismo código.

La función que adquiere el cerrojo o marca el inicio de la transacción podría tener la siguiente forma:

```
void acquire_lock(pthread_mutex_t *mutex, int use_tsx) {
    unsigned int status;

    if (use_tsx) {
        if ((status = _xbegin()) == _XBEGIN_STARTED) {
            // Transacción iniciada.
            if ((*mutex).__data.__lock == 1) {
                // Si, a pesar de estar en transacción, alguien ha
                // adquirido el cerrojo, la transacción termina y
                // esperamos para cogerlo.
                _xend();
                pthread_mutex_lock(mutex);
            }
            // Si no, la transacción continúa sin adquirir el cerrojo.
        } else {
            // La transacción falló.
            // Recogida de estadísticas usando la variable "status".
            pthread_mutex_lock(mutex);
        }
    } else {
        pthread_mutex_lock(mutex);
    }
}
```

La función que libera el cerrojo o termina la transacción podría ser de la siguiente forma:

```
void release_lock(pthread_mutex_t *mutex, int use_tsx) {
    if (use_tsx) {
        // Si la ejecución es transaccional...
        if ((*mutex).__data.__lock == 0) {
```

```
                // y no se tuvo que coger el cerrojo, la transacción
                // termina con éxito.
                _xend();
            } else {
                // De lo contrario, la transacción ha fallado y se libera
                // el cerrojo que se adquirió.
                pthread_mutex_unlock(mutex);
            }
        } else {
            // Si la ejecución no es transaccional, simplemente se libera el cerrojo.
            pthread_mutex_unlock(mutex);
        }
    }
```

3.2.4. Modelo de paralelismo con hilos

Ya hemos hablado de que la aplicación sigue un modelo de paralelismo a través de hilos tradicionales, un modelo que la memoria transacional nos permite paralelizar. La ejecución con cerrojos de exclusión mutua sigue un modelo similar, sin olvidar que en el momento en que se necesite acceder al recurso compartido, la ejecución se serializaría. Con la memoria transaccional podemos evitar este problema sin perder la corrección del programa.

En el siguiente diagrama se describe este modelo que, como decíamos, simula un servidor de alto rendimiento que tiene que servir lecturas y escrituras de datos aleatorias desde una tabla compartida.

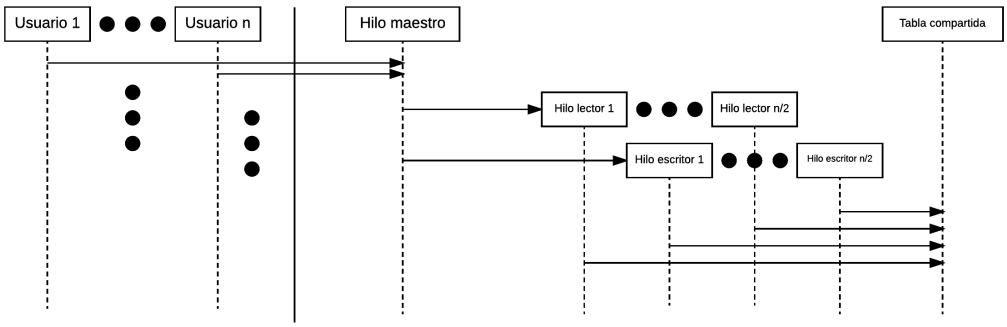

Figura 3.1: Diagrama que describe el modelo de paralelismo con hilos del benchmark sintético

En el diagrama vemos que un número indeterminado de usuarios realizan peticiones que llegan hasta el hilo principal de nuestra aplicación sintética. En nuestra ejecución, estos usuarios se modelan como iteraciones en las que se realiza alguna tarea. Después, el hilo principal es el encargado de lanzar los hilos que realizan las tareas. En nuestro modelo,

dividimos a la mitad entre escrituras y lecturas. Todos estos hilos acceden simultáneamente a una estructura compartida que almacena la información sobre la que se trabaja. En nuestro caso, es una tabla hash.

3.3. Otras herramientas

Además del propio benchmark, será necesario crear otras utilidades auxiliares para el procesado de los datos generados por éste. Para ello optamos por usar Python, dado que crearemos scripts ad-hoc para cada gráfica que deseemos generar y necesitamos la flexibilidad y agilidad que proporciona este lenguaje. Trataremos de reutilizar en la medida de lo posible los algoritmos que creemos para el procesado de los CSV dividiéndolo en varias etapas (procesado inicial, transformación, generación de graficas). El estado final de estos scripts se puede encontrar en el código anexo al trabajo.

3.4. Configuración del entorno

Para el desarrollo de este trabajo es necesario contar con una máquina que disponga de procesador Intel suficientemente reciente como para ejecutar las instrucciones necesarias de la extensión TSX. La Escuela puso a mi disposición una máquina que cumple dicho requisito y que tiene las siguientes características [14] [5] [19] [4]:

- Procesador:
  - Fecha de lanzamiento: 2 de junio de 2015
  - Modelo: Intel(R) Core(TM) i7-5775C CPU @ 3.30GHz
  - Tamaño caché: 6 MB
  - Núcleos: 4
  - Máxima potencia permitida: 65W
  - Frecuencia DRAM: 1600MHz
  - Hilos de ejecución: 8 (Hyper-Threading, 2 procesos por núcleo físico)
  - Arquitectura: x86_64
  - Litografía: 14 nm
  - Caché L1 datos: 4x32KB (8-way)
  - Caché L1 instrucciones: 4x32KB (8-way)
  - Caché L2: 4x256KB (8-way)
  - Caché L3: 6MB (12-way)
  - Caché L4: 128MB
  - Tamaño de línea de caché: 64 bytes
- GCC: `gcc (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4`
- Memoria RAM: DDR3 4GB
- Kernel: `Linux version 3.19.0-25-generic (buildd@lgw01-20) (gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1) ) #26~14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015`

Es interesante fijarse en el primer nivel de caché, pues es en dicho nivel en el que el procesador lleva el conteo de líneas de caché leídas y editadas durante transacciones para detectar los conflictos [22]. Vemos que tenemos en total 256KB de memoria caché de primer nivel, siendo la mitad para datos y la otra mitad para instrucciones.

Como veremos, esta limitación afecta al rendimiento de las ejecuciones con memoria transaccional. También afectará, por tanto, el tamaño de línea de caché que en este modelo está limitado a 64 bytes. Además, el caché tiene una asociatividad de ocho, que teniendo en cuenta el *multithreading* en procesadores Intel como éste que permite que haya más de un hilo compartiendo un mismo núcleo del procesador, hace que la asociatividad de la caché pueda ser cercana a cuatro en la práctica.

Capítulo 4

Modelado de memoria transaccional hardware con una aplicación sintética

Este trabajo comenzó con una introducción en la que se exponían las motivaciones del mismo. A continuación, vimos la introducción al concepto de memoria transaccional para gestionar la concurrencia, concretando en la memoria transaccional por hardware y la implementación de Intel en sus procesadores de nueva generación de este concepto a través de las extensiones TSX. Justo después continuamos con el análisis y diseño de la aplicación de benchmark sintético necesaria para, usando estas extensiones TSX, cumplir con los objetivos del trabajo.

Ahora, el paso lógico es entrar en la sección en que usaremos la aplicación creada para modelar y caracterizar el comportamiento de este acercamiento para gestionar el acceso concurrente a recursos compartidos evitando bloqueos en la medida de lo posible sin poner en juego la corrección de los programas: la memoria transaccional a través de las instrucciones TSX en procesadores Intel.

4.1. Descripción de la aplicación de benchmark sintético creada

Se ha realizado una aplicación sintética diseñada específicamente para probar el comportamiento de las transacciones frente a una aproximación más tradiccional con cerrojos de exclusión mutua, y ver en qué condiciones merece la pena usar memoria transacciona hardware en lugar de cerrojos bajo la implementación que Intel ha puesto a disposición de los consumidores.

4.1.1. Simulando una base de datos

Esta aplicación, que denominaremos `TablaHash`, emula una tabla a la que se accede aleatoriamente, tratando de simular el comportamiento que puede tener una base de datos en un servidor web u otras aplicaciones de gran carga concurrente. Como se expuso anteriormente, entendemos que éste puede ser un buen caso de uso para la memoria transaccional.

Esta tabla está compuesta de dos dimensiones. Cada elemento de la tabla contiene un valor y forma parte de una lista enlazada. El número de listas enlazadas que forman la tabla lo denominaremos **tamaño** de la tabla, mientras que la cantidad de elementos que forman cada lista enlazada será el **número de entradas** de la lista. Nos referiremos a estas dimensiones con los nombres anteriores durante el resto del trabajo.

Lo primero que hace la aplicación en cada ejecución es rellenar esta tabla con valores aleatorios. Con esta aleatoreidad pretendemos evitar la secuencialidad en los accesos a caché y así simular patrones de acceso aleatorios como los que podemos ver en servidores web de alto rendimiento.

Después, lanzará un número de hilos de búsqueda y escritura de valores en esta tabla de forma simultánea, con el objetivo de estudiar el comportamiento en una situación de conflictos entre lecturas y escrituras. Este proceso se repetirá dos veces: primero usando un cerrojo de exclusión mutua sobre la tabla, y después usando la memoria transaccional hardware que nos proporciona el procesador.

4.1.2. Comparativa de la ejecución con cerrojos tradicionales y con memoria transaccional

La implementación con cerrojos de exclusión mutua es muy tradicional: cuando el hilo va a acceder a la sección crítica, que en este caso son los accesos a la tabla, pide el cerrojo. Si nadie lo tiene en ese momento, lo adquiere y entra en la sección crítica bloqueando el acceso al resto de hilos para evitar cualquier tipo de conflicto. Tras realizar su tarea en el recurso compartido, el hilo sale de la sección crítica liberando el cerrojo y permitiendo que uno de los demás hilos pueda adquirirlo para acceder a la sección crítica. Nada nuevo.

En cambio, la implementación que aprovecha la memoria transaccional hardware no bloquea el acceso al resto de hilos al entrar en la sección crítica. Aunque el código es el mismo para los hilos, las funciones que gestionan la adquisición y liberación de los cerrojos pueden usar directamente un cerrojo de exclusión mutua tradicional, que es lo que hace en el caso anterior; o bien puede iniciar una transacción con las instrucciones de la extensión TSX, concretamente las RTM en nuestro benchmark.

En caso de que todo vaya bien, tendríamos potencialmente varios hilos accediendo a un mismo recurso compartido con la garantía de que no se han producido conflictos, ya sea porque todos estaban leyendo, porque se leyeron líneas de caché distintas a las que se editaron, o porque nadie editó a la vez elementos en la misma línea de caché. La clara ventaja de este acercamiento es que esta optimización de la detección de conflictos ha sido totalmente

transparente para nosotros, ya que confiamos en el propio procesador para realizarla de manera óptima, y además se ejecuta directamente sobre hardware. El coste a la hora de programarlo ha sido el mismo que al configurar un cerrojo de bloqueo mutuo global.

Sin embargo, si la transacción va mal y resulta ser abortada, el procesador anulará todos los cambios que haya realizado nuestro hilo de forma transparente para el resto de hilos, sin que nadie pueda llegar a ver los cambios. Además, podremos ver el motivo del fallo y recuperarnos de él. En nuestro caso, aunque usamos las instrucciones RTM que permiten soluciones más creativas y adaptadas al problema y al tipo de fallo de la transacción, la solución que aplicamos es repetir la operación usando un cerrojo de exclusión mutua tradicional, que nos da garantía de que la operación se realizará correctamente. Esto no es problema pues el modelo de memoria transaccional de Intel para TSX, al ser de aislamiento fuerte, detecta los conflictos en los datos, vengan o no de código transaccional. Usamos las instrucciones RTM debido a que nos permiten obtener estadísticas sobre la razón de los fallos, necesarias para la caracterización.

4.2. Parámetros de la aplicación

La aplicación está diseñada para la realización de los benchmarks, con lo que cuenta con cinco parámetros que se pueden configurar para probarla en distintas condiciones:

- Número de hilos: Define la cantidad de hilos de lectura y de escritura que se pondrán a trabajar contra la estructura compartida.
- Número de iteraciones: Define la cantidad de operaciones que hará cada hilo antes de terminar.
- Número de entradas: Define la cantidad de elementos que forman cada lista enlazada de la tabla.
- Tamaño de la tabla: Define la cantidad de listas enlazadas que formarán la tabla.

Además de estos parámetros, se debe indicar un nombre de fichero donde el programa, tras su ejecución, escribirá una línea en formato CSV indicando los parámetros con los que se le ha llamado, el número de transacciones que se realizaron correctamente, el número de transacciones que fallaron, la cantidad de transacciones que han fallado por cada tipo de fallo posible, el tiempo que tardó la ejecución con cerrojos de exclusión mutua, y el tiempo haciendo uso de la memoria transaccional. Por último, se le puede indicar al programa que sea verboso: en ese caso mostrará algunos detalles sobre el tipo de hilos que ejecuta y mostrará más información intermedia durante el proceso por la salida estándar. En caso de un error en la introducción de los parámetros, el programa imprimirá cuáles son éstos para recordarlo.

Es interesante recordar en este punto el análisis realizado en el capítulo anterior en el que remarcábamos las implicaciones de los parámetros sobre las dimensiones de la tabla sobre los tipos de errores.

4.3. Ejecución del benchmark y su entorno

Antes de comenzar a analizar los resultados de la aplicación sintética y extraer información a partir de los datos, pasaremos a hacer algunas consideraciones sobre la ejecución del mismo que pueden influenciar los datos, de forma que el posterior análisis sea completo.

4.3.1. Tipos de ejecuciones: trabajo variable y trabajo constante

Con esta aplicación sintética se han realizado varios benchmarks. En primer lugar, el benchmark era ejecutado de forma que el valor que se pasaba como número de iteraciones no era el número de iteraciones total que se ejecutaría como suma de todo el trabajo de los hilos, sino que cada hilo ejecutaría el número de iteraciones indicado. A esta forma de ejecución la denominaremos "de carga variable", pues la carga de trabajo total varía en función del número de hilos.

Con un sencillo script en bash, se ejecutó el benchmark con 100.000 iteraciones para cada hilo y aumentando progresivamente el resto de dimensiones. El número de hilos subió desde 4 hasta 256 subiendo de cuatro en cuatro, y las dos dimensiones de la tabla variaron entre 16 y 512 aumentando en potencias de 2. De esta forma obtuvimos información sobre el comportamiento en cualquier combinación de estas variables dentro de estos rangos. El benchmark se repitió en las mismas condiciones cinco veces y los datos con los que se generan las gráficas mostradas a continuación en cuanto a ejecuciones con carga variable son la media aritmética de estas cinco ejecuciones.

Posteriormente, se añadió una opción al benchmark de forma que el número de iteraciones que se le hace llegar no representa el trabajo de cada hilo, sino el trabajo total realizado. Por tanto, el valor que se indique como número de iteraciones se dividirá entre el número de hilos, manteniendo constante la carga de trabajo independientemente del número de hilos entre los que se divida. A esta forma de ejecución la denominamos "de carga constante".

4.3.2. Entorno de experimentación

Como ya comentamos, estos experimentos han sido realizados sobre una máquina virtual proporcionada por la Escuela. Esta máquina tenía instalado un hipervisor que permitía compartir el hardware entre varios usuarios de forma transparente. Dado que los benchmarks realizados no se tratan de una simulación, sino que se ejecutan sobre el procesador real, que es compartido con otros usuarios, puede que algunos datos se hayan visto afectados por otras tareas, ya sean de otros usuarios, del hipervisor o del propio sistema operativo.

Para minimizar los efectos que puedan tener estas perturbaciones, las ejecuciones se intentaron realizar en momentos de bajo uso de la máquina, como noches o fines de semana. Sin embargo, es imposible garantizar que la ejecución no se haya visto afectada en algunos puntos por otras tareas como las que indicábamos anteriormente. Esto puede explicar ciertos valores atípicos recogidos.

4.4. Caracterización del comportamiento de la ejecución con memoria transaccional

Para caracterizar el comportamiento de la memoria transaccional partiremos de la principal razón que afecta al rendimiento de este tipo de programas: los fallos que abortan las transacciones. Cuando una transacción es abortada, se hace necesario repetir la ejecución y, en nuestro caso, esta repetición se realiza con un cerrojo global tradicional. Por tanto, variaremos los parámetros que afectan a la forma de la tabla para poder caracterizar el comportamiento de la memoria transaccional.

4.4.1. Sensibilidad al tamaño de la tabla

Procediendo con el análisis de los datos recabados, comenzaremos estudiando los efectos que produce la variación del tamaño de la tabla en el comportamiento de la ejecución con memoria transaccional. Hipotéticamente, los cambios en esta variable deberían reducir el número de conflictos, puesto que al ser aleatorios los accesos a datos y aumentar el número de entradas del vector, se reduce la probabilidad de que haya más de un hilo al mismo tiempo usando la misma línea de caché. Concretamente, los fallos que se verían reducidos al aumentar el tamaño de la tabla serían los fallos de capacidad generados por fallos en la caché.

Podemos ver este efecto cuando duplicamos el tamaño de la tabla, pasando de 32 a 64 elementos. En este caso vemos en las gráficas siguientes que el speedup aumenta de forma generalizada. Estas gráficas usan datos de ejecuciones de carga variable con lo que no podemos comparar las ejecuciones con distinto número de hilos entre sí pero sí podemos comparar

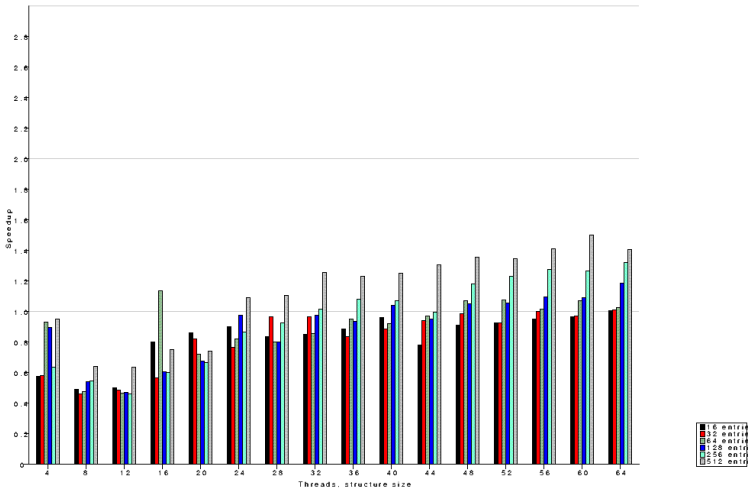

Figura 4.1: Speedup para una tabla de tamaño 32 (pocos hilos, carga variable)

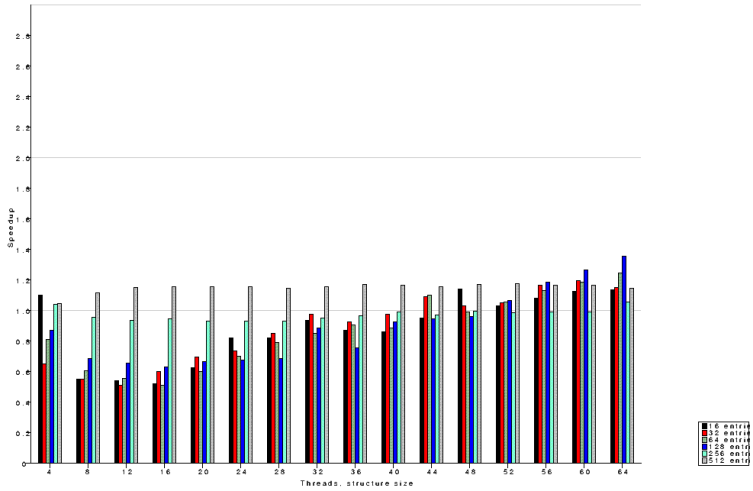

Figura 4.2: Speedup para una tabla de tamaño 64 (pocos hilos, carga variable)

También vemos el mismo efecto cuando hacemos la misma comparativa con un número mayor de hilos.

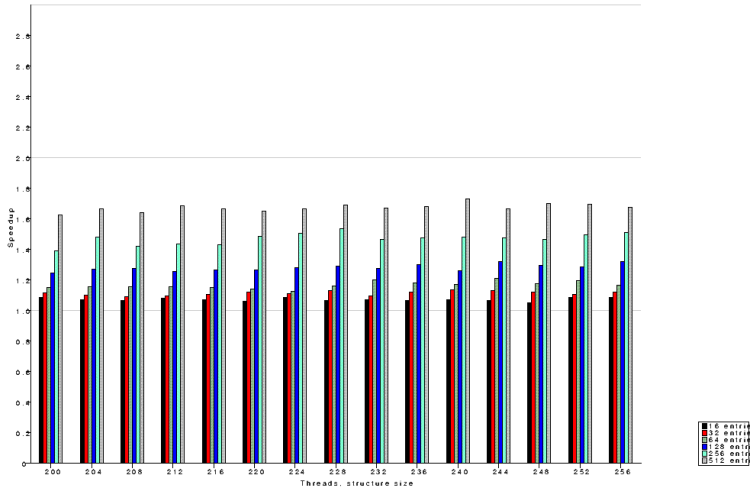

Figura 4.3: Speedup para una tabla de tamaño 32 (muchos hilos, carga variable)

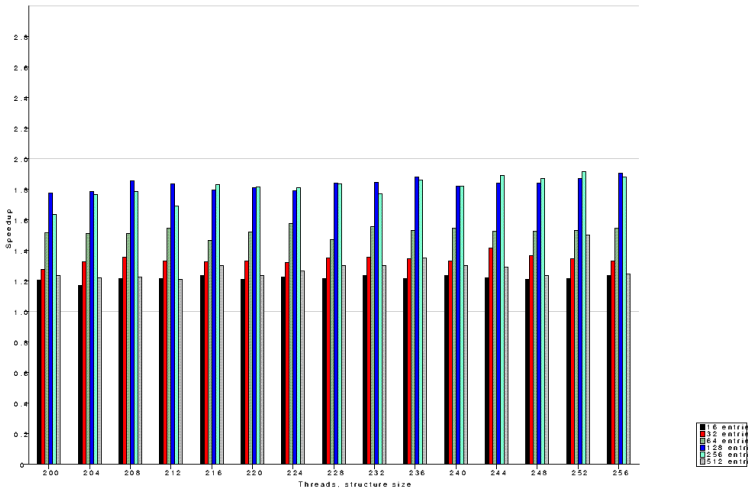

Figura 4.4: Speedup para una tabla de tamaño 64 (muchos hilos, carga variable)

Las gráficas representan también cómo evoluciona el rendimiento al cambiar la longitud de la lista enlazada, que provoca un efecto que contrarresta éste y por eso en mayores longitudes de lista enlazada el rendimiento no aumenta tanto o se reduce. De estas consideraciones hablaremos en el siguiente apartado.

En estas gráficas vemos que para estos tamaños, el overhead que supone la memoria transaccional no nos da un rendimiento mejor que el uso de un cerrojo tradicional. Sin embargo, con un número mayor de hilos vemos cómo la ejecución con el cerrojo se bloquea, serializándose y dándonos un rendimiento peor que con memoria transaccional. Vemos que este efecto se acentúa cuanto mayor sea el número de hilos y también cuanto mayor sea la estructura.

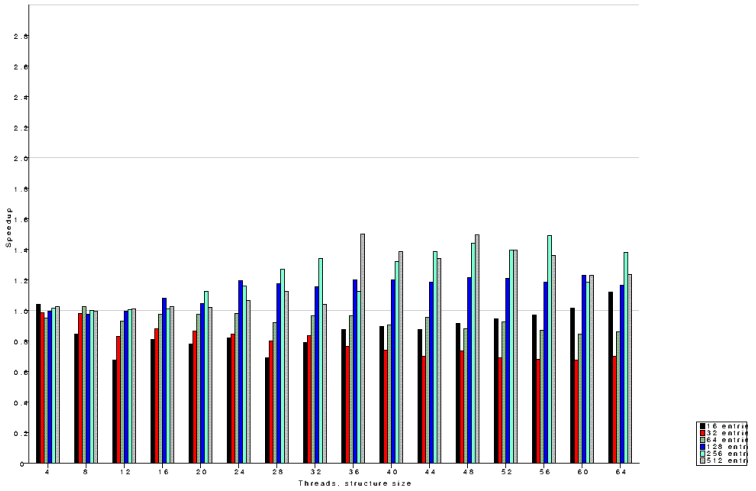

Figura 4.5: Speedup para una tabla de tamaño 128 (pocos hilos, carga variable)

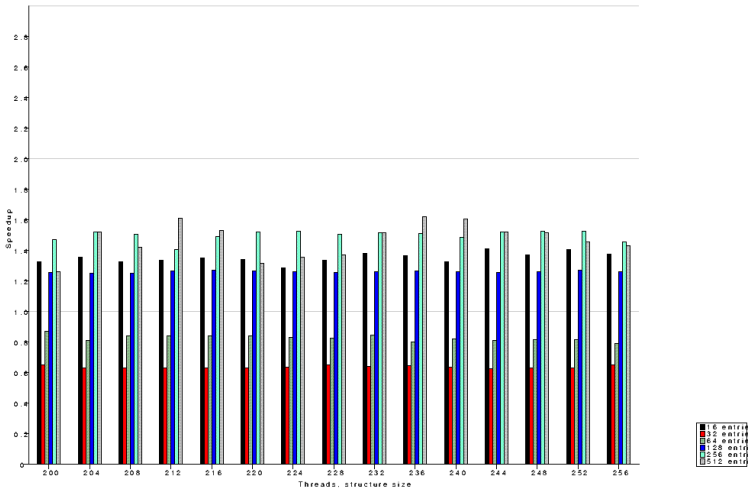

Figura 4.6: Speedup para una tabla de tamaño 128 (muchos hilos, carga variable)

Cuando volvemos a duplicar el tamaño desde 64 hasta 128, vemos que sí hay una ligera mejora pero entran en acción otros efectos que tienden a equiparar el rendimiento de ambas soluciones. Si vamos al caso extremo, que para nosotros es la tabla de tamaño 512, vemos que según vamos subiendo el tamaño de la tabla llega un momento que no se ven mejoras con el uso de la memoria transaccional, sino que se llega a un límite con un speedup constante de algo más del 0,9. Podemos interpretarlo como que el mayor número de filas en la tabla provoca un mayor uso de líneas de caché, con lo que la caché es invalidada más rápidamente,

y la penalización en el rendimiento de las transacciones abortadas hace que el rendimiento de la memoria transaccional no supere al de los cerrojos, que debido al gran número de fallos de las transacciones, pasa a ser más eficiente que la memoria transaccional, aunque no sustancialmente.

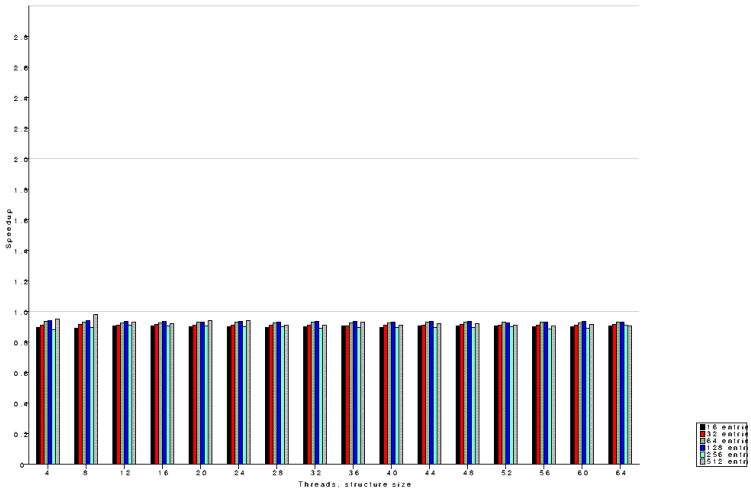

Figura 4.7: Speedup para una tabla de tamaño 512 (pocos hilos, carga variable)

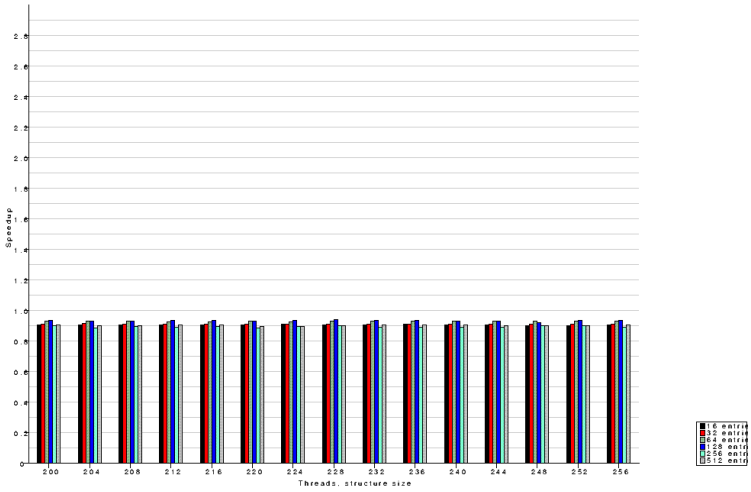

Figura 4.8: Speedup para una tabla de tamaño 512 (muchos hilos, carga variable)

Para tener una visión completa de las características de esta ejecución vamos a ver los datos recogidos con carga constante, donde sí podemos comparar entre sí los resultados con

distinto número de hilos, además de que podemos comparar también la cantidad total de errores gracias a que el trabajo total es siempre el mismo independientemente del número de hilos entre los que se divida.

Para los mismos tamaños que comentamos con anterioridad, pero con trabajo constante, podemos ver las siguientes gráficas.

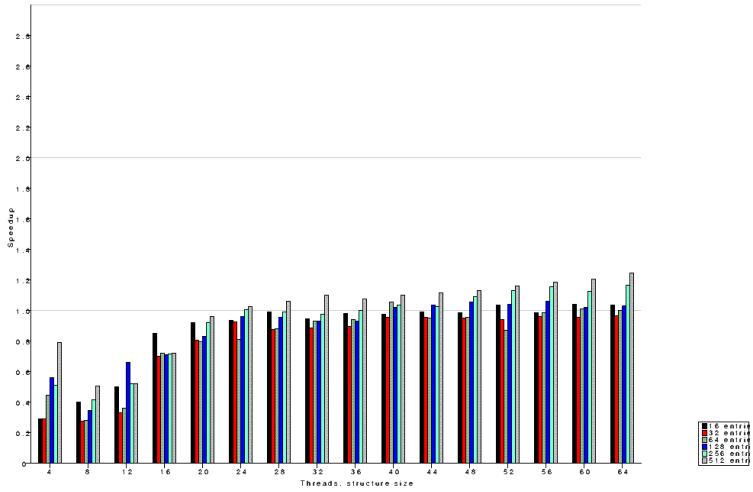

Figura 4.9: Speedup para una tabla de tamaño 32 (pocos hilos, carga constante)

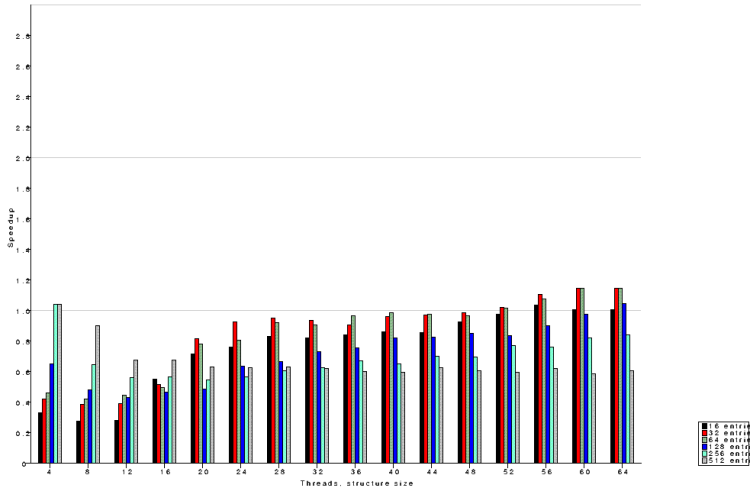

Figura 4.10: Speedup para una tabla de tamaño 64 (pocos hilos, carga constante)

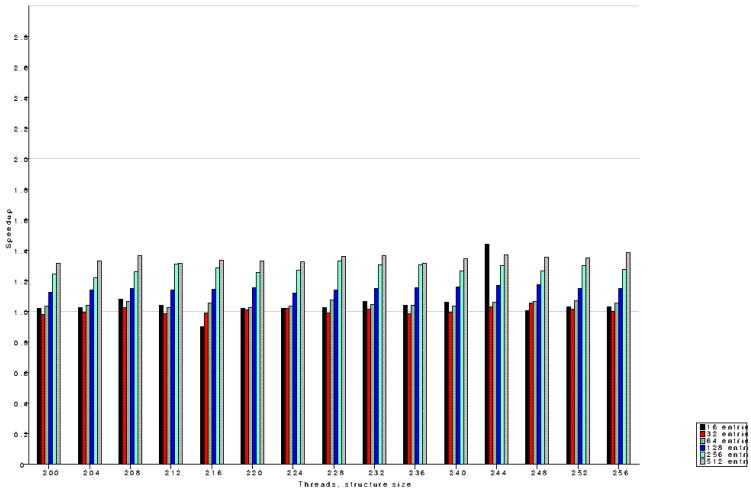

Figura 4.11: Speedup para una tabla de tamaño 32 (muchos hilos, carga constante)

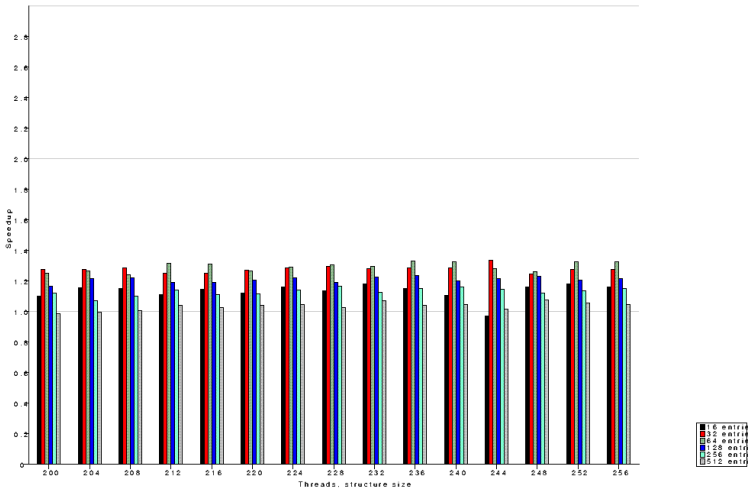

Figura 4.12: Speedup para una tabla de tamaño 64 (muchos hilos, carga constante)

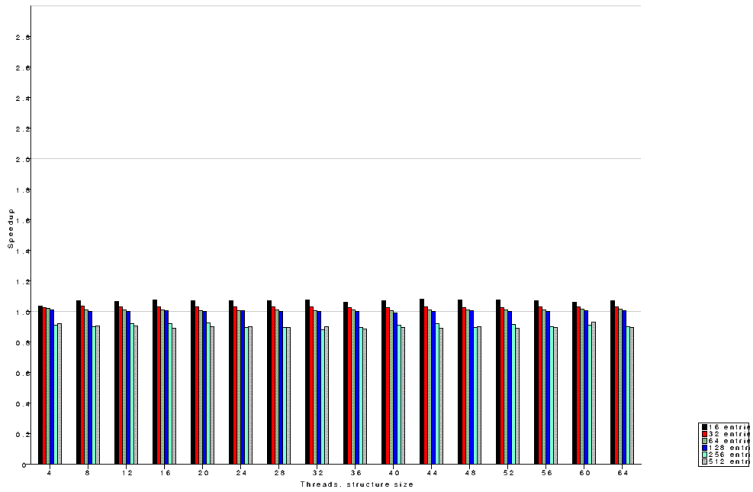

Figura 4.13: Speedup para una tabla de tamaño 512 (pocos hilos, carga constante)

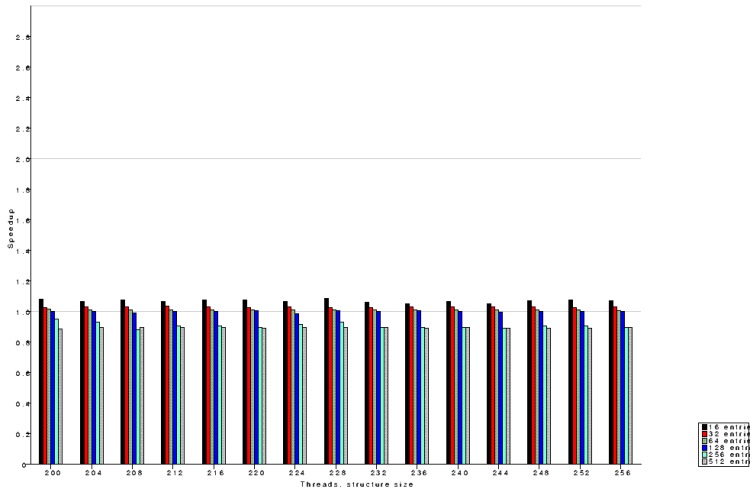

Figura 4.14: Speedup para una tabla de tamaño 512 (muchos hilos, carga constante)

Aunque las gráficas se pueden consultar en el apéndice correspondiente o en el disco adjunto al trabajo, los speedups para nuestra ejecución constante con 128 millones de iteraciones y 64 millones de iteraciones son sustancialmente los mismos. Lo mismo sucede para los errores aunque, claro está, cambia el valor absoluto; pero las gráficas nos muestran la misma forma. Vemos que con estas cantidades de trabajo, aumentar el número de iteraciones quizás sirva para minimizar el efecto de agentes externos en los tiempos medidos, pero no vemos un comportamiento distinto.

Como comentábamos antes, nos fijaremos en los tamaños pequeños de lista enlazada puesto que después estudiaremos cómo afecta la variación en este parámetro. Viendo esos datos vemos que con esta mayor carga de trabajo se mantiene la tendencia de mejora de rendimiento a mayores tamaños de tabla. Con estos datos podemos ver también cómo ganamos en rendimiento a la ejecución con cerrojos cuando tenemos un alto número de hilos. También vemos el mismo efecto que comentábamos anteriormente cuando la tabla aumenta mucho de tamaño: los speedups se hacen constantes y cercanos a 1, independientemente del número de hilos.

Si nos fijamos en algunas gráficas de errores, vemos que la cantidad de errores se reduce a medida que incrementa el tamaño de la tabla, como especulábamos.

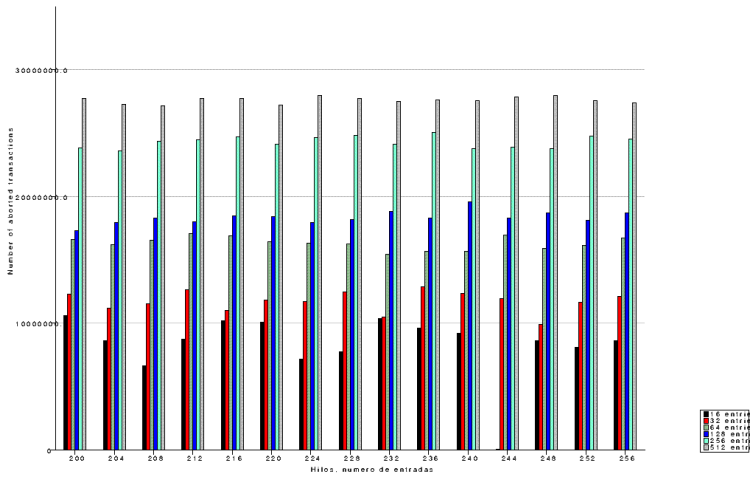

Figura 4.15: Fallos totales para una tabla de tamaño 32 (muchos hilos, carga constante)

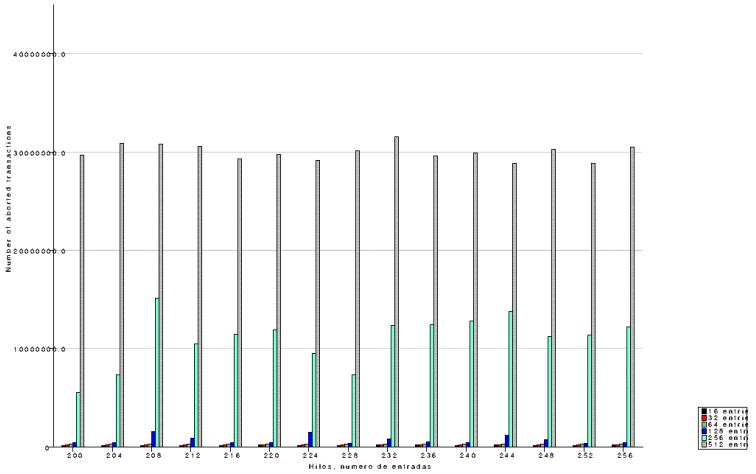

Figura 4.16: Fallos totales para una tabla de tamaño 512 (muchos hilos, carga constante)

En las dos gráficas anteriores vemos como al pasar de una tabla de tamaño 32 a una de 512, el número de fallos baja. Hay que tener en cuenta que los fallos al aumentar el tamaño de la lista enlazada los comentaremos después, y que esas dos gráficas tienen escala distinta.

4.4.2. Sensibilidad al número de entradas de la lista enlazada

Ahora pasamos al siguiente parámetro que afecta al rendimiento de la memoria transaccional: el número de entradas de cada lista enlazada. Como explicábamos, nuestra tabla está formada por un vector de un tamaño determinado, donde cada nodo apunta a un elemento de una lista enlazada. Por tanto, la variable que estudiaremos ahora es la longitud de cada una de estas listas enlazadas que cuelgan de los elementos del vector principal.

Queremos comprobar si, como pensamos, a medida que aumente la longitud de la lista enlazada, los hilos estarán más tiempo usando cada línea de caché hasta encontrar el dato con el que tienen que trabajar, con lo que es más probable que haya conflictos en una misma línea de caché. Esta situación provocará un aumento de los fallos de conflicto, que abortarán las transacciones, provocando penalizaciones en el rendimiento.

Lo primero que observamos a este respecto es que con un tamaño de tabla grande de forma que no nos veamos afectados por los fallos de conflicto, está claro que a medida que aumenta el número de elementos de la lista enlazada se dispara el número total de fallos.

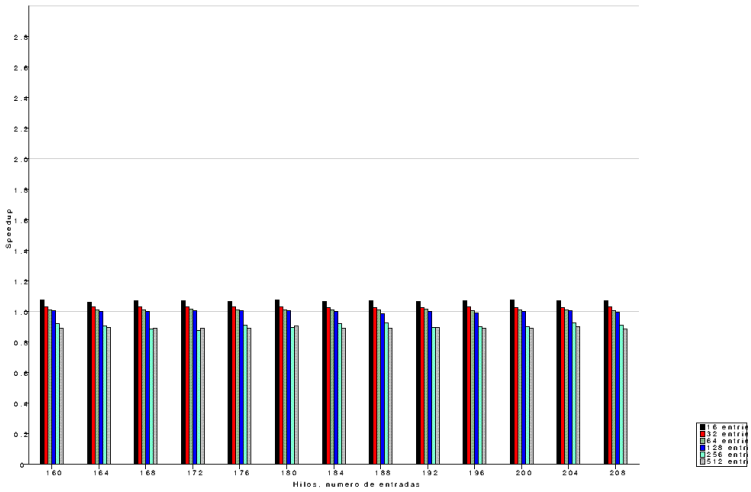

Figura 4.17: Fallos totales para una tabla de tamaño 512 (muchos hilos, carga constante)

Este comportamiento se mantiene con un número menor de hilos también.

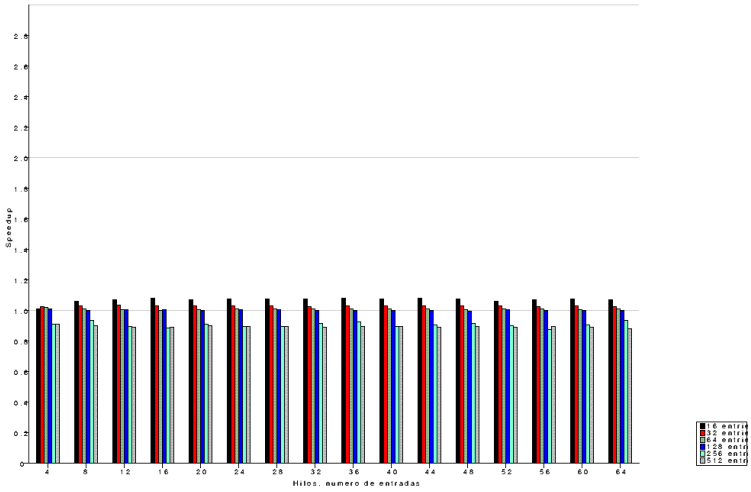

Figura 4.18: Fallos totales para una tabla de tamaño 512 (pocos hilos, carga constante)

Si reducimos el tamaño de la tabla ya no se observan estos patrones, posiblemente por la influencia de los fallos de capacidad como ya hemos descrito anteriormente.

4.4.3. Comparativa con el rendimiento usando cerrojos tradicionales

El tercer análisis que podemos realizar para caracterizar la memoria transaccional, y para poder entender qué aplicaciones reales podría tener la memoria transaccional, será comparar el rendimiento de éste método frente a la ejecución con cerrojos de exclusión mutua.

Hay algunos casos en los que vemos una mejora de rendimiento, con un speedup mayor a uno cuando enfrentamos el tiempo de ejecución de los cerrojos frente a la memoria transaccional. Por ejemplo, como hemos visto, un mayor tamaño de tabla otorga una ventaja a la memoria transaccional. Esto puede significar que en tablas reales de gran tamaño podríamos obtener un beneficio real aplicando memoria transaccional, aunque habría que probar esta situación en más profundidad para afirmarlo con seguridad.

También vemos que hay una oportunidad de mejora de rendimiento cuando el número de hilos aumenta sustancialmente. Por ejemplo, a continuación vemos dos gráficas con mismo tamaño de tabla, siendo la primera con un número bajo de hilos mientras que la segunda muestra los datos para una cantidad de hilos mucho mayor. Podemos ver cómo manteniendo todas las demás variables iguales, el rendimiento sube debido a que los hilos se bloquean entre sí y no pueden aprovechar el paralelismo real del procesador.

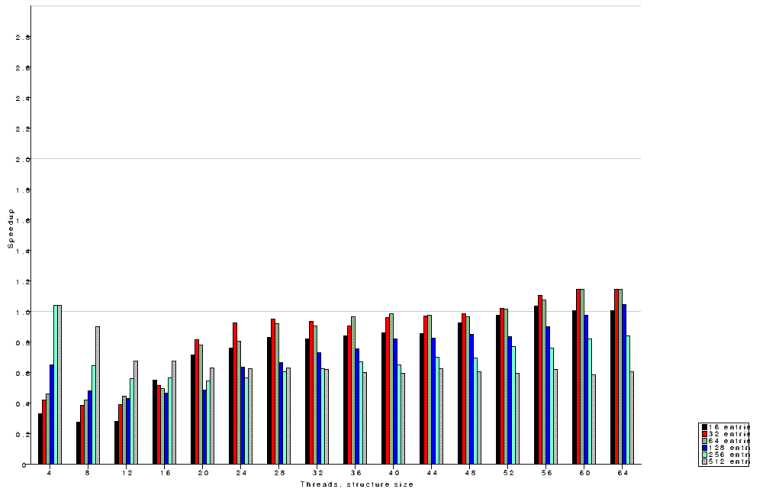

Figura 4.19: Speedup para una tabla de tamaño 64 (pocos hilos, carga constante)

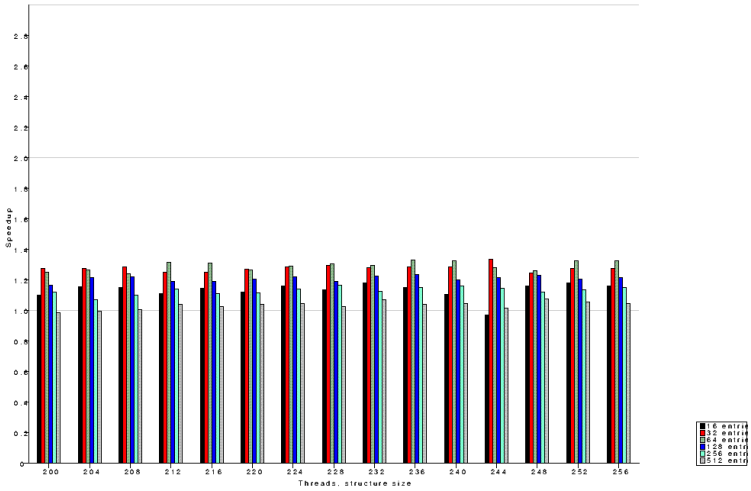

Figura 4.20: Speedup para una tabla de tamaño 64 (muchos hilos, carga constante)

Capítulo 5

Aplicaciones paralelas con estructuras compartidas irregulares

Además del caso que se decidió que vertebrara este trabajo, el de un servidor web, se investigaron otras posibles aplicaciones donde la memoria transaccional pudiera ser de utilidad. Una de ellas es la paralelización en aplicaciones con estructuras irregulares.

Este tipo de estructuras se caracterizan porque no se corresponden con estructuras fácilmente paralelizables, como tablas o árboles equilibrados. Las formas que suelen tomar son grafos o conjuntos, con un número variable de dimensiones, y sin una forma definida. Además, en algunos algoritmos, incluso la forma de la estructura cambia durante la ejecución.

5.1. Ejemplos de aplicaciones con estructuras irregulares

Uno de los ejemplos más de moda de este tipo de aplicaciones es el análisis de redes sociales. Este problema suele modelarse con un grafo donde cada vértice es una persona en la red social y cada arista es una relación entre dos personas. Con esta sencilla premisa se pueden llegar a generar estructuras muy complejas debido al gran número de datos con los que se suelen realizar estas investigaciones. Además, la estructura puede complicarse con pesos en las aristas o más información en los vértices. El análisis de estas redes puede usarse para fines muy diversos, como encontrar el comportamiento de la difusión de información en ellas [1] o analizar su estructura durante eventos concretos [8].

Otro ejemplo es en análisis de estructuras en dos o tres dimensiones para hacerlas más simples con el objetivo de renderizarlas, imprimirlas, etc. Sin lugar a dudas, hay muchos

más ejemplos de este tipo de estructras, y lo que tienen en común es que es muy costoso paralelizar algoritmos que necesitan trabajar con ellas y que es aún más complicado crear frameworks que permitan generalizar esta paralelización para poder usarlo en varios problemas y estructuras irregulares distintas. A continuación, veremos un framework que intenta alcanzar este objetivo.

5.2. Framework Galois

5.2.1. Introducción

El proyecto Galois, desarrollado por un equipo en la Universidad de Texas en Austin es un sistema que facilita la paralelización de código en C++ o Java, aunque sus últimas versiones solo están disponibles para C++. El sistema se enfoca en hacer posible y sencilla la paralelización en computaciones sobre estructuras irregulares, como las que normalmente se encuentran en problemas que involucran estructuras irregulares de datos enlazados que cambian o no su forma durante la ejecución. Algunos ejemplos de este tipo de problemas son la triangulación de Delaunay o la generación de árboles recubridores a partir de un grafo que contengan todos los vértices de éste pero sin ciclos.

Galois consigue este objetivo añadiendo unas nuevas estructuras de datos que ayudan a definir la naturaleza de los datos que representan de forma que el sistema sepa hasta qué punto y con qué condiciones se puede paralelizar el algoritmo. Estos metadatos se hacen llegar al motor de Galois, quien se asegura de que la ejecución sea segura y todo lo rápida que se lo permitan las restricciones indicadas.

Pensamos que el cuello de botella en Galois está en este motor en tiempo de ejecución, ya que implementa en software y en tiempo de ejecución un sistema que lleve cuenta de posibles condiciones de carrera o conflictos entre hilos, y una serie de cerrojos de exclusión mutua y sistemas de recuperación que permitan que la ejecución paralela sea segura. Tradicionalmente se ha visto que este tipo de sistemas tienen una gran penalización en el rendimiento. Queremos saber cuán grande es esta penalización, y si puede ser mejorada haciendo uso de las nuevas instrucciones TSX de memoria transaccional hardware que implementan los procesadores de Intel dirigidos a consumidores a partir de la 6ª generación.

5.2.2. El “Tao” de la programación paralela

El proyecto Galois es la materialización de una investigación realizada por algunos de los mismos autores en los que se proponen convertir la computación paralela en una ciencia. En el paper que se generó como resultado de esa investigación [21] se realiza una categorización de los algoritmos en base a tres propiedades, y la conclusión que sacan es que se puede unificar la técnica para desarrollar algoritmos paralelos decidendo a posteriori si la paralelización se puede realizar en tiempo de compilación o en ejecución, y de qué manera; pensando antes en la estructura de los datos que en el algoritmo que se quiere hacer sobre ellos.

Las propiedades de este "Tao", en base a las que se analizan los algoritmos para categorizar su forma de paralelismo, son:

- **T**opología: La topología se enfoca en el análisis de la estructura sobre la que se centra la ejecución. En base a esta propiedad se categorizan las estructuras según su complejidad de Kolmogorov [16], que está relacionada con la dificultad que supone describirlas.
- Nodos **a**ctivos: Descripción de cómo un nodo se convierte en activo y del orden en que deben procesarse. Puede que el algoritmo exija un orden o no, y puede que los nodos se activen en función de la topología o en función de la información que contienen.
- **O**perador: Los operadores se categorizan en función de cómo modifican el grafo. Pueden modificarlo añadiendo o borrando vértices y aristas o actualizando valores, puede actualizar valores del grafo pero sin cambiar la topología, o simplemente puede no modificarla.

En base a esta categorización, algunas de las estructuras que define Galois [7] enfocadas a paralelizar estructuras irregulares son las siguientes. Todas ellas pueden ser compartidas entre hilos con seguridad:

- `Galois::Graph::FirstGraph`: Un grafo que permite la mutación concurrente del mismo.
- `Galois::Graph::LC_CSR_Graph`: Un grafo que no permite ser mutado. Está optimizado para cargarse en memoria y ser muy rápido.
- `Galois::Bag`: Una generalización de un conjunto que permite elementos repetidos, pero sin orden.

También, se definen entre otros los siguientes operadores:

- Iterador para conjuntos desordenados: Itera sobre un conjunto sin garantías sobre el orden en el que sus elementos serán procesados. Permite que una iteración añada nuevos elementos al conjunto durante la ejecución. Acepta metadatos que indican condiciones sobre el paralelismo que se puede aplicar en la ejecución de las iteraciones del bucle.
- Iterador para conjuntos ordenados: Similar al anterior, pero además exige que siempre se procese el elemento mínimo del set primero. Se puede pasar un parámetro opcional a este operador que indica la política a seguir. Algunos ejemplos son cola de prioridad, FIFO, LIFO...

Estas estructuras y operadores imponen restricciones y aportan metadatos al framework sobre cómo deben ser procesados estos datos cuando se ejecutan de forma concurrente, estableciendo limitaciones que aseguran la corrección del software.

5.2.3. Trabajo futuro en Galois

Actualmente, podemos ver que Galois está usando un acercamiento similar a la memoria transaccional por software para alcanzar sus objetivos, manteniendo un histórico de

operaciones que el motor en tiempo de ejecución comprueba rutinariamente en busca de inconsistencias. Creemos que esta forma de atacar el problema es un cuello de botella que se puede resolver sin cambios muy graves con memoria transaccional por hardware.

Si conseguimos introducir instrucciones RTM en el motor en tiempo de ejecución de Galois de forma que use las extensiones TSX en lugar de su algoritmo software de detección de conflictos, podríamos obtener un paradigma de programación paralela realmente simple que mantenga la seguridad y fiabilidad de los datos pero con una penalización de rendimiento mucho menor que la de los sistemas de paralelización basados en software que se usan a día de hoy.

5.2.4. Trabajo previo sobre memoria transaccional por hardware en Galois

Aunque no es claro a simple vista, analizando el código de Galois se puede ver que ya ellos han realizado algún trabajo previo sobre memoria transaccional hardware, pero no hemos encontrado más referencias a este respecto en la documentación del software o en los papers que han publicado los autores, salvo dos escuetas referencias:

En [17] encontramos la siguiente referencia:

> (...) These characteristics make simple hardware transactional memories (7) unsuitable for our purposes. However, software transactional memories, such as (5, 6), or more advanced hardware approaches (1, 12) may suffice, although their efficiency in the context of long-running transactions must still be studied (...)

> (...) Estas características hacen que las memorias transaccionales hardware simples no sean apropiadas para nuestros objetivos. Sin embargo, las memorias transaccionales software o aproximaciones hardware más avancadas podrían ser suficientes, aunque su eficiencia en el contexto de transacciones ejecutadas durante largo tiempo aún debe ser estudiada (...)

Y en [18], la siguiente:

> (...) It presents important workload characteristics of these programs. One key finding is that the transactions can be quite large, making them less suitable for hardware-based approaches. (...)

> (...) Esto presenta importantes características de la carga de trabajo de estos programas. Un hallazgo clave es que las transacciones pueden ser bastante grandes, haciéndolas menos apropiadas para aproximaciones basadas en hardware. (...)

Para tratar de averiguar qué investigación concreta han llevado a cabo los autores de Galois sobre memoria transaccional, tenemos que explorar el código fuente del proyecto. Lo primero que llama la atención es la bandera `GALOIS_USE_HTM` que podemos configurar al compilar el proyecto, suponiendo que `HTM` significa *hardware transactional memory*. Los cambios que esta bandera realiza en el código es una implementación distinta de la clase `LoopStatistics`, que si no se configura esta bandera, queda definida pero vacía. Al usar la bandera, esta clase recoge los motivos de fallo de las transacciones de forma que se pueda analizar el rendimiento de las ejecuciones con memoria transaccional hardware.

Otro hallazgo esclarecedor es que en la configuración de *CMake* para construir el proyecto se indica una restricción de forma que si se quiere compilar con el flag `GALOIS_USE_HTM`, es necesario que el compilador tenga el identificador `XL`. Este identificador está haciendo referencia al compilador *IBM XL C++* [3], diseñado para optimizar el código para las arquitecturas IBM [26]. Esto tiene sentido, pues algunos procesadores IBM como la serie *POWER8* o los *BlueGene/Q* cuentan con capacidades de memoria transaccional por hardware, aunque conceptualmente distintas a las que proporcionan las extensiones TSX de Intel.

5.2.5. Conclusiones sobre memoria transaccional por hardware en Galois

Dada la falta de publicaciones o información sobre este tema, y basándonos en la poca que hemos encontrado, podemos suponer que los autores no encontraron suficiente el rendimiento de la memoria transaccional en procesadores de IBM como para continuar la investigación por esa línea. Sin embargo, no podemos suponer que las prestaciones y herramientas que proporcionan las instrucciones TSX de Intel repliquen ese rendimiento deficitario.

Aunque esta investigación quedaría fuera del alcance de este trabajo, sí sería interesante ver la complejidad que supondría introducir las instrucciones TSX en el motor de tiempo de ejecución de Galois, evitando los bloqueos que pudieran producir los cerrojos y la penalización en el rendimiento que supone el sistema de recuperación similar a la memoria transaccional por sofware que forman el núcleo de Galois.

Capítulo 6

Conclusiones

La tecnología de memoria transaccional sobre hardware es una de las técnicas más recientes para el tratamiento de acceso a recursos compartidos. Aunque la investigación académica sobre este tema no es nada nuevo, sí lo es el cambio que produce el hecho de que Intel lo ponga a nuestra disposición en sus procesadores dirigidos a consumidores a través de las instrucciones TSX.

La tecnología está en sus primeros momentos pero hemos podido comprobar que está lista para ser utilizada y que proporciona un rendimiento comparable a un lock tradicional cuando no ofrece un rendimiento mejor. En base a los datos proporcionados porla aplicación sintética realizada, vemos esta mejora del rendimiento en ciertos casos como estructuras con muchas filas que eviten las colisiones de caché, o en el uso de un gran número de hilos ya que con la memoria transaccional se evitan bloqueos que serializan la ejecución y por tanto podemos aprovechar mejor el paralelismo de los actuales procesadores multinúcleo.

El análisis que hemos hecho en la introducción a esta tecnología se presenta de utilidad para seguir utilizándola en aquellas aplicaciones en las que, a día de hoy, el uso de memoria transaccional no solo facilita la programación sino que incluso mejora el rendimiento; pero también de cara al día de mañana cuando nuevos procesadores de Intel sigan mejorando los límites técnicos de la memoria transaccional (tamaño de líneas de caché, permitir más operaciones que no aborten la transacción, número de líneas de caché, etc) y su uso comience a ser una opción válida, o incluso la mejor opción, para cada vez más aplicaciones.

Hemos podido comprobar de primera mano cómo la memoria transaccional demuestra su utilidad real, pero sobre todo un gran potencial de cara al futuro, con una apuesta fuerte de una empresa del tamaño de Intel, decidida a llevar esta tecnología a los ordenadores comunes y a añadir la memoria transaccional al cinturón de herramientas de todo programador. Vemos esto como el inicio de una tendencia que, de consolidarse, podría contribuir a continuar el aumento de rendimiento del software a través del hardware, que parece que empieza a ralentizarse en otras áreas; y podría animar a otras empresas que mostraron interés por la memoria transaccional a decidirse a publicar un producto competidor.

6.1. Trabajo futuro

En este trabajo nos hemos enfocado en realizar una prueba muy concreta sobre una tabla relativamente pequeña y con un patrón de accesos aleatorio donde la mitad de ellos son de lectura y la otra mitad de escritura. Por ello sería interesante de cara al futuro continuar con la experimentación y ver cómo se comporta la memoria transaccional en otros tipos de aplicaciones, con otras estructuras compartidas, mayores tamaños, distintas proporciones de tareas de lectura y escritura, etc.

Otra parte que también se presta a la futura investigación y que es dependiente del tipo de aplicación concreta al que se quiera aplicar la memoria transaccional sería explorar posibles estrategias de recuperación cuando usemos RTM, en lugar de caer directamente a un cerrojo global al primer fallo. Otras estrategias podrían ser repetir la transacción esperando un tiempo antes o no antes de caer al cerrojo, o tomar esta decisión dinámicamente teniendo la información sobre el estado y el funcionamiento de la aplicación-

No podemos olvidar tampoco seguir de cerca la evolución de la memoria transaccional hardware para repetir estos experimentos en futuro hardware que nos proporcione un número menor de fallos y por tanto un mayor rendimiento.

Haciendo alusión al anterior capítulo de este trabajo, también es interesante continuar con la investigación sobre Galois y comprobar hasta dónde puede llegar la idea de un framework que generalice el paralelismo, y sobre todo ver si podemos mejorar esta aplicación introduciendo memoria transaccional y de qué forma, evitando la implementación de una pseudo memoria transaccional en software.

Apéndice A

Gráficas generadas

Este apéndice reune todas las gráficas generadas durante el trabajo como complemento a las que son destacadas durante el mismo. Las que ya han sido usadas en el trabajo no aparecerán aquí.

A.1. Ejecución con carga de trabajo variable

En este caso, las gráficas representan un trabajo proporcional al número de hilos lanzados. Todas ellas representan un valor de 100.000 iteraciones por cada hilo. En este caso las gráficas han sido generadas con la media aritmética de 5 ejecuciones iguales.

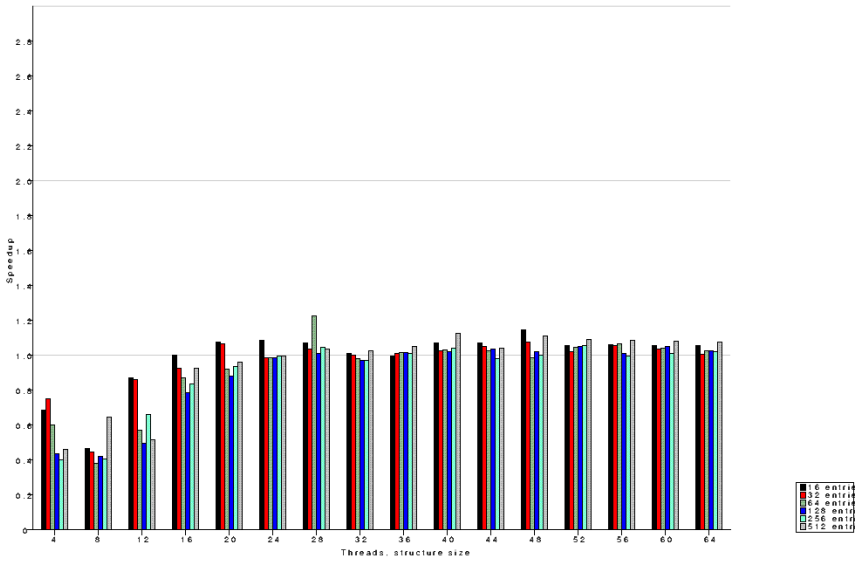

Figura A.1: Speedup para una tabla de tamaño 16

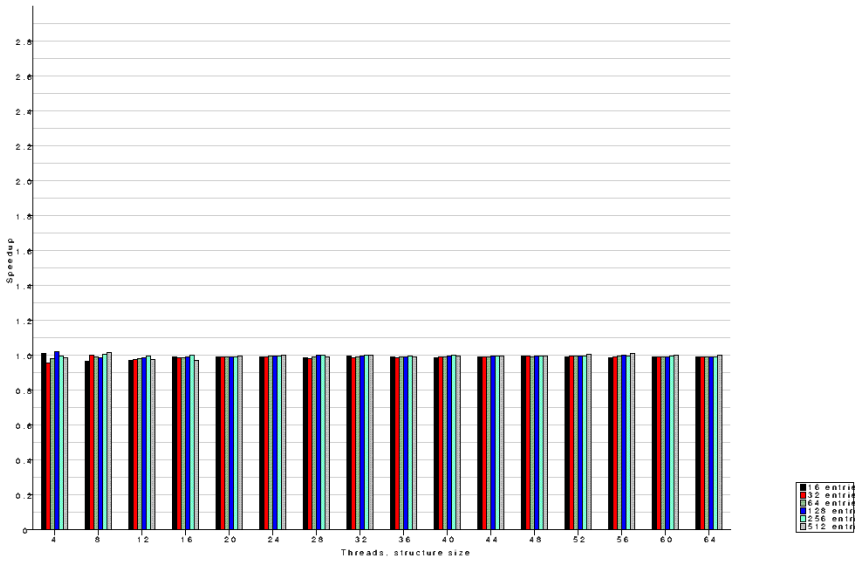

Figura A.2: Speedup para una tabla de tamaño 256

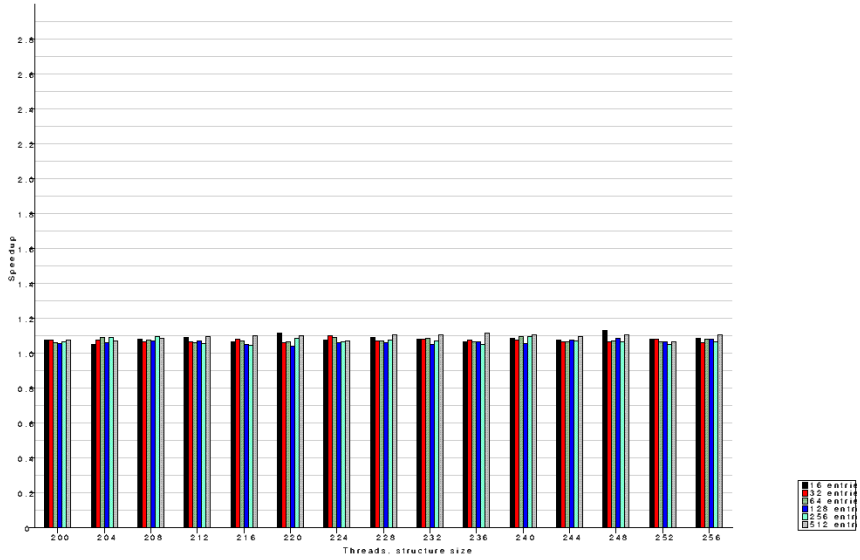

Figura A.3: Speedup para una tabla de tamaño 16 (muchos hilos)

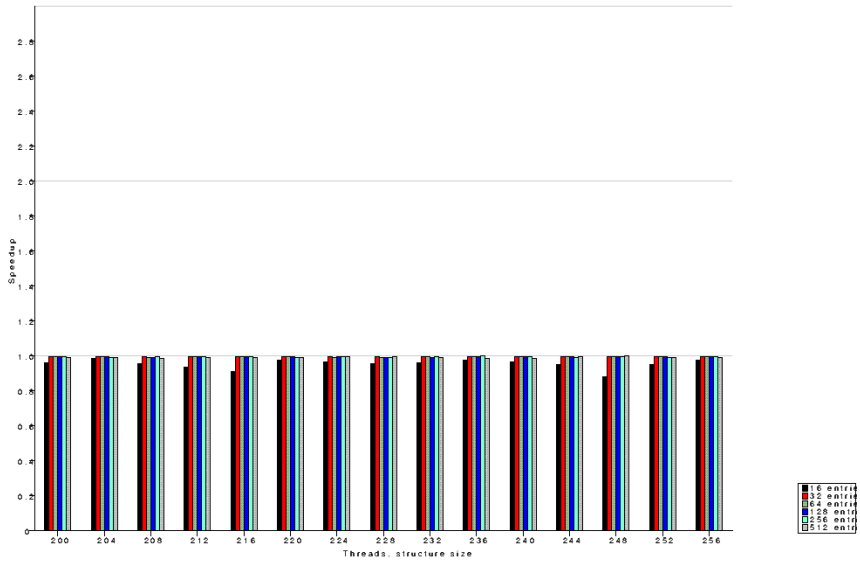

Figura A.4: Speedup para una tabla de tamaño 256 (muchos hilos)

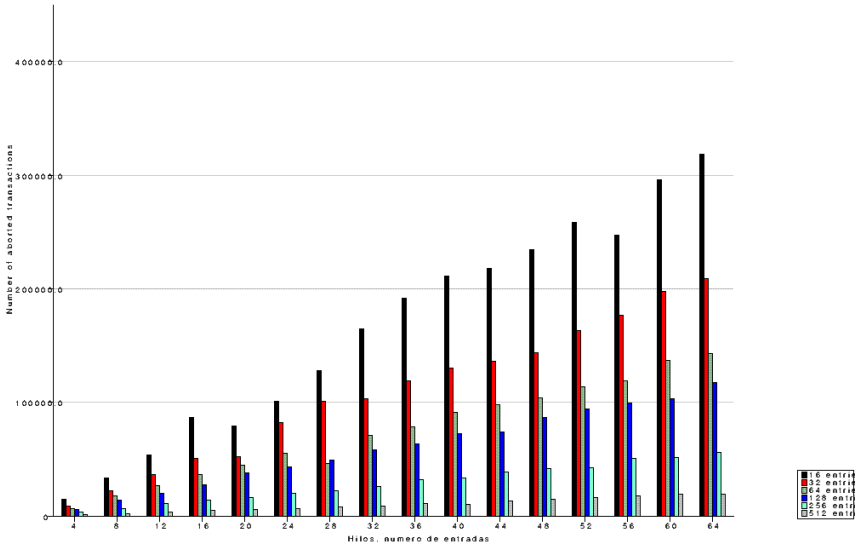

Figura A.5: Total de fallos para una tabla de tamaño 16 entradas

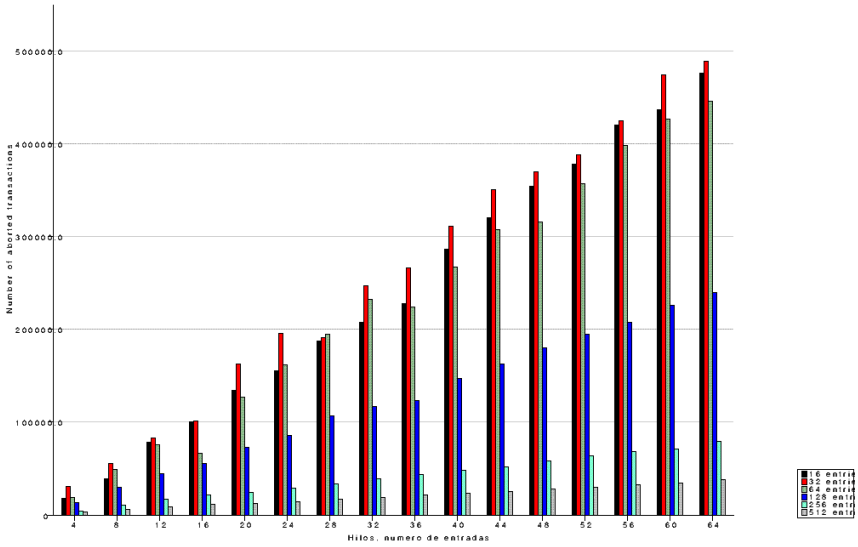

Figura A.6: Total de fallos para una tabla de tamaño 32 entradas

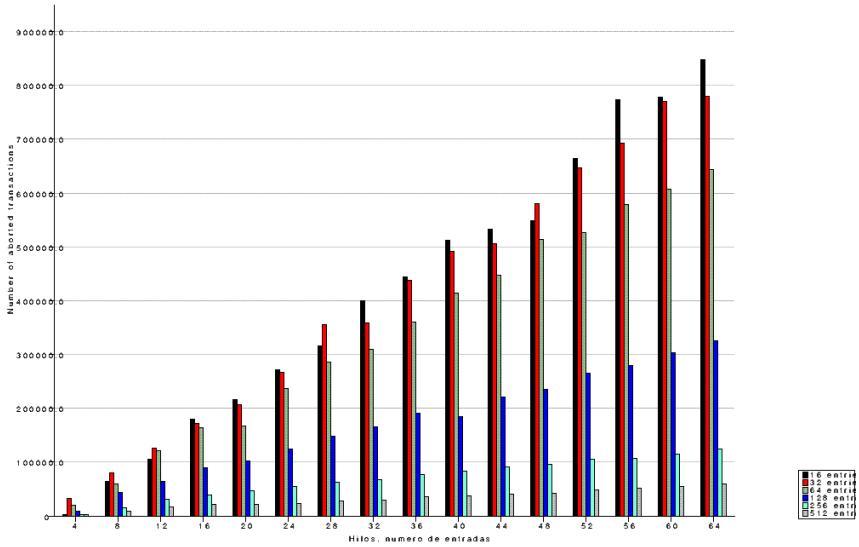

Figura A.7: Total de fallos para una tabla de tamaño 64 entradas

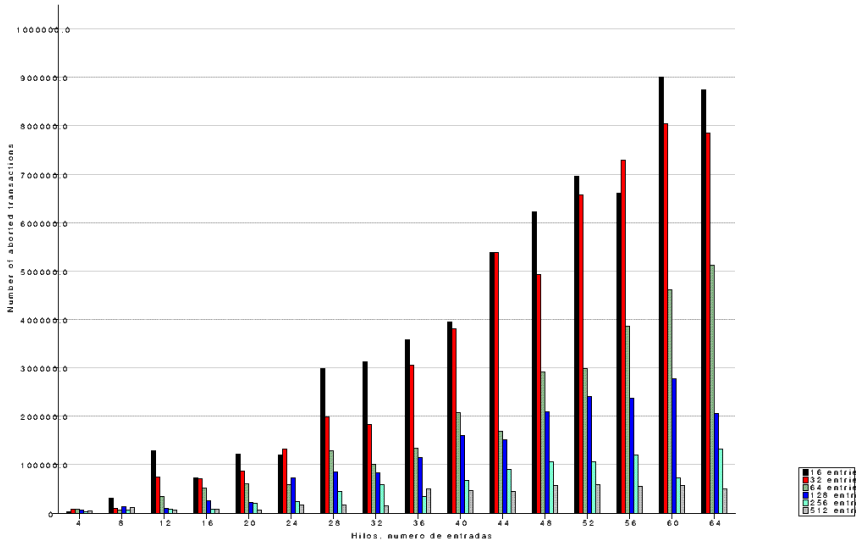

Figura A.8: Total de fallos para una tabla de tamaño 128 entradas

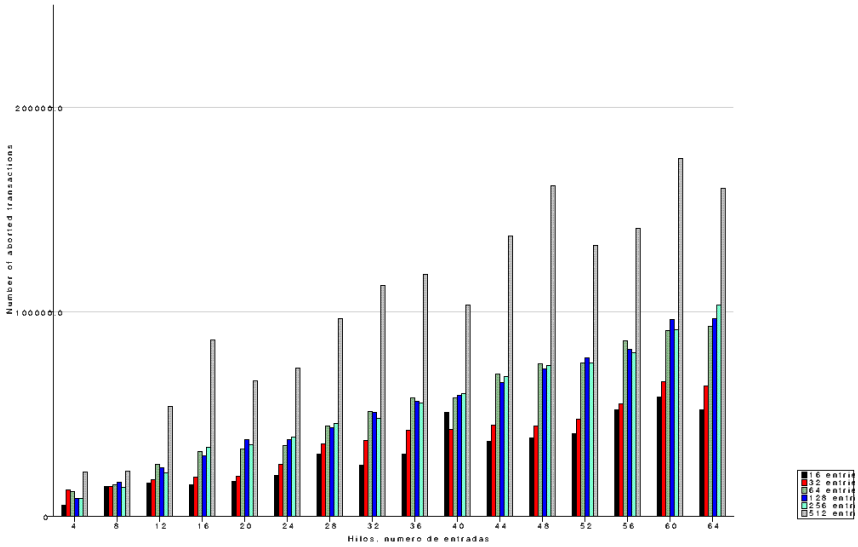

Figura A.9: Total de fallos para una tabla de tamaño 256 entradas

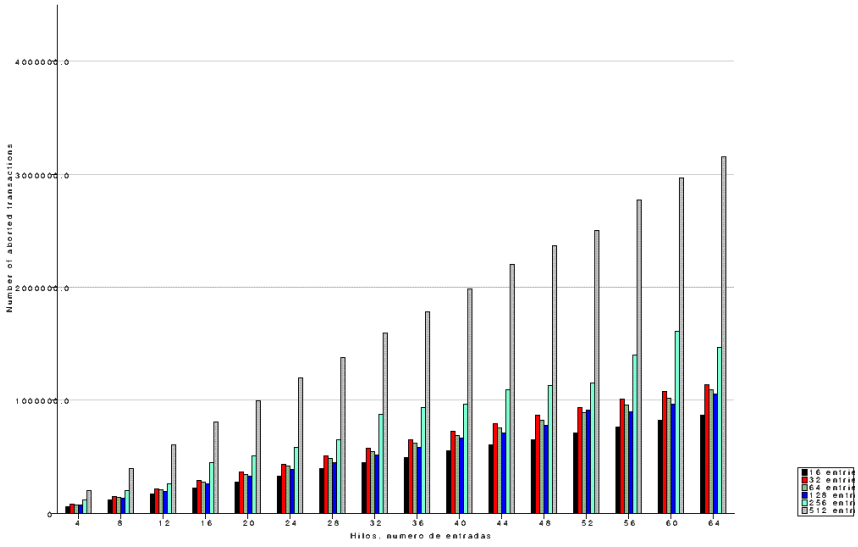

Figura A.10: Total de fallos para una tabla de tamaño 512 entradas

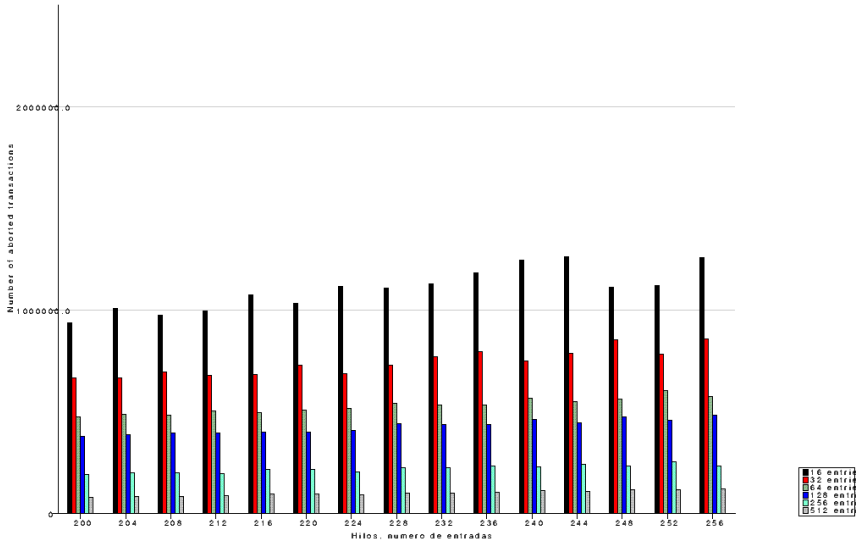

Figura A.11: Total de fallos para una tabla de tamaño 16 entradas (muchos hilos)

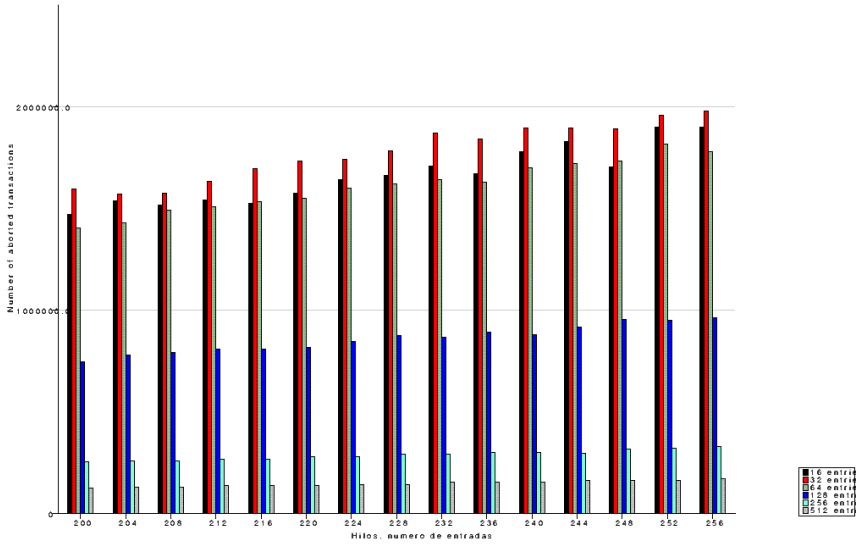

Figura A.12: Total de fallos para una tabla de tamaño 32 entradas (muchos hilos)

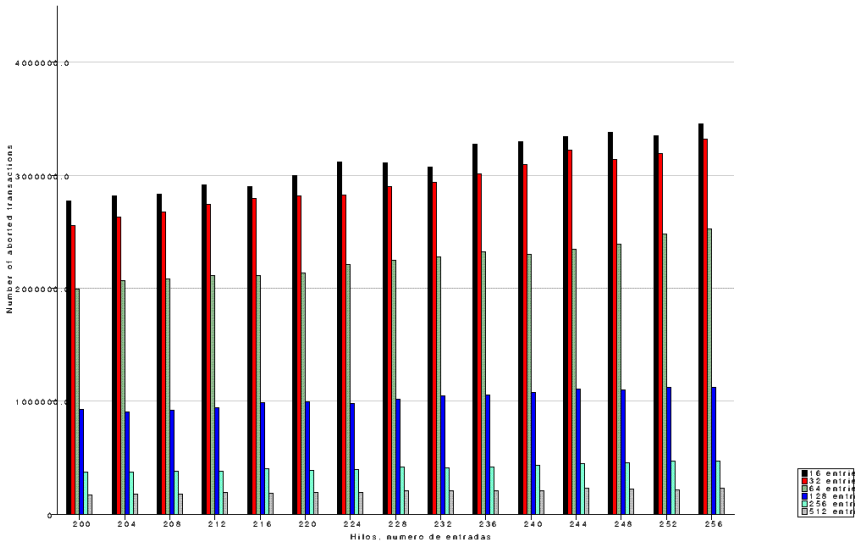

Figura A.13: Total de fallos para una tabla de tamaño 64 entradas (muchos hilos)

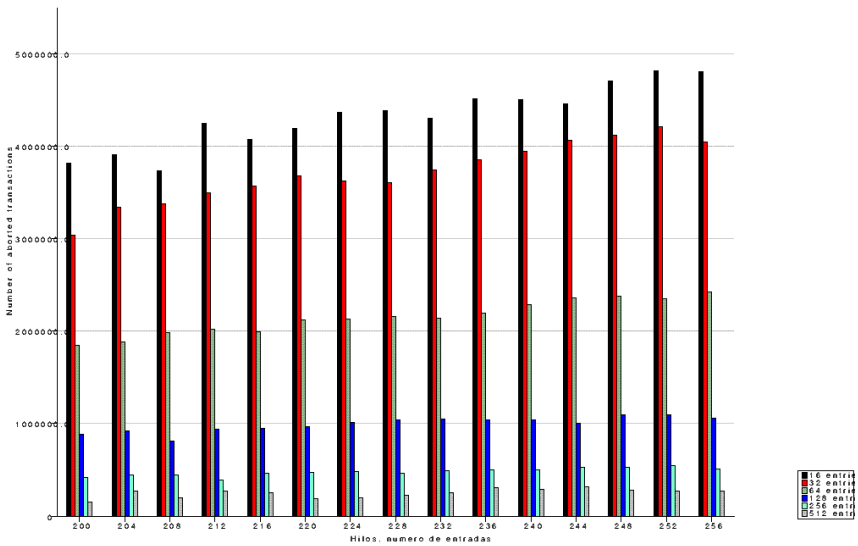

Figura A.14: Total de fallos para una tabla de tamaño 128 entradas (muchos hilos)

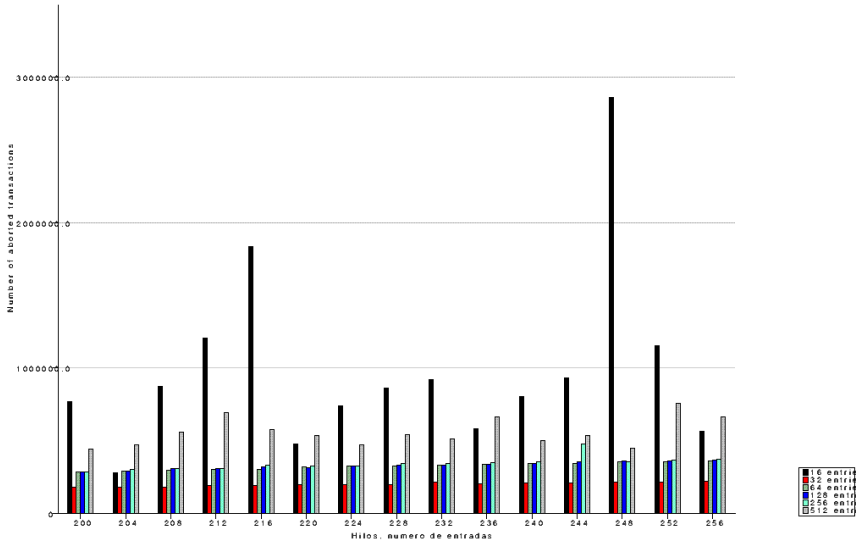

Figura A.15: Total de fallos para una tabla de tamaño 256 entradas (muchos hilos)

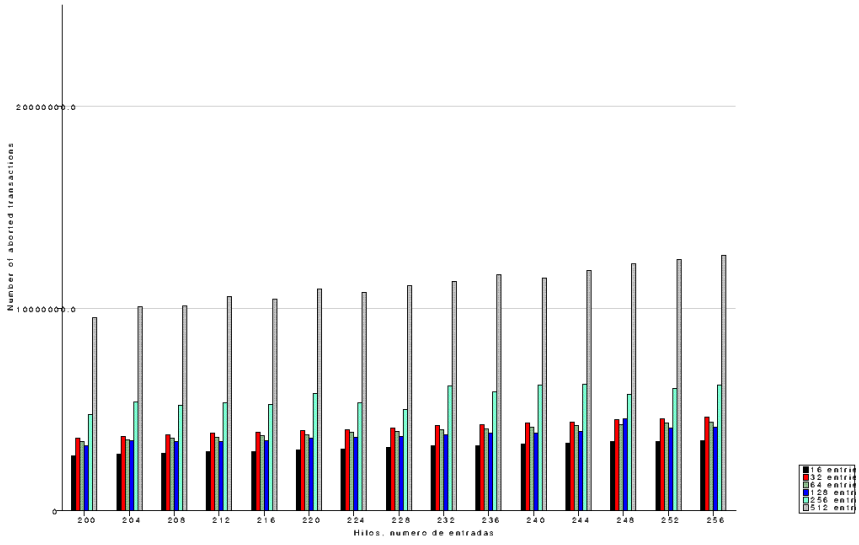

Figura A.16: Total de fallos para una tabla de tamaño 512 entradas (muchos hilos)

A.2. Ejecución con carga de trabajo constante

Las siguientes gráficas muestran los datos obtenidos manteniendo constante el número de iteraciones, y dividiéndolo entre el número de hilos.

A.2.1. Con 64 millones de iteraciones

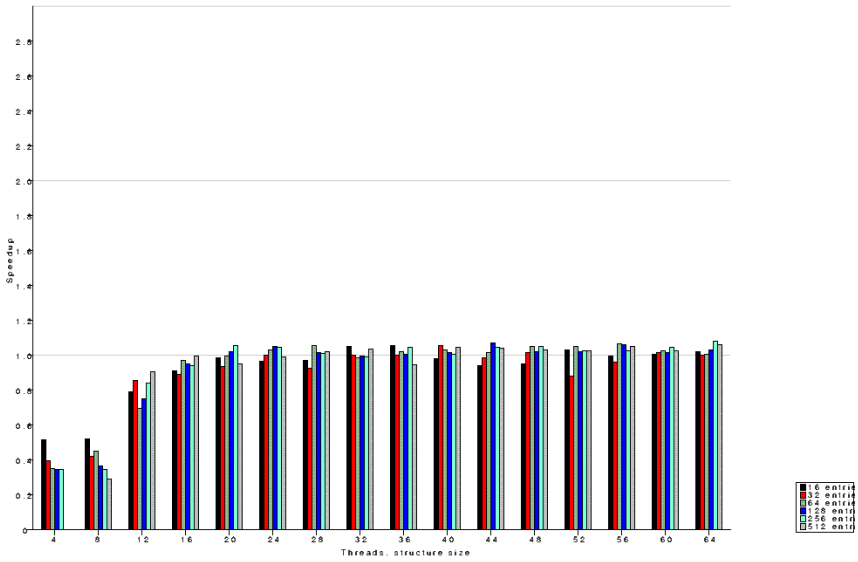

Figura A.17: Speedup para una tabla de tamaño 16 entradas

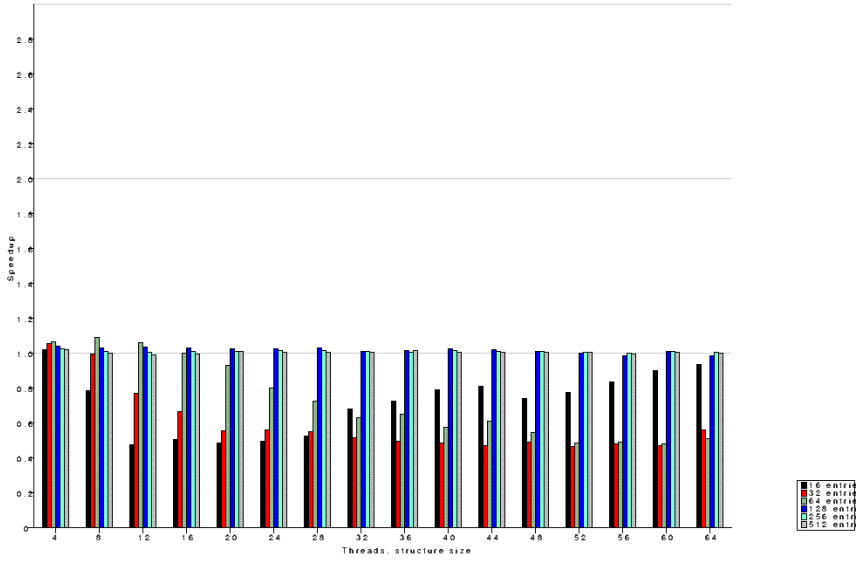

Figura A.18: Speedup para una tabla de tamaño 128 entradas

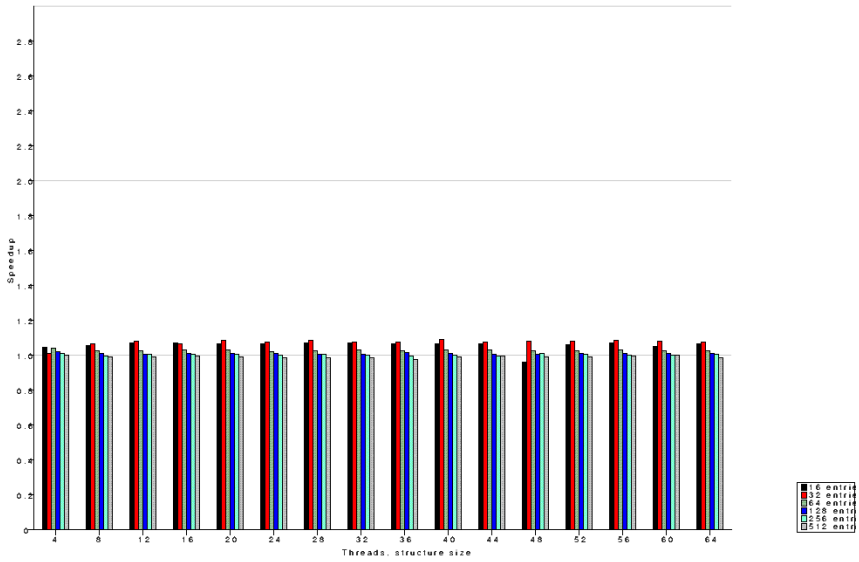

Figura A.19: Speedup para una tabla de tamaño 256 entradas

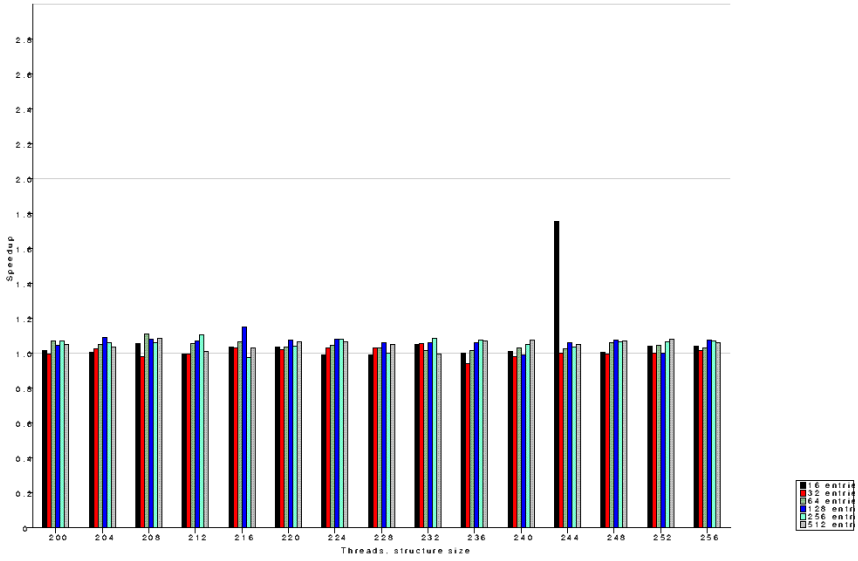

Figura A.20: Speedup para una tabla de tamaño 16 (muchos hilos)

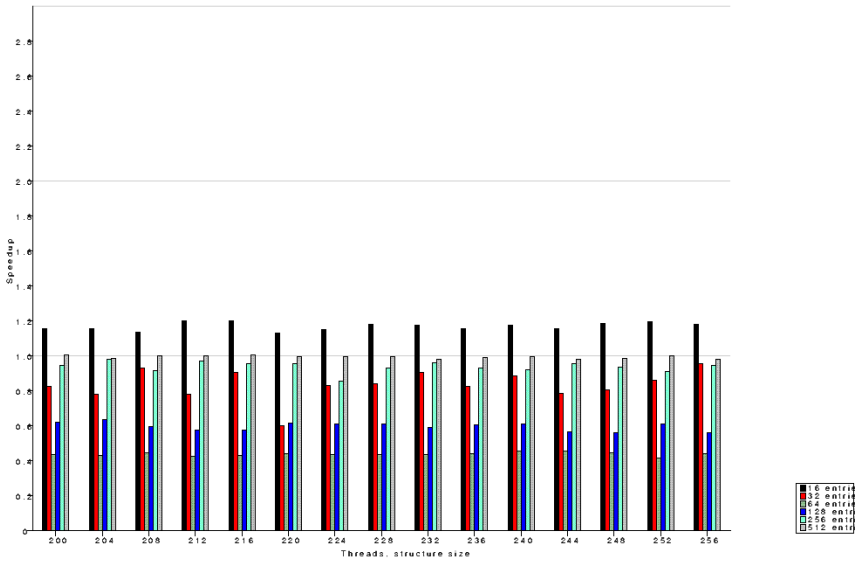

Figura A.21: Speedup para una tabla de tamaño 128 (muchos hilos)

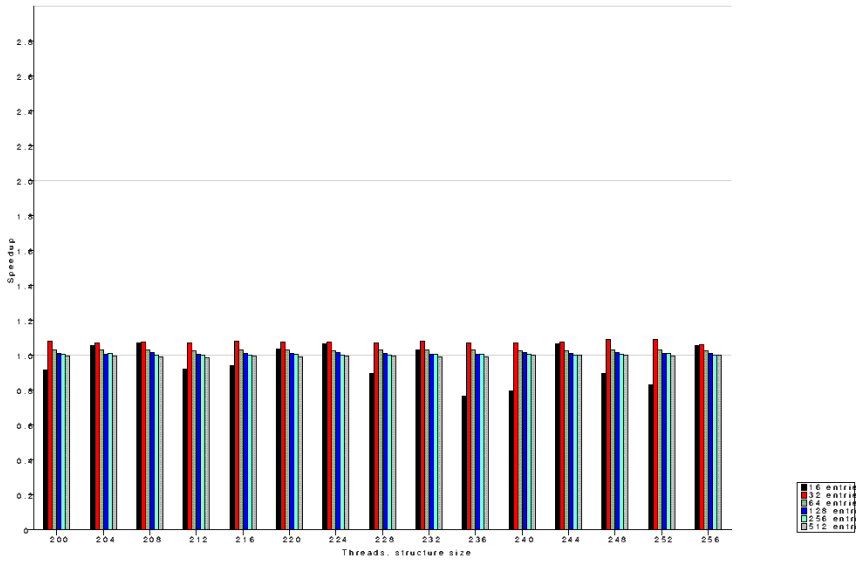

Figura A.22: Speedup para una tabla de tamaño 256 (muchos hilos)

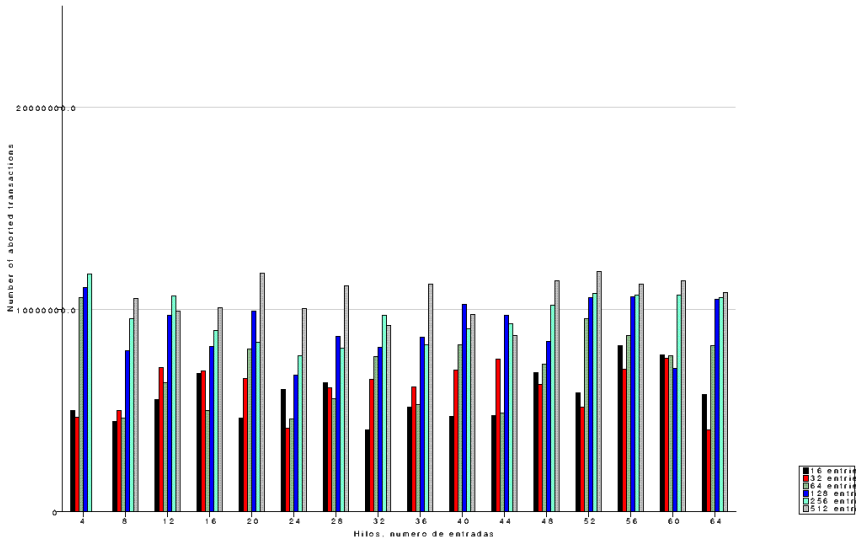

Figura A.23: Fallos para una tabla de tamaño 16 entradas

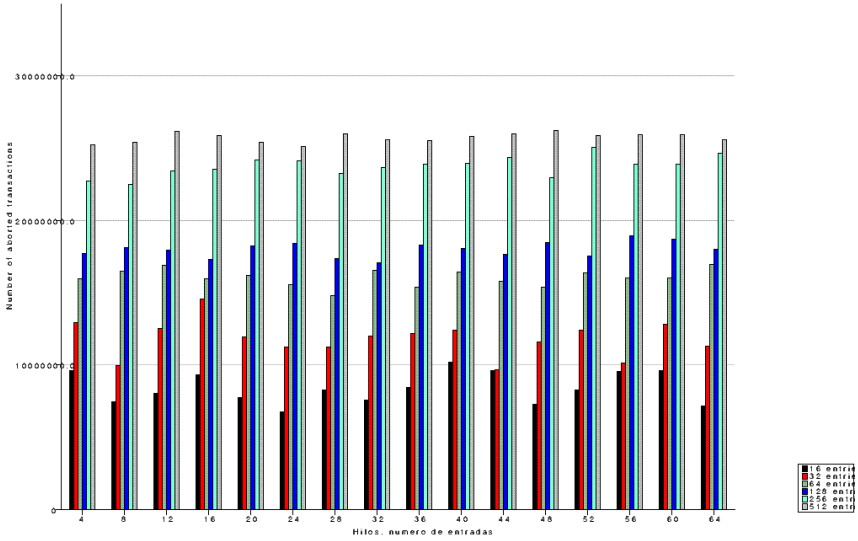

Figura A.24: Fallos para una tabla de tamaño 32 entradas

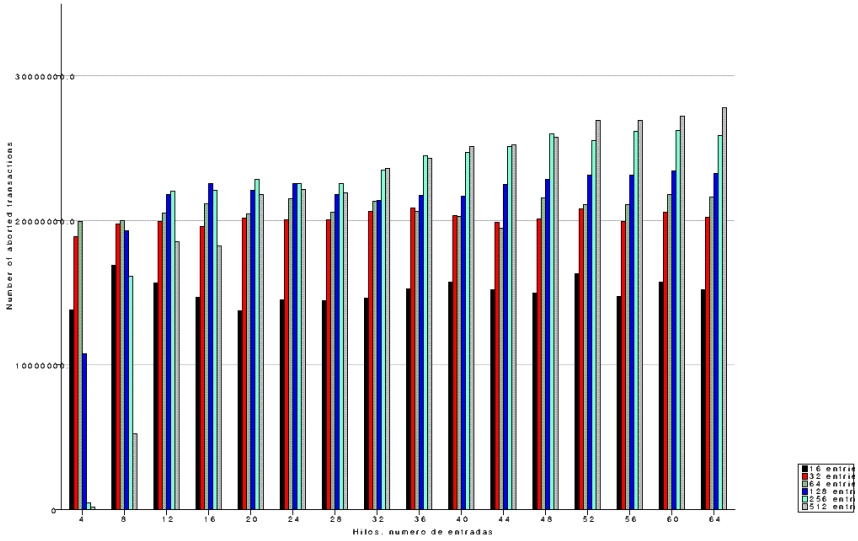

Figura A.25: Fallos para una tabla de tamaño 64 entradas

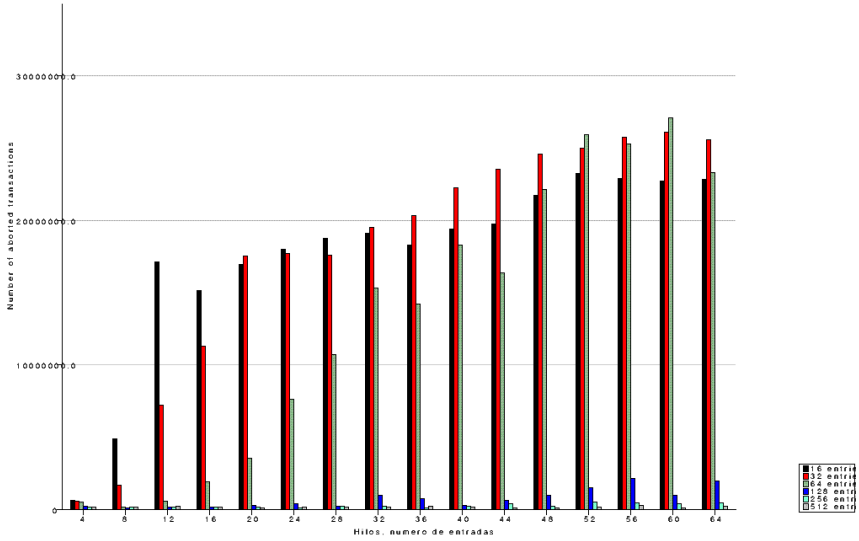

Figura A.26: Fallos para una tabla de tamaño 128 entradas

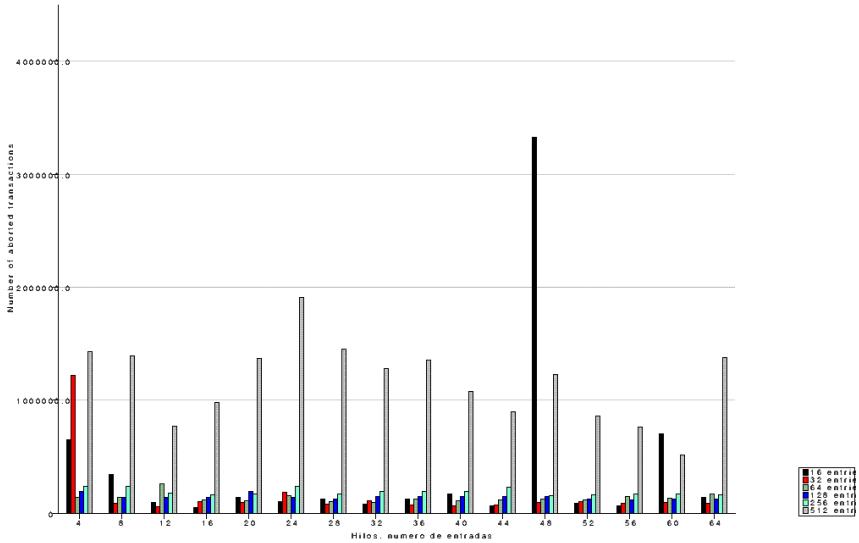

Figura A.27: Fallos para una tabla de tamaño 256 entradas

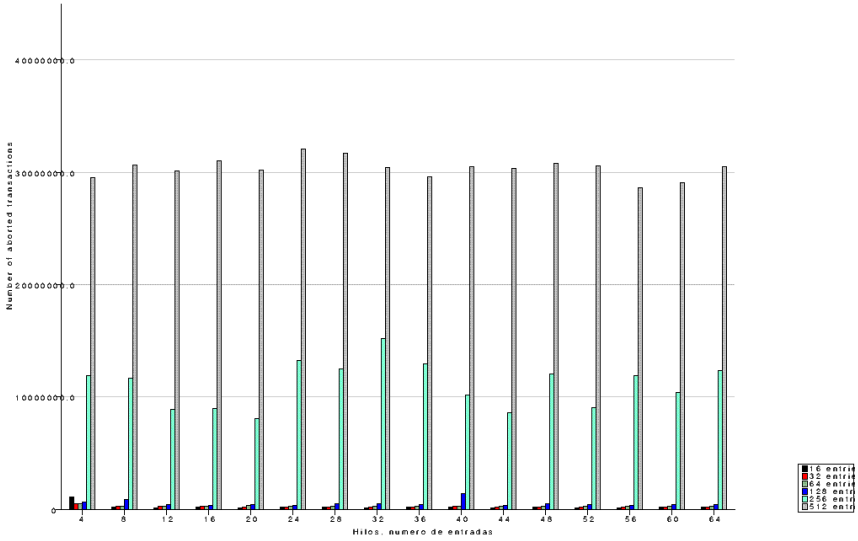

Figura A.28: Fallos para una tabla de tamaño 512 entradas

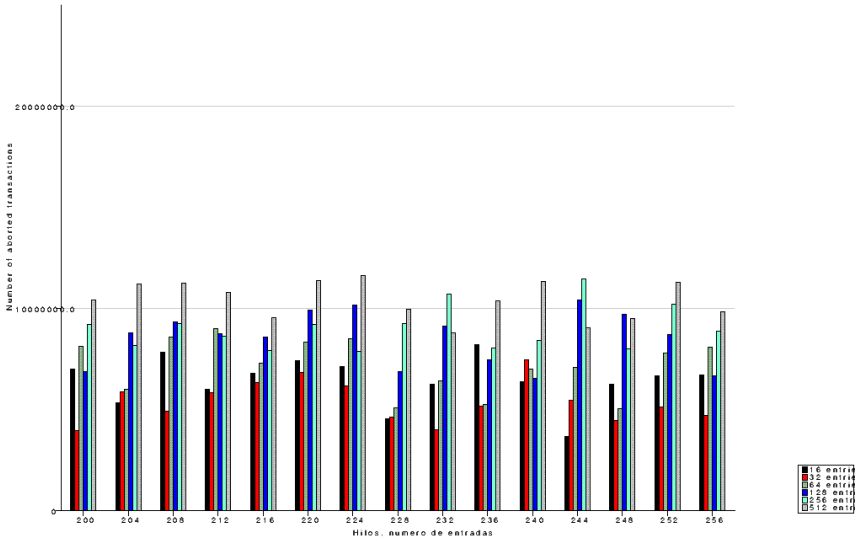

Figura A.29: Total de fallos para una tabla de tamaño 16 entradas (muchos hilos)

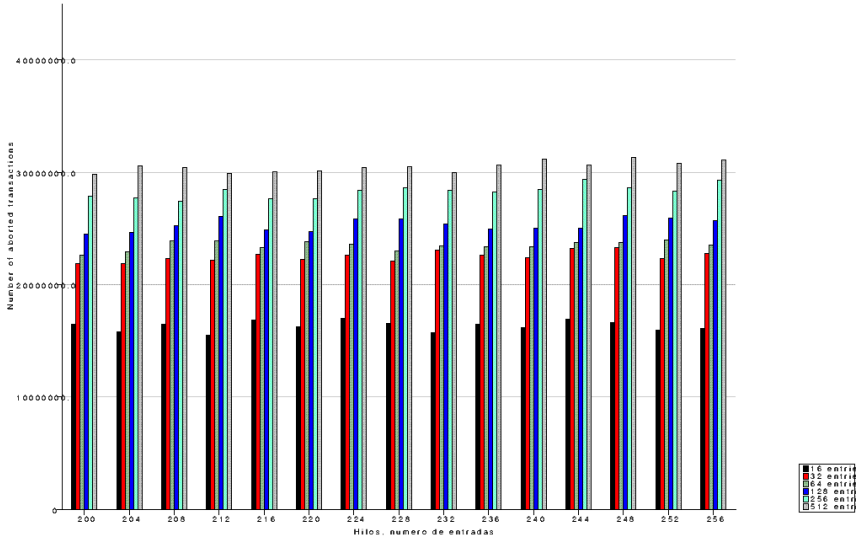

Figura A.30: Total de fallos para una tabla de tamaño 64 entradas (muchos hilos)

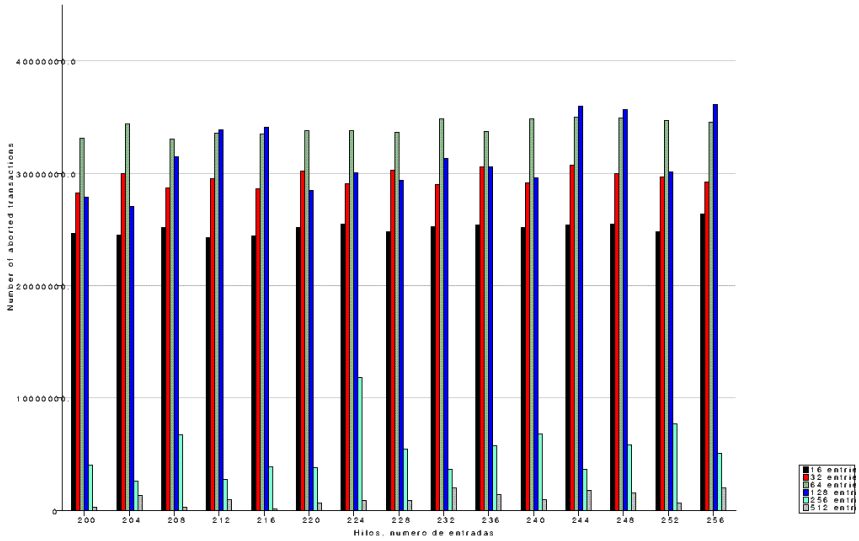

Figura A.31: Total de fallos para una tabla de tamaño 128 entradas (muchos hilos)

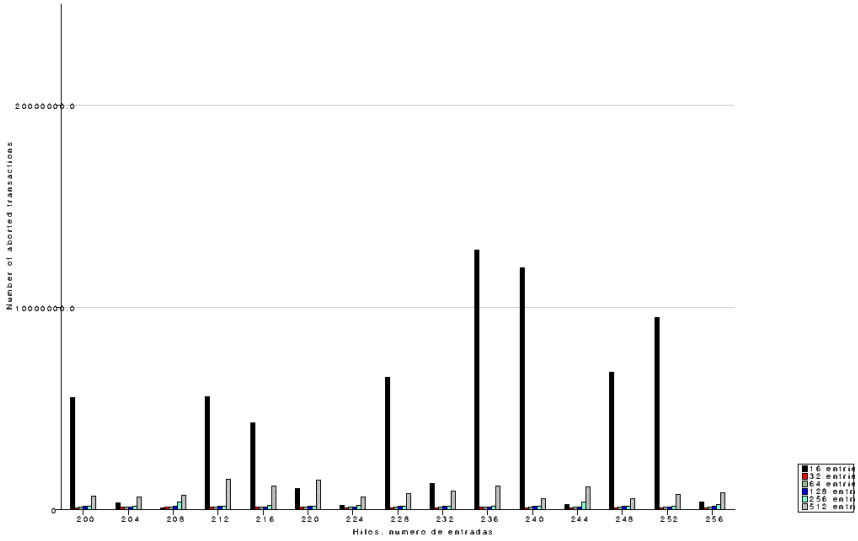

Figura A.32: Total de fallos para una tabla de tamaño 256 entradas (muchos hilos)

A.2.2. Con 128 millones de iteraciones

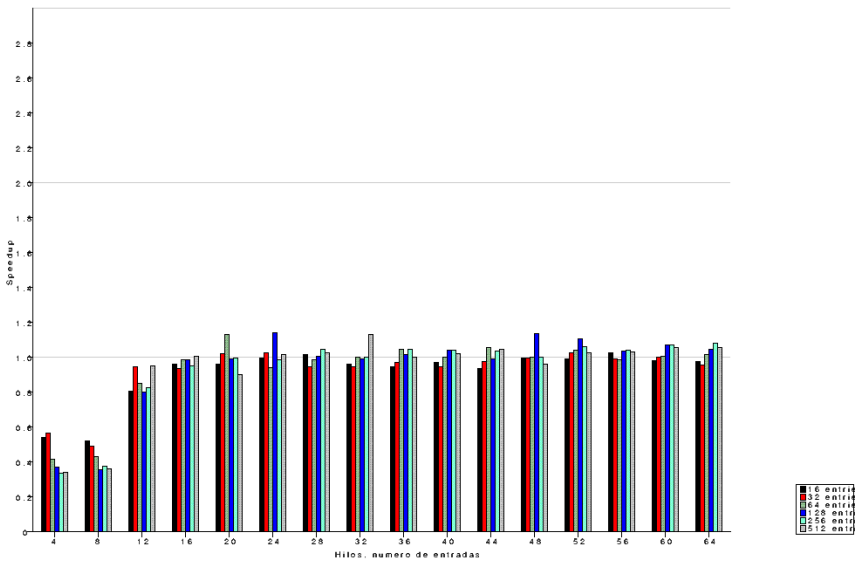

Figura A.33: Speedup para una tabla de tamaño 16 entradas

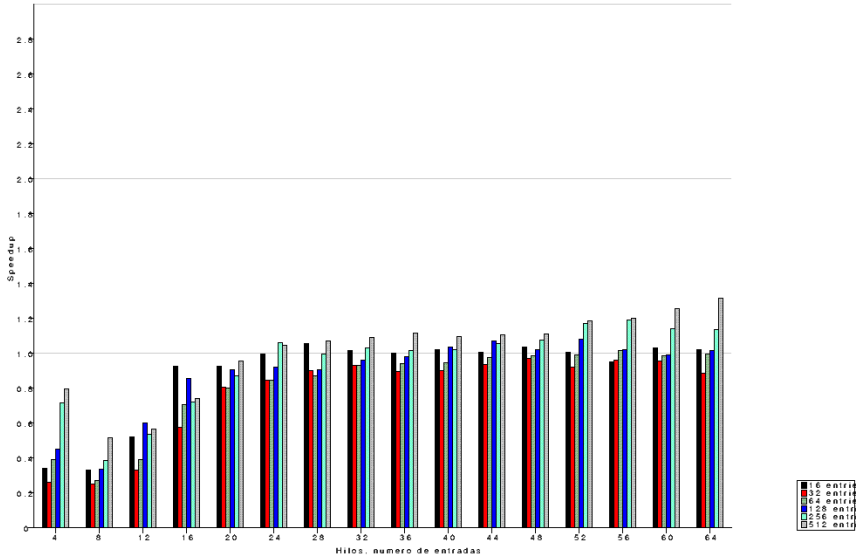

Figura A.34: Speedup para una tabla de tamaño 32 entradas

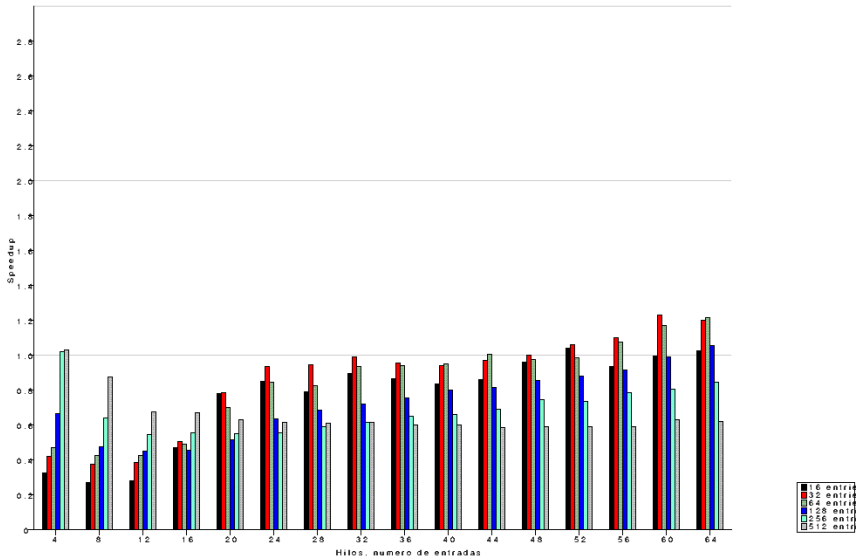

Figura A.35: Speedup para una tabla de tamaño 64 entradas

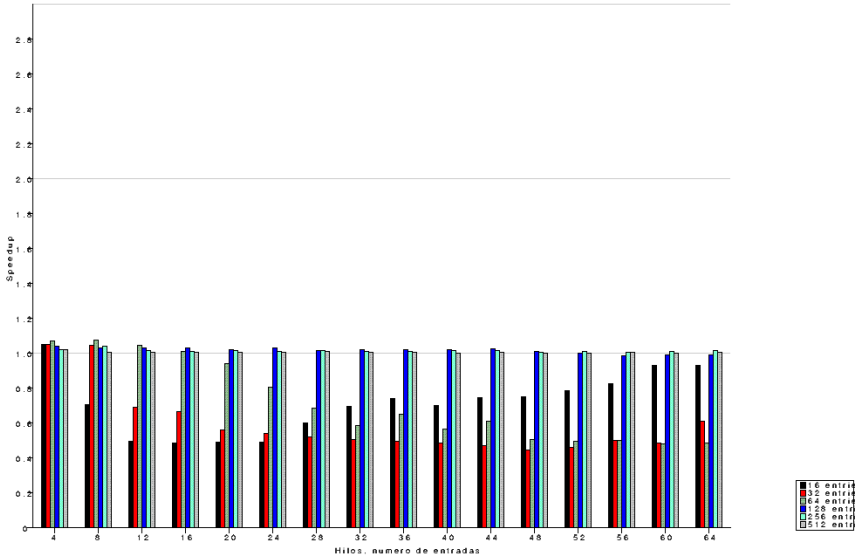

Figura A.36: Speedup para una tabla de tamaño 128 entradas

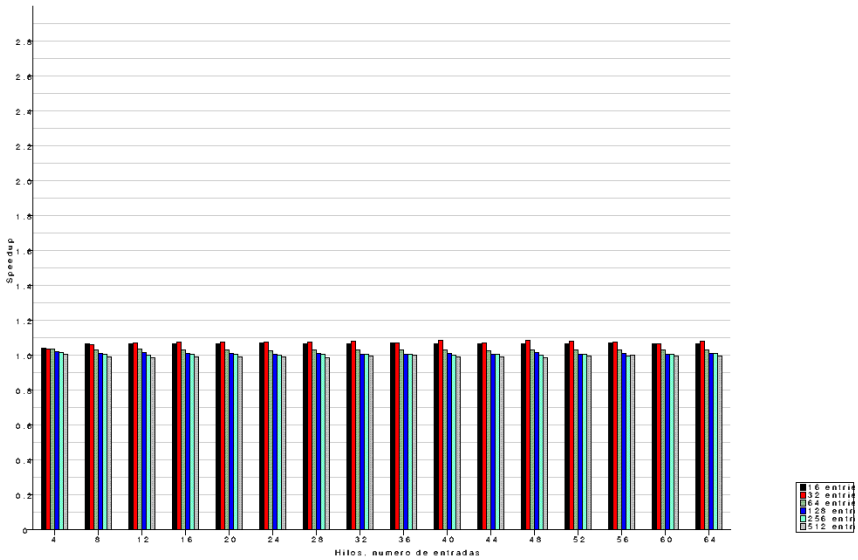

Figura A.37: Speedup para una tabla de tamaño 256 entradas

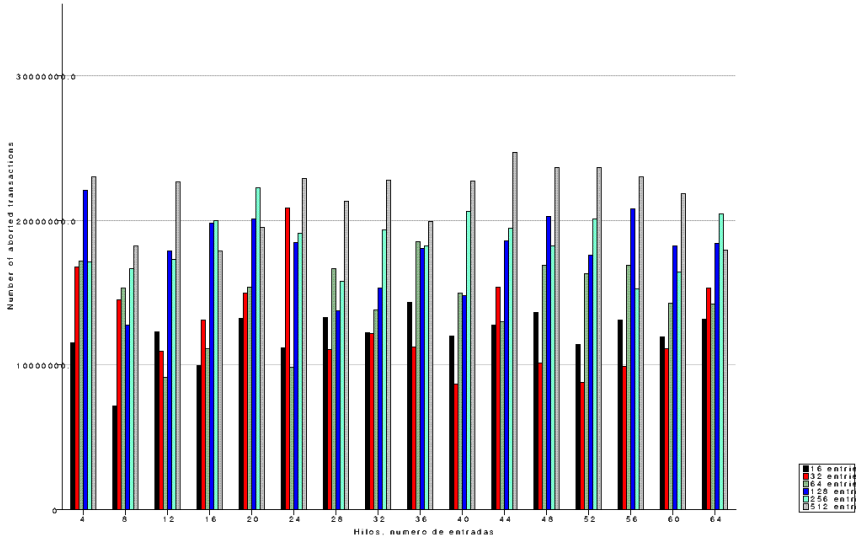

Figura A.38: Fallos para una tabla de tamaño 16 entradas

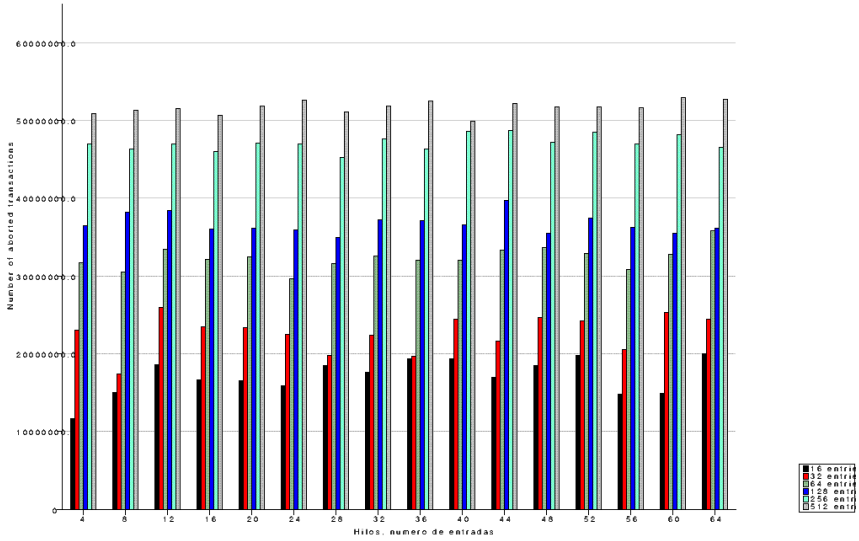

Figura A.39: Fallos para una tabla de tamaño 32 entradas

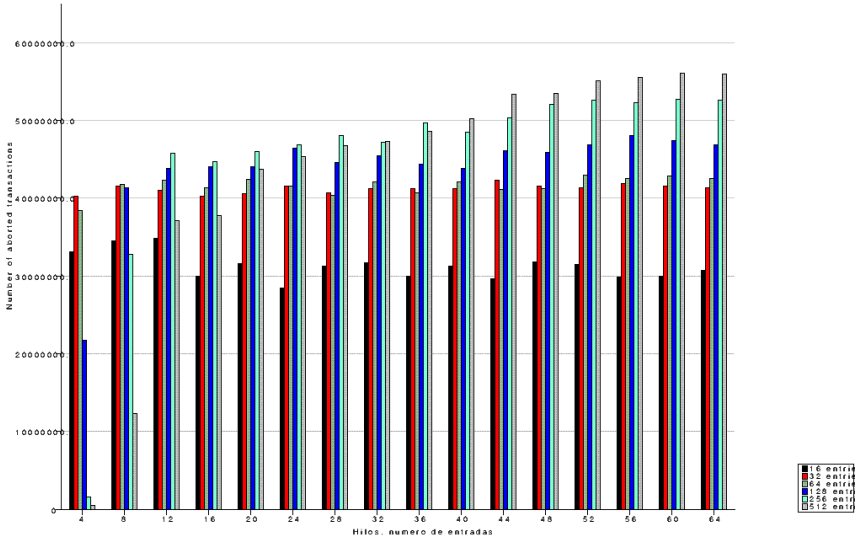

Figura A.40: Fallos para una tabla de tamaño 64 entradas

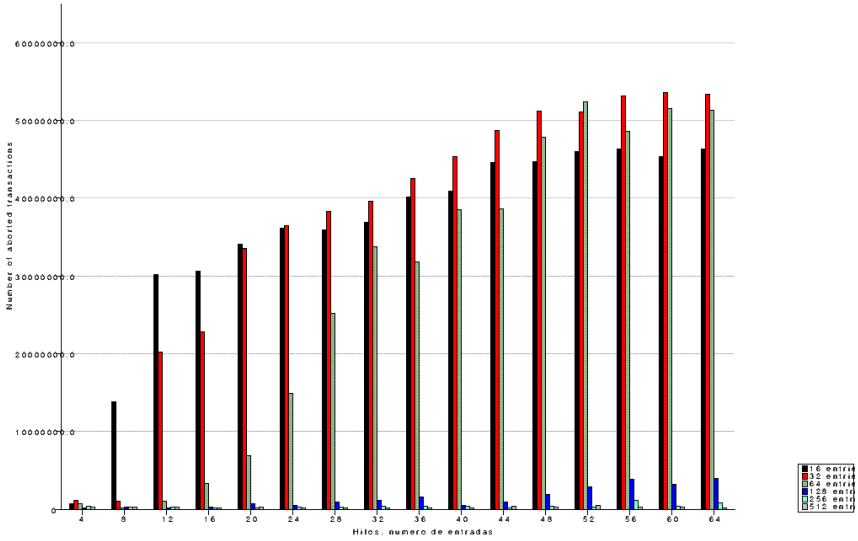

Figura A.41: Fallos para una tabla de tamaño 128 entradas

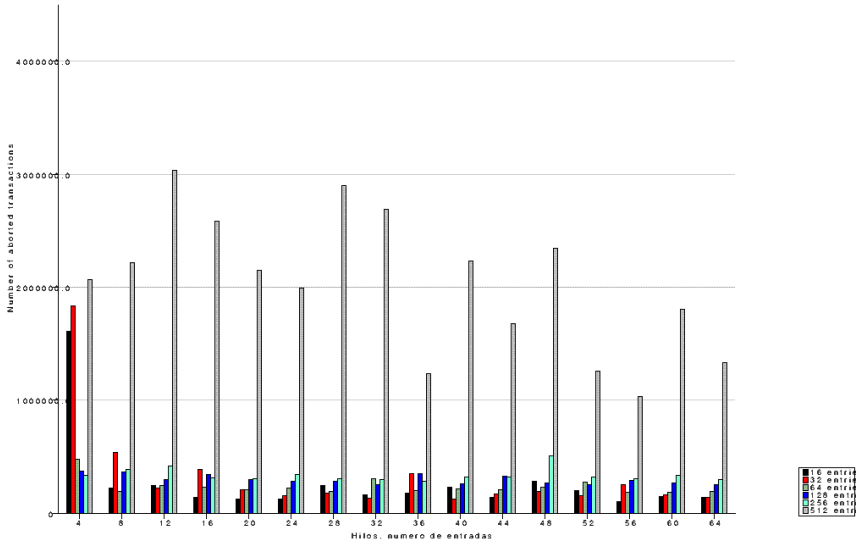

Figura A.42: Fallos para una tabla de tamaño 256 entradas

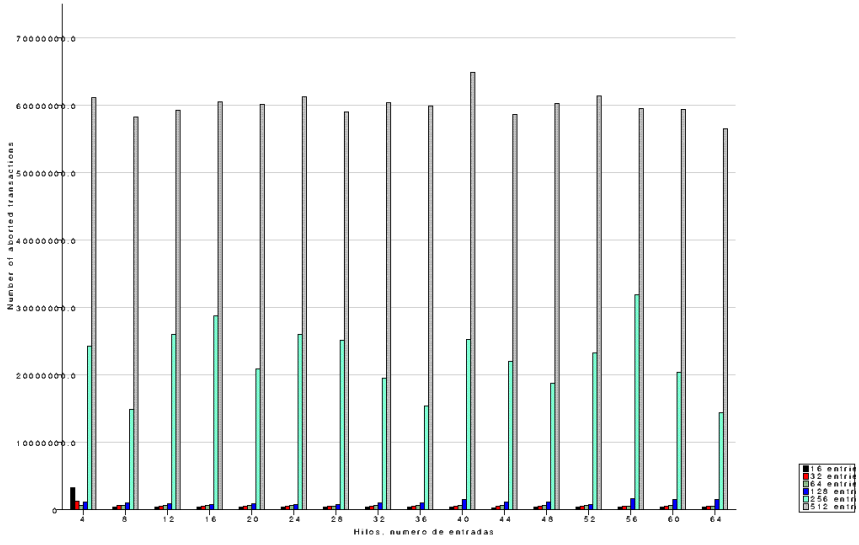

Figura A.43: Fallos para una tabla de tamaño 512 entradas

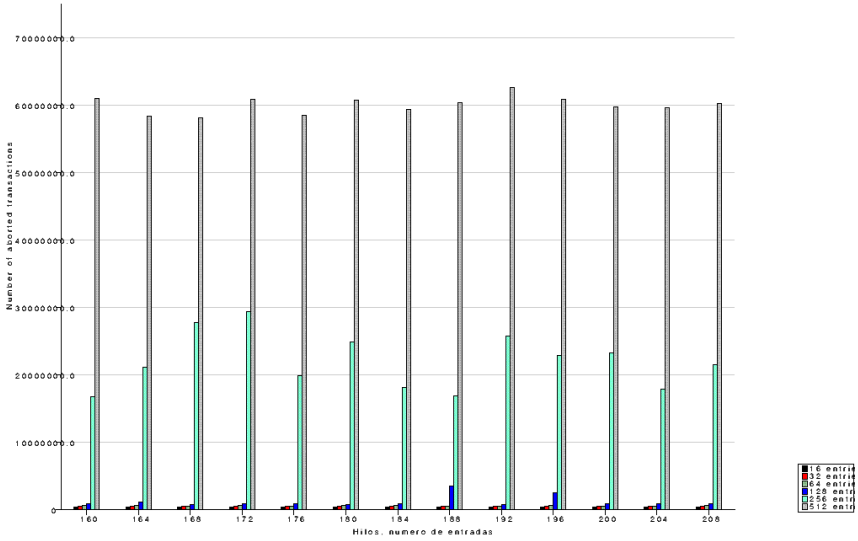

Figura A.44: Total de fallos para una tabla de tamaño 512 entradas (muchos hilos)

Apéndice B

Contenidos del CD-ROM

El CD incluye todos los ficheros generados durante la elaboración del trabajo. Éste se divide en seis directorios que describiremos a continuación:

B.1. Directorio *Benchmark*

Contiene el código del benchmark, compuesto por el fichero `TablaHash.c` y un fichero de encabezado asociado llamado `TablaHash.h`. El código está preparado para ser compilado con el script `haz.sh`, para después ejecutarlo con el script `lanza.sh`. Este último script tiene al inicio unas variables que permiten configurar la forma de las ejecuciones sucesivas (número mínimo y máximo de iteraciones, hilos, etc) y está configurado para que los resultados se vayan añadiendo al fichero CSV que se le indique. Irá imprimiendo por pantalla el estado de la ejecución.

Este directorio contiene además el directorio `galoizedHashTable`, donde quedó una prueba de portar el benchmark a C++ para integrarlo con Galois, que finalmente no progresó.

B.2. Directorio *Benchmark_results*

Aquí se han almacenado todos los resultados en bruto de las distintas ejecuciones del benchmark. En la carpeta `variable_workload_iter_x_thread` están los ficheros CSV ejecutados con carga variable, es decir, con un número fijo de iteraciones por hilo. Las demás carpetas se refieren a ejecuciones con carga constante, es decir, con un número de iteraciones total independiente al número de hilos con que se ejecute. Estas carpetas tienen un nombre que indica el número de iteraciones con las que se ha ejecutado el benchmark, y en su interior almacenan ficheros de distintas ejecuciones con ese parámetro.

B.3. Directorio *Chart_generation*

En este directorio encontramos las utilidades Python que nos sirven para hacer el preprocesamiento de los datos, la generación de las gráficas, etc. La ejecución se realiza llamando al fichero `script.py`, donde se configura el proceso a seguir por los datos. Este fichero llamará a `process.py`, donde se encuentran las rutinas para el preprocesamiento de los datos; y a `chart.py`, que es quien usa la biblioteca PyChart para generar las gráficas.

También se encuentra en este directorio la utilidad `average.py`, que fue usada para hacer una media aritmética de los datos de cinco ejecuciones con carga de trabajo variable. El resultado de esta operación también se encuentra en `Benchmark_results`.

B.4. Directorio *Documentation*

Este es el directorio que contiene los ficheros fuente de la elaboración de esta memoria. El contenido se encuentra en formato markdown dividido en secciones en la carpeta `secciones`. En la carpeta `latex` podemos ver algunos ficheros necesarios para la creación del documento final. La carpeta `img` almacena categorizados en subcarpetas los gráficos generados a partir de los datos, que después se referencian desde el documento. También están en esa carpeta algunos diagramas creados.

En la raiz del directorio está el fichero `bibliografia.bib`, con la información bibliográfica en formato *BibTeX*; y el fichero `main.mdpp` que define la estructura del documento y algunos metadatos como el título, agradecimientos, autor, etc.

B.5. Directorio *Galois*

Ha sido usado para reunir información y notas sobre la investigación con Galois, información que ha sido paulatinamente trasladada a la sección correspondiente de este trabajo.

B.6. Directorio *Papers*

Aquí se reunieron publicaciones de interés para la elaboración del trabajo. Aunque están referenciadas en la bibliografía, aquí se encuentran los documentos originales dispuestos para la consulta cuando sea necesario.

Bibliografía

[1] Matthew E. Brashears y Eric Gladstone. "Error correction mechanisms in social networks can reduce accuracy and encourage innovation". En: *Social Networks* 44 (ene. de 2016), págs. 22-35. ISSN: 0378-8733. DOI: 10.1016/j.socnet.2015.07.007. URL: http://www.sciencedirect.com/science/article/pii/S0378873315000696 (visitado 12-07-2016).

[2] Catalin Cimpanu. *New Record Set for Layer 7 DDoS Attacks by Nitol Botnet*. URL: http://news.softpedia.com/news/new-record-set-for-layer-7-ddos-attacks-by-nitol-botnet-502639.shtml (visitado 20-06-2016).

[3] *CMake - Cross Platform Make*. URL: https://cmake.org/cmake/help/v2.8.10/cmake.html#variable:CMAKE_LANG_COMPILER_ID (visitado 20-06-2016).

[4] *Core i7 5775C processor review: Desktop Broadwell*. URL: http://www.guru3d.com/articles_pages/core_i7_5775c_processor_review_desktop_broadwell,1.html (visitado 20-06-2016).

[5] Ian Cutress. *The Intel Broadwell Desktop Review: Core i7-5775C and Core i5-5675C Tested (Part 1)*. URL: http://www.anandtech.com/show/9320/intel-broadwell-review-i7-5775c-i5-5675c (visitado 20-06-2016).

[6] Ricardo Galli. *Principios y algoritmos de concurrencia*. Español. Jun. de 2105. ISBN: 978-1-5170-2975-3.

[7] *Galois documentation: Main Page*. URL: http://iss.ices.utexas.edu/projects/galois/api/current/index.html (visitado 13-07-2016).

[8] Sandra González-Bailón y Ning Wang. "Networked discontent: The anatomy of protest campaigns in social media". En: *Social Networks* 44 (ene. de 2016), págs. 95-104. ISSN: 0378-8733. DOI: 10.1016/j.socnet.2015.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0378873315000659 (visitado 12-07-2016).

[9] Jim Gray y col. "The transaction concept: Virtues and limitations". En: *VLDB*. Vol. 81. 1981, págs. 144-154. URL: http://infolab.usc.edu/csci599/Fall2008/papers/b-2.pdf (visitado 20-06-2016).

[10] Theo Haerder y Andreas Reuter. "Principles of Transaction-oriented Database Recovery". En: *ACM Comput. Surv.* 15.4 (dic. de 1983), págs. 287-317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://doi.acm.org/10.1145/289.291 (visitado 20-06-2016).

[11] *IBM XL compiler hardware transactional memory built-in functions for IBM AIX on IBM POWER8 processor-based systems*. en. CT316. Nov. de 2014. URL: https://www.ibm.com/developerworks/aix/library/au-aix-ibm-xl-compiler-built-in-functions/index.html (visitado 20-06-2016).

[12] *ICT Facts and Figures – The world in 2015*. URL: http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx (visitado 20-06-2016).

[13] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. (Visitado 10-07-2016).

[14] *Intel® Core™ i7-5775C Processor (6M Cache, up to 3.70 GHz) Especificaciones*. URL: http://ark.intel.com/es-es/products/88040/Intel-Core-i7-5775C-Processor-6M-Cache-up-to-3_70-GHz (visitado 20-06-2016).

[15] David Kanter. *Analysis of Haswell's Transactional Memory*. URL: http://www.realworldtech.com/haswell-tm/ (visitado 18-06-2016).

[16] A. N. Kolmogorov. "On tables of random numbers". En: *Theoretical Computer Science* 207.2 (nov. de 1998), págs. 387-395. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(98)00075-9. URL: http://www.sciencedirect.com/science/article/pii/S0304397598000759 (visitado 13-07-2016).

[17] Milind Kulkarni, L. Paul Chew y Keshav Pingali. "Using transactions in delaunay mesh generation". En: *Workshop on Transactional Memory Workloads*. Citeseer, 2006, págs. 23-31. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.296.7647&rep=rep1&type=pdf (visitado 20-06-2016).

[18] Milind Kulkarni y col. "Lonestar: A suite of parallel irregular programs". En: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, págs. 65-76. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4919639 (visitado 20-06-2016).

[19] Iván Martínez. *Intel Core i7-5775C [Review]*. URL: https://elchapuzasinformatico.com/2015/06/review-intel-core-i7-5775c/2/ (visitado 20-06-2016).

[20] Chi Cao Minh y col. "An effective hybrid transactional memory system with strong isolation guarantees". En: *ACM SIGARCH Computer Architecture News*. Vol. 35. ACM, 2007, págs. 69-80. URL: http://dl.acm.org/citation.cfm?id=1250673 (visitado 20-06-2016).

[21] Keshav Pingali y col. "The tao of parallelism in algorithms". En: *ACM Sigplan Notices*. Vol. 46. ACM, 2011, págs. 12-25. URL: http://dl.acm.org/citation.cfm?id=1993501 (visitado 20-06-2016).

[22] Ravi Rajwar. "Going Under The Hood With Intel's Next Generation Microarchitecture Codename Haswell". En: San Francisco, nov. de 2012. URL: https://qconsf.com/sf2012/dl/qcon-sanfran-2012/slides/RaviRajwar_GoingUnderTheHoodWithIntelsNextGener
pdf (visitado 07-07-2016).

[23] *Rock (processor)*. en. Page Version ID: 723975891. Jun. de 2016. URL: https://en.wikipedia.org/w/index.php?title=Rock_(processor)&oldid=723975891 (visitado 20-06-2016).

[24] *Slashdot effect*. en. Page Version ID: 722756613. Mayo de 2016. URL: https://en.wikipedia.org/w/index.php?title=Slashdot_effect&oldid=722756613 (visitado 20-06-2016).

[25] Scott Wasson. *Errata prompts Intel to disable TSX in Haswell, early Broadwell CPUs.* URL: `http://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus` (visitado 20-06-2016).

[26] *Web Resources about Intel® Transactional Synchronization Extensions Intel® Software.* URL: *https://software.intel.com/en-us/blogs/2013/06/07/web-resources-about-intelr-transactional-synchronization-extensions (visitado 18-06-2016).*

[27] Wgsimon. *English: Transistor counts for integrated circuits plotted against their dates of introduction. The curve shows Moore's law - the doubling of transistor counts every two years. The y-axis is logarithmic, so the line corresponds to exponential growth.* Mayo de 2011. URL: `https://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg` (visitado 20-06-2016).

[28] *X86 transactional memory intrinsics - Using the GNU Compiler Collection (GCC).* URL: `https://gcc.gnu.org/onlinedocs/gcc-4.9.3/gcc/X86-transactional-memory-intrinsics.html` (visitado 06-07-2016).