Ampliando el soporte de Memoria Transaccional en RISCV

Máster en Nuevas Tecnologías de la Informática

Trabajo Fin de Master

Autor:
Javier García Hernández
Tutor/es:
Jose Rubén Titos Gil
Ricardo Fernández Pascual

2 de Junio de 2022

Facultad Informática Universidad Murcia

Ampliando el soporte de Memoria Transaccional en RISCV

Extensiones al ISA para acomodar el esquema de privatización en MT

Autor
Javier García Hernández

Tutor/es
Jose Rubén Titos Gil
DITEC
Ricardo Fernández Pascual
DITEC

Facultad de Informática

Máster en Nuevas Tecnologías de la Informática

UNIVERSIDAD DE MURCIA

Murcia, 2 de Junio de 2022

Resumen

Para obtener el máximo rendimiento de los procesadores multinúcleo, es conveniente que las aplicaciones distribuyan su carga de trabajo en distintos hilos, que podrán ejecutarse en paralelo en sendos núcleos. Este modelo de ejecución no garantiza cuándo se va a ejecutar cada hilo, por lo que el orden de ejecución de determinadas instrucciones puede variar respecto a la versión secuencial de la aplicación. Esto implica que en caso de que al menos un hilo modifique una estructura compartida con el resto de hilos, es posible que otro hilo acceda a la estructura en un estado inconsistente.

Para evitar este problema, se necesita un método de sincronización entre los hilos que garantice una correcta ejecución. Primero se debe identificar las secciones críticas, es decir, las partes del código que se van a ejecutar en paralelo y que acceden a estructuras compartidas, y cuyo reordenamiento incontrolado puede provocar resultados incorrectos. Los cerrojos es el método más habitual, garantizando que una sección crítica sea ejecutada por un único hilo. Un uso correcto de los cerrojos es fundamental para evitar bloqueos y las propias condiciones de carrera. Lo habitual es que conforme mejor rendimiento se quiera obtener, mayor será el número de cerrojos necesarios. De esta forma, un menor número de direcciones de memoria estará asociada a un cerrojo en concreto, incrementando la escalabilidad. Por cada nuevo cerrojo será necesario evaluar como se relaciona con el resto, aumentando la complejidad al programador.

Por ello, la Memoria Transaccional se propone como alternativa a los métodos de sincronización tradicionales, delegando al hardware el garantizar la atomicidad. El usuario únicamente tendrá que indicar dónde se debe garantizar. Una sección crítica se ejecutará especulativamente como una transacción. Durante la transacción, las lecturas y escrituras a memoria se añaden a los conjuntos de lectura y escritura, respectivamente. En el caso de que otro hilo lea una dirección que esté en el conjunto de escritura, o escriba una dirección que este en el conjunto de lectura o escritura, se producirá un conflicto. Detectando los conflictos se garantiza el aislamiento y la atomicidad de las transacciones, es decir, que otros hilos no pueda acceder a datos especulativos y evitando que ocurra una condición de carrera. Si existe un conflicto, una manera habitual de proceder es abortar una de las transacciones. El sistema MT se encarga de deshacer todos los cambios especulativos realizados por la transacción abortada, cuya ejecución normalmente será reintentada más adelante, y así así mantener la atomicidad de la transacción.

Las implementaciones de los sistemas MT en hardware aprovechan elementos existentes en los procesadores comerciales, tales como las cachés y el protocolo de coherencia, para implementar los mecanismos necesarios (conjuntos de lectura/escritura, detección de conflictos, etc.) para soportar transacciones en hardware con una sobrecarga de rendimiento pequeña. Sin embargo, esto implica que las posibilidades de la Memoria Transaccional estarán a su vez limitadas por el propio hardware. En este trabajo se propone extender el soporte a nivel de ISA de RISCV, previamente modificado en mi TFG, con 4 nuevas instrucciones. Permitirán al programador tener un mayor control sobre las transacciones, pudiendo evadir estas limitaciones sin realizar grandes cambios en el hardware. Para poder evaluarlas, se ha escogido el benchmark labyrinth de STAMP, el cual utiliza una única transacción para calcular caminos mínimos entre parejas de puntos. Esto provoca que los conjuntos de lectura y escritura sean considerablemente grandes en relación los tamaños vistos en otras transacciones de la suite STAMP, y que se mantenga el aislamiento de determinadas direcciones durante demasiado tiempo.

Dos de las instrucciones implementan formas concretas de Early Release (Liberación temprana /ER), la cual permite eliminar del conjunto de lectura determinados bloques con el objetivo de evitar posibles conflictos. Durante una transacción, puede que determinadas direcciones no necesiten de aislamiento a partir de cierto punto. Esto ocurre sobre todo en aplicaciones que hagan uso de la privatización como técnica para mejorar el paralelismo de las aplicaciones. Esta técnica permite que se puedan hacer cálculos y modificaciones sobre la copia privada sin afectar a la copia visible al resto de hilos, de tal forma que varios hilos pueden trabajar simultáneamente tratando sobre sus copias privadas. Una vez hecha la copia privada, no es necesario detectar accesos conflictivos sobre la copia compartida, ya que será una vez se tenga una solución cuando se compruebe si las soluciones son compatibles. En las implementaciones comerciales actuales no es posible hacer eliminar bloques del conjunto de lectura. La primera instrucción, TRXSRELEASE, permitirá eliminar un bloque del conjunto de lectura, pero en caso de que el número de direcciones a liberar sea elevado, también lo será el número de Early Release. Por ello se propone una segunda instrucción TRXARELEASE que libere todas las direcciones del conjunto de lectura, pero habrá que tener cuidado de que no ocurran condiciones de carrera en implementaciones de MT que se apoyen en otros mecanismos, como los cerrojos, para garantizar continuidad. Los resultados de la evaluación muestran una mejora de alrededor del 2.25x usando TRXSRELEASE. TRXARELEASE ofrece algo más de rendimiento, pero a costa de una menor flexibilidad que su TRXSRELEASE.

Las otras dos instrucciones, TRXWSRESUME y TRXWSSUSPEND, permitirán definir un intervalo dentro de una transacción en la que las escrituras no se añadan al conjunto de escritura. Los resultados muestran como permitirán reducir, e incluso eliminar por completo, los abortos por capacidad debido a direcciones que no necesitan de mantener su valor original para restaurarlo en caso de aborto, por ejemplo, cuando se

trabaja con memoria privada a cada hilo. Sin embargo, existen determinados obstáculos que dificultan su uso, ya que impone limitaciones sobre la reutilización de memoria. Junto con TRXARELEASE, TRXWSRESUME/TRXWSSUSPEND permite obtener un rendimiento de 3.5x para tamaños de entrada pequeños, y alrededor de 2.5x para tamaños medianos, manteniendo un tamaño de cache de primer nivel modesto.

Las nuevas instrucciones se han comparado con una alternativa software que divide la transacción principal en dos, obligando a unir en software ambas transacciones para mantener un funcionamiento correcto. Los resultados han mostrado que esta versión obtiene los mismos o incluso mejores resultados que utilizando las instrucciones sobre una única transacción. Esto nos hace preguntarnos si las instrucciones, aunque se han demostrado que funcionan, pueden llegar a ser útiles. ¿Merece la pena modificar un ISA con instrucciones cuyo efecto puede obtenerse a nivel software?. Las instrucciones propuestas son simples, por lo que la lógica necesaria para incluirla en un dispositivo será mínima. Además, existirán casos en los que no se pueda dividir la transacción, pero si se puedan aplicar las instrucciones. En mi opinión, no estaría de más poder disponer de estas instrucciones o similares, de tal forma que el programador tenga las máximas posibilidades a su alcance, y sea él mismo quien decida.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

La cada vez más acuciante necesidad de ofrecer mejores prestaciones y nuevos servicios ha provocado la constante mejora de los procesadores. El aumento de la frecuencia como método de mejora de rendimiento ha quedado obsoleto debido a limitaciones técnicas, difícilmente evitables si no se optaba por otra perspectiva: los procesadores multinúcleo. Dichos procesadores disponen de varias unidades de procesamiento casi independientes que permitirán ejecutar distintos programas de forma simultánea. Sin embargo, este cambio a nivel de arquitectura ha implicado la aparición de otro reto. Para que una aplicación pueda aprovechar los núcleos adicionales, será necesario crear hilos entre los que se reparta todo o parte del trabajo total. En el caso de que dos o más hilos accedan una misma estructura compartida, y alguno de ellos la modifique, puede que el resto de hilos observe la estructura en un estado inconsistente. En determinadas aplicaciones, puede que esto no sea correcto. El programador puede querer que un determinado conjunto de operaciones se ejecute sin interferencias de otros hilos, es decir, que no observen su cambios intermedios ni modifiquen los recursos con los que esté trabajando. Para ilustrar la necesidad de sincronización un sencillo programa paralelo que calcula un histograma, en el que cada hilo suma una unidad al total de cada posible valor en función de una entrada. En este caso, es posible que dos hilos quieran hacer la suma sobre el mismo valor del histograma. Si no se hace nada, puede que solo una de las sumas sea visible, ocurriendo lo que se denomina como condición de carrera y siendo el histograma resultante incorrecto. Para obtener un resultado correcto, un hilo debería poder leer el valor previo, sumarle 1 y guardarlo en el histograma sin que otro hilo lea el mismo valor antes de que lo actualice. Estas tres operaciones conformarían una sección crítica, y resulta fundamental poder garantizar que estas secciones se ejecuten de tal forma que no ocurran condiciones de carrera. Si no fuera así, los programas en los que esto ocurra no serían correctos.

Por ello, es necesario introducir en los programas multihilo mecanismos de sincronización, la cuál permitirá al programador garantizar la ausencia de condiciones de carrera, para que el resultado final sea el deseado. A través de los cerrojos (*locks*), el método más habitual de sincronización, la ejecución de un hilo se detiene mientras otro hilo esté accediendo a un conjunto de posiciones de memoria, todas ellas protegidas por el mismo cerrojo. De esta forma, la sección de código que puedan ejecuten varios hilos a la vez se ejecuta por un único hilo, en exclusión mutua, garantizando un resultado correcto. Pongámonos en el caso de que existen muchas de estas secciones críticas. Podemos optar por utilizar un único lock para todas las secciones críticas. Este enfoque

resulta muy sencillo de utilizar, pero se está limitando la concurrencia, ya que se está protegiendo con el mismo cerrojo posiciones de memoria distintas que quizás podrían haber sido accedidas en paralelo sin afectar al resultado del programa. El otro extremo sería detener los hilos justo cuando sea necesario, es decir, locks de grano fino, donde un lock protegerá el mínimo de direcciones de memoria posibles. Esta opción otorgará una mayor escalabilidad, ya que solo bloquea los recursos estrictamente necesarios. Sin embargo, conforme el la complejidad de las estructuras compartidas sea mayor, conllevará más tiempo detectar donde colocar los locks y evitar posibles bloqueos (deadlock) por no adquirirlos y liberarlos correctamente.

¿Porqué he de detener un hilo si no estoy realmente seguro de que cuando se esté ejecutando ocurra una condición de carrera? Un método de sincronización pesimista podría implicar una pérdida de rendimiento considerable. La Memoria Transaccional (MT) propone seguir un enfoque optimista. La sección crítica se ejecutará como una transacción, la cual tiene un inicio y un final. Cada transacción podrá ejecutarse de forma concurrente con el resto, pero será necesario comprobar conforme se ejecutan las transacciones si se ha producido una condición de carrera. A este proceso se le denomina detección de conflictos, y garantiza el aislamiento y la atomicidad de las transacciones, es decir, únicamente trabajarán con datos modificados por sí mismas o ya consolidados. Si no se detectan conflictos, la transacción podrá confirmarse. En caso de ocurrir, se debe abortar la transacción. Esto implica que habrá que deshacer todos las modificaciones a registros y memoria que se hayan hecho durante la transacción. A este procedimiento se le denomina control de versiones. Si no se deshicieran estas modificaciones, se estaría volviendo a ejecutar código sobre datos que no corresponden con los que deberían según el código previo a la transacción. Esto garantiza la atomicidad, a través de la cual los datos modificados en una transacción solo serán utilizables por el resto de hilos una vez se confirme.

La Memoria Transaccional puede implementarse en software, (Software Transactional Memory /STM)[1][2], pero la sobrecarga del manejo de las transacciones afectará al rendimiento considerablemente[3]. Una característica muy interesante de MT es que muchos de los mecanismos necesarios para proporcionar las propiedades de las transacciones pueden implementarse en hardware (Hardware Transacional Memory / HTM) reutilizando elementos presentes en los diseños actuales de los procesadores. Las cachés resultan un lugar idóneo para mantener información adicional sobre cada bloque, de forma que se sepa si pertenece al conjunto de lectura y/o escritura. De igual forma, los mensajes generados por el protocolo de coherencia pueden aprovecharse para detectar conflictos, ya que los cambios en la propiedad de un bloque de caché son indicativos de potenciales condiciones de carrera.

En el mundo académico se han propuesto distintas alternativas para implementar HTM [4][5]. Cada una tiene sus ventajas e inconvenientes, y funcionarán mejor que las

otras dependiendo de qué programa se ejecute. La mayoría de estas propuestas tendría un coste de implementación excesivamente alto, por lo que no se han llevado a la práctica, por lo menos al completo. Por este motivo, las implementaciones en procesadores comerciales [6][7][8] optan por implementaciones de menor complejidad pero que exponen las limitaciones del hardware. Determinados tipos de transacciones no podrán ser ejecutados con el soporte HTM debido a diversos factores, entre los que destaca el tamaño de los conjuntos de lectura/escritura de la transacción.

En 2010, se publica la primera especificación de RISCV, una ISA basado en el concepto RISC y de licencia abierta. Esto permitirá a cualquier entidad poder diseñar y distribuir dispositivos basados en RISCV. A fecha de las escritura de este trabajo, RISCV no dispone de soporte de Memoria Transaccional. Para intentar solventar esta carencia, en mi Trabajo Fin de Grado [9] propuse incluir un soporte mínimo para Memoria Transaccional en RISCV, basado en ARM TME. Consta de las instrucciones básicas para iniciar y confirmar las transacciones, y una adicional que permite abortar una transacción desde el software. Las pruebas realizadas muestran como con este soporte mínimo, una gran parte de los benchmarks evaluados conseguían una buena escalabilidad. Sin embargo, no realicé una comparativa con versiones equivalentes basadas en locks de grano fino, por lo que no puedo asegurar que pueda mejorar su rendimiento, aún en principio siendo más sencillo de utilizar. Resultaría interesante realizar este estudio, pero en este trabajo me centro exclusivamente en probar distintas características de Memoria Transaccional, independientemente de su comparativa con locks de grano fino. Utilizaré lo propuesto en mi TFG como punto de partida para este trabajo.

1.1 Motivación

El soporte para HTM en los ISA existentes de Intel, ARM... limita la aplicabilidad de MT para determinados códigos dada su naturaleza best effort. En concreto, la cantidad de datos especulativos que se pueden gestionar en hardware es limitada, además del control del programador sobre el conjunto de lectura, con el fin de reducir conflictos. A la vista de estas limitaciones, y con el precedente de mi TFG, surge la necesidad de estudiar la conveniencia de extender el soporte MT en el ISA de RISCV para tener mayor control sobre los mecanismos de versionado de datos y detección de conflictos.

Un ejemplo de aplicación que sufre de estos problemas es labyrinth, perteneciente a la suite STAMP[10]. Este benchmark implementa una variante del algoritmo de Lee[11], cuyo objetivo es resolver eficientemente problemas de conexiones en una PCB, encontrando el camino más corto posible entre dos puntos de una matriz, teniendo en cuenta posibles obstáculos y caminos ya calculados. Para poder calcular los caminos,

cada hilo almacena cierta información intermedia necesaria que le permita obtener una solución correcta. En caso de usar una única estructura de datos, común para todos los hilos, esta información puede generar un conflicto. Sin embargo, a nivel de aplicación, estos conflictos no tendrían por que existir, ya que no implican un camino incorrecto.

A través de la privatización se consigue que las modificaciones sobre la copia privada no afecten de forma negativa e innecesaria al resto de hilos. El objetivo es que solo aquellos accesos que afecten realmente a la solución final necesiten de sincronización. Para ello, es necesario duplicar la estructura de datos implicada en la memoria privada de cada hilo. Así, la información que necesita cada hilo estará almacenada de forma independiente al resto, pero implicará un mayor uso de memoria, por lo que una privatización excesiva no es recomendable.

La privatización debe realizarse en bloque para que a la hora de copiar la estructura, los datos estén consolidados, es decir, que mientras se esté realizando la copia nadie escriba en lo que ya se ha leído. Esta circunstancia dependerá de la aplicación. En concreto, en labyrinth ocurre cuando los caminos resultantes estén incluidos al completo en la estructura. Si esto no se garantizara, la solución obtenida a partir de esa copia podría no ser correcta, o bien ser peor de lo que podría ser. Por ejemplo, en labyrinth podrían aparecer caminos que se crucen, o que fuesen más largos de lo necesario. Una forma de implementar este bloque sería con un único lock global(SGL). Permitiría que existieran varios hilos y que los caminos sean correctos, pero se calcularían de forma secuencial, por lo que en realidad no es una solución factible. Otro enfoque totalmente opuesto sería utilizar locks de grano fino. A cada recurso se le asociaría un lock, los cuales se irían adquiriendo conforme se modifique la estructura global, y posteriormente liberándolos. La escalabilidad de esta opción es mucho mejor que con SGL, pero su complejidad también, ya que necesita de una política de adquisición y liberación de los locks que evite los bloqueos (deadlock).

Este bloque se puede definir como una única transacción, permitiendo en el mejor de los casos que distintas soluciones puedan calcularse de forma concurrente sin bloquear ningún hilo. La Memoria Transaccional permitirá obtener mejor rendimiento que con cerrojos de grano grueso, pero manteniendo su sencillez. Además, hay que tener en cuenta que solo se está estudiando un único benchmark, y que una aplicación totalmente distinta necesitaría otra aproximación si se usaran locks de grano fino. Con las transacciones, únicamente es necesario indicar su comienzo y final, obteniendo el mismo comportamiento en cualquier benchmark. Sin embargo, las transacción resultantes puede considerarse excesivamente largas respecto a las prestaciones de las que podrá disponer la Memoria Transaccional en procesadores de uso doméstico. Esto impedirá que no se pueda obtener el máximo rendimiento posible al aplicar transacciones.

1.2 Objetivos

Por estos motivos, la meta principal de este trabajo es ofrecer distintas soluciones que permitan evitar conflictos en posiciones de memoria que han sido previamente accedidas en la transacción, y ejecutar transacciones cuyos conjuntos de escritura excedan el espacio disponible para almacenar los valores especulativos.

Existe la posibilidad de que un recurso que se haya leído en un instante de una transacción no requiera de aislamiento a partir de cierto punto. Las circunstancias para que esto sea posible estarán determinadas por la propia aplicación. En labyrinth, esto ocurre al hacer la privatización de la estructura global, donde la lectura completa provoca que la adición de cualquier nueva solución aborte una transacción que haya hecho la copia. Si bien es cierto que garantiza que ninguna solución sea incorrecta, el aislamiento es de un grano excesivamente gordo, ya que únicamente interesa saber si los caminos se cruzan o no. Resulta factible plantear lo siguiente: si realmente solo me interesa comprobar que los caminos no se crucen, es posible retrasar la comprobación de si es correcto hasta tener el camino completo. Esto implica mantener la solución en memoria privada mientras se esté construyendo, y escribirla en la estructura global una vez completado. Esta escritura permitirá comprobar si las nuevas soluciones son compatibles, ya que en caso contrario se generará un conflicto.

Para que esto sea posible, es necesario tener en cuenta una propuesta ya existente. Early Release(ER) propone que una vez no sea necesario seguir aplicando el aislamiento, el recurso se elimine del conjunto de lectura, no generando conflictos a partir de ese punto, siempre y cuando no pertenezca al conjunto de escritura. Solo aplica al conjunto de lectura porque la eliminación del conjunto de escritura implica no recuperar el valor original, lo que puede que no sea utilizable en muchos casos. De esta forma, en labyrinth se podrá liberar la estructura global una vez hecha la copia de la misma en la memoria privada del hilo. Esta modificación implica que dos soluciones incompatibles puedan no llegar a generar conflicto. Esto ocurrirá cuando una vez liberado la estructura global, y antes de añadir una nueva solución, otra transacción añada la suya. Por ello, es necesario que el programador compruebe explícitamente si la nueva solución es correcta. En caso contrario, tendrá que abortar la transacción para que las escrituras sobre la estructura global al intentar añadir la nueva solución se deshagan. Para estas circunstancias, resulta útil la instrucción de cancelación, que permite abortar una transacción desde el software.

El único que conoce cuando se debería aplicar ER es el programador, por lo que en este trabajo se propone una instrucción que implemente esta funcionalidad. Es necesario recordar que la granularidad del sistema de Memoria Transaccional es el bloque de caché. Por lo tanto, un Early Release en este contexto eliminará el bloque que se le pase como parámetro del conjunto de lectura poniendo el bit correspondiente a 0.

Debido a la granularidad utilizada, con un único ER se pueden eliminar múltiples datos. Por ejemplo, si se están utilizando enteros de 8 bytes, y el bloque de caché es de 64 bytes, con un único ER se liberarán 8 elementos. Esto provoca que, en el caso de querer eliminar, en este caso, un único entero, se deberá guardar en memoria de forma alineada respecto a los bloques de caché (una dirección múltiplo del tamaño del bloque). Si no se tuviera esto en cuenta, al querer liberar un único elemento, se podría estar liberando el resto de elementos del bloque que todavía requieren de aislamiento. Esto resulta ser un inconveniente importante de ER, ya que expone al programador detalles de la arquitectura que de otra forma suelen ser invisibles (p.ej. tamaño del bloque de caché).

Por otro lado, es bastante habitual que para simplificar su uso, las transacciones vigilen todos los accesos. A veces, esto resulta un inconveniente ya que un exceso de recursos vigilados puede impedir la confirmación de una transacción. Esto ocurre en labyrinth, donde de todos los accesos contenidos en la transacción, únicamente es necesario mantener las lecturas al hacer la copia y las lecturas y escrituras al añadir una nueva solución a la estructura global. Al usar únicamente la caché de primer nivel para mantener los conjuntos de lectura/escritura, el tamaño de los conjuntos estará limitado por su tamaño. El conjunto de lectura/escritura de cada transacción se define independientemente del tamaño de las cachés, es decir, van a haber casos donde los conjuntos van a ser si o si de mayor tamaño que las cachés. Esto provocará que un bloque haya sido accedido durante una transacción pueda ser expulsado de la caché de primer nivel. Si en los niveles inferiores los bloques no distinguen si son transaccionales o no, entonces ese bloque dejaría de estar aislado, por lo que podría ocurrir una condición de carrera, pero no generarse el conflicto. Por ello la transacción abortará por capacidad. Si el programador pudiera establecer que accesos no son realmente necesarios, se podría minimizar este problema. En este trabajo se propone una pareja de instrucciones (TRXWSRESUME/TRXWSSUSPEND) que permitan definir intervalos dentro de una transacción donde no sea necesario vigilar recursos en caso de modificarlos. Se ha decidido solo aplicarlo a los modificados ya que son los más costosos de manejar.

Después de observar detenidamente labyrinth, se observa como en realidad no es necesario utilizar una única transacción. Gracias a la privatización de la matriz global (*grid*), la expansión del camino se puede realizar fuera de la transacción. La detección de conflictos se realiza en la copia y en la modificación del grid global. Por lo tanto, es posible reducir la granularidad de la transacción y definir dos transacciones nuevas, una para la copia y otra para la modificación. Ahora bien, si no se realizan más cambios, esta versión no sería equivalente a utilizar una única transacción. En caso de abortar al modificar el grid, únicamente se volvería a intentar hacer la modificación, pero nunca se volvería a copiar el grid y calcular un nuevo camino. Entonces, para que esta versión sea correcta, es necesario que estas dos transacciones se unan a nivel software. En

caso de que la segunda transacción aborte porque haya dos caminos que se crucen, se saltará a la primera transacción. Estos cambios implican una mayor complejidad en el diseño de la aplicación, ya que cada una necesitará de distintos métodos para unir las transacciones. Además, al utilizar un única transacción, se puede utilizar los bloques atomic de muchos lenguajes de programación, y que encapsulan distintos métodos de sincronización, entre ellos la Memoria Transaccional. Sin embargo, al reducir el tamaño de las transacciones, será posible que, sobre el mismo sistema de Memoria Transaccional, se obtenga un mejor rendimiento.

Una vez se consiga implementar las tres alternativas, se evaluarán sobre el propio labyrinth. Para Early Release, se probará con distinto número de hilos para evaluar como evolucionan los conflictos conforme se incrementan el número de transacciones activas. Se utilizará una caché lo suficientemente grande para que no haya problemas de capacidad para los conjuntos de lectura/escritura, y que no alteren el impacto real de los conflictos. De esta forma, se espera concluir que grado de escalabilidad puede aportar ER a un benchmark con transacciones largas y alta probabilidad de conflicto. Para TRXWSRESUME/TRXWSSUSPEND, el objetivo será observar si es posible reducir los abortos por capacidad que ocurran por problemas de espacio. Se utilizará una caché más pequeña y tamaños de entrada más grandes para maximizar este problema y ver si esta propuesta puede solucionarlo. En caso de reducirse, existirá una contención similar a la evaluación de ER, por lo que esta se utilizará conjuntamente con ER para poder valorar si al reducir los abortos por capacidad, se consiguen nuevas oportunidades de concurrencia. La versión de dos transacciones se incluirá en ambos estudios para poder observar hasta que punto una menor granularidad puede ser suficiente para minimizar las limitaciones de la Memoria Transaccional.

2 Estado del arte

2.1 Garantizar una ejecución correcta en un entorno paralelo

Una aplicación paralela requiere de un esfuerzo adicional por parte del programador para poder aprovechar al máximo los distintos núcleos de forma correcta. Dependiendo del tipo de aplicación multihilo que se esté ejecutando, es posible que algunos de los hilos se estorben entre sí. Supóngase una variable A con un valor inicial de 1 y los hilos H1 y H2, cada uno ejecutando el código que se indica en la figura 2.1. El objetivo es que ambos hilos puedan ejecutarse en paralelo y obtener un resultado de A=3. Supóngase consistencia secuencial, es decir, todas los accesos a memoria se realizan en el orden del programa, evitando cualquier tipo de reordenamiento dentro de cada hilo. Aún así, todavía puede existir un reordenamiento entre los distintos hilos respecto a la versión secuencial que provoque un resultado incorrecto, ocurriendo la denominada condición de carrera. Puede (Figura 2.2) que H1 y H2 ejecuten la instrucción 1, y por lo tanto que A=1 para cada hilo. Sumarán 1 a su A y escribirán el valor 2 en memoria. Este resultado es incorrecto, ya que el objetivo es que A=3. También puede ocurrir que uno de los hilo ejecute las instrucciones 1 y 2, y el otro ejecute entonces la 1, obteniendo también A=2.

Para conseguir el valor correcto es necesario que uno de los hilos ejecute todas las instrucciones antes de que el otro hilo ejecute la primera, es decir, que no ocurra una condición de carrera. Se denomina sección crítica a la secuencia de instrucciones que accede a recursos compartidos y que no puede ser ejecutada en paralelo para garantizar el resultado correcto del programa. Para que sea posible ejecutar secciones críticas, es necesario alguna forma de sincronización, de tal forma que se garantice que esas operaciones se ejecuten en un orden correcto (Figura 2.3).

Uno de los métodos de sincronización más utilizados es el lock, o cerrojo. Pueden

```
1. lectura (r1,A)
2. r1 = r1 +1
3. escritura (r1,A)
```

Figura 2.1: Pseudocódigo para ejemplo

```
Hilo 1                          Hilo 2
-------------------------------------------------------------
1. lectura (r1,A) -> A==1      | 1. lectura (r1,A) -> A==1
2. r1 = r1 +1 -> r1==2         | 2. r1 = r1 +1 -> r1==2
3. escritura (r1,A)  -> A==2   | 3. escritura (r1,A)  -> A==2
```

Figura 2.2: Ejemplo de ejecución con resultado incorrecto

```
Hilo 1                          Hilo 2
-------------------------------------------------------------
1. lectura (r1,A) -> A==1      | Sincronización
2. r1 = r1 +1 -> r1==2         | Sincronización
3. escritura (r1,A)  -> A==2   | Sincronización
           ...                 | 1. lectura (r1,A) -> A==1
           ...                 | 2. r1 = r1 +1 -> r1==2
           ...                 | 3. escritura (r1,A)  -> A==2

=============================================================

Hilo 1                          Hilo 2
-------------------------------------------------------------
Sincronización                 | 1. lectura (r1,A) -> A==1
Sincronización                 | 2. r1 = r1 +1 -> r1==2
Sincronización                 | 3. escritura (r1,A)  -> A==2
1. lectura (r1,A) -> A==1      |           ...
2. r1 = r1 +1 -> r1==2         |           ...
3. escritura (r1,A)  -> A==2   |           ...
```

Figura 2.3: Las únicas ejecuciones correctas gracias a la sincronización

implementarse de distintas formas, según el procesador donde se ejecuten, pero se usan siempre igual. El cerrojo se utiliza como un indicador: si un hilo esta ejecutando un bloque, el cerrojo estará cerrado. Si no, estará abierto. En el caso de estar cerrado, el hilo que intente adquirir el cerrojo no podrá continuar su ejecución hasta que el cerrojo sea liberado por el hilo que lo adquirió en un principio. Este método tiene la ventaja de que, de forma sencilla, permite ejecutar una sección crítica de forma exclusiva, es decir, únicamente un hilo podrá ejecutarla hasta que libere el lock. Sin embargo, conforme la aplicación se complique, pueden surgir una gran cantidad de problemas que es necesario resolver. En caso de no realizar correctamente la adquisición y liberación de los locks, es posible que un hilo no libere nunca un lock, provocando que él mismo y el resto de hilos que lo quieran adquirir se queden bloqueados a la espera de algo que no va a ocurrir. A este problema, bastante común, se le denomina deadlock. También puede ocurrir lo opuesto, es decir, una liberación del lock de forma prematura. En ese caso, más de un hilo estarían ejecutando una sección crítica que según el diseño de la aplicación, solo debería ejecutarla un único hilo. Esto provocará que sigan existiendo las condiciones de carrera que se pretendían eliminar, y por lo tanto, el resultado de la aplicación no será correcto. Otro problema bastante reseñable es la inversión de prioridad: un hilo con baja prioridad adquiere un lock, por lo que el resto de hilos tendrán que esperar a uno cuyo avance será mucho más lento. Todos estos problemas disponen de solución, pero dependiendo de la aplicación, puede resultar bastante compleja.

Dependiendo de cómo los locks se ajusten a las necesidades reales de sincronización, se suelen clasificar en dos categorías. Los locks de grano grueso (coarse-grained) abarcan secciones críticas de mayor tamaño del necesario, reduciendo la complejidad de uso al ser más intuitivos. Se sabe que en una sección es posible que se acceda a cualquier recurso, pero no se conoce exactamente cuales hasta su ejecución. Por lo tanto, se define un lock para esa sección, el cual protegerá todos los recursos. Esto implica que se estarán bloqueando hilos innecesariamente en caso de que los recursos utilizados no estén compartidos.

Si por el contrario se asocia un lock exclusivamente a cada recurso o recursos cuyos accesos necesiten de sincronización, se estarán usando locks de grano fino (fine-grained). La escalabilidad será mucho mayor que los locks de grano grueso, ya que los hilos se detienen solo cuando de otra forma ocurriría una condición de carrera. Alcanzar esta solución variará de complejidad dependiendo de la aplicación. Si existen muchas posibilidades en las que pueda producirse una condición de carrera, diseñar y evaluar la implementación puede llegar a ser costosa, sobre todo si los programadores no disponen de experiencia.

¿Pero y si realmente no fuese necesario utilizar cerrojos para garantizar la sincronización? El enfoque lock-free (libre de cerrojos) propone que, ciertas secciones de la aplicación, pueden reescribirse de tal forma que se garantiza una ejecución correcta

```
while (flag == 0);     |    A=5;
print(A);              |    flag=1;
```

Figura 2.4: Ejemplo de uso de barrera de memoria

pero sin utilizar cerrojos. Para ello, lo habitual es utilizar directamente determinadas operaciones atómicas implementadas en hardware, las cuales garantizan el acceso exclusivo a un recurso. Esto elimina la sobrecarga de adquirir y liberar el lock, y también evita que un hilo que tenga adquirido el lock bloquee al resto al cambiar de contexto. En el mejor de los casos, esto permitirá una mejora de rendimiento. Sin embargo, realizar los cambios necesarios en determinadas situaciones puede llegar a ser muy costoso, y además, tampoco está libre de problemas en caso de no hacerse de forma correcta. Por ejemplo, es necesario conocer el modelo de consistencia de memoria del procesador. En caso de no disponer de consistencia secuencial, como ocurre en la mayoría de procesadores actuales, pueden ser necesarias las denominadas barreras de memoria para poder garantizar cierto orden en los accesos a memoria. Si no se garantizara este orden, es posible que determinados accesos se reordenen y provoquen un comportamiento no deseado sobre el resto de hilos. Un caso bastante habitual es la modificación de una variable de control. La figura 2.4 muestra un ejemplo de este problema. La intención de este código es que el hilo 1 no imprima el valor de A hasta que el hilo 2 modifique su valor. Una vez escrito, actualiza la variable flag de tal forma que el hilo 1 sale del bucle e imprime el nuevo valor. Con reordenamientos, podría ocurrir que la actualización de flag ocurra antes que la modificación de A por parte del hilo 2, por lo que el hilo 1 podría acabar imprimiendo el valor inicial de A. Para evitar esto, es necesario insertar una barrera de memoria en el código del hilo 2 entre ambas escrituras. De esta forma, se evitará cualquier reordenamiento entre la parte previa y posterior a la barrera.

En caso de plantearse estos cambios en un proyecto ya avanzado, podría implicar rediseñar buena parte de él. En caso de plantearse esta alternativa en fases iniciales del proyecto, podría ocurrir que llegar una solución correcta consuma tanto tiempo que hiciese la aplicación inviable. También existe la incertidumbre de que la posible mejora de rendimiento no sea tan significativa como para justificar una alta inversión de recursos.

2.2 Memoria Transaccional Hardware

Con el objetivo de ofrecer un equilibrio entre complejidad de uso y rendimiento, M.Herlihy y J. Eliot B. Moss (H&M)[12] propuso implementar en hardware la detección de condiciones de carrera. De esta forma, el programador solo tendrá que indicar donde cree que puede ocurrir una condición de carrera (coarse-grained), pero podrá conseguir un rendimiento cercano a proteger cada recurso (fine-grained).

En bases de datos, una transacción es un conjunto de operaciones que cumplen las propiedades ACID. Una transacción debe ser atómica (A): en caso de que no se puedan completar todas las operaciones, la estructura mantendrá el estado previo a ejecutar la transacción, garantizando que no se corrompa, por ejemplo por una avería. Otro caso en el que una transacción no puede completarse es cuando se accede a una estructura compartida. Si ambas transacciones modificaran la misma estructura al mismo tiempo, el estado resultante no sería coherente. Por ello, es necesario el aislamiento (Isolation/I) de las transacciones para que la estructura compartida solo sea modificada por una única transacción en un momento concreto. Si se detecta que dos transacciones están modificando la estructura, se abortará una de ellas para que la otra pueda terminar. La transacción abortada podrá reintentarse posteriormente utilizando los datos de la otra transacción. Estas dos propiedades permiten que las transacciones se ejecuten en paralelo de forma especulativa, es decir, se comprueba si se está accediendo a una estructura compartida al ejecutar las operaciones en vez de al principio. Las transacciones que se puedan confirmar deberán ser también consistentes (C), es decir, encajar dentro de las restricciones de la de datos. Por último, todas las escrituras que se realicen deberán ser persistentes (Durable/D), de tal forma que los datos se puedan recuperar en cualquier momento.

La Memoria Transaccional aplica el concepto de transacción a memoria, que correspondería con una estructura compartida en bases de datos. Una transacción es un conjunto de instrucciones ejecutadas por un hilo que realiza una serie de cambios especulativos sobre memoria. Entre las transacciones debe existir **aislamiento**, es decir, los cambios realizados por una no podrán afectar a otra transacción activa. En caso de que esto ocurra, se generará un conflicto, y una de las transacciones deberá abortar y deshacer todos los cambios especulativos sobre memoria y también sobre los registros. Esto garantiza la **atomicidad** de las transacciones, de tal forma que la otra transacción podrá continuar sin verse afectada por sus cambios. Estas dos propiedades permiten que el resultado de ejecutar las dos transacciones sea el mismo que si se ejecutaran una detrás de otra, es decir, sin condiciones de carrera. Si una transacción no ha tenido que abortar podrá confirmarse (commit) y hacer visible al resto de hilos todos sus cambios especulativos.

Para poder utilizar las transacciones, es necesario añadir nuevas instrucciones al ISA objetivo para que el usuario pueda manejarlas. Como mínimo serán necesarias dos instrucciones: una para iniciar una transacción y otra para confirmarla. A partir de este punto, cada implementación puede decidir por añadir instrucciones adicionales que hagan más flexible el sistema de MT o no. Por ejemplo, muchas implementaciones añaden otra instrucción que permita al software abortar una transacción por motivos ajenos al propio sistema de MT.

Para detectar los conflictos, es necesario que determinadas direcciones de memoria se añadan a los denominados conjuntos de lectura y escritura (Read Set (RS)/ Write Set (WS)). En el RS se incluirían las direcciones provenientes de una lectura, y en el WS de una escritura. De esta forma, para determinar si existe un conflicto, bastará con comprobar si la dirección implicada está contenida en estos conjuntos. En caso de pertenecer al WS, cualquier acceso sobre esta dirección provocará un conflicto. Si pertenece al RS, únicamente lo hará una escritura, ya que otra lectura no implica una condición de carrera. Se puede optar por añadir todas las direcciones accedidas, simplificando el uso de las transacciones, pero añadiendo direcciones que pueden ser no necesarias. Otro enfoque sería indicar explícitamente que accesos deben modificar los conjuntos a través de nuevas instrucciones de acceso a memoria, una para leer y otra para escribir. Esto aumenta la complejidad tanto de uso como de implementación, pero permitirá que los conjuntos de lectura/escritura únicamente contengan las direcciones necesarias.

También suele estar presente la pregunta de cuándo detectar los conflictos. La forma más intuitiva sería detectarlos justo en el momento que se produce el conflicto. Permite que una transacción que ha provocado un conflicto no siga ejecutando instrucciones innecesarias. Sin embargo, se puede dar la circunstancia de que ninguna transacción pueda confirmarse, es decir, que las transacciones se aborten continuamente. Si por el contrario se retrasa la detección de conflictos hasta la confirmación, se garantiza que al menos una transacción pueda confirmar. Sin embargo, las transacciones que al final tengan que abortar habrán ejecutado instrucciones innecesarias. Esta aproximación suele estar más asociada a la Memoria Transaccional Software (STM), aunque también es posible implementarla en hardware.

Uno de los puntos fuertes de la Memoria Transaccional es que muchas de sus características pueden implementarse con mecanismos ya existentes en el hardware (Memoria Transaccional Hardware/HTM). Los protocolos de coherencia se encargan de controlar la propiedad de los recursos en las cachés. En el momento en que se produzca un mensaje para modificar la accesibilidad de un recurso, se puede determinar cómo ha sido accedido desde otro hilo. Por lo tanto, se puede detectar conflictos con ese acceso. Será necesario que puedan diferenciarse que direcciones se han accedido dentro de la transacción o no. Es bastante habitual que los bloques dispongan de un par de bits que indiquen si pertenecen a los conjuntos de lectura y escritura.

Utilizar el bloque de caché como granularidad de detección de conflicto puede provocar los denominados falsos conflictos, donde un acceso no conflictivo genere conflicto, ya que no se puede diferenciar a nivel de palabra, es decir, cargas y almacenamientos a un recurso concreto. Un ejemplo bastante simple pero ilustrativo sería el siguiente. Una transacción A modifica un objeto, construido con cualquier lenguaje orientado a objetos. Justo después, otra transacción B lee otro objeto totalmente distinto, y que no mantiene ninguna relación con el otro objeto. En caso de que uno de los atributos del

primer objeto esté en el mismo bloque de caché que otro atributo del segundo objeto, la transacción A abortará. Este tipo de problemas es posible solucionarlo a través del alineamiento de memoria, es decir, rellenar memoria para garantizar que un bloque de caché solo contenga datos de un único objeto. El malgasto de memoria puede llegar a ser considerable dependiendo de la compatibilidad por defecto del tamaño del objeto y del tamaño del bloque de caché. Además, en lenguajes de bajo nivel es probable que sea necesario indicar este alineamiento manualmente para cada tipo de dato. Otra implicación de utilizar las cachés para mantener los conjuntos de lectura y escritura es que sus tamaños estarán limitados por la asociatividad de la caché. En caso de que alguna de las direcciones accedidas durante una transacción no tenga espacio en el conjunto, la transacción abortará por capacidad, ya que no se puede garantizar el aislamiento de la transacción. El tamaño de la propia caché también afecta a los conjunto de lectura y escritura pero de forma indirecta: el tamaño de la caché determinará cuántos conjuntos habrán, por lo que a mayor tamaño de caché, habrá mejor distribución de los bloques a lo largo de la caché.

En caso de ocurrir un conflicto, una de las transacciones debe abortar para poder garantizar la atomicidad de las transacciones. Esto implica que durante una transacción, una dirección modificada tendrá dos valores: un valor especulativo, es decir, el que está utilizando durante la transacción, y el valor antiguo, es decir, el valor que la dirección tenía antes de ser modificada durante la transacción. A esto se le denomina **control de versiones**, y según como se aplique, existen dos enfoques. Un control de versiones **ansioso** (eager) almacena el valor especulativo directamente en memoria y el antiguo en un buffer. En caso de que la transacción confirme, solo se tendrá que vaciar el buffer, pero en caso de abortar tendrá que volver a copiar los valores antiguos en memoria. Si por el contrario se aplica un control de versiones **perezoso** (lazy), se guardarán los valores especulativos en un buffer y se mantendrán los antiguos en memoria. En caso de abortar solo habrá que vaciar el buffer, pero en caso de confirmar habrá que mover los valores especulativos a memoria. Aunque cada opción funcionará mejor dependiendo del número de abortos, el enfoque perezoso tiene una gran ventaja: puede reutilizar las cachés como buffer para almacenar los valores especulativos, de tal forma que los cambios necesarios para soportarlo son mínimos. Únicamente habrá que copiar el bloque con el valor antiguo a niveles inferiores de la jerarquía de memoria. En caso de aborto, se invalidarán los bloques modificados durante la transacción y el valor antiguo podrá volver a leerse.

Al aplicar Memoria Transaccional sobre una CPU, hay que tener en cuenta que el estado completo de la ejecución lo representa la memoria, pero también los registros. Por lo tanto, se deben preservar sus valores antiguos para que en caso de aborto se pueda recuperar el estado previo a la transacción. Una opción es guardar los valores a través del software en memoria. Esto implica que al iniciar la transacción se ejecutarán una serie de instrucciones que accedan a los registros y almacenen los valores en memoria.

	Eager VM	Lazy VM
Eager CD	LogTM,UTM	TSX,TME,IBM,Rock
Lazy CD	X	TCC

Tabla 2.1: Tabla resumen que agrupa las distintas implementaciones mostradas en la sección según el control de versiones y la detección de conflictos

En caso de aborto se tendrán que ejecutar instrucciones que vuelvan cargar los valores antiguos. Sin embargo, la opción más utilizada es utilizar las estructuras existentes en los procesadores, como puede ser la Register Alias Table (RAT) o el Arquitecture Register File (ARF), que pueden llegar a permitir hacer la copia de todos los registros en un único ciclo.

Las distintas propuestas comerciales tienen en común que utilizan una detección de conflictos ansiosa y un control de versiones perezoso (Figura 2.1). Sin embargo, cada implementación tiene sus peculiaridades, tal y como se puede ver en la tabla 2.2. **Rock HTM**[13] utiliza la caché de segundo nivel para mantener el conjunto de lectura, y la Store Queue (SQ) para el conjunto de escritura. **IBM BlueGene/Q**[6] utiliza también la caché de segundo nivel para mantener el conjunto de escritura. Otra característica relevante es la posibilidad de tener transacciones irrevocables, las cuales se garantiza que siempre se confirmen. **IBM System Z**[14] mantiene el conjunto de lectura en la caché de primer nivel y el conjunto de escritura en un buffer independiente. También incorpora una nueva política de resolución de conflictos: en vez de abortar directamente, la transacción que provoque el acceso a memoria conflictivo espera a la otra durante un intervalo determinado. Si esta última transacción se confirma, entonces se podrá utilizar ese valor y no tendrá que abortar, si no, terminará por abortar.**IBM POWER**[8] también incluye soporte transaccional. Destacan las instrucciones TRESUME y TSUSPEND. Los accesos a memoria entre estas instrucciones no se añadirán a los conjuntos de lectura ni escritura y serán visibles inmediatamente por el resto de hilos, además de que tampoco se resolverán conflictos hasta que se ejecute TRESUME. La propia IBM indica que su función principal es la depuración de las transacciones, pudiendo hacer llamadas al sistema durante la transacción. **Intel TSX**[7] utiliza la caché de primer nivel para mantener tanto el conjunto de lectura como el conjunto de escritura. Incluye también Hardware Lock Elision, que permite que un código basado en locks utilice transacciones en caso de que el procesador tenga soporte de RTM, y en caso de que no, se ejecuta con los locks. Destacan también las instrucciones XSUSLDTRK y XRESLDTRK, que permiten definir un intervalo dentro de una transacción donde las lecturas no se añadan al conjunto de lectura. Sin embargo, si se llama a XSUSLDTRK de forma anidada la transacción abortará. Esto implica que no se podrán utilizar librerías que también utilicen estas instrucciones. Respecto a **ARM TME**, todavía no se han publicado detalles concretos de implementación. Solo se conocen las instrucciones

Mantenimiento de	Conjunto Lectura	Conjunto Escritura	Versiones
TSX	L1	L1	L1
TME(*)	L1	L1	L1
POWER	L2	L2	L2
TCC	L1	L1	L1
LogTM	L1	L1	Memoria virtual
BlueGene/Q	L2	L2	L2
Rock	L2	Store buffer	Store buffer
SystemZ	L1	Store buffer	Store buffer
UTM	Memoria virtual	Memoria virtual	Memoria virtual

Tabla 2.2: Tabla resumen que agrupa las distintas implementaciones mostradas en la sección según dónde mantienen los conjuntos de lectura y escritura, y los valores especulativos. (*) Todavía no se han publicado detalles de la implementación. Estos datos corresponden con el modelo que han implementado en gem5.

que incluye.

En el mundo académico también se han realizado otras propuestas (Figuras 2.1 y 2.2). Las más relevantes tienen en común que priorizan explorar las posibilidades de la Memoria Transaccional sin dar mucha importancia al posible coste y complejidad de implementación, tanto a nivel software como hardware. **H&M** acuña por primera vez el término de Memoria Transaccional. No plantea una implementación específica, sino que define las bases para cualquier implementación. **TCC**[15] aplica un enfoque peculiar de las transacciones: en vez de solo aplicarlas a las secciones críticas, se aplicará a todo el código paralelo. Esto implica una modificación en el funcionamiento del protocolo de coherencia, que en vez de actuar en cada acceso, retrasará los mensajes de coherencia hasta la confirmación de las transacciones. **LogTM**[4] propone una implementación novedosa de control de versiones ansioso: guarda los valores antiguos en la memoria virtual del proceso, teniendo que realizar un acceso a memoria adicional. Esto permite eliminar la necesidad de un buffer adicional, pero habrá que acceder a memoria para poder abortar una transacción. **Unbounded Transactional Memory** (UTM)[5] se apoya en el Sistema Operativo para que las transacciones puedan ejecutarse aunque haya cambios de contexto, fallos de página o sustituciones de bloques de caché transaccionales. Sin embargo, las modificaciones del Sistema Operativo obligan a que exista una colaboración entre el procesador y el Sistema Operativo para manejar las transacciones.

Un detalle importante es que las propuestas comerciales son best-effort. Garantizan que las transacciones que se ejecuten lo hagan sin que ocurran condiciones de carrera,

pero no garantizan que una transacción se ejecute. Por ello, para garantizar la continuidad de la ejecución, se suele acompañar a las aplicaciones con una librería que implemente un camino alternativo (*Fallback path*). En caso de que se compruebe que una transacción no ha confirmado en varios intentos, es posible que esté ocurriendo un livelock. Puede ocurrir que varias transacciones intenten acceder a los mismo recursos al mismo tiempo siempre que se ejecuten, abortándose continuamente entre sí. En este caso, se optará por otro método que sí garantice continuidad, tal y como lo son los locks, habitualmente un Sigle Global Lock (SGL). Cuando una transacción se inicia correctamente, añade el SGL al conjunto de lectura. De esta forma, cuando una transacción acabe adquiriendo el SGL, la escritura provocará que todas las transacciones activas aborten. Así se consigue que ninguna otra transacción acceda a datos modificados por la ejecución no transaccional. Será determinante para el rendimiento de la aplicación cuándo optar por el fallback. Si se coge demasiado pronto, se estará despreciando la concurrencia de transacciones que habitualmente sí pueden ejecutarse a la vez. Si por el contrario se retrasa demasiado, se ejecutarán transacciones que en ningún momento podrán confirmarse por tener una alta probabilidad de conflicto. Escoger un fallback adecuado requiere de un proceso de investigación y evaluación que no corresponde a este trabajo, por lo que se utilizará el disponible en el repositorio del grupo de investigación CAPS de la Universidad de Murcia.

2.3 Simulador de computadores: gem5

Si todas las ideas en el mundo de los microprocesadores hubiesen tenido que ser implementadas en su totalidad para probarlas, hoy en día no existirían tantos avances en el sector. Gracias a los simuladores, es posible verificar a nivel lógico el funcionamiento de cualquier planteamiento. Al estar implementados en software, cualquier persona con una máquina más o menos funcional podrá implementar sus diseños, y también aportar a otros proyectos.

Cuanto más nivel de detalle ofrezca el simulador más lento será, pero se podrá obtener mayor información de las ejecuciones. Esto aportará más valor y confianza a las conclusiones que se lleguen a obtener, tanto por parte del programador como de los que interpreten los resultados. La simulación también puede enfocarse a la funcionalidad y eficiencia. Los detalles obtenidos de la simulación serán mucho menores, pero el tiempo de simulación se reducirá considerablemente. Qué alternativa utilizar dependerá del objetivo del estudio, y en todo caso, no tienen porqué ser excluyentes. Se puede realizar un boceto inicial en un simulador más sencillo y eficiente, y una vez se verifique que funciona correctamente, trasladar el diseño a un simulador más detallado. El proceso es más costoso que decantarse directamente por una de las opciones, pero permitirá detectar un error antes de modificar un simulador más complejo.

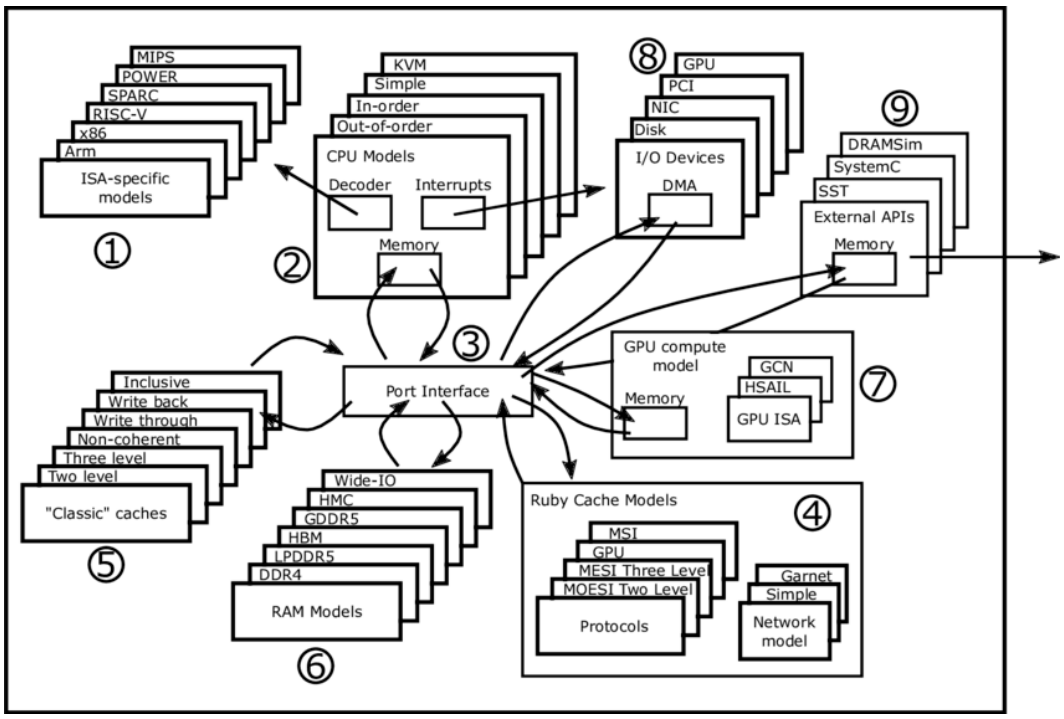

Figura 2.5: Componentes gem5

Para este trabajo, la elección de un simulador ha sido sencilla. gem5[16] es un simulador de arquitecturas multinúcleo que dispone de un gran número de opciones de configuración y de muchos diseños ya implementados, además de un detallado modelo de coherencia. En la figura 2.5 se recogen todos los componentes que conforman toda la estructura de simulación de gem5. Los más relevantes para este trabajo se explican a continuación:

- ISA. Con soporte para un gran número de ISAs, incluido RISCV, gem5 permite simular casi cualquier aplicación. Cada una de las instrucciones que se definan en el Decoder de cada ISA, se encargará de leer los registros que le hagan falta, indicará a la CPU que hacer, y modificará los registros con la respuesta. La interfaz con la CPU es la misma para todos los ISAs, por lo que muchas de las instrucciones en gem5 de todos los ISAs tiene una implementación similar. Esto facilita bastante la inclusión de nuevas instrucciones al poder reutilizar otras ya existentes. Esta característica se ha podido aprovechar en este trabajo, utilizando como base la implementación de ARM de Memoria Transaccional.

- CPU. Ofrece principalmente tres modelos distintos. AtomicSimple simula un procesador en orden con accesos a memoria atómicos, es decir, en cuanto se produce

un acceso, éste se resuelve inmediatamente. TimingSimple simula el mismo tipo de procesador, pero incluye mayor detalle en los accesos a memoria. Ahora cada acceso a memoria se lanzará, y se seguirán ejecutando instrucciones sin esperar a su resolución. Será ciclos más tarde cuando se obtenga la respuesta de memoria, siendo más fidedigno al comportamiento real de los accesos a memoria. Por último, DerivO3 se comporta como un procesador fuera de orden, donde cada etapa es simulada y pudiendo obtener estadísticas de cada una. Los accesos a memoria funcionan de la misma forma que TimingSimple.

- Ruby. Uno de los principales atractivos de gem5. Unifica todos los componentes de la jerarquía de memoria en una misma interfaz. Una petición a memoria recorrerá toda la jerarquía encapsulada como un Packet, el cuál contendrá información del tipo de petición, quién ha realizado la petición... Además, permite definir protocolos de coherencia a través de un lenguaje propio denominado SLICC, teniendo únicamente que trabajar con conceptos relacionados con la coherencia independientemente del lenguaje de programación, en este caso C++.

2.3.1 Memoria Transaccional en gem5 público

Buscando seguir mejorando una de sus mayores virtudes, gem5 incluye a partir de su versión 20.0 soporte para Memoria Transaccional. Son miembros de ARM quien lo implementan, por lo que únicamente incluye las instrucciones de TME. Sin embargo, gracias al diseño modular de gem5, se podrá reutilizar toda la lógica de Memoria Transaccional para cualquier otro ISA. Cada una de estas instrucciones llamará a la nueva función initiateHtmCmd. A través de esta interfaz, únicamente será necesario indicar que tipo de operación se quiere realizar. Las instrucciones TSTART,TCOMMIT y TCANCEL tendrán asociadas las flags HTM_START,HTM_COMMIT y HTM_CANCEL. Esta función creará un Packet que contendrá una de estas flags y lo enviará a memoria. Uno de los motivos para ofrecer una interfaz distinta a los comandos transaccionales y al resto de accesos a memoria, es que realmente no acceden a ninguna dirección de memoria. Por ello, no es necesario realizar una traducción de dirección virtual a dirección física.

Una vez creado el paquete, seguirá el camino habitual de cualquier acceso a memoria hasta llegar a la caché privada de la CPU. Es aquí donde se implementa toda la lógica transaccional. Al recibir una petición de CPU y antes de acceder a los bloques, se comprueba si es un comando transaccional o no. En caso afirmativo, aplicará los cambios necesarios sobre el estado de la transacción en función del comando, y responderá a la CPU sin haber accedido a ningún bloque de datos. En caso de que la petición sea un acceso a datos, el protocolo comprobará si se ha ejecutado previamente un HTM_START. En caso afirmativo, una lectura añadirá el bloque al conjunto de lectura, mientras que una escritura hará lo propio, poniendo las variables correspon-

dientes a verdadero. Si además, la escritura es la primera que se realiza sobre un bloque dentro de una transacción, el protocolo hace un Writeback del bloque al siguiente nivel de caché. De esta forma, guarda el valor antiguo en memoria, y mantiene el valor especulativo en la L1. En caso de que el algoritmo de remplazo determine que un bloque perteneciente al conjunto de lectura/escritura debe salir de la L1 para poder satisfacer otro acceso a memoria, la transacción abortará. La caché responderá a CPU con un HtmCommandFailed, en vez de con datos. Esto indicará a la CPU que se ha producido el aborto, y que debe restaurar el checkpoint del procesador.

La integración de la detección de conflictos en un protocolo Ruby es relativamente sencillo. Cuando se detecte una invalidación por un cambio de propiedad a otra caché, el protocolo abortará la transacción y se lo hará saber a la CPU en el próximo acceso a la caché, ya sea un acceso a memoria o un comando transaccional.

2.3.2 Memoria transaccional gem5 UMU

La versión pública tiene una limitación importante. Desde su publicación, únicamente dispone de una configuración para ejecutar HTM. En el caso de querer evaluar esa configuración en concreto, entonces la versión pública sería suficiente. Sin embargo, existen muchas configuraciones posibles y que interesaría poder evaluar sobre distintas aplicaciones. La versión UMU pretende soportar la mayor variedad posible de características descritas en la literatura. En su versión actual, ofrece distintas políticas de resolución de conflictos, control de versiones, detección de conflictos... Esta variedad de opciones me permitirán poder evaluar de forma más precisa las soluciones que propongo en este trabajo, y por lo tanto, obtener conclusiones más certeras.

2.4 Instruction Set Arquitecture: RISCV

La mayoría de los ISAs populares son propietarias. Sus empresas obtienen beneficio de la venta de dispositivos basados en su ISA, ya sea por la propia empresa o por las licencias otorgadas, por lo que no les interesa que los diseños sean de libre distribución. Es cierto que esto no evita la investigación alrededor de estos ISAs, pero impide las ideas resultantes puedan ser comercializadas. Además, estos ISAs suelen ser bastante complejos de implementar. Se puede pensar en crear un subconjunto más sencillo del ISA podría simplificar su uso, pero entonces el software compilado para el ISA al completo podría no funcionar. En esta categoría destacan tanto x86 como ARM, siendo los ISAs que dominan el mercado de computadores y dispositivos móviles, respectivamente. Aunque ya descontinuadas, Alpha y MIPS también pertenecen a esta categoría.

También hay disponibles ISAs de libre distribución, tal y como lo son actualmente SPARC y POWER, pero también tienen sus inconvenientes. SPARC, por ejemplo, fue

diseñado pensando en procesadores en orden de 5 etapas, lo cual no corresponde con los procesadores actuales. Este supuesto provoca que los procesadores agresivos estén limitados por una serie de requisitos que impone el ISA. Un ejemplo de estas limitaciones es que SPARC utiliza retardos de salto, es decir, mientras no se resuelva un salto no se podrá ejecutar ninguna instrucción adicional. POWER, por otro lado, no tiene este tipo de restricciones de diseño. Sin embargo, como el resto de ISAs, los dispositivos que lo implementen deben incluir todas las características del ISA, es decir, para garantizar que un programa compilado sobre POWER, o cualquier otro ISA, funcione correctamente, se debe cumplir la especificación al completo.

Como solución a estos problemas, Andrew Shell Waterman (Universidad de Stanford) propone RISCV[17], un ISA de libre distribución basado en MIPS. El objetivo principal con el que se ideó RISCV fue para facilitar la docencia e investigación sobre la arquitectura de los procesadores. Actualmente, RISCV también ha evolucionado hacia sector comercial, con el nuevo objetivo de que pueda utilizarse en casi cualquier dispositivo. Esto implica que los requisitos del ISA deben evitar afectar excesivamente al diseño del dispositivo, tal y como ocurre con SPARC. Así se evita que una restricción de diseño que mejore un poco a unos dispositivos no incremente el coste de implementación de otros.

Un factor importante del diseño de RISCV es su modularidad, que se consigue mediante las extensiones (Figura 2.6). Cada una incluye un conjunto de instrucciones que aportan una determinada funcionalidad, y es totalmente independiente del resto de extensiones. La extensión RV32/64I es la extensión troncal alrededor de la cuál se deberían añadir el resto de extensiones. Aporta todas las instrucciones que se consideran necesarias para ejecutar cualquier aplicación, incluido un Sistema Operativo. El estándar indica que el resto de extensiones se podría emular con RV32I, a excepción de las instrucciones atómicas, que requieren de hardware adicional. Esto implica que la adición de nuevas extensiones implicaría una mejora de rendimiento, pero la falta de extensiones no limitaría la funcionalidad.

Uno de los aspectos positivos de ser de libre distribución, es que la implicación de la comunidad es mayor. Cualquier persona puede colaborar activamente en el proyecto, ya sea en el propio ISA o herramientas como compiladores y depuradores. Esto ha provocado que multitud de programas y librerías soporten oficialmente RISCV, como los son los compiladores GCC o LLVM, el kernel de Linux y distribuciones basadas en éste. También ha tenido un gran impacto en el mundo de la educación. Un ensamblador fácil de entender suaviza la curva de aprendizaje a alumnos que se estén iniciando en el comportamiento de las computadoras. Existen diversos simuladores de RISCV, algunos de los cuales están enfocados a la docencia, como RIPES[18].

Extensión	Descripción
RV32I	Extensión troncal 32 bits
RV32E	Extensión troncal 32 bits con solo 16 registros
RV64I	Extensión troncal 64 bits
RV128I	Extensión troncal 128 bits
M	Multiplicación de enteros y División
A	Instrucciones Atómicas
F	Punto flotante de precisión simple
D	Punto flotante de precisión doble
G	Agrupa las extensiones anteriores
Q	Punto flotante de precisión cuádruple
L	Punto flotante decimal
C	Instrucciones comprimidas
B	Manipulación de bits
J	Lenguajes traducidos dinámicamente
T	Memoria Transaccional
P	Empaquetado-SIMD Instrucciones
V	Operaciones de Vector
N	Interrupciones de nivel de usuario

Figura 2.6: Extensiones RISCV

2.5 Trabajo Fin de Grado: Instrucciones básicas MT

La especificación actual de RISCV no soporta Memoria Transaccional. Por ello, en mi Trabajo Fin de Grado[9] implementé tres instrucciones que permiten utilizar Memoria Transaccional Hardware en gem5. Las instrucciones TRXBEGIN y TRXCOMMIT permiten iniciar y confirmar las transacciones respectivamente. Estas dos instrucciones son estrictamente necesarias, ya que permiten definir la sección crítica sobre la que aplicar MT. Además, se incluye la instrucción TRXCANCEL que permite al software abortar una transacción. Aporta mayor flexibilidad al programador, pudiendo aplicar adecuadamente técnicas de programación que no necesiten de un aislamiento continuo. En la sección siguiente se explicará más en detalle. Estas instrucciones se evaluaron sobre el sistema de Memoria Transaccional incluido en la versión 20.0 de gem5, el cual es similar a TSX de Intel. El proceso de evaluación mostró una buena escalabilidad de las aplicaciones que no tuvieran transacciones excesivamente grandes. En este trabajo se pretende extender el soporte de Memoria Transaccional a nivel de ISA de tal forma que este tipo de aplicaciones puedan también escalar adecuadamente.

3 Diseño y resolución del trabajo realizado

En esta sección se exponen dos limitaciones de las implementaciones actualmente disponibles del soporte para TM, y una serie de alternativas, basadas en la modificación del ISA, para poder aliviar estas limitaciones. Se mostrará como caso de estudio que hace visibles las limitaciones anteriores el benchmark labyrinth de la suite STAMP (Stanford Transactional Applications for Multi-Processing)[10]. También se propone una mejora a nivel software de labyrinth para comparar rendimiento y complejidad respecto a las nuevas instrucciones.

3.1 Limitaciones HTM

Los diseñadores de procesadores han tratado hasta la fecha de incluir soporte HTM introduciendo la mínima complejidad adicional, para lo cual se reutilizan determinadas características de los procesadores.

3.1.1 Tamaño del conjunto de lectura y escritura

La gran mayoría de procesadores actuales disponen de cachés hardware para aprovechar la localidad espacial y temporal de los accesos a memoria. Cuando se ejecuta un acceso a memoria, primero se comprueba si los datos están en las cachés antes de acceder a la memoria principal. Por lo tanto, las cachés privadas de cada núcleo son un lugar idóneo para mantener tanto el conjunto de lectura como de escritura, ya que todos los bloques accedidos por la transacción pasarán por la caché privada. En caso de que el sistema de MT utilice esta técnica, los conjuntos de lectura y escritura estarán limitados por el tamaño de las cachés que soporten memoria transaccional. En el caso del conjunto de lectura, resulta relativamente sencillo modificar la caché, bastando con incluir nuevos bits que indiquen que un bloque ha sido leído durante una transacción. Sin embargo, para el conjunto de escritura, además de otro bit adicional, será necesario modificar las cachés para poder mantener el valor no especulativo de un bloque que se modifique durante una transacción. Todos estos cambios se tendrán que hacer sobre cada nivel de caché que quiera que soporte MT.

Si se supone que el coste de extender el conjunto de escritura a otros niveles es excesivo, se podría optar por aumentar el tamaño de las cachés que lo soporten. Sin

embargo, hay que tener en cuenta que las CPU también tienen que lidiar con problemas de otra índole, y aumentar el tamaño de las cachés de primer nivel provocaría un aumento de latencia de los accesos a memoria. Se da la circunstancia de que las transacciones pueden contener accesos a memoria que realmente no es necesarios vigilar, como lo pueden ser las variables locales, de acceso privado a cada hilo. El único que puede conocer cuáles son el programador y, en segunda instancia, el compilador. ¿Y si pudiera comunicar al hardware qué direcciones no son necesarias que vigile? Sería posible tener más recursos para las que sí son estrictamente necesarias.

3.1.2 Modificación de los conjuntos de lectura

Otro de los problemas es un aislamiento demasiado estricto. Por defecto, es habitual que se aplique el aislamiento a todas las direcciones durante toda la transacción. Esto garantiza que se detecte un posible conflicto, pero también que pueda producirse sin que sea estrictamente necesario. Por ejemplo, en aplicaciones que utilicen la técnica de privatización, por la cual una estructura compartida se copia en la memoria privada de un hilo, no es necesario mantener el aislamiento de la estructura original una vez se haya hecha la copia [19][20]. Se trabajará con la copia privada y posteriormente se tratará de actualizar la estructura compartida con los cambios locales. Esta técnica permite que se puedan hacer cálculos y modificaciones sobre la copia privada sin afectar a la copia visible al resto de hilos, de tal forma que varios hilos pueden trabajar simultáneamente tratando sobre sus copias privadas. Después tendrán que comprobar sobre la estructura compartida si sus cambios son incompatibles con los de otros hilos.

3.1.3 Soluciones propuestas en este TFM

En respuesta a estas dos limitaciones, en este trabajo, se propone modificar el ISA RISCV, previamente extendido en mi TFG[9], con cuatro nuevas instrucciones que permitan al programador tener un mayor control sobre el sistema de Memoria Transaccional. La instrucciones TRXSRELEASE y TRXARELEASE permitirá eliminar recursos del conjunto de lectura durante la transacción, mientras que las instrucciones TRXWSRESUME/TRXWSSUSPEND delimitarán una región de la transacción en la cual las direcciones escritas no se añadan al conjunto de escritura.

3.2 Caso de estudio: labyrinth

A la hora de evaluar una determinada propuesta, resulta interesante enfrentarla a una serie de entornos que muestren su comportamiento ante diferentes circunstancias. En concreto, para Memoria Transaccional, se suele optar por STAMP. Contiene una

serie de benchmarks que usan sincronización mediante transacciones, y que tienen diferentes características en términos de los tamaños de los conjuntos de lectura/escritura, contención, número de instrucciones en las transacciones, etc. Para este trabajo, interesa el benchmark labyrinth, el cual tiene una alta probabilidad de conflicto, y conjuntos de lectura/escritura considerables.

labyrinth implementa un algoritmo de búsqueda de caminos mínimos entre dos puntos, teniendo en cuenta los posibles obstáculos que puedan haber, similar a los algoritmos utilizados para enrutar conexiones en PCBs. Así como la mayoría de benchmarks de STAMP, se divide en tres secciones. Una fase de inicialización de las variables, donde se leerá el fichero que define el laberinto y se inicializará la matriz global (*grid*), una segunda fase donde se calculan los caminos en base al laberinto cargado, y una tercera fase donde se validan las soluciones. Un camino será válido si tiene un principio y un final, y si no se cruza con otro camino. Por lo tanto. una solución será válida si todos los caminos que la conforman son válidos. Esta validación es fundamental para garantizar que no hayan ocurrido condiciones de carrera al paralelizar el algoritmo.

La segunda fase es la que se ha estudiado en este trabajo (Figura 3.1), ya que es la región de interés (ROI) del benchmark por ser la fase paralela. Por cada par de puntos, uno de inicio y otro de fin del camino, se realiza una expansión. Para cada punto que se ha visitado, se comprueba el estado de sus vecinos, es decir, las posiciones adyacentes. Se empieza desde el inicio del camino, y se detiene al llegar al punto de destino, o hasta que se determina que es imposible establecer un camino entre los dos puntos. Esto puede ocurrir si hay obstáculos de por medio que aíslen a un de los puntos del otro. Una vez determinado que existe al menos un camino, se procede a calcular un camino mínimo (traceback). A partir del punto de destino, se comprueba cuales de las celdas visitadas anteriormente garantizan el camino más corto hasta el punto de inicio.

Según lo descrito anteriormente, el benchmark se comportaría de forma secuencial. Si el número de caminos a calcular es alto, la ejecución del algoritmo puede llegar a alargarse de forma excesiva. Por ello, resulta conveniente aplicar alguna forma de paralelización. En concreto, se opta por utilizar un esquema MIMD basado en hilos, ya que STAMP está enfocado a CPUs de propósito general. En una situación ideal, cada uno de los caminos se podría calcular totalmente en paralelo, por ejemplo, si se pudiese garantizar que no se van a cruzar. Sin embargo, esto no está garantizado, por lo que hay que manejar el peor caso, es decir, suponer que puede ocurrir una condición de carrera. Es posible que los caminos que estén calculando cada uno de los hilos utilicen una misma celda, lo que llevaría a caminos cruzados, lo cual es incorrecto. Este problema puede surgir en cualquier fase del enrutamiento, por lo que una solución sencilla y funcional sería definir una sección crítica bloqueante que empieza abarcaría desde la línea 1 a la 3 (Figura 3.1). Hacerlo de esta forma limitaría bastante la concurrencia, ya que el procesamiento de todos los caminos se serializaría.

```
Grid global;
Para todas las parejas de puntos {
  1. Expansión desde origen hasta destino
  (lee y escribe en global)

  2. Traceback desde destino hasta origen
  (lee y escribe en global)

  3. Eliminar valores no utilizados de la Expansión
}
```

Figura 3.1: Enrutamiento secuencial

```
Grid global;
Para todas las parejas de puntos {
  adquirir(SGL)
    1. Expansión desde origen hasta destino
    (lee y escribe en global)

    2. Traceback desde destino hasta origen
    (lee y escribe en global)

    3. Eliminar valores no utilizados de la Expansión
  liberar(SGL)
}
```

Figura 3.2: Enrutamiento basado en Single Global Lock

```
Grid global;
Para todas las parejas de puntos {
  Grid local;
  1. Privatización
  (lee de global y escribe en local.
  Protege contra escrituras cada posición
  utilizada de global)

  2. Expansión desde origen hasta destino
  (lee de local y escribe en local.)

  3. Traceback desde destino hasta origen
  (lee y escribe en local)

  4. Añadir solución
  (lee de local y global, y escribe en global.
  Protege contra lecturas y escrituras
  cada posición utilizada de global)
}
```

Figura 3.3: Enrutamiento basado en locks de grano fino

```
Grid global;
Para todas las parejas de puntos {
  Grid local;
  beginTransaction {
    1. Privatización
    (lee de global y escribe en local.
    Protege contra escrituras cada posición
    utilizada de global)

    2. Expansión desde origen hasta destino
    (lee de local y escribe en local)

    3. Traceback desde destino hasta origen
    (lee y escribe en local)

    4. Añadir solución
    (lee de local y global, y escribe en global)
  }
}
```

Figura 3.4: Enrutamiento basado en transacciones

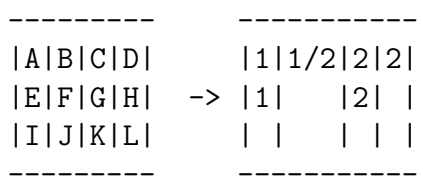

Figura 3.5: Ejemplo de laberinto. A la izquierda se muestran las posiciones. A la derecha los hilos que han adquirido determinados cerrojos en la matriz global. Múltiples números indican que hay contención por el cerrojo.

Otra alternativa sería utilizar locks de grano fino. A cada posición del laberinto se le asocia un lock, el cual se adquirirá en caso de que un hilo quiera escribir en esa posición. Si bien esta opción aporta mucha más escalabilidad que la anterior al únicamente bloquear las posiciones necesarias, conlleva una mayor complejidad de diseño. Será necesario establecer una política de liberación y adquisición de los locks que eviten los bloqueos infinitos (deadlock) y las condiciones carrera, además de tener que resolver determinadas cuestiones de diseño propias de cada aplicación.

Por ejemplo (Figura 3.5), si un hilo calcula el camino entre A y B, al expandir A accederá a B y E. En caso de que otro hilo quiera calcular el camino de C a D, al realizar la expansión sobre la misma matriz tendría que poder acceder a también a B, D y G. Para que estos dos caminos puedan calcularse correctamente, se debe adquirir los locks correspondientes a cada posición. En caso de que alguno de ellos esté adquirido, el otro hilo debe cancelar su expansión, liberando los cerrojos que ha ido adquiriendo hasta el momento. Además, el programador tendrá que determinar cuándo volver a intentar calcular el camino de C a D. ¿Lo intenta inmediatamente o espera un intervalo determinado de tiempo? Utilizar una granularidad reducida y tener que manejarla en software puede provocar la aparición de diversas cuestiones de diseño, además del manejo de los locks.

Con este ejemplo también se puede observar una limitación importante. Para realizar la expansión de un camino, cada hilo necesita guardar cierta información intermedia que le permita reconocer en la fase de traceback que posiciones se han explorado para ese camino. En caso de utilizar un único grid para todos los hilos, estas escrituras tendrán que hacerse en memoria compartida. Esto provoca que, aunque determinadas posiciones visitadas durante la expansión no pertenezcan a la solución, estas se tengan que proteger adquiriendo su lock.

Por ello, en labyrinth se opta por aplicar privatización. Antes de calcular los caminos, se hace una copia del laberinto en memoria privada de cada hilo, pero garantizando que nadie vaya escribir en el laberinto original mientras tanto y así evitar que se tra-

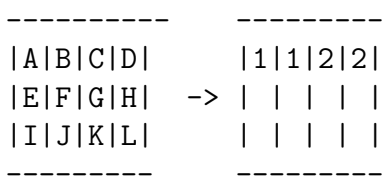

Figura 3.6: Ejemplo de laberinto con privatización. A la izquierda se muestran las posiciones. A la derecha los hilos que han adquirido determinados cerrojos en la matriz global.

baje con caminos incompletos. De esta forma, es posible paralelizar la expansión de los caminos, ya que la información intermedia está contenida en memoria privada. Una vez calculados, se debe actualizar el laberinto global con los nuevos caminos. En este punto, también es necesario garantizar que nadie esté accediendo a la copia global, tanto si está escribiendo, ya que podrían generarse caminos no válidos, como si si está leyendo, de tal forma que una privatización que se esté haciendo en ese momento no tenga caminos incompletos. Con el esquema de una sección crítica, esta técnica solo resulta útil en caso de utilizar sincronización de grano fino (Figura 3.3). Por ejemplo, en caso de utilizar un Single Global Lock(SGL, un único lock global), solo habrá un hilo ejecutando la sección crítica, por lo que ya se está garantizado que trabaje con un laberinto correcto, y que no se va a modificar por otro hilo mientras tanto. Con locks de grano fino, únicamente habría que adquirir los locks de las posiciones que forman parte de la solución, lo que aportaría una mayor escalabilidad. Partiendo del ejemplo anterior, y suponiendo que realizan los dos primeros hilos hacer la privatización al mismo tiempo, solo se tendrán que proteger las posiciones de las soluciones (Figura 3.6), pudiendo ambos hilos calcular los caminos completamente en paralelo.

Con Memoria Transaccional, se podrá mantener la sencillez del esquema del SGL (Figura 3.4), ofreciendo mayores oportunidades de concurrencia al enrutar los caminos, similares a utilizar cerrojos de grano fino. Sin embargo, con la implementación actual, la mayoría de las transacciones deberán acudir al fallback para evitar un bloqueo. Para privatizar la copia global hay que leerla al completo, por lo que todas las posiciones estarán en el conjunto de lectura durante toda la transacción. Esto implica que se generará un conflicto al añadir cualquier nuevo camino, independientemente de si se cruza o no con el camino que se esté calculando. El único caso donde esto no ocurre es si todavía no se ha completado la copia, y el nuevo camino recorre una sección perteneciente a la parte sin copiar. Esto implica que al final, muy pocas transacciones podrán añadir su camino en paralelo, serializando el enrutamiento, tal y como ocurre con SGL. Además, en este caso, SGL podría incluso ser más eficiente, ya que evita la ejecución de transacciones que nunca van a confirmar.

Entonces, ¿resulta imposible que labyrinth pueda aprovecharse de la Memoria Tran-

saccional para mejorar su rendimiento? No lo es, y es en las siguientes secciones donde se intentarán resolver algunas de las limitaciones del benchmark, entre las cuales se incluye el número reducido de transacciones que se pueden confirmar al mismo tiempo.

3.3 Extensión del ISA (1): Early Release

En determinadas transacciones se pueden llegar a añadir determinados recursos al conjuntos de lectura, los cuales a partir de cierto punto no es necesario seguir aislándolos. La única entidad que conoce qué direcciones de memoria cumplen esta propiedad es el programador. Por lo tanto, puede resultar interesante que el ISA ofrezca una alternativa que permita eliminar determinadas direcciones de memoria del conjunto de lectura del conjunto de lectura. A esta técnica se le suele denominar Early Release (Liberación Temprana / ER) y en este trabajo se ha implementado a través de una nueva instrucción en el ISA de RISCV.

Respecto a labyrinth, ER es aplicable en la privatización (Figura 3.7). Durante la copia del laberinto global al local, se están añadiendo los bloques de la matriz global al conjunto de lectura. Esto permite detectar si otro está añadiendo un camino nuevo mientras se hace la privatización. En este caso, esta privatización no será válida, ya que es posible que contenga caminos incompletos, o incluso que contenga caminos que al final no se hayan incluido en el grid global. Una vez se ha terminado de hacer la copia privada del laberinto, es posible eliminar el aislamiento de los bloques correspondientes. ¿Por qué querríamos hacer esto, si la intención de la Memoria Transaccional es evitar las condiciones de carrera? Una vez hecha la privatización, solo es necesario detectar conflictos sobre el nuevo camino. Esto permitirá eliminar los conflictos provocados sobre el grid global por caminos que en ningún momento se vayan a cruzar con el camino que se esté calculando. Sin embargo, antes de añadir el nuevo camino al grid global, habrá que volver a leerlo para comprobar que mientras se calculaba el camino no se haya añadido otro que interfiera con él. Esto puede ocurrir por haber eliminado el aislamiento del grid global, ya que ahora las escrituras no generarán conflicto. En caso de detectar que una de las posiciones ya pertenece a otro camino, será necesario abortar la transacción manualmente para deshacer las escrituras previas sobre el grid global. Para esto es imprescindible la instrucción de aborto explícito. Además, las lecturas sobre el grid global permitirán detectar a través de un posible conflicto si se está añadiendo otro camino al mismo tiempo y si se están cruzando.

Una forma de implementar Early Release sería a nivel de la granularidad de la detección de conflictos, que para este trabajo, y también para las implementaciones comerciales existentes, será de un bloque de caché. Cada instrucción TRXSRELEASE eliminará del conjunto de lectura un único bloque, por lo que se podrá escoger de forma precisa sobre qué recursos eliminar el aislamiento. Sin embargo, en el caso de

```
Grid GLOBAL;
START_TRANSACTION;
Grid LOCAL;

GRID_COPY(LOCAL, GLOBAL);
Para cada DIRECCION en LOCAL hacer:
  EARLY_RELEASE(DIRECCION);

si existe (EXPANSION(ORIGEN,DESTINO,LOCAL)) hacer:
    SOLUCION = TRACEBACK(LOCAL, DESTINO);

   //ADD_PATH
    si existe SOLUCION hacer:
        Para cada POSICION en SOLUCION hacer:
          si GLOBAL[POSICION] == OCUPADA hacer:
            CANCEL_TRANSACTION;
COMMIT_TRANSACTION;
```

Figura 3.7: Pseudocodigo Early Release

que el número de recursos que se quiera liberar sea relativamente grande, el número de instrucciones de Early Release que deben ejecutarse también lo será. Por lo tanto, aunque durante la privatización no hayan ocurrido conflictos, el hecho de tener que liberar los recursos uno a uno aumenta la probabilidad de que los que más tarden en liberarse puedan generar un conflicto.

Partiendo de lo comentado en el párrafo anterior, se puede plantear un funcionamiento diferente del Early Release. ¿Y si en vez de una mayor flexibilidad, se prefiera un mayor rendimiento en un conjunto de aplicaciones concretas? Si aplicamos este enfoque a labyrinth, la instrucción TRXARELEASE podría liberar toda la copia global en una única instrucción. Eliminaría por completo la limitación de tener que liberar bloque por bloque, pero impediría utilizar la instrucción en otra circunstancia. En este caso, como únicamente es necesario liberar la copia global, resulta factible. Sin embargo, hay que tener especial cuidado con esta alternativa.

En las implementaciones de HTM best-effort, se requiere del uso combinado de un cerrojo (SGL) para la implementación del camino alternativo (fallback). Por lo tanto, se tiene que evitar eliminar el SGL del conjunto de lectura para seguir garantizando el aislamiento entre las ejecuciones transaccionales y no transaccionales. En caso de no pertenecer al conjunto de lectura, podría ocurrir que una transacción activa modifique un recurso que esté siendo utilizado en una ejecución no transaccional, provocando

una condición de carrera. Estando en el conjunto de lectura, en el momento en que se adquiera el SGL, cualquier transacción abortará, teniendo que esperar a que termine la ejecución no transaccional. Esto implica que tiene que existir una relación entre la nueva instrucción y el software, lo cual evitaría que el programador pudiera utilizar un fallback distinto.

3.4 Extensión del ISA (2): Desactivar escrituras transaccionales

La abstracción ofrecida al programador por el modelo de sincronización de TM es que todos los accesos realizados dentro de una transacción son implícitamente transaccionales, es decir, todos los accesos a memoria dentro de una transacción se añadirán a los conjuntos de lectura/escritura. De esta forma, se garantiza que la transacción mantenga sus propiedades fundamentales, el aislamiento y la atomicidad. Sin embargo, en muchos casos habrá accesos que en realidad no es necesario vigilar, como pueden ser los que referencien a memoria privada. Si la transacción accede a pocos datos, este problema se minimiza, ya que, o bien todos los accesos deben añadirse, o aunque haya accesos que no se tengan que añadir, el hardware puede mantener los conjuntos de lectura/escritura sin problemas. Conforme el tamaño de los conjuntos de lectura y escritura de la transacción vaya aumentando, es posible que los accesos que no necesiten de aislamiento vayan aumentando. Por ejemplo, es posible que se necesiten más variables auxiliares para guardar resultados intermedios. De esta forma, al aumentar el conjunto de lectura/escritura, los abortos por capacidad serán más probables. Por ello, podría resultar conveniente poder excluir a estos accesos de dichos conjuntos.

Solo el programador (o el compilador) conoce que accesos no necesitan de aislamiento, por lo que será necesario añadir nuevas instrucciones. Puede aplicarse tanto a las lecturas como a las escrituras, pero tal y como se ha explicado al inicio del capítulo, soportar un mayor conjunto de lectura en hardware resulta mucho más sencillo que un mayor conjunto de escritura. Por ello, se ha priorizado el poder reducir el conjunto de escritura, permitiendo una reducción de complejidad considerable en el hardware en caso de que estas instrucciones cumplan con su objetivo. En el caso que el porcentaje de estos accesos sea bastante alto, se podrá conseguir reducir considerablemente los abortos por capacidad, pudiendo eliminarlos por completo.

Una alternativa sería incluir instrucciones de escritura a memoria no transaccionales. En cada escritura, se debería sustituir la instrucción por su equivalente no transaccional. Esto implica que, o bien el programador tiene que indicar una a una que escritura es no transaccional, o que el compilador debe extenderse de tal forma que en una determinada sección de código debe utilizar escrituras no transaccionales. Hay que tener en cuenta que existen distintas instrucciones de escritura a memoria, por lo que habría

que incluir una nueva instrucción no transaccional por cada una.

Por ello, se propone seguir otro enfoque. A través de dos nuevas instrucciones: TRXWSRESUME/TRXWSSUSPEND, ya sea el programador o el compilador, se podrá indicar al sistema de Memoria Transaccional un intervalo donde cualquier escritura no se añadirá al conjunto de escritura. Además de ahorrar en instrucciones, esta aproximación tiene otra gran ventaja : permitirá que código previamente compilado pueda aprovecharse de la nueva funcionalidad sin tener que modificarlo.

Esta propuesta provoca la aparición de un problema. Se puede pensar que una asignación a una variable únicamente implica un almacenamiento. Aunque en algunos casos puede ser así, en muchos otros esa asignación implicará más cambios. Para comprender cuáles, es necesario entender qué es y cómo funciona la pila.

La pila tiene varios usos, como por ejemplo almacenar las direcciones de retorno de las funciones y pasarles parámetros, pero el más relevante para entender el problema es que la pila se utiliza para almacenar variables locales. Será el propio programa quién administre este espacio, o lo que es lo mismo, el compilador determinará cómo usarlo en función de lo que el programador indique qué quiere hacer a través del código fuente. Para intentar reducir el máximo de pila utilizada en un instante de tiempo, el compilador reutiliza partes de la pila durante una función, y entre llamadas a distintas funciones. En una ejecución no transaccional, no hay inconveniente. Sin embargo, si una transacción aborta y existen recursos que no se han añadido al conjunto de escritura, es posible que se lean direcciones de pila cuyos valores corresponden con los de otras variables.

En el entorno que he utilizado para este trabajo, para iniciar una transacción es necesario llamar a la función beginTransaction de la librería contra la que se enlazan los benchmarks (que contiene el manejador de aborto, implementa el camino de respaldo, etc...) Esta reserva su espacio de pila y retorna en caso de haber podido iniciar la transacción, desapilando la pila (Figura 3.9). Durante la transacción, se hacen llamadas a otras funciones, como puede ser la función Expansion, por lo que reutilizarán el espacio reservado previamente por beginTransaction 3.10. En caso de abortar la transacción, si se ha utilizado TRXWSRESUME/TRXWSSUSPEND para escribir en la pila su estado no será consistente, ya que alguna de estas funciones pueden haber modificado una zona de la pila que se solape con la del beginTransaction. Al leer su zona de pila, beginTransaction trabajará con datos que provienen de otro contexto, pudiendo generar resultados incorrectos o violaciones de segmento. Como solución a este problema, se ha implementado beginTransaction como un macro, de tal forma que no se reserve espacio en la pila al llamar a la función

Sin embargo, la medida descrita anteriormente no es suficiente. En una función re-

lativamente larga, existen variables que solo se usan durante un momento concreto. Si son bastantes, el porcentaje de uso efectivo de la pila en un instante concreto podría resultar bastante bajo. Es habitual que los compiladores reutilicen estas direcciones de pila que estén inactivas para almacenar otras variables que si lo estén. Es una optimización relativamente sencilla de hacer, y que permitirá reducir el tamaño máximo de pila utilizada por una función. Sin embargo, aunque se haya eliminado la llamada a beginTransaction, si se reutilizan direcciones de pila, es posible que una de las variables que se lea después de un aborto comparta dirección de pila con otra que se haya escrito durante la transacción. De esta forma, el valor leído corresponderá con el de una variable distinta, resultando en una ejecución inconsistente con el código fuente. Por suerte, los compiladores ofrecen la opción de desactivar esta optimización. En concreto, en GCC se desactiva con -fstack-reuse=none.

Incluso con estas medidas, el problema todavía persiste. Se ha observado, en lo que aparenta ser un bug, que GCC sigue reutilizando la pila para variables distintas dentro de una misma función. Partiendo de que mi conocimiento del compilador es limitado, he probado desactivando las opciones por defecto hasta comprobar que con la opción -fno-expensive-optimizations ejecuciones que antes fallaban ahora terminan correctamente. En estos momentos, desconozco el motivo exacto ya que está opción en concreto afecta a distintos elementos de la compilación, los cuales no están lo suficientemente documentados para que pueda obtener conclusiones. Con estos tres cambios, la mayoría de ejecuciones terminan, pero todavía hay algunas en las que el problema persiste. Como último recurso, modifico las funciones de expansión y de traceback para que cada una tenga su propio espacio en la pila, en vez de compartirlo con la función principal. De esta forma, he conseguido que todas las ejecuciones de la evaluación terminen correctamente.

Esto muestra que para poder utilizar TRXWSRESUME/TRXWSSUSPEND en cualquier caso de forma segura y sencilla, el compilador debe garantizar que no reutilice direcciones de pila reservadas fuera de la transacción, dentro de una transacción. Sí que se podrían utilizar sin problemas si únicamente se llamaran dentro de funciones que se llamen dentro de la transacción, siempre que al iniciar la transacción no se retorne. De esta forma, TRXWSRESUME/TRXWSSUSPEND solo afectará a la pila de la función, y que no será leída en caso de aborto.

Estos problemas que se han descrito son una circunstancia concreta de otro más general: en caso de desactivar las escrituras transaccionales, se debe garantizar que memoria reservada antes de iniciar una transacción no se libere durante la transacción en caso de que un aborto pueda provocar una lectura sobre esa memoria. Además de con la pila, también podría ocurrir con memoria dinámica. Por ejemplo, si se ha reservado memoria antes de la transacción, durante esta se libera es misma memoria y se vuelve a reservar más memoria, es posible que se reutilicen las mismas direcciones. Entonces, en

```
Grid GLOBAL;
START_TRANSACTION;
Grid LOCAL;

GRID_COPY(LOCAL, GLOBAL);
Para cada DIRECCION en LOCAL hacer:
  EARLY_RELEASE(DIRECCION);
SUSPEND();
si existe (EXPANSION(ORIGEN,DESTINO,LOCAL)) hacer:
    SOLUCION = TRACEBACK(LOCAL, DESTINO);

    RESUME();
   //ADD_PATH
    si existe SOLUCION hacer:
        Para cada POSICION en SOLUCION hacer:
          si GLOBAL[POSICION] == OCUPADA hacer:
            CANCEL_TRANSACTION;
COMMIT_TRANSACTION;
```

Figura 3.8: Pseudocodigo TRXWSRESUME/TRXWSSUSPEND

caso de aborto, se seguirán accediendo las mismas direcciones, pero contendrán datos de otro contexto.

3.5 Solución software sobre instrucciones básicas

Las instrucciones anteriormente propuestas podrán aplicarse a cualquier programa. Sin embargo, cada programa tiene sus particularidades. Aunque la Memoria Transaccional simplifique la sincronización de programas paralelos, no es conveniente abusar de transacciones excesivamente grandes. Para poder reducir la granularidad de las sincronización, es necesario conocer más en profundidad el algoritmo a paralelizar. Una granularidad más fina permitirá reducir, entre otras cosas, el tamaño de los conjuntos de lectura/escritura, pero implicará que hay que unir las transacciones en software para mantener el funcionamiento correcto, aumentando la complejidad de cara al programador. Para demostrar esta circunstancia, se utilizará el caso de estudio.

Tal y como se menciona en la sección 3.2, el algoritmo de enrutamiento de labyrinth se divide en tres secciones. En la primera, se privatiza el laberinto (gridCopy). Interesa incluir esa sección en una transacción para poder garantizar que la copia privada es consistente, es decir, no contiene caminos incompletos. La tercera sección (addPath), la que se encarga de incorporar los nuevos caminos al laberinto global, también interesa

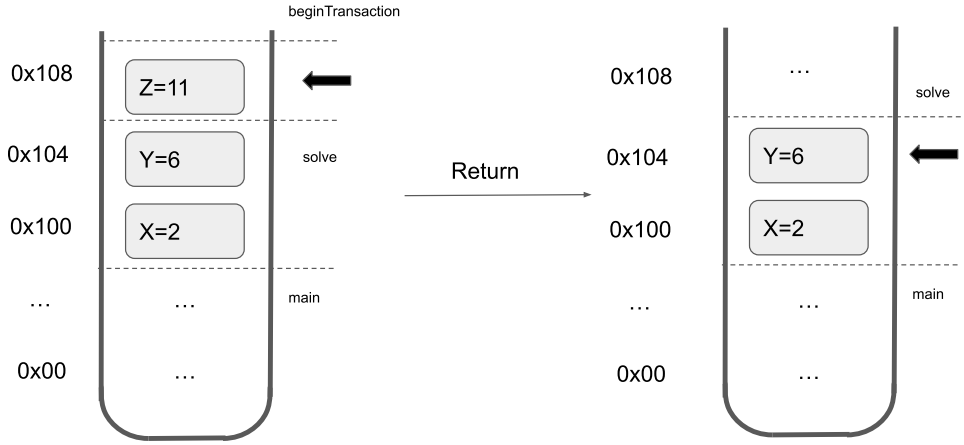

Figura 3.9: Ejemplo de reutilización de pila (1/2)

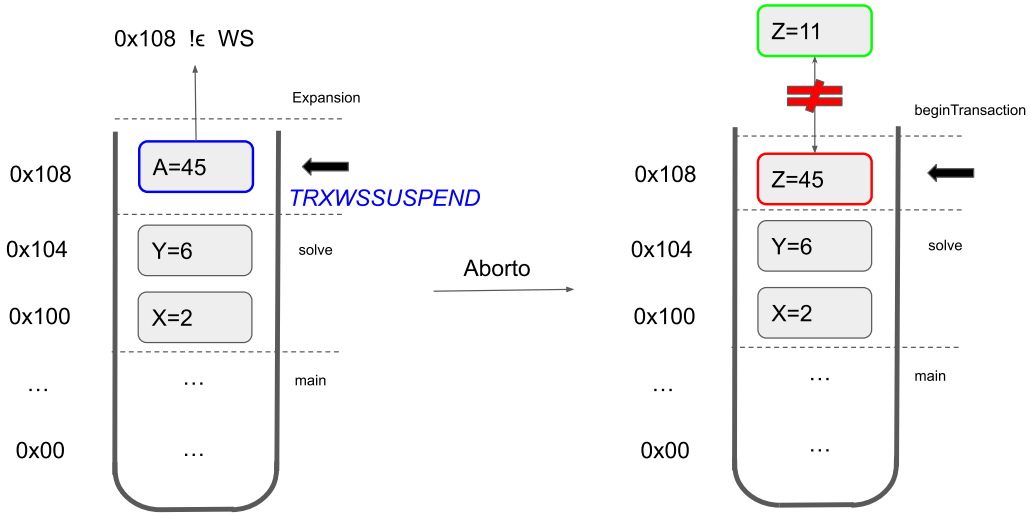

Figura 3.10: Ejemplo de reutilización de pila (2/2)

que se ejecute en modo transaccional. De esta forma, se evitará que en la copia global se escriban caminos no válidos. La segunda sección (Expansión y Traceback) se encarga de calcular los caminos, utilizando la copia privada generada en gridCopy. ¿Por qué incluir esta sección en una transacción, si en ningún momento va a acceder a datos compartidos, y por lo tanto es imposible que aparezca un conflicto? Al trabajar cada hilo con su copia, no ocurrirá ninguna condición de carrera, por lo que, durante esta sección, resulta innecesario ejecutar en modo transaccional. Por ello, se propone dividir la transacción del benchmark original en dos (Figura 3.11), una para gridCopy y otra para addPath.

Aunque la transacción original se divida en dos, en caso de encontrarse un camino no válido en addPath, se deberá volver a ejecutar gridCopy. Con una transacción, esto ocurría de forma automática, ya que el aborto provoca que se restaure el estado anterior, que corresponde a la ejecución de gridCopy. Con las dos nuevas transacciones, el aborto de la segunda transacción no implicaría una nueva copia, sino un nuevo intento de añadir el nuevo camino. La Memoria Transaccional, junto con el fallback, garantiza la atomicidad de una transacción, pero no garantizan nada entre transacciones distintas. Por ello, es necesario a nivel software que se enlacen las dos transacciones. De esta forma, cuando la segunda transacción aborte de forma explícita por un camino incorrecto, en vez de reintentarla, saltará a la primera transacción. Esta necesidad provocará un incremento de complejidad de la programación, teniendo que manejar de forma directa el estado de las transacciones.

Además de este tipo de abortos, hay que poner especial atención a los abortos por conflicto. Recordemos que, al utilizar los bloques de caché para mantener los conjuntos de lectura/escritura, pueden aparecer falsos conflictos. En labyrinth para representar cada punto se utilizan enteros de 8 bytes. Esto implica que un bloque de caché de 64 bytes pueda llegar a contener hasta 8 puntos distintos. Por ejemplo, recuperando el ejemplo 3.5, las posiciones de la A a la H estarán en el mismo bloque. Esto implica que una escritura sobre la posición A por parte de un hilo, provocará un conflicto si otro hilo escribe en cualquiera de las otras 7 posiciones.

Este problema puede enfocarse de dos maneras distintas en la segunda transacción. Si utilizamos un enfoque optimista, suponemos que el conflicto es en realidad un falso conflicto, y que si se reintentara transacción, el nuevo camino podrá añadirse a la solución. Si la suposición es correcta, se habrá evitado volver a calcular un camino que era válido, al contrario que con una única transacción. En caso de ser errónea, se reintentará la transacción hasta que se aborte explícitamente. Si por otro lado, se aplica un enfoque pesimista, en caso de conflicto se supondrá que el camino no es válido, y por lo tanto se descarta. Si es cierto, se evitará reintentar la segunda transacción, como pasaba con el enfoque optimista. En caso de que el conflicto fuese en realidad un falso conflicto, se volverá a calcular el camino de forma innecesaria.

```
   Grid GLOBAL;
 - START_TRANSACTION();
|  Grid LOCAL;
|
|  GRID_COPY(LOCAL, GLOBAL);
|  COMMIT_TRANSACTION();
|
|  si existe (EXPANSION(ORIGEN,DESTINO,LOCAL)) hacer:
|     SOLUCION = TRACEBACK(LOCAL, DESTINO);
|
 ---- START_TRANSACTION();
      //ADD_PATH
       si existe SOLUCION hacer:
           Para cada POSICION en SOLUCION hacer:
             si GLOBAL[POSICION] == OCUPADA hacer:
               CANCEL_TRANSACTION;
      COMMIT_TRANSACTION;
```

Figura 3.11: Pseudocodigo 2 transacciones

3.6 Instrucciones básicas

Actualmente, RISCV no dispone de soporte de Memoria Transaccional. Por ello, en mi TFG propuse tres instrucciones que permiten manejar las transacciones: TRXBEGIN, TRXCANCEL y TRXCOMMIT. Habrá que tener en cuenta una serie de requisitos que impone la especificación de RISCV. Uno de ellos es el formato de las instrucciones, que determinará qué y cómo se codifica cada instrucción. Para este trabajo, a la hora de elegirlo, se ha tenido en cuenta que los operandos que necesitan las instrucciones probablemente sean más grandes que la propia instrucción, por lo que habrá que usar registros para guardar los operandos.

También hay que tener en cuenta que, para diferenciar una instrucción de otra, se utiliza el campo opcode. Cada instrucción o grupo de instrucciones tendrá asignado un opcode único. Los opcode son limitados, por lo que es necesario utilizar un campo adicional que permita diferenciar múltiples instrucciones que utilizan un mismo opcode. Existen distintos formatos que encajan con estos dos requisitos, ya que las nuevas instrucciones apenas necesitan operandos. Me he decantado por el formato R (Figura 3.12), pero por ejemplo, el formato I también sería válido para todas las instrucciones.

El uso de transacciones dentro de otras se denomina anidamiento. *Open nesting*[21] permite una transacción interna puede confirmarse, haciendo visible sus cambios al

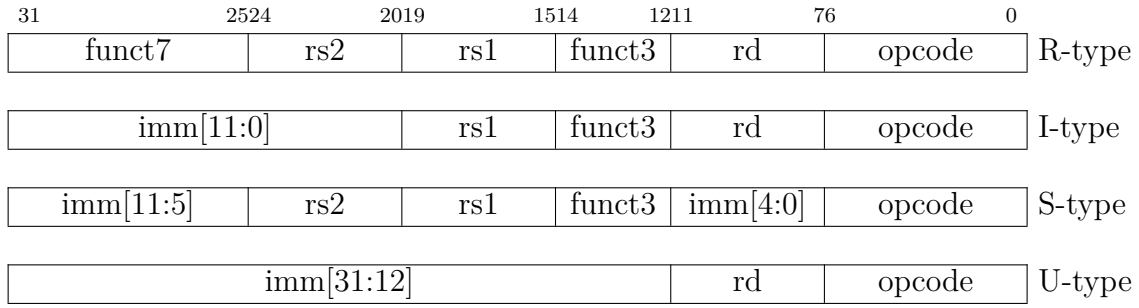

Figura 3.12: Formatos básicos RISCV.

resto de hilos. Esto ofrece mejor rendimiento, ya que permite eliminar el aislamiento de las direcciones de las transacciones internas. Sin embargo, requiere de una alta complejidad tanto a nivel hardware como software. *Closed nesting*[21] no requiere de tanta complejidad, pero impide que las transacciones internas hagan visibles sus modificaciones hasta que confirme la primera transacción. Aún así, los cambios necesarios en hardware para *Closed nesting* hacen que las implementaciones actuales lo simplifiquen aún más. *Flattened nesting* permite anidamiento unificando las transacciones en una sola. De esta forma, un aborto deshará los cambios de todas las transacciones, y los cambios solo serán visibles cuando se confirme la primera transacción. Entonces, para determinar cuándo eliminar el aislamiento, es necesario un *Current Status Register* (CSR) que indique cuantas transacciones están anidadas. Para este trabajo, se le llamará TRXDEPTH. Este registro forma parte del contexto del hilo, ya que de ellos depende que las transacciones se ejecuten correctamente. Por lo tanto, en caso de que las transacciones soporten cambio de contexto, este registro tendrá que almacenarse.

A la hora de escoger un opcode, no es recomendable escoger uno al azar. En caso de querer que las nuevas instrucciones sean compatibles con otras extensiones, habrá que escoger un opcode adecuado (Figura 3.1). Si el opcode está siendo utilizado actualmente, las nuevas instrucciones no serían compatibles con la extensión que lo utilice. Si está reservado, funcionará en los dispositivos que sigan la especificación actual, pero es posible que no lo haga en las próximas. Para asegurar la compatibilidad a lago plazo con todas las extensiones, se escogerá uno de los opcodes que están libres.

El formato de las instrucciones se muestra en la figura 3.14, y a continuación, las descripciones más detalladas de las instrucciones y del nuevo registro:

- Registro TRXDEPTH: indica la profundidad del anidamiento. El valor inicial es 0, incrementándose con TRXBEGIN y decrementándose con TRXCOMMIT. En caso de aborto, se volverá al valor inicial.

TRXFAILURE_REASON	0x00007fffu	Codifica el parámetro de CANCEL
TRXFAILURE_RTRY	0x00008000u	La transacción no puede ejecutarse
TRXFAILURE_CNCL	0x00010000u	Aborto explícito
TRXFAILURE_MEM	0x00020000u	Conflicto
TRXFAILURE_SIZE	0x00100000u	Aborto por capacidad
TRXFAILURE_NEST	0x00200000u	Anidamiento excesivo
TRXLOCK_IS_ACQUIRED	65535	Otro hilo ha adquirido el SGL
TRXBEGIN_STARTED	0	Transacción iniciada

Figura 3.13: Códigos de aborto

- TRXBEGIN: la instrucción TRXBEGIN inicia una transacción, escribiendo el valor 0 en *dest* si la transacción se ha iniciado correctamente. En caso de abortar la transacción, *dest* contendrá el motivo del aborto (Figura 3.13). En caso de que ya exista una transacción activa, se incrementará el valor de TRXDEPTH en 1.
- TRXCOMMIT: la instrucción TRXCOMMIT confirma una transacción activa, decrementando en 1 el valor de TRXDEPTH. Si el resultado es 0, los datos modificados por la transacción o transacciones se harán visibles al resto de hilos. En caso de que el valor de TRXDEPTH sea 0, es decir, no se ha iniciado una transacción, se generará una excepción. No requiere de operandos.
- TRXCANCEL: la instrucción TRXCANCEL permite abortar una transacción desde el software. En caso de existir anidamiento, todas las transacciones abortarán. Esto implica que un motivo concreto para abortar una transacción afectará a todas las transacciones que la contengan. En src se puede especificar un motivo personalizado de aborto, y que será escrito en el registro indicado en *dest* de la instrucción TRXBEGIN.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Tabla 3.1: Mapa de opcodes de RISCV

3.7 Formalización de las nuevas instrucciones

Una vez se han planteado las limitaciones que exhiben las implementaciones actuales de HTM y formas de evitar su impacto en el rendimiento de ciertos programas paralelo,

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
5	5	5	3	5	7	
0	0	0	000	dest	0001011	TRXBEGIN
0	0	0	001	0	0001011	TRXCOMMIT
0	0	src	010	0	0001011	TRXCANCEL

Figura 3.14: Formato de las instrucciones básicas

el siguiente paso es formalizar las instrucciones. Se ha escogido el mismo formato de instrucción que las instrucciones básicas por los mismos motivos.

Buscando facilitar que el software sea componible, será necesario seguir un enfoque similar al anidamiento de las transacciones para TRXWSRESUME/TRXWSSUSPEND. Se dedicará un CSR llamado TRXRESUS para contabilizar cuántos TRXWSSUSPEND y cuántos de los TRXWSRESUME correspondientes se han ejecutado. Si no existiera este soporte, llamar a TRXWSSUSPEND/TRXWSRESUME dentro de otra pareja provocaría que entre el TRXWSRESUME más interno y el externo las escrituras se añadirían al conjunto de escritura, provocando un funcionamiento incorrecto de las instrucciones. Este nuevo registro también deberá almacenarse, al igual que TRXDEPTH, en caso de que se soporten cambios de contexto.

En la figura 3.15 se muestra el formato de las nuevas instrucciones, y a continuación, descripciones más detalladas de estas y del registro TRXRESUS, además de un cambio necesario en TRXBEGIN:

- Registro TRXRESUS: si el valor del registro es 0, las escrituras transaccionales deben incluirse en el conjunto de escritura. En caso de ser mayor de 0, las escrituras no se añadirán. Se modifica a través de las instrucciones TRXBEGIN, TRXWSRESUME y TRXWSSUSPEND. El valor inicial es 0. En caso de aborto, el registro recupera este valor.

- TRXBEGIN: la instrucción TRXBEGIN inicia una transacción, escribiendo el valor 0 en *dest* si la transacción se ha iniciado correctamente. En caso de abortar la transacción, *dest* contendrá el motivo del aborto (Figura 3.13). En caso de que ya exista una transacción activa, se incrementará el valor de TRXDEPTH en 1. Inicializa el valor del registro TRXRESUS a 0, de tal forma que todas las escrituras se añadan al conjunto de escritura.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
0	0	0	011	0	0001011	TRXWSRESUME
0	0	0	100	0	0001011	TRXWSSUSPEND
0	0	src	101	0	0001011	TRXSRELEASE
0	0	src	110	0	0001011	TRXARELEASE

Figura 3.15: Formato de las nuevas instrucciones.

- TRXWSSUSPEND: la instrucción TRXWSSUSPEND permite que las escrituras siguientes dentro de una transacción no se añadan al conjunto de escritura. Incrementa el valor del registro TRXRESUS en 1. No requiere de operandos.
- TRXWSRESUME: la instrucción TRXWSRESUME disminuye el valor del registro TRXRESUS en 1. Si el valor resultante es 0, las escrituras siguientes dentro de una transacción se añadirán al conjunto de escritura. No requiere de operandos.
- TRXSRELEASE: la instrucción TRXSRELEASE permite eliminar del conjunto de lectura de una transacción la dirección de memoria contenida en el registro *src*.
- TRXARELEASE: la instrucción TRXARELEASE permite eliminar todas las direcciones del conjunto de lectura, excepto la dirección física correspondiente a la dirección virtual indicada en *src*, que debe contener el SGL.

3.8 Implementación de las instrucciones en gem5

Una vez incluidas las instrucciones en el Decodificador correspondiente a RISCV, es necesario implementar la funcionalidad de cada instrucción. Para ello, habrá que conocer la estructura que maneja las transacciones (Figura 3.16). Estas instrucciones alteran el estado de una transacción, por lo que deberán incluirse como comandos transaccionales en el TransactionalSequencer, intermediario entre CPU y memoria cuando se está ejecutando una transacción. También se deberán incluir en el TransactionalInterfaceManager (TIM), que actúa como interfaz para el resto de componentes de Memoria Transaccional. En concreto estas instrucciones modifican los conjuntos de lectura y escritura, por lo que habrá que modificar también el TransactionalIsolationManager (TISM).

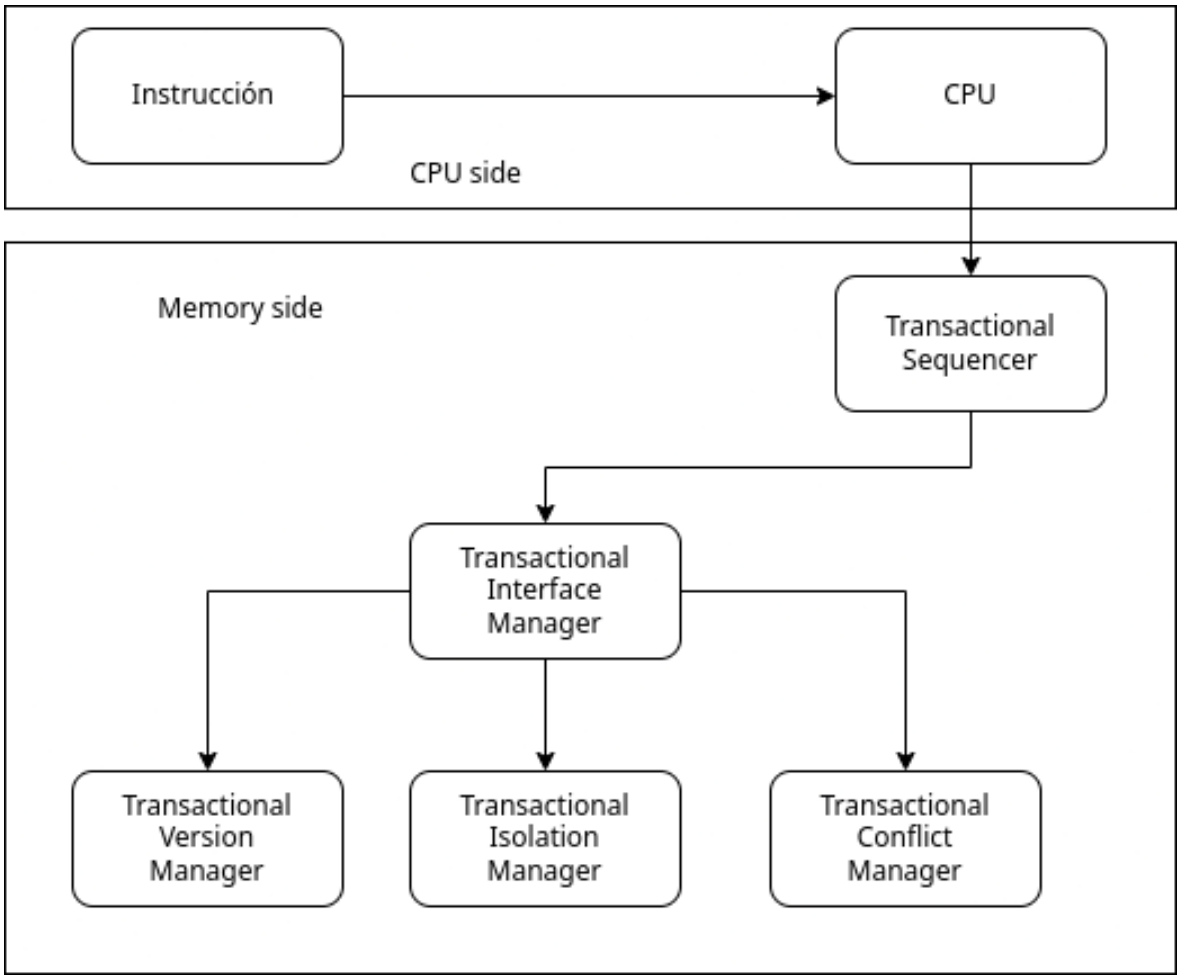

Figura 3.16: Estructura de la Memoria Transaccional en gem5-UMU

La instrucción TRXSRELEASE elimina a un bloque del conjunto de lectura. Ninguna otra instrucción hace algo similar, por lo que se añadirá una nueva función earlyRelease(Addr block) al TIM. Esta función debe acceder al TISM para modificar el conjunto de lectura, por lo que será necesario una función análoga a la anterior pero en el TISM. Una vez se puede acceder a los conjuntos, se comprobará si el bloque a eliminar está en el conjunto de lectura. En caso afirmativo, se elimina. Por otro lado, la instrucción TRXARELEASE funcionará de una forma similar, pero eliminando todos los bloques del conjunto de lectura, excepto el indicado como parámetro, que en este caso debe corresponder son el SGL.

Las instrucciones TRXWSRESUME/TRXWSSUSPEND únicamente cambian a la transacción a un modo distinto donde las escrituras no se añaden al conjunto de escritura. Para implementar este modo, se define un entero en el TIM que indique cuántos TRXWSSUSPEND hay activos. Será la instrucción TRXWSSUSPEND quien incremente esta variable, por lo que se añade una función suspendTransaction(bool suspend) al TIM, a la que se le pasa el valor verdadero y incrementará en uno el entero. La instrucción TRXWSRESUME podrá reutilizar esta función, indicando como parámetro un valor falso y decrementando el entero. Una vez definido el nuevo modo, es necesario tenerlo en cuenta para que las escrituras posteriores no se añadan al conjunto de escritura. Para ello, se incluye en el TISM una comprobación que solo añade el bloque al conjunto de escritura en caso de que el entero esté a 0.

Alterar el conjunto de escritura puede provocar resultados incorrectos si no se hace de forma adecuada. Por ello, para facilitar el proceso de depuración de TRXWSRESUME/TRXWSSUSPEND, se ha incluido un proceso de verificación en el que se almacenan todos los bloques que se deja de añadir al conjunto de escritura por estar la transacción suspendida. En caso de aborto, cada lectura y escritura posterior se comprueba si accede a uno de esos bloques. Si el primer acceso es una escritura, el bloque se elimina de ese conjunto de verificación, ya que el valor modificado la transacción abortada no se va a utilizar. Si por el contrario, se produce una lectura, se genera un mensaje de aviso que indique que se está leyendo un valor modificado en una transacción abortada, cuyo contenido no ha sido restaurado por no estar en el conjunto de escritura. Este método me ha facilitado enormemente las tareas de validación de estas instrucciones.

4 Entorno de evaluación y resultados

Una vez establecido cómo se pretende resolver en cierta medida las limitaciones de las implementaciones HTM *best-effort* descritas en el capítulo anterior, y qué resultados se espera de ello, resulta imprescindible comprobar su desempeño en un entorno realista. Cierto es que sería interesante poder implementar las instrucciones en un chip RISCV ya existente, y comprobar de primera mano su rendimiento. Sin embargo,eso resultaría demasiado complejo y costoso. En lugar de ello, utilizaremos un simulador, que es más adecuado para recabar información de la ejecución. En concreto, gem5[16] incluye por defecto un gran número de estadísticas y facilidades para depuración, además de poder incluir de manera sencilla las propias instrucciones que se proponen en este trabajo .

Se comparará por separado a las técnicas Early Release y TRXWSRESUME/TRXWSSUSPEND con la versión base y las dos versiones con 2 transacciones. Para cada versión, se mostrará el número de abortos que ha producido y el rendimiento en ciclos, desglosados según el motivo del aborto y según qué estén ejecutando, respectivamente. Con el objetivo de observar como evoluciona la escalabilidad de labyrinth con las nuevas instrucciones, se muestran datos utilizando 1, 2, 4 y 8 hilos. Los tamaños de entrada tienen especial relevancia en los resultados, ya que afectará al número de elementos en los conjuntos de lectura y escritura. Se han utilizado los tamaños descritos en STAMP como small y medium de labyrinth. Dependiendo del estado actual del entorno, el tiempo de ejecución de la aplicación puede variar drásticamente. Por ello, se han realizado 4 ejecuciones con los mismos parámetros pero introduciendo pequeñas perturbaciones aleatorias, para así poder observar un comportamiento "habitual" y descartar los valores inusuales.

4.1 Configuración

Para poder sacar conclusiones de los resultados, es necesario conocer de qué características dispone el sistema simulado, en especial las que afecten al comportamiento de las transacciones. Una determinada solución puede encajar bien en un sistema, pero en otro puede que sea inútil. Se ha optado por una configuración bastante cercana a las implementaciones más accesibles al público, véase Intel TSX[7]. Así, las conclusiones obtenidas podrán aplicarse a un mayor número de usuarios.

Una transacción se iniciará con una instrucción TRXBEGIN. A partir de este punto, las lecturas y escrituras se añadirán a los conjuntos de lectura y escritura, respectivamente. En este sistema se utilizan los bloques de caché como unidad de los conjuntos, añadiendo a cada bloque un bit R para el conjunto de lectura, y un bit W para el conjunto de escritura. El acceso a alguna de las direcciones del bloque pondrá el bit correspondiente a 1, lo que implica que no se podrá discernir qué dirección en concreto del bloque han sido accedido durante la transacción. Solo se podrá saber si alguna de las direcciones del bloque han sido accedidas, lo cual puede provocar falsos conflictos. En caso de no abortar, la ejecución de la instrucción TRXCOMMIT confirmará la transacción.

La detección de conflictos se basará en reutilizar el protocolo de coherencia para detectar cuándo se produce un acceso que pueda poner en riesgo la atomicidad. Un cambio de propiedad en una caché privada de un bloque indica que otro hilo quiere leerlo y previamente se ha modificado en esta caché, o que quiere modificarlo habiéndose modificado o leído previamente. Esto significaría una condición de carrera, lo cual se quiere impedir para garantizar le ejecución atómica de la transacción. Se abortará la transacción que reciba el mensaje justo en ese momento, siendo por tanto una detección de conflictos ansiosa (*eager*).

En el caso de que el acceso en la caché privada sea una escritura, el valor antiguo se propagará a niveles inferiores de caché. De esta forma, en caso de aborto, únicamente habrá que invalidar los bloques escritos para que el valor previo se recupere. Esto implica que los bloques escritos deberán mantenerse en el primer nivel para no aumentar la complejidad en exceso. En caso de tener que expulsar uno del primer nivel, la transacción deberá abortar por ser incapaz de rastrear ese bloque a partir de ese momento. Este comportamiento significa un control de versiones perezosa (*lazy*).

El mantenimiento del conjunto de lectura es más sencillo. Solo será necesario asociar un bit de lectura a los bloques, lo que facilita la extensión del conjunto de lectura a la caché de segundo nivel. Así se podrán expulsar bloques que solo hayan sido leídos al siguiente nivel de mayor tamaño. También se podrían incluir instrucciones que permitieran no añadir al conjunto de lectura determinadas direcciones, pero ya que el soporte del conjunto de lectura en el segundo nivel ya está implementado en la versión UMU de gem5, se ha optado por esta última opción.

Otra característica importante de la que dispone el sistema utilizado es un algoritmo de remplazo de caché que tiene en cuenta los recursos en los conjuntos de lectura/escritura. Resulta sencillo de implementar comprobando los bits R y W, y omitiendo los bloques que los tengan activos. Sin esta característica, sería posible que, aún disponiendo de mucho espacio libre en la caché para los conjuntos de lectura/escritura, se expulsara alguno de los bloques en los conjuntos. Se estarían infrautilizando el hard-

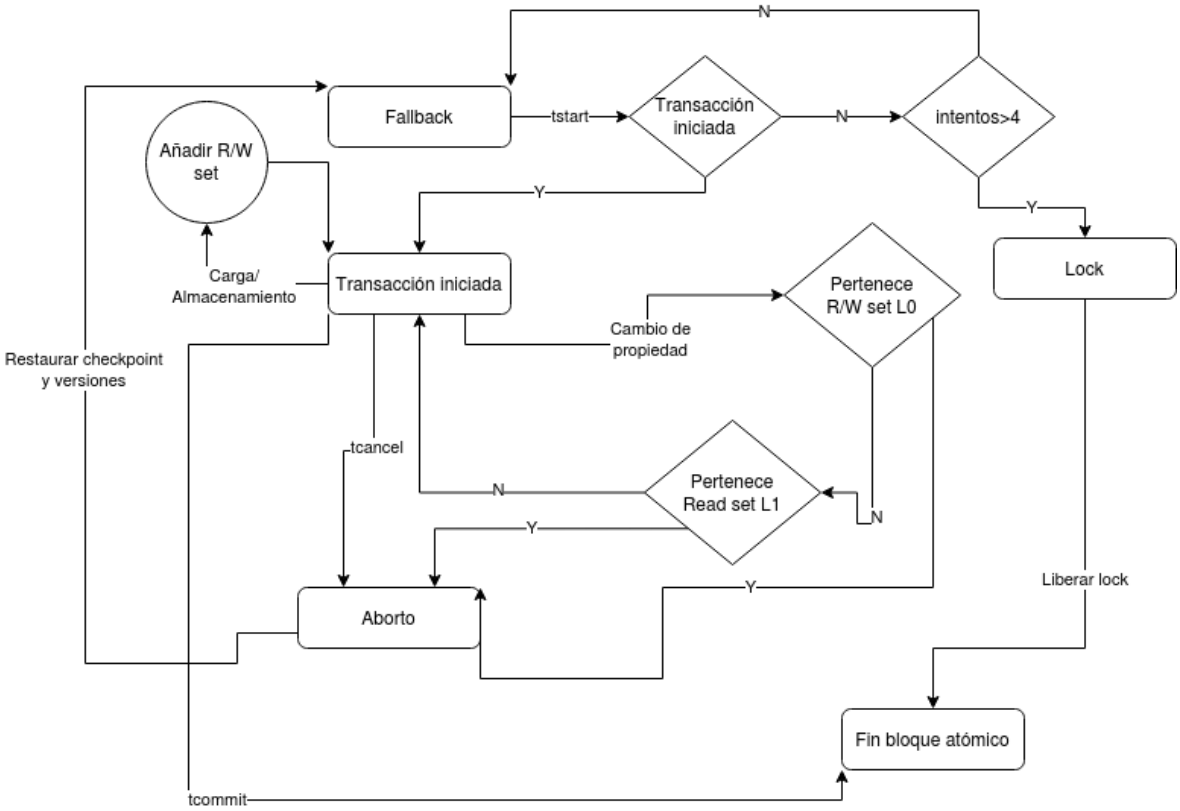

Figura 4.1: Diagrama lógico del sistema de Memoria Transaccional

ware disponible y generando más abortos por capacidad.

También es conveniente conocer las características generales del sistema. Se ha utilizado el modelo detallado de memoria proporcionado por Ruby, junto con el modelo de procesador fuera de orden O3CPU. El procesador es un CMP con un número de nodos variable, como se describe en la tabla 4.1. Cada nodo contiene un núcleo con cachés L1 privadas de datos e instrucciones, una L2 privada y una porción de la caché L3 compartida con las entradas de directorio asociadas. Se utiliza una red Crossbar. Las coherencia se mantiene a través de un directorio distribuido que mantiene vectores de bits de los compartidores e implementa el protocolo MESI.

4.2 Benchmarks

Tal y como se ha dicho al principio de la sección, se ha utilizado el benchmark labyrinth de STAMP. Este programa utiliza una única transacción para calcular un camino mínimo entre dos puntos. Consta de una fase de privatización, donde la matriz global

Tabla 4.1: Características del sistema

Configuración del núcleo	
Núcleos	1/2/4/8, fuera de orden F|D|R|IEW|C
Cola de cargas	32
Configuración de memoria	
L1 I&D cachés	Privada, 32/128 KiB, 8-way, acierto 2 ciclos
L2 caché	Privada, 512 KiB, 8-way
L3 caché	Compartida, 16 MiB, unificada, 16-way, inclusiva
Memoria	3GB, DDR3_1600_8x8
Protocolo	MESI, directorio, vector de bits
Configuración de la red interconexión	
Topología y enrutamiento	Crossbar, Simple
Latencia enlace	1 ciclo

se copia en memoria privada del hilo. La segunda fase es la de expansión, la cual determina si existe o no un camino entre la pareja de puntos. Gracias a la privatización, esta fase puede hacerse sin condiciones de carrera. La tercera fase se encarga de determinar el camino más corto en caso de que la expansión determine que al menos existe un camino entre ambos puntos. Por último, el nuevo camino se añade a la matriz global para comprobar si es correcto. El realizar todas estas operaciones en una única transacción genera un conjunto de lectura y escritura grande que conforme aumente el tamaño de la entrada, será más costoso de mantener. STAMP ofrece por defecto distintos tamaños de entrada (Tabla 4.2), los cuales afectarán al tamaño de los conjuntos de lectura y de escritura. Para la evaluación, se han utilizado los tamaños small y medium. Un tamaño demasiado grande puede incrementar el tiempo de simulación exageradamente, y con estos tamaños es suficiente para poder estudiar este problema. Además, hacer la privatización en la misma transacción que el resto de fases, provoca que se mantenga el aislamiento de la matriz global durante demasiado tiempo. A lo largo de este trabajo se han propuesto distintas alternativas para paliar estos problemas, las cuales van a evaluarse en la siguiente sección. A continuación se recogen los nombres que utilizará cada versión a lo largo del proceso de evaluación:

- labyrinth : versión base de labyrinth. Utiliza una transacción y no utiliza ninguna de las nuevas instrucciones.
- labyrinth_2_transactions : utiliza dos transacciones. En caso de aborto por conflicto en la segunda transacción, se vuelve a intentar la segunda transacción.
- labyrinth_2_transactions_always_copy : utiliza dos transacciones. En caso de

	Dimensiones	Caminos a enrutar
small	32 × 32 × 3	96
medium	48 × 48 × 3	64
large	512 × 512 × 7	512

Figura 4.2: Tamaños de entrada de labyrinth en STAMP

aborto por conflicto en la segunda transacción, se salta a la primera transacción para hacer una nueva copia.

- labyrinth_release_single_addr_from_rs : utiliza una transacción. Después de la privatización, se utiliza TRXSRELEASE para liberar la matriz global del conjunto de lectura.
- labyrinth_release_all_addr_from_rs : utiliza una transacción. Después de la privatización, se utiliza TRXARELEASE para liberar la matriz global del conjunto de lectura.
- labyrinth_disable_ws_adding : utiliza una transacción. Después de la privatización, se utiliza TRXARELEASE para liberar la matriz global del conjunto de lectura. También se utiliza TRXWSRESUME/TRXWSSUSPEND desde el final de la privatización hasta antes de añadir la nueva solución.

4.3 Resultados

Antes de interpretar los resultados, resulta indispensable entender qué implicaciones tiene cada tipo de aborto. A continuación se explican aquellos que aparecen a lo largo de todas las gráficas expuestas:

Transaction size wset. Es necesario mantener el conjunto de escritura en la caché de primer nivel. En caso de tener que expulsar un bloque, se aborta la transacción, ya que las cachés de siguientes niveles no tienen soporte para mantener el conjunto de escritura. En este caso, es equivalente al término aborto por capacidad, ya que no van a ocurrir abortos en otros niveles ni tampoco referentes al conjunto de lectura. Uno de los objetivos de este trabajo es minimizarlos a través de las instrucciones TRXWSRESUME/TRXWSSUSPEND.
Exception/Interruption. La ejecución del Sistema Operativo durante una transacción no es trivial. Durante su ejecución, no sería correcto abortar la transacción en caso de conflicto ya que el kernel podría haber modificado una serie de recursos y dejarlos en un estado inconsistente. Por ello, la solución más habitual es abortar la transacción en cuanto ocurre una excepción, como un fallo de página, o una interrupción, como lo

es un Timer Interrupt. En concreto, los fallos de página implican leer una página de disco, la cuál es posible que expulse a otra página a disco. En caso de abortar, la tabla de páginas y los datos de la propia página volverían a su estado original, pero no así el disco, por lo que serían inconsistentes. Por otro lado, los Timer Interrupt son utilizados para cambiar el contexto. Un cambio de contexto de un proceso que esté ejecutando una transacción implicaría que los conjuntos de lectura/escritura deben mantenerse también en el contexto del proceso, lo cual haría necesario modificar el Sistema Operativo.

Memory conflict. La detección de conflictos es el motor principal de la Memoria Transaccional. Cuando se produzca un acceso que viole el aislamiento de una transacción, ésta deberá abortar para garantizar una ejecución correcta. Sin embargo, al utilizar una granularidad de bloque de caché, también podrán ocurrir falsos conflictos, incluidos en esta estadística. Este trabajo tiene entre otros objetivos reducir este tipo de abortos a través de la instrucción Early Release.

Memory conflict fallback lock. Este tipo de aborto también mide los conflictos, pero sobre un recurso concreto. El fallback utilizado se apoya en un Single Global Lock (SGL) que se utiliza en caso de determinarse que una transacción no va a poder completarse. Esto implica que todas las transacciones comparten el mismo recurso, generando una alta contención. Resulta interesante tener una estadística independiente para poder medir el comportamiento a nivel de conflicto de la propia aplicación, independientemente del uso del fallback lock.

Explicit. Este tipo de aborto indica que el software a abortado la transacción a través de TRXCANCEL. En labyrinth, esto implica que al añadir el nuevo camino a la matriz global se ha detectado un conflicto previo.

Cabe destacar que, aunque se muestren en una misma gráfica, no todos los abortos tienen las mismas implicaciones en el rendimiento. Por ejemplo, es bastante probable que un aborto de capacidad en el primer nivel ocurra mucho antes que un aborto por conflicto al final de la transacción. Además, dentro del mismo tipo de aborto, cada uno puede implicar deshacer una cantidad distinta de trabajo. Por ejemplo, no es lo mismo que un aborto por conflicto ocurra al principio de la transacción que al final. Es importante tener esto en cuenta, ya que muchas veces resulta en mejor rendimiento que un tipo de aborto más prematuro ocurra antes que otro más tardío. Se espera que estas diferencias se puedan visualizar en la gráfica de tiempos, y que apoyándose en la de abortos totales, se puedan obtener conclusiones correctas.

Antes de comentar los resultados en detalle, he creído conveniente explicar el comportamiento de la Memoria Transaccional con un único hilo. No existirán conflictos ya que todos los caminos se calculan secuencialmente, pero seguirán aplicándose las limitaciones de las transacciones. Esto provoca que, aún sin realmente estar haciendo uso de la detección de conflictos se produzcan abortos asociados a estas limitaciones, como lo son de capacidad, excepción... Por lo tanto, si la intención es usar un único

hilo, por ejemplo para un tamaño de entrada pequeño, es recomendable usar la versión secuencial del programa.

Tal y como se ha comentado en la sección anterior, en una transacción excesivamente larga puede suceder que una determinada dirección se deje de utilizar a partir de cierto punto. En caso de que ese recurso provoque un conflicto, se estará abortando la transacción innecesariamente, limitando así el rendimiento máximo que el hardware podrá obtener. Como solución a este problema, se ha propuesto las dos instrucciones Early Release para que el programador pueda liberar los bloques que considere necesario. Para observar su utilidad real, se ha comparado el total de abortos en la versión base de labyrinth, las dos alternativas de 2 transacciones y utilizando Early Release. De esta forma se espera poder observar la evolución de los abortos, en especial los provocados por conflicto, conforme se aumenta el número de hilos. Es necesario recordar que por defecto, labyrinth genera un conjunto de lectura/escritura relativamente grande, por lo que muchas transacciones tendrán que ejecutarse a través del fallback. Esto implica que no sería posible obtener resultados detallados respecto a los conflictos, por lo que se ha decido utilizar un tamaño de caché ideal, en la que no ocurren abortos por capacidad.

En la figura 4.3 se observa la evolución de los abortos por capacidad para distintos tamaño de caché, con la que se pretende conocer ese tamaño ideal. Se ha utilizado la versión *labyrinth_release_all_addr_from_rs* por que en caso de que la contención disminuya, más caminos podrán seguir calculándose, pero puede que provoquen un aborto por capacidad. Además, se muestran los resultados para 1 y 8 hilos, ya que a mayor número de hilos es más probable que el orden en el que se calculen los caminos varíe. Esto puede provocar que el proceso de expansión de algunos caminos sea mayor, y por lo tanto su conjunto de escritura.

Se puede observar como, además de abortos por capacidad, aparecen abortos por excepción e interrupción. En este caso, los abortos por capacidad estaban enmascarándolos. Este tipo de abortos no dependen de la aplicación en sí, sino del Sistema Operativo, por lo que el momento en el que ocurran será indeterminado. En este trabajo no se espera resolver estos problemas, pero para poder obtener unos resultados que tengan una relación más directa con los problemas que sí se quieren resolver, se ha decidido evitar que estos abortos ocurran. Las interrupciones se pospondrán hasta que la transacción termine, ya sea por aborto o por confirmación. Para las excepciones, antes de iniciar la transacción por primera vez, se accede a todas las posibles páginas de memoria que la transacción vaya a utilizar. De esta forma, si ocurre un fallo de página ocurrirá en ese momento, y no durante la transacción. La figura 4.4 muestra cómo evolucionan los abortos por capacidad sin la influencia de las interrupciones ni excepciones. Se observa como para el tamaño de entrada pequeño, sería necesaria una caché de 128 KiB para que no ocurra ningún aborto por capacidad, lo cual es suficiente para evaluar los conflictos.

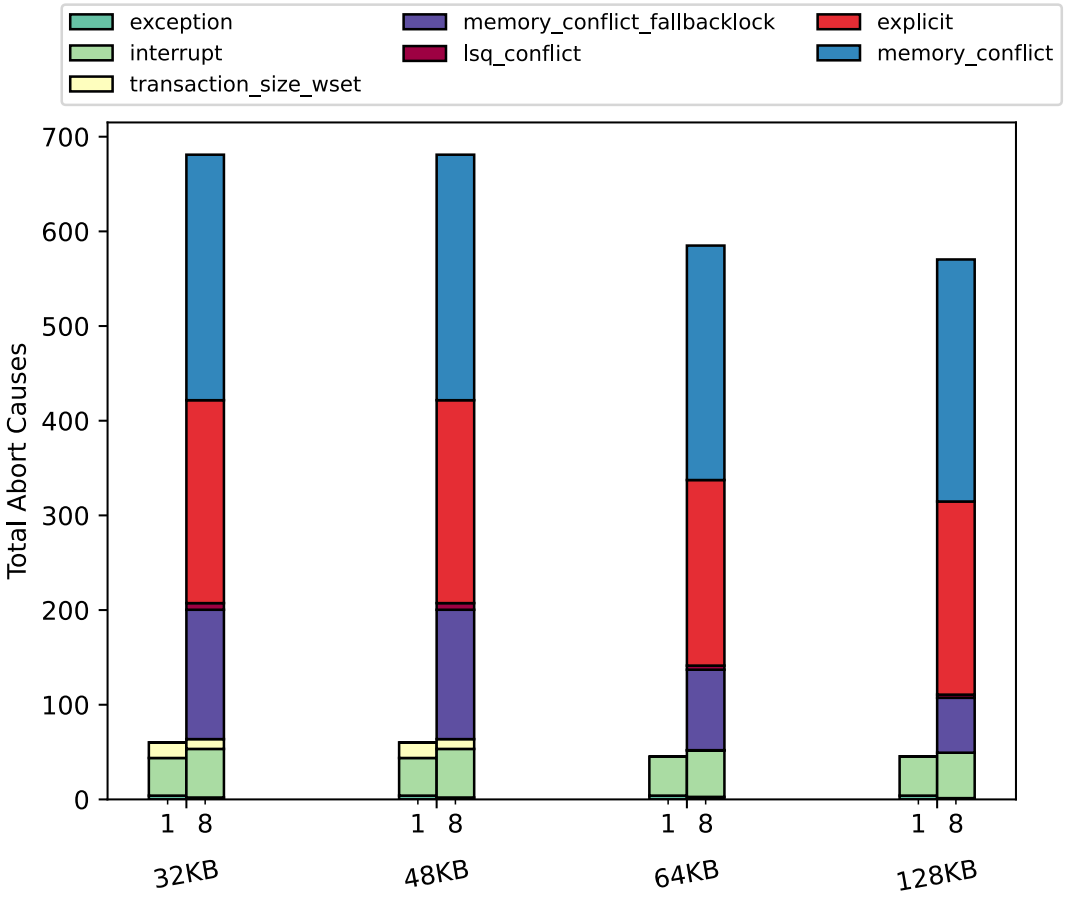

Figura 4.3: Número de abortos totales, diferenciados según su causa, agrupados número de hilos y tamaño de la caché de primer nivel. Sobre labyrinth_release_all_addr_from_rs y tamaño de entrada small

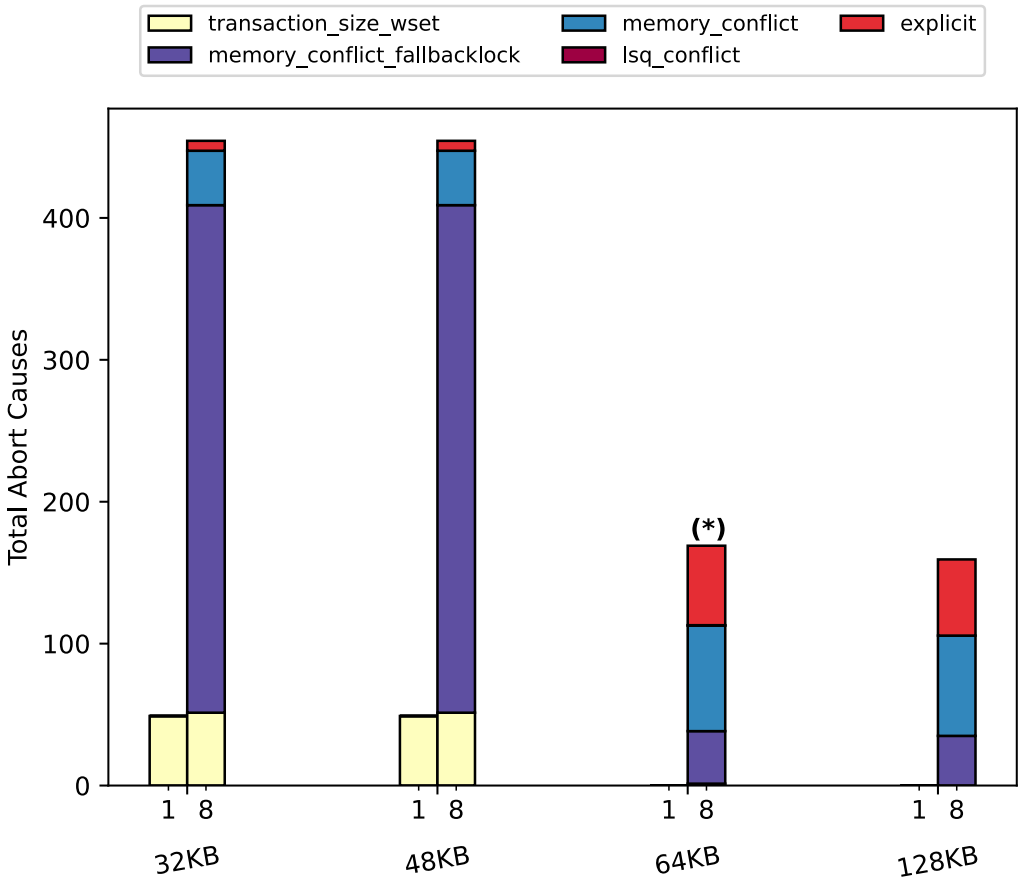

Figura 4.4: Número de abortos totales, diferenciados según su causa, agrupados por tamaño de entrada y tamaño de la caché de primer nivel. Sobre labyrinth_release_all_addr_from_rs y tamaño de entrada small. Excepciones e interrupciones evitadas. (*) Se dan 2 abortos por capacidad.

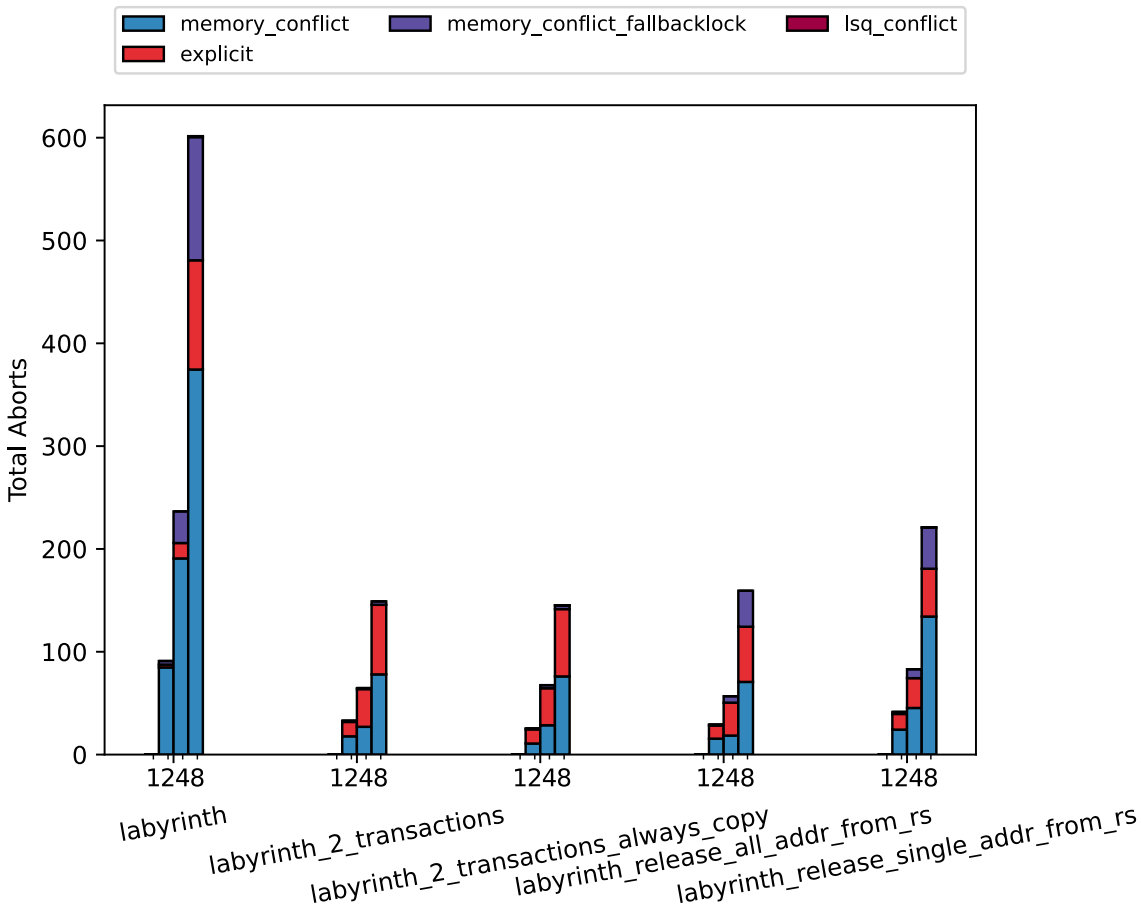

Figura 4.5: Número de abortos totales, diferenciados según su causa, para un tamaño de entrada pequeño y agrupados por número de CPUs y versión de labyrinth (Early Release). Tamaño de L1: 128 KiB

Los resultados (Figura 4.5) muestran una reducción de los conflictos de hasta un 30 por ciento gracias al TRXSRELEASE respecto a la versión base. Es necesario recordar que TRXSRELEASE está limitada cuando el número de recursos a liberar es elevado. Aunque se sepa que a partir de cierto punto una serie de recursos pueden ser eliminados de los conjuntos, hasta que no se ejecute el último Early Release podrán generarse conflictos sobre ese grupo de recursos. De esta forma, aunque la idea de este método es muy útil, se verá lastrada por una granularidad a veces demasiado pequeña. Esta limitación se observa al compararla con TRXARELEASE, donde liberar todas las direcciones con una única instrucción permite reducir aún más los conflictos. Si comparamos ambas versiones con la de 2 transacciones, se observa como la contención sobre el SGL es casi inapreciable al utilizar 2 transacciones. Ya que este recurso debe mantenerse constantemente en el conjunto de lectura, conforme más larga sea la tran-

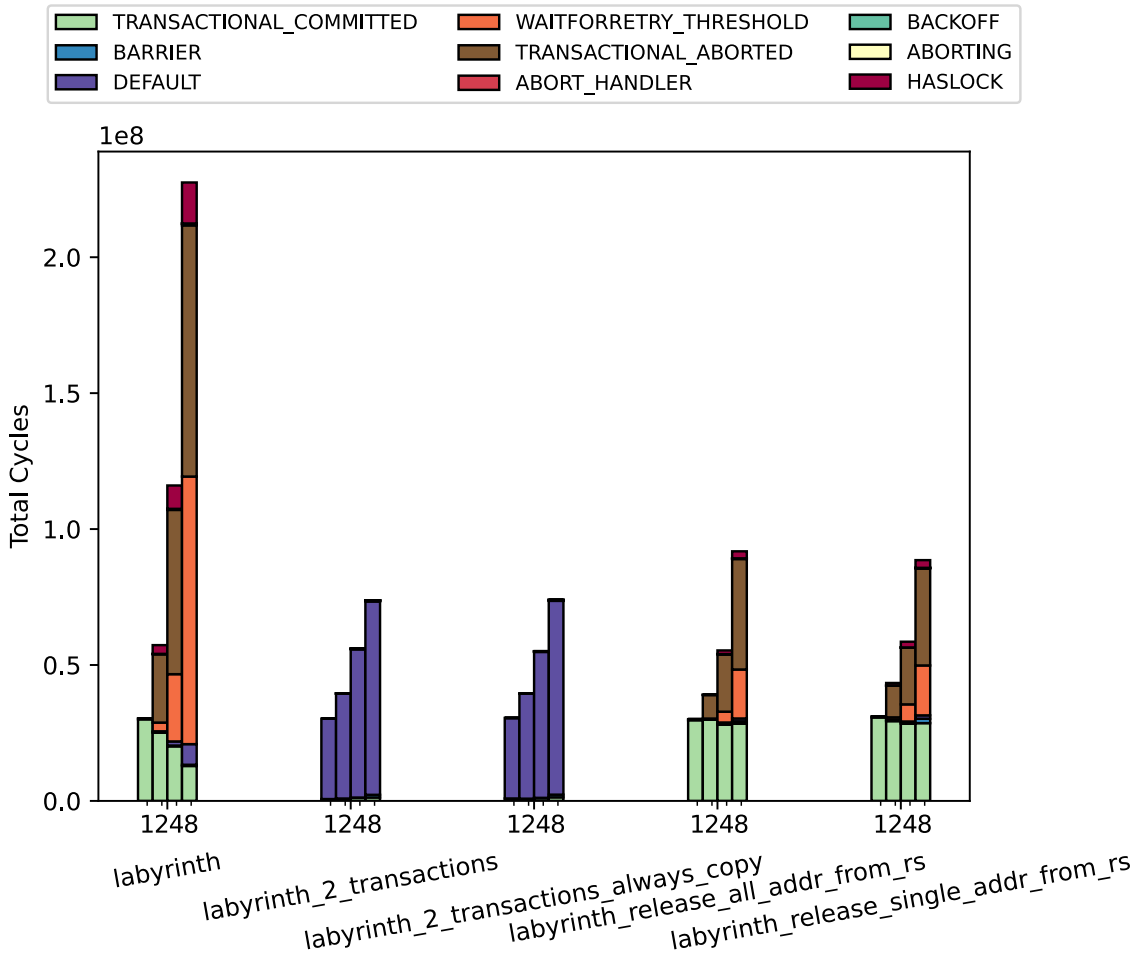

Figura 4.6: Ciclos ejecutados para un tamaño de entrada pequeño y agrupados por número de CPUs y versión de labyrinth (Early Release). Tamaño de L1: 128 KiB

sacción, más probable es que otra transacción adquiera el lock. Este conflicto provocará que todas las transacciones activas aborten. Para solventar este problema utilizando una única transacción, sería necesario utilizar otro fallback que no utilizara un único lock.

La reducción de los conflictos sobre el grid global muestra una mejora de rendimiento considerable respecto a a la versión base (Figura 4.6). Que ocurran menos conflictos implica que se reduce el número de instrucciones que tienen que volver a ejecutarse en caso de aborto, lo cual se puede observar en la figura con TRANSACTIONAL_ABORTED. Si se comparan las versiones de 2 transacciones con el resto, se puede entender lo que implica utilizar 2 transacciones en vez de 1. La mayor parte del tiempo, se está ejecutando código no transaccional (DEFAULT). Esto es porque la expansión y el traceback se realizan fuera de las transacciones, siendo la parte de labyrinth más intensiva computacionalmente. Esto también permite visualizar como el SGL apenas va a provocar conflictos, ya que solo está presente en los conjuntos de lectura durante durante la privatización y al añadir la solución.

Incluso trabajando con un tamaño de entrada pequeño, labyrinth genera un conjunto de lectura/escritura bastante grande en proporción a los tamaños de caché actuales. Con las instrucciones TRXWSRESUME/TRXWSSUSPEND que se han propuesto, se espera poder reducir los conjuntos de escritura lo suficiente como para reducir los abortos por capacidad. Se ha comparado la versión base de labyrinth, las dos alternativas de 2 transacciones y por último, la versión que utiliza TRXWSRESUME/TRXWSSUSPEND. Ya que se ha comprobado que se producen un alto número de conflictos sobre el grid global al privatizarlo, esta última versión también utiliza Early Release para que solo se observen los conflictos estrictamente necesarios. En concreto, se utiliza TRXARELEASE, que es el que mejores resultados proporciona.

En la figura 4.7 se puede observar como se producen alrededor de 50 abortos por capacidad en labyrinth, que resulta ser más del 50% de los caminos a generar. Esto muestra que todavía hay oportunidad de concurrencia sin explotar. En las dos variantes de 2 transacciones se observa como los abortos por capacidad desaparecen gracias a que las transacciones resultantes tienen un conjunto de escritura pequeño. Si las comparamos con TRXWSRESUME/TRXWSSUSPEND, los abortos por capacidad también desaparecen, aunque todavía aparecen los conflictos sobre el SGL, tal y como se ha mencionado en el estudio de Early Release. Con estos resultados se puede demostrar como, por una lado, que el uso de TRXWSRESUME/TRXWSSUSPEND consigue el objetivo de reducir los abortos por capacidad. Por otro lado, también se observa como una granularidad adecuada de las transacciones puede llegar a aportar una mejora de rendimiento considerable respecto a la versión base, aunque a costa de una complejidad mayor (Figura 4.8). Hay que tener en cuenta además que gracias a TRXWSRESUME/TRXWSSUSPEND, se consiguen resultados similares (Figura 4.9)

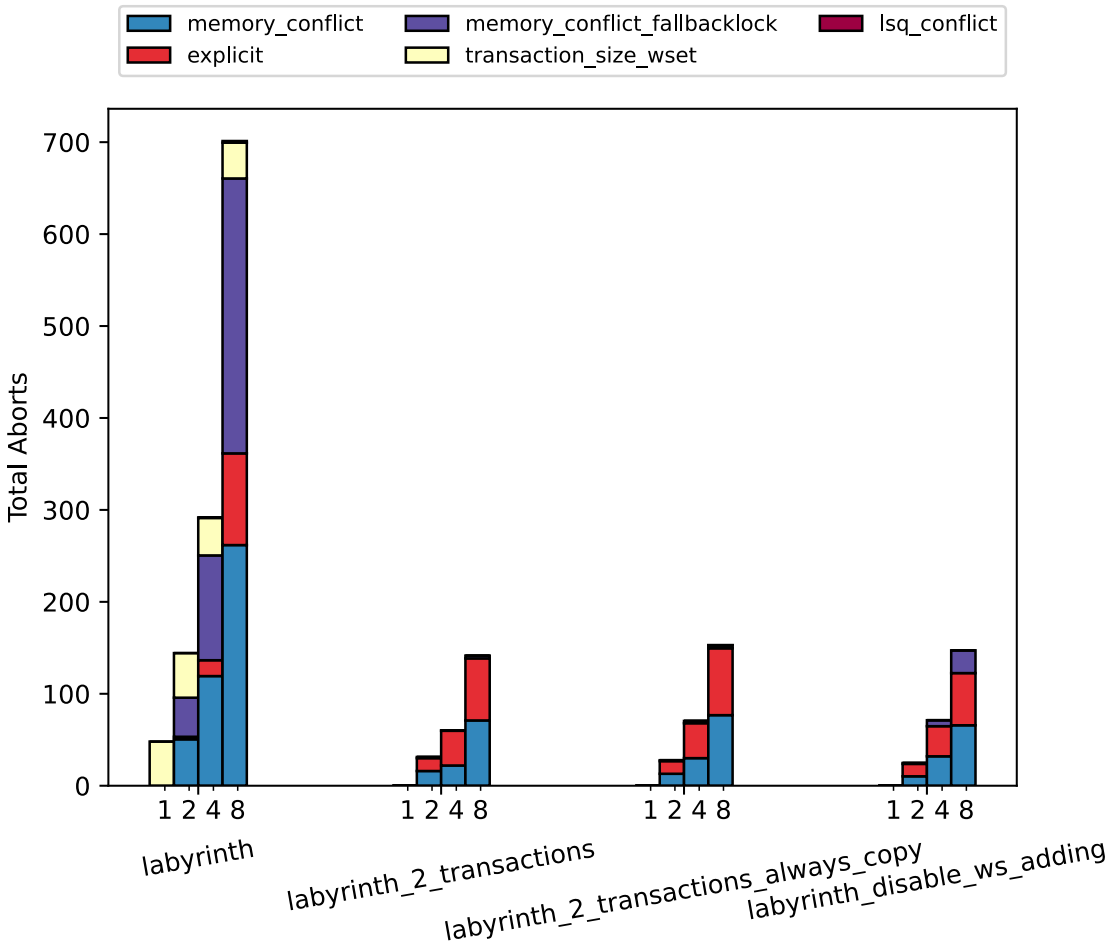

Figura 4.7: Número de abortos totales, diferenciados según su causa, para un tamaño de entrada pequeño y agrupados por número de CPUs y versión de labyrinth

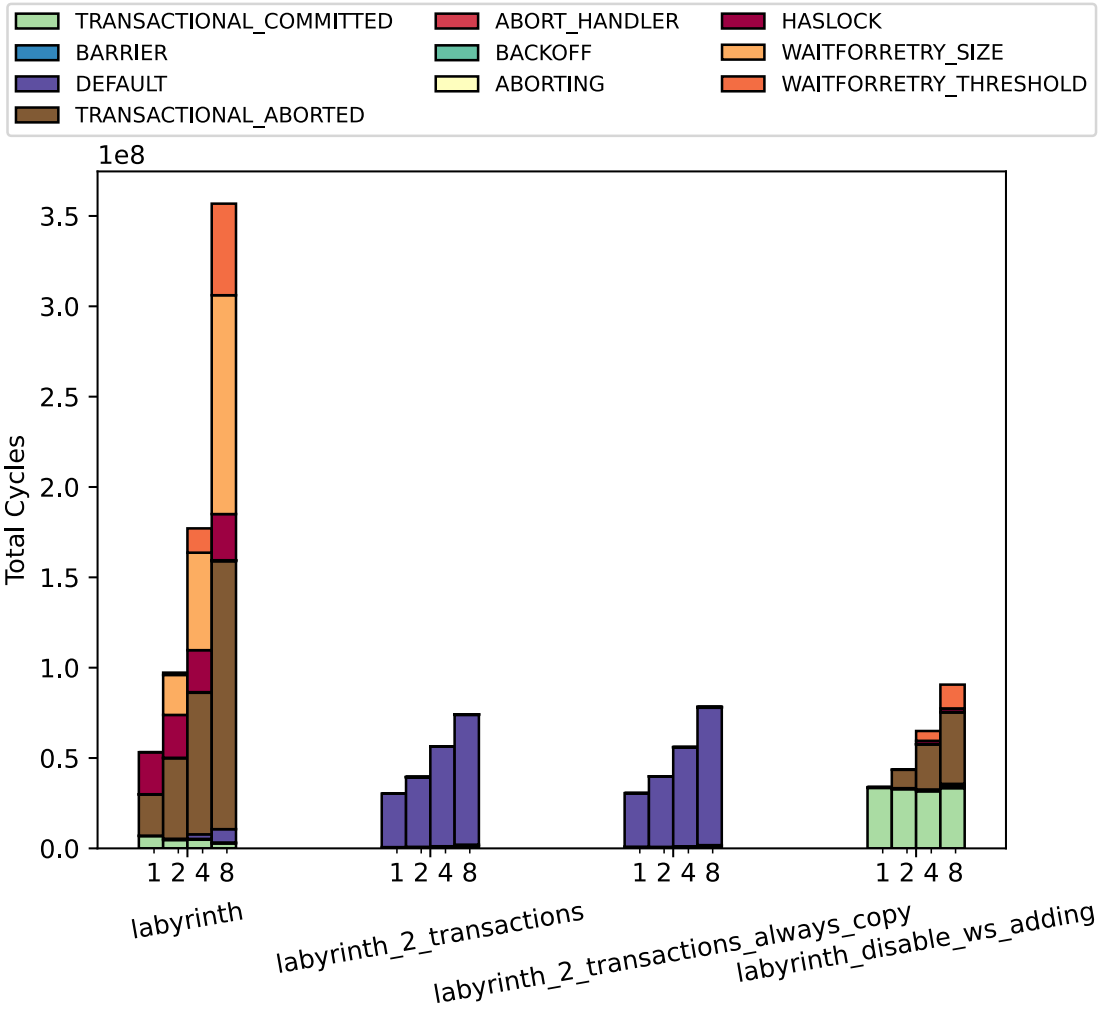

Figura 4.8: Ciclos de ejecución para un tamaño de entrada pequeño y agrupados por número de CPUs y versión de labyrinth

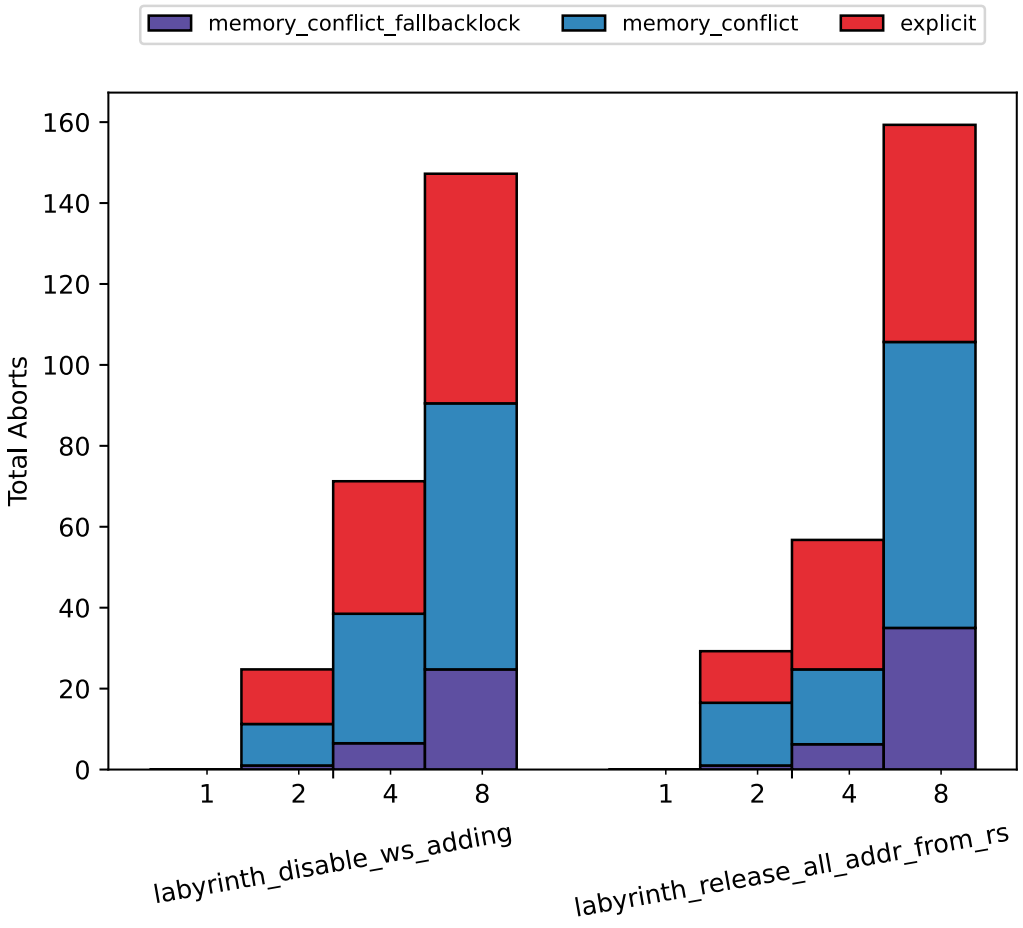

Figura 4.9: Comparativa entre TRXWSRESUME/TRXWSSUSPEND con 32 KiB de L1 y sin TRXWSRESUME/TRXWSSUSPEND con 128 KiB de L1 (Ambas usan TRXARELEASE)

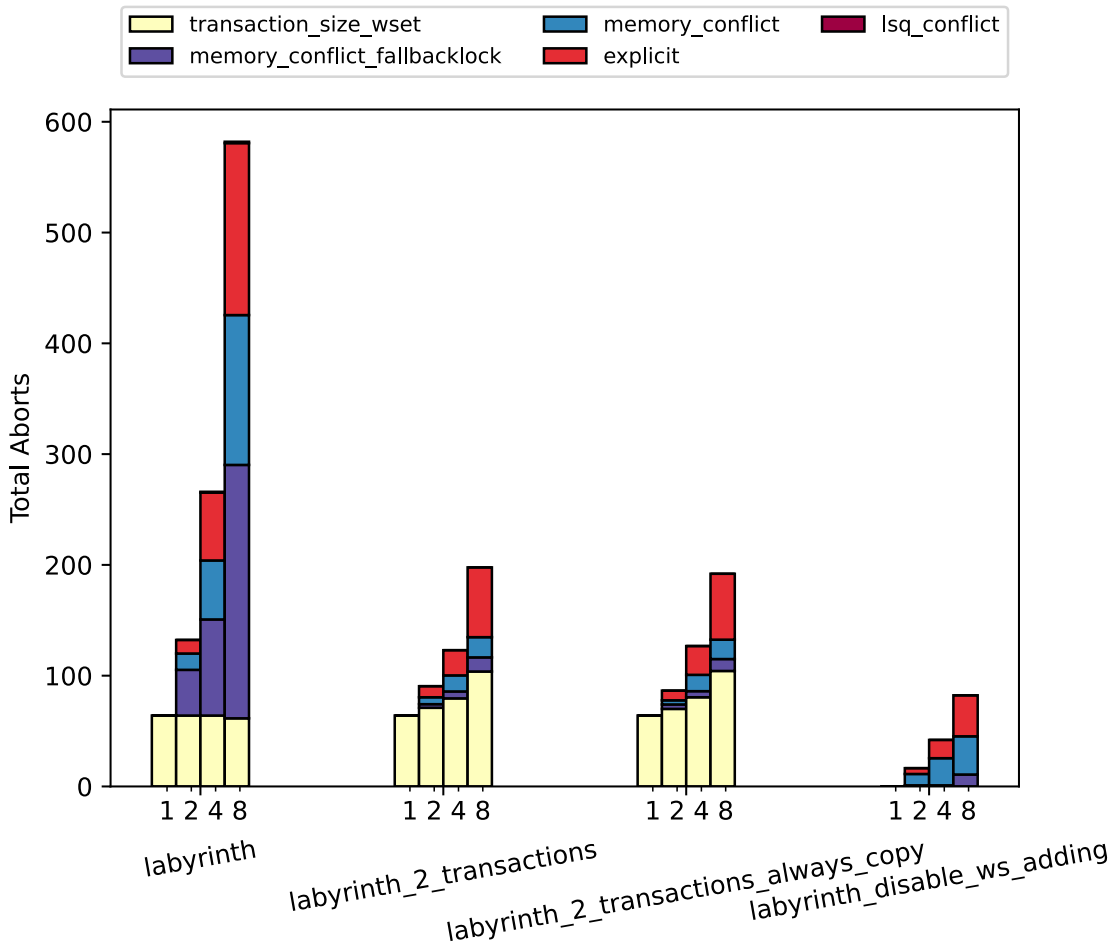

Figura 4.10: Número de abortos totales, diferenciados según su causa, para un tamaño de entrada medio y agrupados por número de CPUs y versión de labyrinth

que utilizando una caché cuatro veces mayor.

Para estresar más el sistema de Memoria Transaccional, se han evaluado las distintas versiones sobre un tamaño de entrada mayor. Solo intentar hacer la copia privada del laberinto va a provocar un aborto por capacidad. En estas circunstancias, aún dividiendo la transacción en dos, sigue teniendo la misma limitación que una única transacción (Figura 4.10). En el mejor de los casos, el conjunto de escritura estará limitado por el tamaño de la caché. En la transacción 1, solo se podrá copiar un laberinto de como mucho 32 KiB, perteneciendo todos los bloques al conjunto de escritura. TRXWSRESUME/TRXWSSUSPEND mantiene los abortos por capacidad a 0 incluso aumentando el tamaño de entrada. Por lo tanto, nos podemos plantear la siguiente pregunta: ¿Y si, además de las dos transacciones, dispusiéramos también de TRXWSRESUME/TRXWSSUSPEND? La primera transacción estaría limitada entonces solo

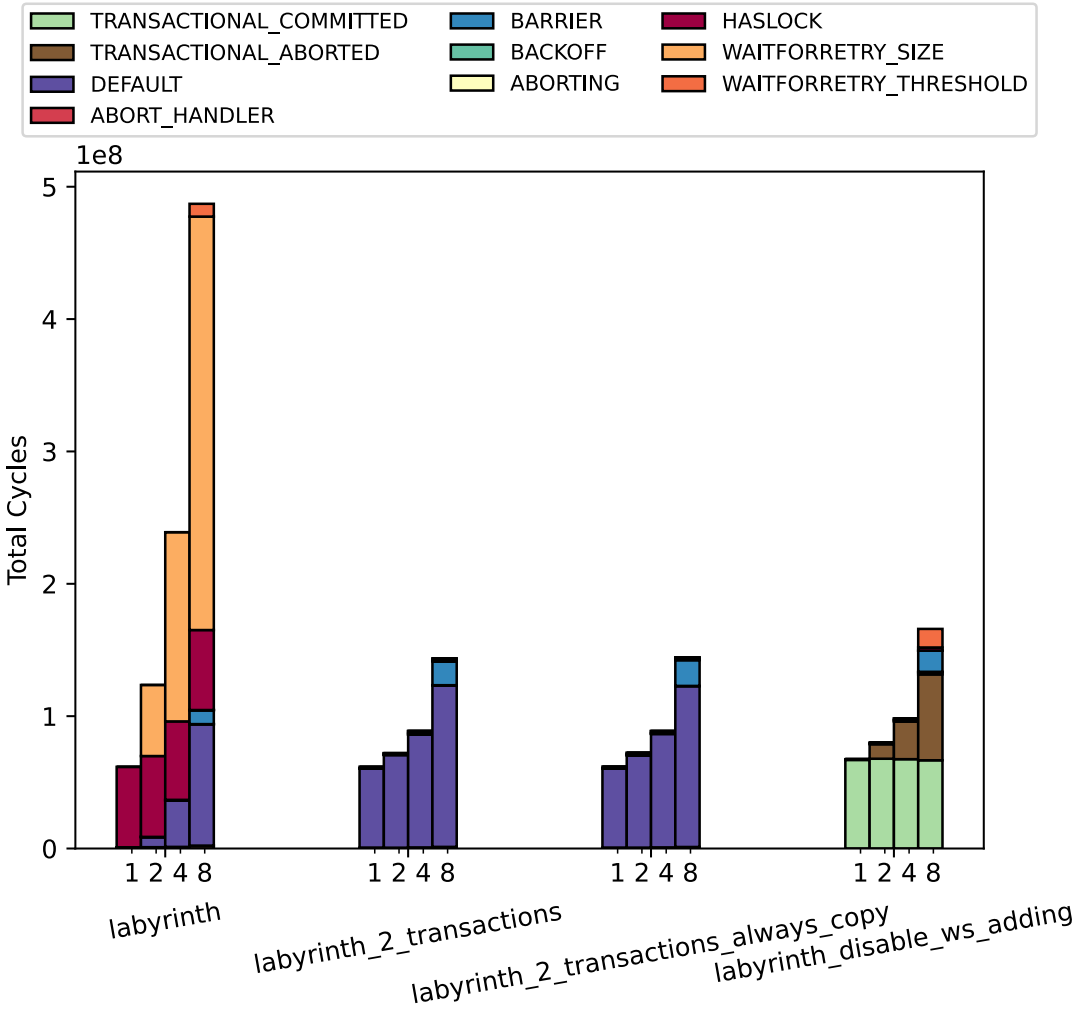

Figura 4.11: Ciclos de ejecución para un tamaño de entrada medio y agrupados por número de CPUs y versión de labyrinth

por el conjunto de lectura, que en este caso puede abarcar tanto la caché de primer nivel como la del segundo. Esto significa que determinados abortos por capacidad no podrán evitarse a no ser que el sistema de MT permita al usuario modificar su comportamiento por defecto. En este caso concreto, los abortos por capacidad usando 2 transacciones no tienen un gran impacto en el rendimiento, ya que la mayor parte del cómputo se realiza en la expansión (Figura 4.11). Sin embargo, en otras aplicaciones en las que las transacciones resultantes sigan siendo demasiado grandes, el evitar ejecutarlas sin acudir al fallback ofrecerá una mejora de rendimiento considerable.

Gracias a este proceso de evaluación, se ha podido observar como, gracias un sistema de Memoria Transaccional más transparente y flexible, es posible conseguir un rendimiento similar a una alternativa software de dos transacciones, utilizando una única transacción. Para que las nuevas características puedan rendir al máximo, será fundamental que el programador conozca la aplicación de tal forma que no provoquen resultados incorrectos. Únicamente será necesario saber si sobre un recurso puede producirse una condición de carrera, y en caso afirmativo, si puede ocurrir a lo largo de toda la transacción o solo hasta cierto punto. Por un lado, TRXWSRESUME/TRXWSSUSPEND permite reducir los abortos por capacidad debido a recursos que no necesitan de aislamiento no añadiéndolos al conjunto de escritura. Por otro lado, Early Release reduce los conflictos provocados por recursos que solo necesiten un aislamiento temporal, y no durante toda la transacción.

5 Conclusiones

Con usuarios cada vez más exigentes, las aplicaciones necesitan aprovechar al máximo los procesadores multinúcleo. La Memoria Transaccional se propone como alternativa a los métodos de sincronización tradicionales, detectando las condiciones de carrera en el hardware, pudiendo conseguir un rendimiento similar a alternativas más complejas con una complejidad menor para el programador. Además, su integración en el hardware actual implica un coste mínimo, ya que puede reutilizar muchos de los mecanismos hardware actualmente disponibles, como lo son las cachés y los protocolos de coherencia. Sin embargo, esta ventaja tiene un inconveniente. Las posibilidades de la Memoria Transaccional estarán limitadas por el hardware sobre el que se implemente.

En este trabajo, se han propuesto 4 nuevas instrucciones que permiten evadir alguna de estas limitaciones. Dos de las instrucciones se basan en el concepto Early Release. Permitirán eliminar direcciones del conjunto de lectura durante una transacción, y así evitar conflictos por un aislamiento demasiado estricto. La primera, TRXSRELEASE, elimina un único bloque, por lo que será útil cuando se deba seguir manteniendo el aislamiento sobre otras direcciones. Sin embargo, conforme aumente el número de direcciones que haya que liberar, habrá que ejecutar un mayor número de instrucciones. Por ello, la segunda instrucción TRXARELEASE permite liberar, con una única instrucción, todos los bloques del conjunto de lectura, excepto el que contenga el SGL.

Las otras dos instrucciones, TRXWSRESUME/TRXWSSUSPEND, permitirán definir un intervalo dentro de una transacción en el que las escrituras no se añadan al conjunto de escritura. Las alternativas similares existentes tienen ciertas limitaciones. Las instrucciones de TSX, XSUSLDTRK y XRESLDTRK, no se pueden ejecutar de forma anidada, por lo que si un módulo software las utiliza, y una aplicación utiliza también las instrucciones y el módulo, la transacción abortará. Por otro lado, las instrucciones de POWER, TRESUME y TSUSPEND, permiten definir una región de código no transaccional dentro de una transacción, desactivando incluso la detección de conflictos. Aunque son útiles, su implementación es excesivamente compleja en caso de únicamente querer controlar los conjuntos de lectura/escritura.

Los resultados de la evaluación muestran como Early Release permite reducir los conflictos provocados por la estructura compartida una vez hecha la privatización. Se ha observado una mejora de alrededor del 2.25× usando TRXSRELEASE. TRXARELEASE ofrece un rendimiento ligeramente superior, pero a costa de una menor

flexibilidad que la primera alternativa.

Con TRXWSRESUME/TRXWSSUSPEND se ha observado como se pueden reducir, e incluso eliminar, los abortos por capacidad provocados por un conjunto de escritura grande, permitiendo que un mayor número de caminos puedan calcularse en paralelo. Sin embargo, implica que hay que controlar la reutilización de memoria reservada antes de una transacción, teniendo que evitar reutilizarla durante la transacción, lo cual requiere de soporte por parte del compilador. Junto con TRXARELEASE, TRXWSRESUME/TRXWSSUSPEND permite obtener un rendimiento de 3.5× para tamaños de entrada pequeños, y alrededor de 2.5× para tamaños medianos, manteniendo un tamaño de caché de primer nivel modesto.

Las nuevas instrucciones se han comparado con una alternativa software que aumenta la granularidad de las transacciones. Los resultados han mostrado que esta versión obtiene los mismos o incluso mejores resultados que utilizando las instrucciones sobre una única transacción. Esto nos hace preguntarnos si las instrucciones, aunque se han demostrado que funcionan, pueden llegar a ser útiles. ¿Merece la pena modificar un ISA con instrucciones cuyo efecto puede obtenerse a nivel software?. Las instrucciones propuestas son simples, por lo que la lógica necesaria para incluirla en un dispositivo sería mínima. Aprovechando la estructura de RISCV, podrían incluirse en una nueva extensión. De esta forma, aunque el coste de integración resultara elevado para determinados dispositivos, podrán decidir si incluir estas instrucciones dependiendo de sus clientes. En mi opinión, no estaría de más poder disponer de estas instrucciones o similares, de tal forma que el programador tenga las máximas posibilidades a su alcance, y sea él mismo quien decida.

Bibliografía

[1] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10, 02 1970. doi: 10.1007/s004460050028.

[2] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, page 92–101, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137087. doi: 10.1145/872035.872048. URL https://doi.org/10.1145/872035.872048.

[3] Christos Kozyrakis. The case for hardware support for transactional memory, 2008. URL https://web.archive.org/web/20220121114544/https://web.stanford.edu/~kozyraki/publications/2008.case4htm.slides.pdf.

[4] Kevin Moore, Jayaram Bobba, Michelle Moravan, Mark Hill, and David Wood. Logtm: Log-based transactional memory. volume 2006, pages 254–265, 01 2006. doi: 10.1109/HPCA.2006.1598134.

[5] C. Ananian, Krste Asanovic, Bradley Kuszmaul, Charles Leiserson, and Sean Lie. Unbounded transactional memory. volume 26, pages 316–327, 01 2005. doi: 10.1109/HPCA.2005.41.

[6] Amy Wang, Matthew Gaudet, Peng Wu, José Amaral, Martin Ohmacht, Christopher Barton, Raúl Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. pages 127–136, 09 2012. ISBN 9781450311823. doi: 10.1145/2370816.2370836.

[7] Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. 11 2013. doi: 10.1145/2503210.2503232.

[8] Hung le, G. Guthrie, D. Williams, M. Michael, B. Frey, William Starke, C. May, R. Odaira, and Takuya Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59:8:1–8:14, 01 2015. doi: 10.1147/JRD.2014.2380199.

[9] Javier Garcia Hernandez. Implementación de memoria transaccional en riscv, 2021. URL `https://github.com/javiergh12/TrabajoFinGrado/blob/main/tfg.pdf`.

[10] Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. pages 35–46, 10 2008. doi: 10.1109/IISWC.2008.4636089.

[11] C. Y. Lee. An algorithm for path connections and its applications. *IRE Trans. Electron. Comput.*, 10:346–365, 1961.

[12] Maurice Herlihy, J. Eliot, and Eliot Moss. Transactional memory: Architectural support for lock-free data structures. volume 21, pages 289–300, 06 1993. ISBN 0-8186-3810-9. doi: 10.1109/ISCA.1993.698569.

[13] Mark Moir, Kevin Moore, and Daniel Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster). page 362, 01 2008. doi: 10.1145/1378533.1378595.

[14] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36, 2012. doi: 10.1109/MICRO.2012.12.

[15] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 102–113, 2004. doi: 10.1109/ISCA.2004.1310767.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark Hill, and David Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39:1–7, 08 2011. doi: 10.1145/2024716.2024718.

[17] Krste Asanovic Andrew Waterman. The risc-v instruction set manual, volume i: User-level isa, document version 20191213. 1, 12 2019.

[18] Morten B Petersen. Ripes: A visual computer architecture simulator. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, pages 1–8. IEEE, 2021.

[19] Pascal Felber, Christof Fetzer, and Torvald Riegel. Snapshot isolation for software transactional memory. 05 2022.

[20] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *CoRR*, abs/cs/0701157, 01 2007. doi: 10.1145/223784.223785.

[21] Eliot Moss and Antony Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming - SCP*, 63:186–201, 12 2006. doi: 10.1016/j.scico.2006.05.010.