Universidad de Murcia

Facultad de Informática

Trabajo Fin de Grado

Implementación de Memoria Transaccional en RISCV

Autor : Javier García Hernández

Tutores : Rubén Titos Gil y Ricardo Fernández Pascual

Septiembre de 2021

Abstract

Los sistemas de Memoria Transaccional por Hardware son cada vez más relevantes al simplificar el proceso de programación paralela y ofrecer un incremento del rendimiento en la ejecución de estas aplicaciones, lo que permite el aumento de la escalabilidad. En este trabajo, se propone la especificación de una extensión del ISA RISCV para incluir soporte de Memoria Transaccional por Hardware. Se implementará esta propuesta en el simulador gem5 para evaluar su funcionamiento. Se indicarán algunas alternativas en otras arquitecturas, los pasos seguidos para su implementación, los medios utilizados y finalmente, una evaluación de la solución propuesta.

Contents

Chapter 1

Extended Abstract

The appearance of multicore processors has made it possible to increase the performance of applications by means of running several threads at the same time. The order in which the instructions and memory acesses of different threads are interleaved is not deterministic. Because of that, it may not correspond to the order that the programmer intended.

This situation can lead to race conditions, where data accessed by one thread can be modified by another in an unexpected way. To avoid this, it is necessary to perform some synchronization between threads to allow the programmer to achieve atomic access to critical sections, constraining the order of execution. Locks have been traditionally the way to go, but some of its drawbacks have encouraged the development of Transactional Memory as an alternative.

Considering the relevance of Transactional Memory, the first objective of this work is to specify an extension to support Hardware Transactional Memory in RISCV, an ISA with an open specification. This extension will be implemented in the gem5 simulator, which already has support for RISCV and Transactional Memory. Finally, the resulting system will be evaluated using the STAMP benchmarks, which will provide an overview of its performance.

As we said, Transactional Memory is proposed as an alternative to locks. To use it, you only have to indicate which resources must be accessed atomically. The system itself will dynamically check that the resources have not been accessed in a conflicting way from other threads. Implementing the transactional memory support in hardware reduces the overhead involved in using it, although it brings more complexity to the design of the processor. To alleviate this problem, it is common to reuse existing hardware features, like the coherence protocol to detect conflicts. The definition of how a conflict is generated, when it is detected and how to solve it will define a Transactional Memory system.

The most common definition of conflict is an access, either a read or write, on a resource written by another transaction, or a write access on a resource read by another processor. Memory accesses performed within transactions are added to the transaction read and write sets, which are grouped in units of a given size which is called granularity. It is important to define a suitable granularity. If it is too low the read / write sets will be more difficult to maintain. And if it is too high, acesses to different parts of the unit may be conflated, increasing the number of conflicts.

Once we have defined what a conflict is, we have to establish when to detect them. If we choose an eager approach, the detection procedure will be applied to each access. This will avoid executing unnecessary instructions. If, on the other hand, we choose the lazy option, conflicts will be checked later, usually at the end of the transaction, eliminating the overhead of checking each access but executing unnecessary instructions in many cases.

If a conflict is detected, it needs to be resolved. The simplest and most direct action is to abort one of the transactions involved in the conflict. This will return the processor and memory accesed bu the transaction to the state just before starting the transaction. In order for this to be carried out, a copy of the previous values must be kept. For the processor, it is common to copy the values of all registers in auxiliary registers or in memory. For memory, a versioning mechanism is needed.

In order to implemnt versioning, it is necessary to have a buffer to handle speculative writes, that is, those made within the transaction. The eager approach uses the buffer to hold the previous values. If the transaction aborts, the values contained in the buffer will have to be copied back into memory. On the other hand, the lazy approach uses the buffer to write the new values. In case of commit, it will be necessary to copy the values of the buffer to memory.

Once we have defined the main concepts, we can proceed to design the solution. The first step to specify and implement Hardware Transactional Memory for RISCV is choosing its characteristics. As there is no enough time to implement every possible solution, a comparation between HTM systems offered by other ISAs will be made. The most common characteristics will be part of our new HTM system. Using the table 4.1 as reference, we can say that the system will be implicit, so only the limits of the transaction will need to be specified by the programmer to use it. Three new instructions will be added. One for starting the transaction, one for commiting and another to abort the transaction from software. Conflict detection will be eager, and versioning will be lazy, with the speculative buffer implemented in the L1 data cache. Finally, the new system will be best effort, delegating the restart of transactions to software.

The next step is to specify each of the instructions. The Begin instruction allows you to indicate in which register to save the result of the transaction. The Commit instruction will behave as a memory barrier, in order to work in out-of-order processors. Otherwise, some memory accesses might not be treated correctly. The Cancel instruction will have a parameter that indicates the register that keeps the reason of the cancellation.

Once the instructions have been specified, an encoding must be associated to them. There are various instruction formats in RISCV, among which the R format is chosen, which allows indicating both source and destination registers, and which contains an additional field to differentiate instructions with the same opcode, or instruction code. In this way, we will only have to use an opcode for the three instructions. When selecting the opcode, the first two bits must be 11, since the other combinations are reserved for compressed instructions. For the rest of the bits, there are values that are already being used and others that are reserved, so a free value will have to be chosen. In this case, the value 2 has been chosen. (Figures 4.2, 4.3 and 4.4)

In addition, it is necessary to include an specific register in the encoding. The instructions are not included in the compiler, so the register that will contain the parameter of the Begin and Cancel instructions are unknown at compilation.

Another consequence of the instructions not being emitted by the compiler is that using them can be complex. To alleviate this problem, a macro has been defined for each instruction. Furthermore, since this TM system is best effort, a library has been defined that implements an "alternative path", or fallback, using a lock for retrying aborted transacions. This will guarantee progress when transactions are continuously aborted due to contention or lack of resources.

The next step is to implement the instructions in gem5. A class must be associated with each one of them. The BEGIN and CANCEL instruction (Figure 4.5) will inherit from the RiscvStatisInst class to work as an instruction. The COMMIT instruction (Figure 4.6) is a peculiar case, since, as we have said, it must behave as a memory barrier. To do this, the COMMIT instruction will be defined as a macro operation. In this way, at code level only one instruction will be necessary, but two will actually be executed: the mfence microinstruction and another microinstruction, which will be called microCommit, which will implement the commit itself, inheriting from RiscvMicroInst. Microinstructions must be used so that the Program Counter is not modified until the macroinstruction completes. Finally, in the constructors of each class, it will be

necessary to indicate what type of instructions they are (memory references), and for each one, what type of TM instruction (start, commit or cancel).

Once the classes are defined, the initiateAcc and completeAcc methods must be implemented. They will allow the instructions to work in gem5's timing access mode, which will be necessary to interact with the memory response, in this case, the notification of an abort. The initiateAcc methods of the instructions will be responsible for initiating their corresponding actions through the ExecContext::initiateHtmCmd interface, passing the flags indicated on the slide as parameters. The completeAcc methods will execute as long as no abort has been detected. The BEGIN instruction will checkpoint the processor. The CANCEL instruction will be in charge of communicating the aborts with its return value. Finally, the COMMIT instruction will restart the checkpoint for subsequent transactions.

Once the instructions have been implemented, they must be associated with their encoding. In gem5, the keyword decode is used to indicate the start of a decode block, which will compare the value of the indicated field of the decoded instruction, with the values indicated in braces. These values can indicate the start of a new decode block to check the value of another field, or directly indicate the identification of an instruction, accompanying them with the class name of the instruction.

For the new instructions, 3 decode blocks will be needed. The first one will check the first two bits, which will be 11. The next one will check the remaining 5 bits of the opcode. Here we will have to add the value 2. To this new value, another decode block is associated that will check the value of the FUNCT3 field, with which the instructions can be identified. Inside the decode block, the format keyword is observed. The gem5 formats will help with the definition of the instruction classes, although in this case, we only use it to define the constructor call.

To evaluate the resulting implementation, the STAMP benchmarks will be used. They are 8 applications that will allow us to check the performance of the TM system in different domains. Before proceeding, a small modification must be made in STAMP to support our new TM system. Thanks to the definition of the library described above, we will only need to call these functions within the macros that STAMP uses in its benchmarks.

Once the modification is made, two studies will be carried out, one on scalability and the other on the effect of cache associativity. The first one will show how increasing the number of threads affects performance. With the second one, we'll see how associativity of L1 affects the capacity aborts, caused by the replacements of blocks that are in the read / write set. Throughout the studies, explicit and exception aborts also appear to show a more complete context, but they have relatively minor importance with respect to what will be demonstrated.

Each benchmark will have characteristics that will affect the performance of the system (Table 6.1). Contention indicates the probability of conflicts to appear, and the size of the read / write set will give a rough idea of the relevance of capacity aborts. Length of transaction shows how many instructions are executed in each transaction. Finally, the total time of the application that the transactional part represents indicates how it will affect the general performance of the application.

With the first study we can see how, in some benchmarks (Figure 6.1), the increase of the number of threads will provide a performance improvement only up to a certain point. In others, especially those with high contention, we will see no improvement at all. Benchmarks with low contention will show very positive performance improvements.

To better understand the reason for these variations, we have to take a look at the relation between aborts and commits (Figure 6.4). In the cases in which the total number of aborts exceed the commited transactions, or are very close to doing so, it coincides with the drop in performance. It is also observed that, with the increase in these aborts, the transactions executed with the fallback lock also increase, serializing the application. Why does this increase in aborts occur, and what causes the drop in performance?

Taking a closer look to the different types of aborts (Figure 6.2), we can observe how the increase in aborts coincides with the increase in conflict aborts with each increase in the number of threads.

With the second study, we can see how all benchmarks reduce capacity aborts when increasing associativity (Figure 6.5). It is also observed that there are variations in the number of conflict aborts. The main reason is that by reducing capacity aborts, more transactions will be executed without using the fallback lock. This can cause these to generate more conflicts, or to be reduced by establishing a certain order after the aborts.

Looking at the means of the sizes of read/write sets of aborted and commited transactions (Figure 6.6), most benchmarks can commit a greater number of transactions with the raise of the average, so the reduction in aborts due to capacity has allowed more transactions to be executed. In labyrinth, on the other side, we observe how the mean of aborts increases but not that of commits. This may indicate that the cache size is not large enough.

Although most benchmarks reduce aborts by capacity and commit more transactions, only two of the benchmarks are favored by the increase in associativity (Figure 6.7). Because we are using the small size variants, the generated read / write sets may not be excessively large for capacity aborts to be critical to performance. The only ones where it improves are precisely where conflict aborts decrease significantly.

As conclussions of this TFG, we can say that Transactional Memory is the alternative to locks to improve scalability. A completely functional Transactional Memory system has been defined for RISCV, with three new instructions being added to the ISA. It's been implemented in gem5, and evaluated with STAMP benchmarks. This evaluation has shown the relation between the number of threads and performance of the applications, and between associativity of L1 and capacity aborts.

Chapter 2

Introducción

Un sistema informático suele tener como elementos más relevantes la Unidad de Procesamiento Central (CPU) y la unidad de memoria. No hace demasiados años, las CPU estaban formadas por una única unidad de procesamiento (núcleo), encargada de leer las instrucciones, realizar la aritmética... Esto limitaba el número máximo de instrucciones ejecutándose en un mismo instante, lo que conllevaba un importante cuello de botella. La reducción del tamaño de los transistores con el paso de los años, permitió la aparición de un nuevo diseño de CPU denominado multinúcleo, disponiendo de varios núcleos interconectados entre sí en un mismo chip. En cada núcleo se ejecuta un conjunto de instrucciones denominado hilo. Estos hilos pueden, o bien ser procesos distintos, o por otra parte, ser capaces de acceder a una misma zona de memoria de forma concurrente. Estos hilos comparten el procesador con otros tantos, por lo que el cuando se van a ejecutar dependerá del Sistema Operativo sobre el que se ejecuten. De esta forma, no será posible determinar cuando se van a ejecutar ciertas instrucciones respecto a otras, pudiendo no corresponder con lo que el programador pretendía. Esta situación se denomina condición de carrera, por la cual los datos accedidos por un aplicación pueden acabar en un estado inconsistente.

Será necesario definir una sincronización entre hilos que permitirá al programador definir ese orden de ejecución en las secciones de la aplicación que considere. Cómo se consiga esta sincronización será fundamental, ya que una especificación errónea conllevará una penalización del rendimiento considerable, resultados incorrectos o una ejecución que nunca termine.

Una de las opciones más utilizadas para conseguir un acceso atómico a ciertos datos es utilizar los denominados locks. Estas estructuras están conformadas en su mayoría por instrucciones atómicas o LL/SC concretas de cada ISA. Estas instrucciones permiten que un único hilo acceda a cierto dato y que el resto aguarda a que finalice. Por lo tanto, podemos decir que el hilo que primero acceda al dato podrá continuar la ejecución con la seguridad de que ningún otro hilo se encuentra en la misma sección. Cuando termina, deberá liberarlo para permitir que otro lo pueda adquirir. En el momento en que este proceso de adquirir/liberar no se realice de forma correcta, el funcionamiento de la aplicación se verá comprometido. Aunque el concepto base de los locks resulta bastante simple, su aplicación en un entorno con muchas secciones críticas puede llegar a resultar complejo a nivel de código fuente, y poco eficiente.

Cada vez más servicios requieren de una mayor capacidad de cómputo, por lo que se suele incrementar el número de núcleos. Cuantos más núcleos se utilicen, mayor será el número de hilos ejecutables y por lo tanto mayores serán las posibilidades de intentar acceder de forma concurrente a una sección crítica. Esto provoca un incremento de tiempo invertido en el manejo de estas secciones y la posible serialización de la aplicación, generando la necesidad de optimizar este procedimiento. La opción de los locks puede resultar demasiado compleja en estos casos.

El modelo de Memoria Transaccional se propone como alternativa, consiguiendo simplificar el establecimiento de secciones críticas. Además, en situaciones complejas donde existan un gran número de secciones críticas, la aplicación de un mismo criterio para verificar la atomicidad permitirá obtener un rendimiento más uniforme. Su funcionamiento se basa en garantizar que un dato haya sido modificado únicamente por un hilo registrando si el dato ha sido accedido de forma conflictiva desde otro hilo. La definición de como se genera el conflicto y como solucionarlo determinará como funciona un Modelo de Memoria Transaccional. Para utilizarlo, únicamente habrá que indicar el inicio de la transacción y donde termina. El sistema determinara si la transacción puede completarse o no, liberando al programador de esta tarea.

Por otro lado, la arquitectura RISCV se propone como una alternativa a los ISAs más populares. Surge de las complicaciones de que en su mayoría sean opciones de pago, limitando la capacidad de creación y distribución de nuevas implementaciones. Dispone de licencia de libre distribución y ofrece gran flexibilidad con varias configuraciones en función del objetivo del chip.

Como objetivos de este trabajo, se mostrará la especificación diseñada para definir la extensión T del ISA RISCV, el procedimiento seguido para incluir el soporte en el simulador y se evaluarán los resultados obtenidos para comprobar el rendimiento de la configuración elegida. Se utilizará el simulador gem5 para poder implementar la propuesta y poder evaluarla. Nos proporcionará una simulación lo suficientemente detallada para obtener las conclusiones que se buscan.

Chapter 3

Estado del arte

3.1 Paralelismo

Aunque hoy en día todos los procesadores que podemos adquirir tienen al menos dos núcleos, no era lo habitual hace algunos años. Todas las ejecuciones se realizaban de forma secuencial, una instrucción detrás de otra. La solución más habitual para mejorar el rendimiento era aumentar la frecuencia. Esto producía que cada instrucción se ejecutará en menos tiempo, y por lo tanto, un mayor número de instrucciones se ejecutaban en un intervalo determinado. Sin embargo, el principal problema que hizo este método obsoleto es que tal como aumenta la frecuencia, también lo hace la potencia (insertar fórmula). Además de un mayor consumo de energía, también aumentaba la temperatura, pudiendo bajar el rendimiento del procesador, e incluso deteriorarlo hasta dejarlo inservible. Un evento concreto que marcó el final del escalado por frecuencia fue la cancelación de los modelos Tejas y Jayhawk de Intel en 2004 [1].

La solución propuesta para seguir aumentando el rendimiento y evitar estos problemas fue introducir múltiples núcleos. Esta estrategia permitirá dividir los problemas secuenciales en otros problemas más pequeños y asignar cada uno a un núcleo, permitiendo realizar los cálculos de forma paralela.

3.1.1 El problema de la paralelización

La posibilidad de que exista más de un hilo accediendo sobre los mismos datos, hace que aparezca un nuevo problema: la sincronización. El orden de ejecución de estos hilos no es a priori conocida por el programador. Esto puede provocar una situación donde, de forma paralela, un hilo lea un dato, le sume 1, y a su vez, otro hilo haga lo mismo. De esta forma, si la intención inicial era que se contabilizarán las dos sumas, solo se contabilizará una, lo que conllevará a un programa incorrecto. La posibilidad de que el orden de ejecución de las instrucciones no sea el inicialmente esperado se denomina condición de carrera. Al conjunto de instrucciones que pueden ejecutarse de forma concurrente y referencian a los mismos datos, pudiendo replicar problemas similares al mencionado, se le denomina sección crítica. Será tarea del programador establecer cuando un hilo puede proceder a acceder a un dato sin generar conflictos con otro de los hilos que acceda al mismo dato.

3.1.2 Uso de Locks

Uno de los métodos más habituales y que implementan la mayoría de lenguajes de programación, son los locks o mutex. Este mecanismo garantiza que un único hilo esté ejecutando una sección crítica en un determinado instante.

Suelen implementar dos acciones. La primera se denomina adquisición del lock, donde el hilo comprueba si la sección crítica está libre. La segunda consta de la liberación del lock. Una vez ejecutada la sección crítica, se indica al resto de hilos. En función de lo que haga el hilo al no conseguir adquirir el lock, se pueden definir dos implementaciones. Una posible implementación es un spinlock, el cual intenta constantemente adquirir el lock. Otra es el sleeplock, en el cual se fuerza un cambio de contexto si el hilo no consigue adquirir el lock y de esta forma no desaprovechar tiempo intentando adquirirlo. La primera opción es la más adecuada cuando el hilo consuma poco tiempo intentando adquirir el lock. La segunda opción conviene cuando las secciones sean lo suficientemente largas como para que un cambio de contexto se realice en menos tiempo. Para conseguir una implementación eficiente, es habitual el uso de operaciones atómicas y de instrucciones LL/SC, las cuales están presentes en la mayoría de ISAs.

Uno de los inconvenientes de este método es que si se utilizan de forma incorrecta, puede darse el caso de que la ejecución de un hilo puede bloquearse indefinidamente, produciéndose un deadlock. Ocurre cuando no se liberan los locks una vez ejecutada la sección crítica. Lo peor de todo es que, en un programa complejo, hasta la persona más experta puede encontrar estos problemas.

Además de estos problemas, hay que tener en cuenta que por cada lock que definamos habrá una variable más que manejar a nivel de código, y será necesaria más memoria. Sin embargo, si utilizamos menos de los necesarios, la probabilidad de que un hilo se bloquee será mucho mayor. Para cada aplicación será necesario determinar este equilibrio para maximizar el uso de todos los núcleos. Otro problemas comunes al utilizar locks es la inversión de prioridad. Si un hilo con prioridad baja adquiere el lock, el resto de hilos que lo intenten adquirir no podrán seguir ejecutándose hasta que el otro lo libere, lo cual conllevará más tiempo al no estar en CPU demasiado tiempo.

También se puede producir el efecto convoy, en el que un hilo que tenga adquirido el lock lo libere más tarde de lo normal, por diversas razones, como por ejemplo, esperar a que se resuelva un fallo de pagina. Todos estos problemas deberán ser resueltos por el propio programador, teniendo que invertir tiempo en estas cuestiones en vez del núcleo de la aplicación en si. Debido a esto, es habitual que resulte más complejo utilizar locks que la solución propuesta a continuación.

3.1.3 Memoria Transaccional (MT)

El aumento de la necesidad de utilizar aplicaciones que se ejecuten en paralelo, y la complejidad que puede implicar hace necesario la introducción de nuevos métodos que simplifiquen el proceso. El modelo de Memoria Transaccional [2] propone comprobar de forma dinámica que un conjunto de operaciones se ejecute de forma atómica por un procesador, garantizando que cada conjunto se ejecute de forma secuencial determinando si determinados accesos son incompatibles con la atomicidad. Todo este proceso se oculta al programador.

La unidad de organización del modelo de Memoria Transaccional es la transacción. Está formada por un inicio y un final de transacción, los cuáles delimitan que instrucciones se deben ejecutar de forma atómica. Una vez iniciada una transacción, se verifica que no se ejecuten instrucciones que violen la atomicidad. Si durante la transacción no ha habido problemas, acabará por confirmarse, indicando que las instrucciones se han ejecutado atómicamente. Si ha habido algún conflicto, se deberán tomar medidas para seguir garantizando la atomicidad.

Un sistema de Memoria Transaccional debe garantizar las siguientes tres propiedades. La atomicidad por fallo garantiza que los cambios generados por una transacción abortada no sean visibles por el resto de

hilos y solo sean visibles por una transacción completada. La consistencia garantiza que todos los recursos accedidos por cualquier hilo hayan sido modificados por transacciones confirmadas. De esta forma, si dos transacciones pretenden incrementar una variable, primero deberá confirmarse una y después la otra, ya que de lo contrario alguna de las sumas podrá no contabilizar. Por último, el aislamiento establece que las escrituras realizadas en una transacción no podrán ser visibles para el resto de hilos hasta confirmarse.

No todos entornos donde se utiliza MT son idénticos. Dependiendo de en que aplicación se ejecute, resulta interesante disponer de distintas alternativas para implementar el manejo de las transacciones. Una elección adecuada al caso concreto permitirá una mejora en el rendimiento.

Dependiendo de como se indique que accesos pertenecen a una transacción, el sistema puede ser explícito o implícito. La primera opción requiere indicar en cada acceso a memoria que queremos realizar el acceso de forma atómica. Esto implica un mayor esfuerzo a la hora de programar, pero permite una mayor flexibilidad y mayor legibilidad. La segunda opción supone que todos los accesos a memoria dentro de una transacción van a ser atómicos, por lo que simplifica el proceso.

Un sistema de Memoria Transaccional concreto se caracterizará también por como define tres conceptos esenciales. Primero, será necesario definir que es un conflicto. Otro concepto a definir es cuando se detecta el conflicto, durante la transacción o al final. Por último, se deberá definir una acción que permita resolver el conflicto y continuar la ejecución.

Un conflicto define cuando una transacción interfiere con otra, realizando una secuencia operaciones incompatibles sobre el mismo recurso. La definición más habitual es que se produzca un acceso sobre un recurso escrito en una transacción, o una escritura sobre un recurso leído. Por ejemplo, si leemos A en dos transacciones T1 y T2, y la modificamos en T1, se generará un conflicto. Si no se marcara como conflicto, T2 tendría un valor distinto al que correspondería si quisiéramos ejecutar T1 y T2 de forma secuencial, infringiendo la atomicidad.

Una vez sabemos qué es un conflicto, es necesario detectarlo y proceder a su resolución. En función de cuando pretendamos hacerlo, existen dos métodos principales. Si optamos por un enfoque ansioso, el conflicto se detectará y resolverá en el momento en que ocurra. Esto tendrá la ventaja de poder evitar abortos aplicando otras políticas de resolución. Sin embargo, es necesario comprobar si hay conflicto en cada acceso a memoria, y en el peor caso, podría ocurrir un livelock en el que una transacción interfiere con la otra sin poder ninguna confirmar. Habrá que ofrecer soluciones para evitarlo, como introducir un retraso en el inicio de una de las transacciones.

Si optamos por un enfoque perezoso, la detección y resolución del conflicto pueden ocurrir después de que tenga lugar, siendo habitual hacerlo en la fase de confirmación. Esto reduce el número de veces que hay que buscar un conflicto y evita los livelocks, al haber siempre una transacción que podrá confirmar. El principal inconveniente de este método es que la penalización por el reiniciar la transacción suele ser mayor, al ocurrir la detección más tarde.

En general, si los conflictos son frecuentes, la opción ansiosa resulta la alternativa adecuada, al evitar ejecutar continuamente instrucciones que no van actualizar el estado. Si son poco frecuentes, la opción perezosa evita tener que comprobar de forma continua si ha habido un conflicto.

Como hemos dicho anteriormente, las transacciones son una forma de ejecución especulativa, es decir, es posible que los cambios realizados por sus instrucciones deban deshacerse. Por un lado, hay que tener en cuenta durante una transacción se van a modificar los registros del procesador. Si al final no se confirma, debemos ser capaces de recuperar los valores previos. La metodología no varía mucho, siendo habitual copiar los valores en registros auxiliares o en memoria.

Por otro lado, durante una transacción también se modifican datos en memoria, por lo que habrá que ofrecer soporte para guardar copias de los valores modificados. Este proceso se le denomina versioning, o control de versiones. Se suelen aplicar dos aproximaciones.

La primera se denomina ansiosa, o eager, y procede realizando las escrituras directamente en memoria. Para no sobrescribir el valor previo y poder encontrarnos en una situación inconsistente por lo que los valores previos se mantienen en un buffer independiente. Si se aborta la transacción, se copian los valores de este buffer en memoria. La otra opción es la perezosa, o lazy, y realiza las escrituras en un buffer independiente, en vez de directamente en memoria. De esta forma, los accesos a estas direcciones se harán sobre el buffer. Si se confirma la transacción, se escriben estos valores en memoria. Si utilizamos un control eager, una escritura sobre el dato X implica en realidad dos escrituras, una a memoria con el nuevo valor y otra al log con el valor previo. Si la transacción se confirma, únicamente tenemos que descartar el log, la cual suele ser una operación rápida. En el caso de que la transacción aborte, tendremos que leer el log el valor previo de X y escribirlo en memoria.

Si por el contrario, nos decidimos por un control perezoso, una escritura sobre X implicará únicamente una escritura en el log. En el caso de confirmarse, tendremos que leer el valor de X del log y escribirlo en memoria. En el caso de abortar la transacción, únicamente habrá que invalidar el contenido del log, quedando X con el valor inicial.

Como se puede observar, cada opción tiene sus ventajas e inconvenientes. Escoger una u otra dependerá de una aproximación general de como se van a comportar las transacciones. Si sabemos que es posible que el total de transacciones abortadas será similar al de transacciones iniciadas, lo correcto serie optar por la opción lazy. Si por el contrario, sabemos que van a abortar pocas, eager sería una mejor opción.

Sin embargo, estas condiciones pueden resultar imposibles de determinar. Como ya veremos en siguientes secciones, se suele optar por implementar la opción lazy, al requerir en principio un menor número de operaciones.

También será necesario establecer cual va a ser la unidad utilizada para la detección del conflicto, también denominada granularidad. Si las acciones conflictivas se realizan sobre una dirección que esté en una de estas unidades, el conflicto afectará a todos los elementos contenidos en la unidad. Esto provoca que accesos a direcciones que previamente no han sido accedidas en otra transacción, provoquen un conflicto. Si la unidad es pequeña, se obtendrá una mejor precisión a la hora de determinar los conflictos, pero habitualmente conlleva mayor complejidad de diseño. Si es demasiado grande, se aumentará el número de conflictos de forma innecesaria. Por lo tanto, resulta adecuado establecer una granularidad intermedia adecuada a cada sistema.

Cada una de estas unidades pertenecerá a los denominados Read/Write (R/W) sets, o conjuntos de lectura/escritura. Si un acceso a memoria realiza una lectura, la unidad se añadirá al read set. Si realiza una escritura, se añadirá al write set. De esta forma, podemos determinar a lo largo de una transacción cómo se ha referenciado la unidad. Así podremos determinar si un acceso a una de las unidades es compatible en el contexto de una transacción activa.

Un sistema de Memoria Transaccional puede implementarse por software(STM), o por hadware(HTM). STM ([3],[4]) ofrece mayor flexibilidad a costa de rendimiento. Si la ejecución de secciones críticas implica un porcentaje relevante de una aplicación, HTM permitirá minimizar la penalización del mantenimiento de estas secciones.

3.1.4 Hardware Transactional Memory

En 1986, Tom Knight [5] propuso un método para facilitar y mejorar las ejecuciones paralelas del lenguaje Lisp. Es la primera referencia al control de la atomicidad en hardware. Años más tarde, en 1993, Herlihy, Maurice y Moss, J [6] definieron por primera vez el concepto de Memoria Transaccional. Su trabajo ha sido la base para todos los sistemas de Memoria Transaccional existentes.

Sin embargo, la complejidad que añadía al diseño del procesador hizo que los principales fabricantes tardarán en añadir soporte para Memoria Transaccional en sus procesadores. Hubo diversas implementaciones, pero

ocultas al programador. No fue hasta 2009 cuando Sun Microsystems añadiera este soporte a sus procesadores. Aunque el proyecto se canceló y no llego a ponerse a la venta, diversos prototipos fueron entregados a diversos investigadores. Aún en la actualidad, no existen demasiadas alternativas comerciales, siendo las más destacas ARM Transactional Memory Extension y soporte de Memoria Transaccional de IBM.

La razón de ser de HTM es optimizar el mantenimiento de las transacciones haciendo uso de un espacio reservado en el procesador y del aprovechamiento de otras estructuras existentes del hardware. Los objetivos son mejorar el tiempo de ejecución, reducir el consumo de energía y reducir la complejidad de uso al ofrecer el servicio a través del ISA.

Será necesaria la inclusión de nuevas instrucciones. Si se opta por una solución implícita, será necesario introducir dos nuevas instrucciones que establezcan el inicio y final de una transacción. Si por el contrario, se utiliza una solución explícita, además de estas dos instrucciones, habrá que añadir una variante transaccional de las lecturas/escrituras a memoria. De esta forma, solo los accesos realizados con estas nuevas instrucciones formarán parte de la transacción.

Como mencionábamos, una de las claves que han facilitado la aparición de procesadores con soporte HTM ha sido la reutilización del hardware existente. Una de las opciones más utilizadas es apoyarse en el sistema de cachés. Estas memorias están divididas en unidades denominadas bloques de caché. Cada uno de estos bloques contiene un conjunto de datos de memoria, lo que implica que cuando se accede a alguno de estos datos, se referencia al bloque entero. Podemos ampliar el hardware de cada bloque añadiendo bits de pertenencia al conjunto de lectura/escritura y establecer la granularidad del sistema de Memoria Transaccional a esos bloques. Cuando se produzca una lectura sobre un bloque, se añadirá al read set. Si se produce una escritura, se incluirá en el write set. Esto únicamente implica poner el bit de pertenencia a 1. Cuando una transacción aborte o se confirme, los conjuntos de lectura/escritura se reiniciarán al poner los bits a 0, operación con muy bajo coste de tiempo y energía.

También es habitual utilizar las cachés de datos como buffers de escritura especulativa. Antes de modificar un bloque durante una transacción, se realiza un Writeback, es decir, se copia el bloque actual en otra caché adyacente. De esta forma, podemos mantener el valor previo a la transacción y el nuevo. Si la transacción aborta, se invalidarán los bloques modificados, provocando que el valor propagado previamente tenga que ser leído, manteniendo la atomicidad.

Para detectar los conflictos, se puede utilizar el protocolo de coherencia . Los cambios de estado implican una serie de acciones que se pueden usar para identificar si se está violando la atomicidad. Por ejemplo, en un protocolo MESI, cualquier escritura de un hilo distinto sobre un bloque en estado M, E o S, y que haya sido añadido al read set previamente, resulta en un conflicto de memoria. También se detectaría como conflicto si se realiza una lectura desde otro hilo sobre un bloque que pertenezca al write set y que esté en estado M. En la figura 3.1 se muestra la inclusión de la detección de conflictos en el protocolo MESI:

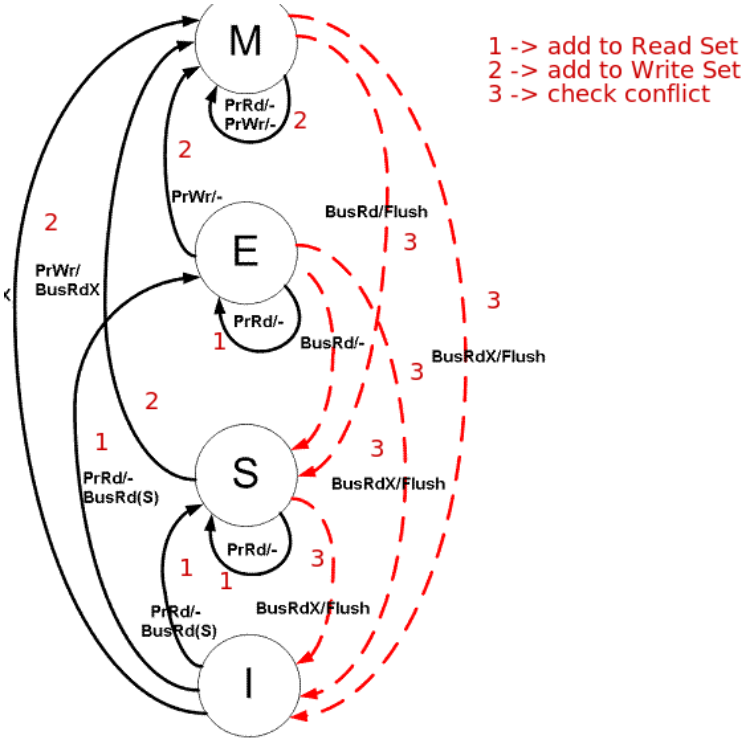

Figure 3.1: Ejemplo integración detección de conflictos en MESI

Por último, es habitual utilizar los registros adicionales para el reordenamiento en procesadores fuera de orden con el objetivo de realizar un checkpoint del procesador. También pueden pueden incluirse nuevos registros exclusivamente para esta tarea, pero no podrán utilizarse en aplicaciones que no utilicen HTM.

Actualmente no existen muchas alternativas comerciales. Resulta complicado definir un sistema de Memoria Transaccional que sea eficiente en la gran variedad de aplicaciones que existen. Además, hay que tener en cuenta que hay otras características a mejorar y problemas por resolver por parte de las empresas antes que emprender la inclusión de nuevas características.

En 2009, SUN Microsystems fue la primera empresa en introducir soporte de memoria transaccional, denominado Rock HTM [7], a sus procesadores de la familia Rock, basados en el ISA SPARC v9.El proyecto fue cancelado en 2010. Años más tarde, IBM propone en 2011 su primer sistema HTM para sus servidores Blue Gene/Q [8]. Más tarde, en 2014, IBM propone un nuevo modelo de Memoria Transaccional [9] para el ISAs POWER 8 y que se seguirá usando en POWER 9. En POWER 10 ya no se incluye esta característica. En 2013 fue Intel quien propuso TSX [10] para introducir un modelo de memoria transaccional en sus procesadores. Apareció por primera vez en la familia Haswell. No tuvo mucho recorrido ya que se relacionó con un fallo de seguridad [11]. Por último, en 2019 ARM anunció su modelo de memoria transaccional denominado Transactional Memory Extension(TME) [12], incluido en el ISA Armv9. Cabe destacar, que AMD también tiene una propuesta denominada Advanced Synchronization Facility (ASF) [13], pero todavía está en fase de especificación. Se detallará el funcionamiento de cada sistema en el capítulo 4.

Una de las características que comparten es que no garantizan la confirmación de una transacción, siguiendo un modelo *best effort*. Uno de los casos más habituales es el aborto por la limitación del tamaño máximo del conjunto de lectura/escritura. Es posible que una transacción genere un conjunto de lectura/escritura tal que no es posible mantenerlo. En este caso, se puede determinar que la transacción nunca podrá finalizar por el camino habitual. Otro caso que puede darse al aplicar una detección de conflictos ansiosa, es que un conjunto de transacciones se aborte constantemente, evitando el progreso de las mismas. El modelo *best effort* delegará al software la tarea de como reiniciar la transacción, pudiendo responder de forma flexible

a cada una de estas causas. Una de las soluciones más habituales es usar un lock global para garantizar la ejecución atómica. Es una solución sencilla, pero en determinados casos poco eficiente. Otra solución posible es utilizar un sistema de STM [14].

Cabe destacar que de las mencionadas, únicamente perdura la implementación de ARM. Esto refleja la complejidad de definir e implementar un sistema de Memoria Transaccional por hardware funcional y eficiente.

Como hemos podido observar en esta sección y la anterior, aunque el objetivo de los locks y de la memoria transaccional son los mismos, garantizar el acceso atómico a una sección crítica, lo consiguen de formas muy diferentes. Con locks se garantiza que ciertas instrucciones se ejecuten una única vez, es decir, la ejecución implica que se ha conseguido garantizar la atomicidad. En memoria transaccional, las instrucciones se ejecutan aunque no se pueda garantizar atomicidad. El acceso atómico solo se garantiza una vez se comprueba que no ha habido ningún conflicto.

Por lo tanto, si sabemos que cierta sección crítica va a generar un número considerable de conflictos, podría ser conveniente utilizar locks. Si no está determinado, puede resultar conveniente el uso de Memoria Transaccional al ser bastante más sencillo de utilizar.

3.2 RISCV

Un Instruction Set Arquitecture(ISA) es la primera abstracción que existe entre el software y el hardware. A través de las llamadas instrucciones, define la funcionalidad de la dispondrá una unidad Central de Procesamiento(CPU), así como el número de registros, como se accede a memoria, entrada/salida a dispositivo externos, y aritmética, entre otros muchos.

Los ISAs pueden clasificarse según diversos criterios. Según la complejidad, un ISA puede pertenecer al tipo Complex Instruction Set Computer (CISC) si dispone de un gran número de instrucciones especializadas, o del tipo Reduced Instruction Set Computer (RISC) si dispone de pocas instrucciones. En CISC, aunque resulta interesante tener una instrucción optimizada para cada acción, la mayoría de ellas no serán utilizadas por el compilador, por lo que estaremos desaprovechando sus capacidades. En RISC, aunque el número bajo de instrucciones simplifique el diseño de la CPU, las aplicaciones generarán un mayor número de instrucciones al tener que componer ciertas acciones a través de otras más simples. Cual es el equilibrio optimo entre especialización y simplificación todavía está por determinar.

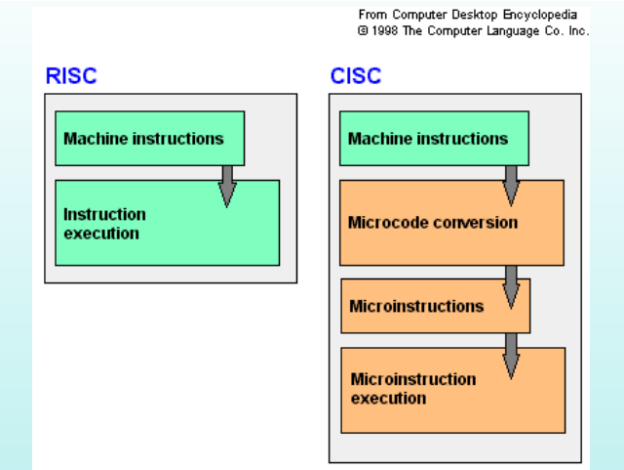

Figure 3.2: Comparativa del proceso de decodificación entre RISC y CISC

En 2010, Yunsup Lee, Krste Asanovic, David Patterson, y Andrew Shell concibieron el ISA RISCV [15], basado en RISC, con el objetivo de apoyar la investigación y educación en la universidad de Berkeley. A lo largo de los años, muchas otras empresas [16] [17] también han empezado a colaborar en el proyecto. En la figura 3.3 se muestra una cronología del desarrollo de RISCV. Ninguna de los ISAs disponibles entonces se ajustaba a sus necesidades, por lo que se propusieron diseña la suya propia, que además de cumplir con sus requisitos, tuviera la flexibilidad de cumplir con los de un mayor número de personas posibles, y evitando así los problemas que ellos sufrieron y que les llevaron a este punto.

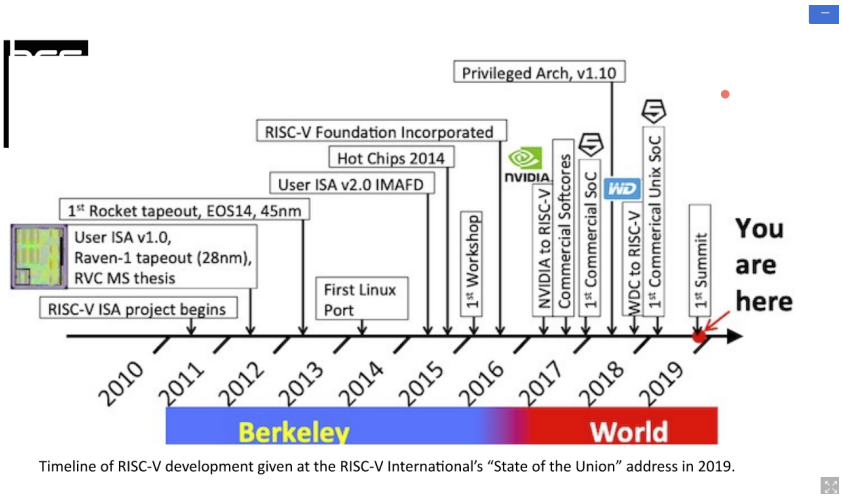

Figure 3.3: Cronología del proyecto RISCV

Uno de los principales motivos de la creación de RISCV por parte de sus autores, fue que los ISAs más populares eran propietarias de empresas cuyo objetivo es maximizar el beneficio obtenido de su producto. Por lo tanto, no se podían permitir que terceras personas, como los investigadores, crearan un nuevo producto a partir del suyo propio y no obtener beneficio. El segundo de los motivos era la gran complejidad de estos ISAs. Disponían de un hardware complejo que dificultaba posibles implementaciones basadas en este, y que a su vez podía implicar una reducción de rendimiento al abarcar demasiado.

Para que estos problemas no estén presentes en RISCV, uno de los objetivos es conseguir que las instrucciones no sean demasiado especificas y que permitan al fabricante mayor libertad al diseñar el chip. RISCV es que está conformado por distintos módulos, denominados extensiones. A más extensiones incluyamos, mayor será la funcionalidad e incluso el rendimiento de la aplicación. Sin embargo, será más complejo de fabricar. Hay extensiones para operaciones enteras, de punto flotante, de 32 y 64 bits... El estado de Memoria Transaccional Hardware en RISCV únicamente está contemplada con la extensión T, y no existe todavía ninguna especificación. Uno de los objetivos de este trabajo será ofrecer una posible especificación para esta extensión. Además, todo el proyecto será de libre distribución, lo que permitirá a cualquiera utilizar el ISA y comercializar cualquier producto basado en él.

En RISCV una instrucción podrá estar formada por distintos campos, cada uno conteniendo una información. Según cómo se organicen estos campos, se definen una serie de formatos de instrucciones, cada uno adecuado para según que parámetros requiera la instrucción. La figura 3.4 resume los cuatro principales.

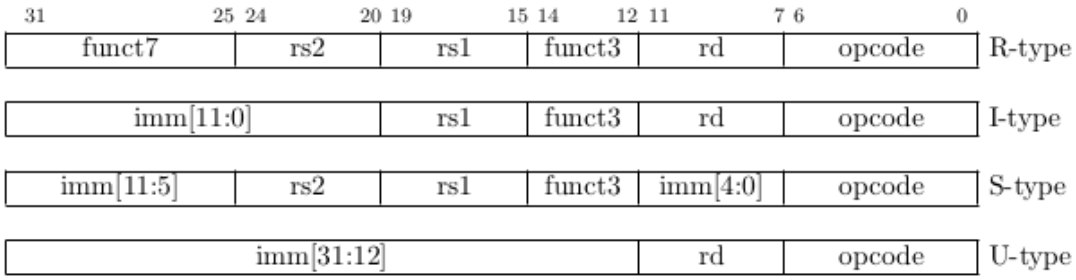

Figure 3.4: Formatos de instrucciones de RISCV

Se puede observar algunas de las características que hereda de RISC, como un opcode de tamaño fijo para cualquier instrucción, inclusión de datos en la propia instrucción y referencia homogénea a los registros desde las instrucciones.

Como todos los proyectos de código abierto, cualquier persona puede contribuir al desarrollo de RISCV. A través de su repositorio en GitHub [18], cualquiera puede realizar modificaciones y publicarlas. Si se prefiere trabajar de forma más coordinada, existe la opción de integrarse en uno de los grupos de trabajo existentes donde se te asignarán tareas que facilitarán el desarrollo de esta arquitectura. La RISCV Foundation se encarga de establecer estos grupos.

3.3 Alternativas de simulación

En este trabajo, la etapa de implementación no implicará de forma directa la construcción de un nuevo chip. Esto conllevaría un gran desembolso económico y un tiempo del que no disponemos. Por ello, se utilizará una herramienta que nos facilite el proceso, permitiéndonos obtener detalles del proceso de ejecución que nos interese y reutilizar determinado hardware del que dispongamos para realizar la ejecución.

A la hora de implementar y verificar una solución, siempre se dispone gran cantidad de alternativas a la hora de simular la arquitectura RISCV. Es fundamental escoger el entorno de simulación adecuado, ya que se podría malgastar tiempo en implementar ciertas funciones que no estén relacionados de forma directa con el trabajo, y que otra opción ya lo tenga implementado. La opción más habitual es utilizar un simulador software. Resultan más sencillos de utilizar y son más variados.

3.3.1 Simulador

Un simulador imitará la ejecución de un programa con unos parámetros que representen el entorno a simular sobre otro ordenador totalmente distinto. Una de sus principales ventajas es la flexibilidad que ofrece poder evaluar un gran número de configuraciones hardware sin disponer de ellos físicamente. Además, suelen ofrecer una serie de estadísticas de la ejecución y permiten manejar los errores de ejecución de forma más dinámica.

Al ser la simulación uno de los pilares fundamentales de la investigación, tanto en informática como en muchas otras especialidades, hay disponibles una gran variedad de alternativas a la hora de elegir un simulador. Cada uno tendrá sus ventajas y sus inconvenientes:

- QEMU : uno de los emuladores más conocidos. Tiene soporte para una gran cantidad de ISAs, incluido RISCV. Su objetivo es realizar las simulaciones en el menor tiempo posible, por lo que no ofrece soporte

para estadísticas. Tiene un gran número de opciones en cuanto a como ejecutar el simulador en si, pero no demasiadas en cuanto a como ejecutar una simulación.

- GEM5 : ofrece distintos modos de ejecución que varían entre eficiencia y nivel de detalle. Ofrece una gran cantidad de estadísticas, por lo que el rendimiento es menor.
- SPIKE :se centra en ofrecer una implementación funcional y eficiente, y no en el nivel de detalle.
- Ripes : focalizado en visualizar el funcionamiento de la arquitectura, tanto a nivel de instrucción como de componentes. más orientado al entorno educativo.
- TinyEMU : incluye lo mínimo necesario para una ejecución eficiente y correcta. Podría verse como una versión reducida de QEMU.
- MARSS-RISCV : basado en TinyEMU, añade un mayor nivel de detalle. Se centra sobre todo a nivel de CPU. A nivel de memoria, es detalla pero poco flexible.
- Web-Based : ofrece una portabilidad interesante para el sistema educativo, pero tiene una funcionalidad limitada.

3.3.2 FPGA y Hardware Description Language(HDL)

Un HDL define la estructura de un circuito electrónico a través de un lenguaje de programación. Las plataformas más utilizadas son las Field-programmable gate array, o FPGA. Conlleva una complejidad mayor que un lenguaje de programación software. Ofrece un mejor rendimiento que los simuladores software, pero dificulta la recolección de estadísticas.

Los HDL más utilizados son:

- Chisel : suele ser la opción favorita para manejar diseños complejos, ya que se cimenta sobre un Domain Specific Language(DSL) que simplifica la construcción de un diseño. Sigue un modelo orientado a objetos y funcional. Permite exportar el diseño a Verilog. Se ha utilizado para realizar una implementación de RISCV, el procesador Rocket [19].
- VHDL/Verilog : son los C/C++ de los HDL. Gran parte de la información referente a los HDL hace referencia a estos dos. Ofrecen un nivel bajo de abstracción, pudiendo optimizar cada detalle del diseño.

3.3.3 Elección de una plataforma de simulación

En la figura 3.5 se muestra un resumen de las principales características entre las opciones más utilizadas.

Feature	gem5	Chisel	spike	QEMU
Binary translation	✔		✔	✔
Checkpoints	✔			✔
Multicore simulation	✔[3]	✔	✔	✔
Performance statistics	✔	✔[4]		
RTL imulation		✔		
System call emulation	✔	✔[5]	✔[5]	✔
ASIC synthesis		✔		
FPGA tools		✔		
Phase analysis	✔			
Stats-based tools	✔	✔[4]		

Figure 3.5: Resumen funcionalidad

La mayoría de las opciones se centran en una simulación funcional y eficiente. Esta ultima característica, aunque positiva en un sistema de simulación, sustituye una característica que nos interesa aún más: el nivel de detalle. En esto, gem5 resulta ser el más adecuado. gem5 es un simulador tanto de la arquitectura de un procesador como de su interacción con el sistema de memoria. Su diseño modular y su alto nivel de configuración aporta una gran flexibilidad a la hora de ejecutar las simulaciones. Soporta los ISAs más utilizados y ofrece también la capacidad de modelar el consumo de energía y potencia. Ofrece estadísticas de prácticamente todos los aspectos relevantes del sistema simulado.

3.4 gem5

En 2011 se publicó la primera versión de GEM5 [20]. Permitió unir en una misma interfaz las principales características de dos simuladores. Por un lado, M5 aporta una gran flexibilidad a la hora de configurar las simulaciones, soporta distintos ISAs y distintos formas de simular una CPU. Por otro, GEMS aporta un sistema de memoria muy detallado y flexible. Muchas empresas y universidades, como ARM, MIT y HP, han contribuido al desarrollo de este proyecto. Esto, junto con el hecho de que se ha sido utilizado en un gran número de artículos de investigación, demuestran el alto nivel de utilidad que ofrece gem5. El objetivo principal del simulador, que a su vez es el principal atractivo, es la flexibilidad y la variedad de opciones de configuración de las que dispone.

La figura 3.6 muestra los módulos principales de gem5:

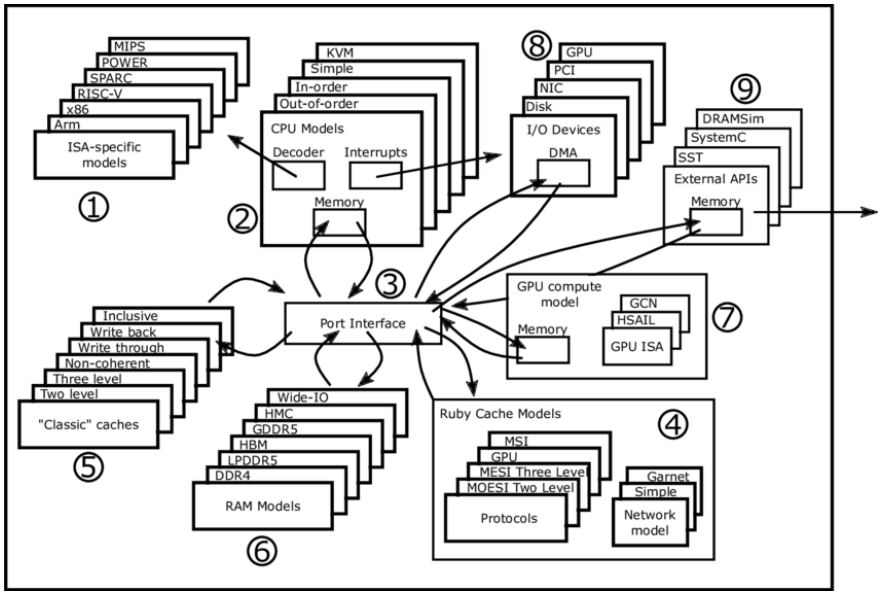

Figure 3.6: Módulos de gem5

3.4.1 Opciones de simulación

Como hemos dicho, una las grandes cualidades de GEM5 es la gran variedad de configuraciones posibles. Gracias a que GEM5 es fruto de la unión de dos simuladores, se pueden utilizar las opciones que ofrecen ambos de forma uniforme.

La primera variante que ofrece gem5 a la hora de configurar la simulación, es escoger el modo de simulación. El modo *Syscall Emulation*(SE) nos permitirá exclusivamente simular un binario concreto. Las llamadas al sistema se ejecutarán como si estuviéramos ejecutando el programa sobre Linux. Esta simplificación nos permitirá una simulación más sencilla y obtener tiempos de simulación menores. Si por otro lado nos interesa simular un sistema completo, podemos usar el modo *Full System*(FS). Su objetivo es simular el mayor número de elementos posibles dentro de un sistema informático habitual. Se simulará el kernel, lo que implica que las llamadas al sistema se simularán instrucción a instrucción. También habrá que preparar una imagen de disco, sobre la cual se realizarán las lecturas/escrituras al dispositivo de almacenamiento persistente. Todo este detalle implicará un despliegue más complejo, al tener que preparar la imagen y encontrar un kernel que se pueda simular correctamente. Además, el tiempo de simulación será mayor que *Syscall Emulation*. Para este trabajo, usaremos *Syscall Emulation*, ya que para evaluar el sistema de Memoria Transaccional propuesto no es necesario la simulación de todo el sistema.

También deberemos escoger el modelo de CPU a simular. Si por un lado, nos interesa evaluar el comportamiento de una CPU *out-of-order*, deberemos escoger el modelo *O3CPU*, el cual nos dará detalles de cada etapa del pipeline. En caso contrario, es recomendable escoger entre el modelo *AtomicSimpleCPU* y el modelo *TimingSimpleCPU*. Ambos implementan un procesador *in-order*, con la diferencia de que cada uno utiliza un modo de acceso a memoria distinto. El modo *atomic* (AtomicSimpleCPU) permite un flujo unidireccional desde CPU a Memoria, impidiendo que Memoria pueda generar otra petición en respuesta. Ofrecerá un mejor rendimiento, pero no será viable si nos interesa recibir respuesta. La otra opción, el modo *timing*, si que permitirá recibir una respuesta por parte de Memoria a una petición realizada por la CPU. Además, intenta simular de la forma más realista posible los conflictos que pueden surgir en memoria, ofreciendo un mayor realismo. El rendimiento será peor que el modo *atomic*, pero permite una comunicación bidireccional entre CPU y Memoria necesarios en algunos casos. Los modos de acceso a memoria también afectarán a los modelos de memoria Classic y Ruby. Mientras que en modo *atomic* únicamente se aplicarán los cambios que provocan las instrucciones de acceso a memoria, en modo *timing* tendrán que generar una respuesta concreta a la petición entrante, y así aprovechar la comunicación bidireccional.

Para este trabajo, se utilizará el modelo de *TimingSimpleCPU*, ya que debemos de ser capaces de recibir una respuesta de memoria que nos indique el estado de la transacción y además no necesitamos de un alto nivel de detalle en la simulación de la instrucción.

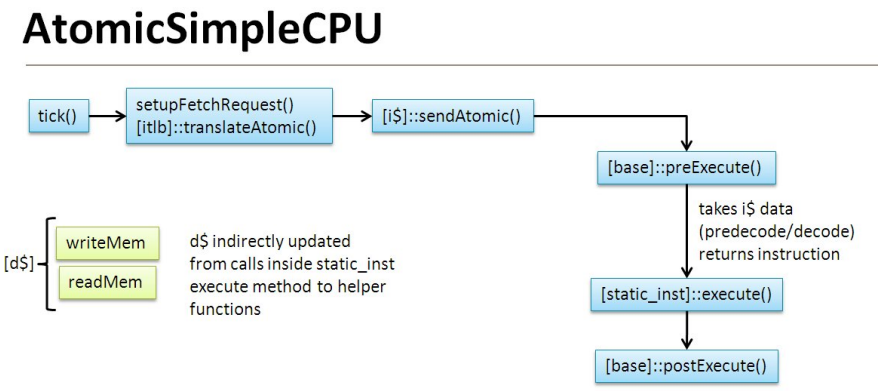

Figure 3.7: Proceso seguido por una instrucción en AtomicSimpleCPU

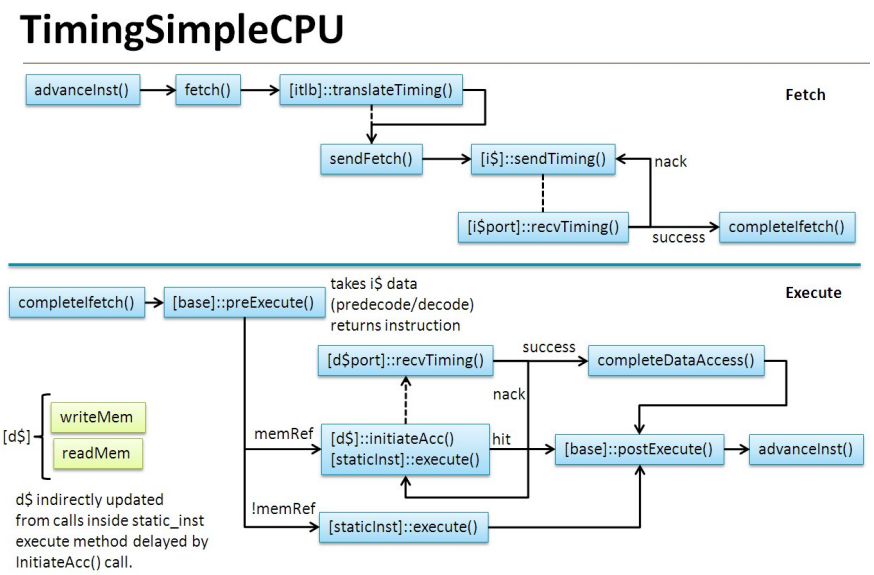

Figure 3.8: Proceso seguido por una instrucción en TimingSimpleCPU

Comparando las figuras 3.7 y 3.8, el modelo *TimingSimpleCPU* aplica procedimientos distintos entre accesos a memoria y el resto de instrucciones. Si la instrucción genera un acceso a memoria, el acceso de desglosa en distintas etapas que permiten proceder a la CPU en función de la respuesta generada por memoria.

La tabla 3.9 resume las opciones de simulación y su nivel de rendimiento y detalle:

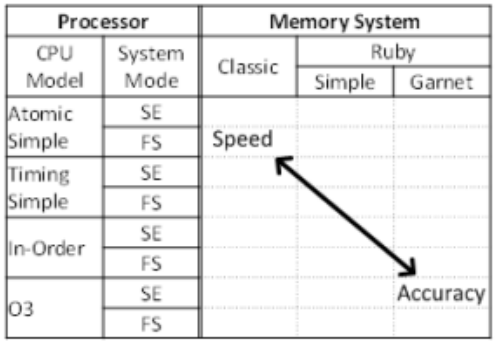

Figure 3.9: Opciones Simulación

gem5 dispone de dos modelos de simulación del sistema de cachés:

- *Classic model*: este modelo implementa MOESI snooping. Se recomienda usar este modelo cuando el funcionamiento del protocolo de coherencia no es objetivo de investigación, y a que adaptación de este modelo a otros protocolos resulta bastante costosa. Además, no soporta estados transitorios, por lo que la modelación de la contención del protocolo no es tan detallada. Permite cierta flexibilidad al soportar la especificación de la jerarquía de cachés en los scripts Python. Este modelo soporta tanto accesos atómicos como timing. Los primeros son más rápidos de simular que los segundos. Además, estos últimos son también más rápidos que los correspondientes con el otro modelo. En el modo FullSystem es habitual utilizar este modelo con accesos atómicos para iniciar el kernel, y después cambiar a un modo más detallado si procede.

	Classic	Ruby
Acceso atómico	Si	No
Acceso Timing	Si	Si
Flex. Coherencia	No	Si
Flex. Jerarquía Caches	Si	No

Table 3.1: Tabla comparativa entre Classic y Ruby

- *Ruby model*: esta opción es la más utilizada por su nivel de detalle a la hora de simular el sistema de memoria. Dispone de distintos módulos, como por ejemplo políticas de remplazo, protocolos de coherencia, redes de interconexión... Cada uno de los módulos es independiente y se comunican el resto a través de una interfaz común. Esto facilita la depuración y evita consumir tiempo en modificar la comunicación entre módulos. También disponen de un gran número de opciones de configuración. Además, permite modificar determinados módulos sin modificar directamente el código fuente, a través de Domain Specific Language (DSL).

Sin embargo, la simulación detallada conlleva un costo de tiempo. No soporta accesos atómicos, por lo que no hay opción de acelerar la simulación de los accesos. Además, aunque *SLICC* ofrece una gran flexibilidad al modelar los protocolos, el código generado está optimizado para una jerarquía muy concreta, por lo que resultará complicado añadir otro nivel de caché, siendo necesario un trabajo similar que al modificar el protocolo en el modelo Classic.

Para este trabajo se utilizará Ruby como modelo de memoria. Ofrece mayor nivel de detalle y, gracias a SLICC, nos facilitará tanto la comprensión como la edición del protocolo de coherencia utilizado.

En la tabla 3.1, se muestra un resumen de las características de cada modelo de memoria.

3.4.2 ISA en gem5

Para simular una aplicación, es necesario simular cada instrucción. Se debe poder identificar cada una inequívocamente y ejecutar los cambios que especifique el ISA. Para cada ISA, se deberá indicar qué instrucciones incluye y qué funcionalidad implementa cada una. gem5 divide este proceso en dos secciones. Por un lado, define un DSL que facilite la inclusión de nuevas instrucciones. Constará de varios *decode blocks* anidados. Cada uno especifica que campo de la instrucción se debe decodificar y la acción a llevar a cabo en función del valor de estos campos. Una acción sería proceder a decodificar otro campo introduciendo otro *decode block*. Otra acción será la identificación de una instrucción. A través de los llamados formatos, se facilitará la declaración de las clases de cada instrucción. Los formatos simplificarán el proceso de especificar los constructores y la implementación de las funciones. Permiten aplicar unas plantillas las cuales se podrán reutilizar entre instrucciones similares. Por ejemplo, las instrucciones suma y resta son ambas aritméticas, por lo que tendrán constructores idénticos y definen las mismas funciones. Únicamente variarán estas implementaciones. Podemos aplicar un mismo formato cambiando únicamente el parámetro que indica el contenido de las funciones. A partir de este DSL, se generarán los ficheros C++ que contengan las declaraciones de las clases con sus constructores y funciones, además de la función que se encargará de identificar las instrucciones.

Una vez identificada una instrucción, se ejecutan los métodos *execute* o *initiateAcc/completeAcc*. Si se trata de un acceso a memoria en modo *timing*, se ejecutará el par de funciones. En el resto de casos, se ejecutará *execute*. El primer módulo con el que interactúa el ISA es la CPU. Puede haber una gran variedad de implementaciones de CPU, por lo que gem5 define una interfaz común para todas ellas con una funcionalidad básica denominada ExecContext.

3.4.3 Memoria Transaccional en gem5

Una de las ventajas de las que dispone gem5 en su versión actual, es que ya ofrece una implementación de protocolo de coherencia en Ruby con soporte de memoria transaccional. De esta forma, gracias a la interfaz entre la parte del ISA y el resto de módulos, podemos reutilizar este protocolo para el nuevo sistema en RISCV.

Además, gracias al lenguaje SLICC, resulta más sencillo comprender el funcionamiento del protocolo y observar que las diferencias respecto a un protocolo MESI estándar son mínimas. Así se demuestra como al ser la unidad de transacción el bloque de caché, se puede reutilizar toda la lógica del protocolo existente para detectar los abortos en las transacciones. Esto permite una implementación a nivel hardware más sencilla.

La implementación del sistema de memoria transaccional en gem5 sigue el modelo ansioso de detección de conflictos, es decir, se cancela la transacción en cuanto se detecta el conflicto. Utiliza el protocolo de coherencia para determinar cuando un bloque de caché se debe añadir al conjunto de lectura/escritura, y cuando un cambio de estado generado por otro nodo genera un conflicto. Cuando esto pasa, se marca la transacción como abortada en el hilo que detecta el conflicto, y en la siguiente instrucción que acceda a memoria de este mismo hilo, se generará un mensaje desde memoria que indique el conflicto. Cuando la CPU lo recibe, inicia el procedimiento de aborto de transacción. Primero se deben invalidar los bloques que estaban en el conjunto de escritura para mantener la atomicidad. También se vacían los conjuntos de lectura/escritura para estar preparado para el inicio de otra transacción. Es posible que los registros del procesador se hayan modificado durante la transacción, por lo que hay que modificarlos con el valor previo restableciendo el Checkpoint. Por último se modifica el *Program Counter* con la dirección de la instrucción que sigue al inicio de la transacción y se reanuda la ejecución no especulativa.

Para el mantenimiento de versiones utiliza un enfoque perezoso, reutilizando la caché de datos como buffer de escritura especulativa. Cuando se produce un acceso a un bloque durante una transacción, se añade al conjunto correspondiente. Si es una escritura, se hace un Writeback al siguiente nivel de caché. De esta forma, tendremos una copia del valor previo al inicio de la transacción.

3.4.4 Compilación de gem5

Al disponer de un gran número de configuraciones, gem5 busca optimizar el tamaño del simulador especificando que arquitectura queremos simular, que tipos de CPUs y que protocolo Ruby se va a utilizar. Esto se realiza habitualmente a través de un fichero incluido en el directorio build_opts. Este fichero tendrá el nombre de la arquitectura acompañado opcionalmente por el protocolo Ruby utilizado. A continuación, se muestra el fichero creado para este trabajo:

```
TARGET_ISA = 'riscv'
CPU_MODELS = 'AtomicSimpleCPU,TimingSimpleCPU'
PROTOCOL = 'MESI_Three_Level_HTM'
```

Con la etiqueta TARGET_ISA indicamos la arquitectura que vamos a simular. Con CPU_MODELS indicamos el tipo de CPU, con el modo de acceso a memoria. Debemos indicar el modelo en función de como vamos a ejecutar la simulación. Si solo vamos a usar Ruby, podemos indicar únicamente la opción TimingSimpleCPU. En caso de utilizar solo el Classic, podemos poner o AtomicSimpleCPU, o TimingSimpleCPU, o los dos si vamos a alternar los modos de acceso. . Por último, indicamos el protocolo Ruby a utilizar con PROTOCOL. Actualmente, gem5 solo soporta un protocolo SLICC por compilación.

gem5 también ofrece distintas variantes de compilación, unas más centradas en depuración y otras en rendimiento:

Build variant	Optimizations	Run time debugging support	Profiling support
debug		X	
opt	X	X	
fast	X		
prof	X		X
perf	X		X

La opción debug se centra en el uso conjunto del simulador y de otras herramientas de depuración como gdb, además de utilizar el sistema de mensajes del simulador. La opción opt nos permite utilizar el sistema de mensajes disponiendo también de optimización en el código del simulador. La opción fast prioriza el rendimiento. Por último las opciones prof y perf están focalizadas en comprobar el funcionamiento del simulador en si.

Para este proceso de evaluación, usaremos las opciones opt y fast. La primera se utilizará para verificar el correcto funcionamiento de las transacciones. La segunda, para ejecutar todos los benchmarks en el menor tiempo posible y poder obtener las estadísticas que nos permitan analizar el funcionamiento.

Chapter 4

Implementación de Memoria Transaccional por Hardware para RISCV en gem5

El objetivo principal de este trabajo es aplicar los conceptos de las secciones anteriores para incluir soporte de Memoria Transaccional por Hardware [2] en RISCV. Para ello, primero se debe establecer que características definirán el nuevo modelo. Cómo se establecen los límites de una transacción, cómo se detectan los conflictos y cómo se mantiene el estado previo del núcleo serán los principales aspectos que definirán el comportamiento del modelo. Se tendrá que definir por completo, ya que actualmente la extensión T que hace referencia a memoria transaccional en RISCV no incluye nada especifico [21].

4.1 Especificación

Antes de proceder, resulta interesante conocer cómo están diseñados los sistemas de Memoria Transaccional por Hardware de los ISAs nombrados en la sección 3.1.4. A continuación, se van a mostrar sus principales características, entre las que se encuentran qué instrucciones añaden al ISA, cómo detectan los conflictos, cómo aplican el control de versiones y peculiaridades concretas de cada sistema. De entre todas las opciones expuestas, únicamente TME de ARM está incluida en gem5, teniendo en cuenta además que no tiene porque asemejarse a una futura implementación real de TME.

Rock HTM [7] sigue un modelo best-effort, añadiendo un nuevo registro donde se almacena la razón del aborto de la transacción para que sea el usuario quien determine como relanzar la transacción. Utiliza el modelo perezoso respecto al control de versiones, utilizando un buffer independiente donde se almacenarán las escrituras especulativas. Se escribirán en memoria al confirmarse la transacción. Sigue un enfoque ansioso respecto a la resolución de conflictos. Introduce dos nuevas instrucciones en el ISA. Con la instrucción checkpoint se inicia una transacción. Además permite indicarle por que instrucción continuar después de un abort. Con la instrucción commit se indica el final de la transacción, realizando la confirmación.

A diferencia del resto de implementaciones, el sistema HTM de IBM para sus servidores Blue Gen/Q [8] se utiliza la caché L2 para almacenar las escrituras especulativas hasta la confirmación. Por este motivo, habrá en el mismo buffer conjuntos de lectura/escritura de distintos hilos. Para poder diferenciarlos, los bloques se acompañan con un ID de transacción. A cada transacción se le asocia uno de estos IDs. En caso de que al iniciar una transacción no haya IDs disponibles, se bloquea hasta que lo haya. De forma periódica, se comprueba línea a línea si todos los bloques de con un ID se han invalidado o confirmado para liberarlo. Este

proceso se denomina *ID scrubbing*. Proporciona dos formas de interactuar con los accesos a memoria. *Short running mode* implica que todas las escrituras especulativas y posteriores lecturas se hagan sobre la L2, sin implicar a la L1. En *long running mode* si se utilizará la L1 en vez de la L2. Aplica un modelo ansioso para detectar los conflictos. Sigue un modelo best effort, pero el reinicio de la transacción está predefinido, pudiendo el usuario únicamente establecer el número máximo de intentos.

El sistema de IBM [9] para el ISAs POWER 8 y POWER 9 incluye 5 nuevas instrucciones. Con la instrucción tbegin inicia la transacción, suponiéndose todos los accesos a memoria dentro de la transacción. La instrucción tend permite confirmar la transacción. En caso de aborto, se aplica el modelo best-effort, utilizando el registro Transaction EXception And Summary Register (TEXASR) para determinar el motivo del aborto. Permite iniciar transacciones dentro de otras, pero procede uniéndolas en una sola, aplicando *flattened nesting*. Los cambios realizados por estas transacciones no se confirmarán hasta que lo haga la primera. Si existe un conflicto, se abortarán todas las transacciones. Además, implementa el sistema denominado transacciones elásticas. A través de las instrucciones tsuspend y tresume, se pueden indicar que conjunto de transacciones no deben evaluarse como parte de la transacción activa. Una transacción puede abortar por utilizar la instrucción tabort, demasiadas transacciones anidadas, desbordamiento del buffer de escritura especulativa...

El sistema TSX [10] de Intel utiliza una detección de conflictos ansiosa, es decir, comprueba en cada acceso si existe conflicto. Sigue un modelo pasivo en cuanto a la resolución de conflictos., abortando la transacción que detecta el conflicto. Aplica un modelo best effort, añadiendo un registro específico para que el usuario determine la causa del aborto. Utiliza un control de versiones Lazy, utilizando la caché L1 como buffer. Para que el usuario pueda utilizar el sistema, se incluyen tres nuevas instrucciones. XBEGIN permite indicar el inicio de una transacción y a partir de dónde continuar la ejecución en caso de aborto. XABORT permite abortar una transacción activa y XEND permite confirmarla.

La Transactional Memory Extension (TME) [12] de ARM también sigue un modelo "best effort", utilizando de nuevo un registro concreto para guardar el motivo del aborto. Garantiza que otros accesos a direcciones dentro de la transacción generen un conflicto. Si se inicia una transacción dentro de otra, la nueva se integra en la primera, es decir, no se confirmará hasta que se confirme la primera, o si se aborta, se aborta también la primera. Incluye cuatro nuevas instrucciones. Tstart permite iniciar una nueva transacción e indicar un registro para guardar el estado de la transacción. Tcommit permitirá confirmar la transacción, mientras que Tcancel permitirá abortarla. Por último se incluye la instrucción Ttest para conocer la profundidad actual de la transacción. En el momento de la redacción de este trabajo, todavía no se han especificado el modelo de detección de conflictos o de control de versiones. La única referencia es la implementación que ofrecen para gem5, la cuál establece una detección de conflictos ansiosa y un control de versiones perezosa reutilizando la caché de datos como buffer especulativo.

Por último, la Advanced Synchronization Facility (ASF) [13] de AMD es el único sistema de MT que es explícito. Incluye instrucciones que permiten indicar que accesos pertenecen a la transacción y la instrucción release, que permite eliminar un bloque del conjunto de lectura/escritura. Además, añade las instrucciones speculate, para indicar el inicio de la transacción, commit para confirmarla y abort para cancelarla de forma explícita. Aplica un control de versiones perezoso utilizando la caché L1 y una detección de conflictos ansiosa. Siguiendo un modelo best effort, utiliza el registro rAX para guardar el estado de la transacción y los motivos de un posible aborto.

En la tabla 4.1 podemos observar un resumen de las características principales de cada una de las implementaciones. Como podemos observar, todas las opciones tienen en común que son best effort. Permitirá al hardware liberarse de manejar el reinicio de las transacciones y permite al programador determinar cómo proceder. Se utiliza un registro específico para almacenar el motivo del aborto de una transacción. La única diferencia existente es que Blue Gene/Q predefine el manejador de abortos. Aunque todas siguen el modelo perezoso, el buffer no se localiza en el mismo sitio. El buffer independiente de Rock le permitirá evitar los abortos provocados por la asociatividad de las cachés, pero aumentará la complejidad del procesador. El uso de cachés como buffers simplificará el diseño, siendo L1 más rápida, pero de menor tamaño que L2. La primera reducirá el tiempo de ejecución de la transacción, pero tendrá más probabilidad de un aborto por

	ROCK	POWER	Blue Gene/Q	TSX	TME	ASF
Declaración	Implícita	Implícita	Implícita	Implícita	Implícita	Explícita
Instrucciones	chkpt,commit	tbegin,tabort, tend,tresume, tsuspend	X	xbegin, xabort, xend	tstart,tcancel, tcommit,ttest	speculate, commit, abort,release
Conflictos	Ansiosa	Ansiosa	Ansiosa	Ansiosa	Ansiosa	Ansiosa
Versioning	Perezoso	Perezoso	Perezoso	Perezoso	Perezoso	Perezoso
Buffer	Buffer independiente	Cache L2	Cache L2	Cache L1	Cache L1	Cache L1
Garantía	Best effort	Best effort	Best effort	Best effort	Best effort	Best effort

Table 4.1: Tabla resumen de HTM comerciales

asociatividad que L2, al disponer de menor espacio. Todas son implícitas excepto ASF, simplificando así el uso del sistema. Aunque un sistema explícito permita reducir los conjuntos de lectura/escritura, y por lo tanto los posibles conflictos, si las transacciones son excesivamente largas se convierte en un proceso tedioso. Además, puede que las direcciones que no se incluyan en la transacción sean tan pocas que no merezca la pena. Tampoco hay una referencia real de como se puede llegar a comportar un sistema explícito en aplicaciones grandes, ya que ASF todavía está en proceso de especificación y ningún procesador lo implementa. Cada una de ellas incluye al menos dos nuevas instrucciones para iniciar y confirmar la transacción. También incluyen una para abortar la transacción de forma explícita, excepto Rock HTM. Reutilizan el protocolo de coherencia para aplicar una resolución de conflictos ansiosa. Pretenden evitar que transacciones grandes se tengan que ejecutar de nuevo casi al completo. Las principales diferencias vienen a la hora de elegir como implementar el control de versiones. Partiendo de esto, definiremos nuestro sistema con las características más comunes, suponiendo que las empresas propietarias estos ISAs han realizado determinadas pruebas que les han permitido llegar a la conclusión de que estas opciones son las más viables. Para este trabajo, resulta imposible realizar ese proceso de evaluación.

Incluiremos dos nuevas instrucciones al ISA que nos permitirán definir los límites de la transacción siguiendo el modelo implícito. Así podremos aplicar el modelo transaccional justo dónde sea necesario. Además, añadimos también una tercera que permita al usuario cancelar una transacción en función de criterios distintos a los conflictos. De esta forma, se podrá evitar otros tipos de abortos más costosos en situaciones más concretas de cada aplicación.

- RVTM_BEGIN : se encarga de iniciar una transacción. Requiere de un parámetro que indique en que registro almacenar si la transacción se ha iniciado, o el motivo por el que ha abortado.
- RVTM_COMMIT : se encarga de confirmar una transacción. No requiere de ningún parámetro ni devuelve ningún valor. Esta instrucción deberá comportarse también como una barrera de memoria, forzando la ejecución de cualquier acceso a memoria previo antes de continuar la ejecución. Si no lo hiciera, cabría la posibilidad de que se produjera un acceso desde otra transacción sobre una dirección que, tras el reordenamiento, se va ejecutar después del commit. Esto estaría violando la atomicidad, ya que según como se define el sistema, se debería generarse un conflicto.
- RVTM_CANCEL : se encarga de abortar una transacción de forma explicita. Permite indicar el motivo de la cancelación a través del registro indicado.

Para facilitar la tarea al manejador de abortos, se define un conjunto de valores (Tabla 4.2) que representa el estado de una transacción. Estos valores son los que devolverá la instrucción RVTM_BEGIN, en función de si la transacción se ha iniciado correctamente, o en caso de abortar, indica también el motivo. Se definen una serie de errores fijos, que son los que detecta el propio sistema de MT. También se reserva un 'espacio' en RVTM_TMFAILURE_REASON para que el software pueda definir otros códigos a través de el parámetro pasado a RVTM_CANCEL.

Estado	Descripción	Valor
RVTM_TMFAILURE_REASON	Contiene el valor pasado a RVTM_CANCEL	0x00007fff
RVTM_TMFAILURE_RTRY	Indica si es recomendable volver a iniciar la transacción	0x00008000
RVTM_TMFAILURE_CNCL	Indica si el aborto es explícito	0x00010000
RVTM_TMFAILURE_MEM	Indica si el aborto es por conflicto	0x00020000
RVTM_TMFAILURE_ERR	Indica si el aborto es por excepción	0x00080000
RVTM_TMFAILURE_SIZE	Indica si el aborto es por capacidad	0x00100000
RVTM_LOCK_IS_ACQUIRED	Valor que se pasa a RVTM_CANCEL	0xffff
RVTM_TBEGIN_STARTED	Indica si se ha iniciado correctamente	0

Table 4.2: Tabla que indica los códigos definidos

Cada instrucción debe poder diferenciarse de otra. Para ello, se utiliza el campo opcode. Para las nuevas instrucciones, escogeremos el mismo opcode para las tres y utilizaremos un campo adicional (FUNCT3) para diferenciarlas entre ellas. Esto minimizará el número de opcodes utilizados y permitirá incluir mayor número de instrucciones en un futuro. Si no se utilizase un campo adicional, el número total de instrucciones que se podrían incluir en el ISA estaría limitado por la longitud del campo opcode.

A la hora de escoger un opcode en RISCV, se deberá tener en cuenta la distribución actual. Hay opcodes que ya están siendo utilizados, por lo que si escogemos uno de ellos, estaremos sobrescribiendo otras instrucciones. También hay opcodes que están reservados, lo que quiere decir que a fecha de la publicación de la especificación no están siendo utilizados, pero que está previsto utilizarlos para nuevas instrucciones. Si utilizáramos uno reservado, a corto plazo no habría problema, pero con la publicación de una nueva versión se podría generar un conflicto entre las instrucciones oficiales y las nuestras. Para evitar estos problemas y facilitar a los usuarios la personalización del ISA, se ofrecen varios opcodes que están libres, es decir, que ni se están usando ni en principio se van a utilizar.

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

Table 4.3: RISC-V base opcode map, inst[1:0]=11

Para poder diferenciarlos, se dispone una tabla 4.3 en la especificación de RISCV que indica el estado de cada opcode. Como se puede observar, únicamente se hace referencia a los bits del 2 al 6. Los dos restantes, el 0 y el 1, deben ponerse a 1. Esto se debe a que los valores 00,01,10 van asociados a la extensión de instrucciones comprimidas. Analizando la tabla detenidamente, podemos observar como unos opcodes están asociados a instrucciones, otros están *reserved* y otros *custom*. Estos últimos son los que quedan libres, y por lo tanto, habrá que escoger uno de ellos.

Una vez se ha escogido un opcode, cada instrucción se deberá codificar con un formato concreto para que pueda ser decodificado por RISCV. En la figura 3.4 se resumen los más utilizados. En los tres casos nos basaremos en el formato R, ya que para RVTM_BEGIN(4.2) y RVTM_CANCEL(4.4) necesitamos indicar un registro. RVTM_COMMIT(4.3) no necesita un registro, pero utilizamos el mismo formato. También necesitamos un campo adicional para diferenciar las instrucciones aunque usemos el mismo opcode. En las figuras anteriormente mencionadas se muestran los formatos.

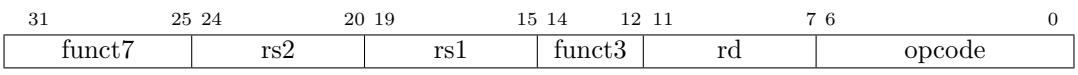

Figure 4.1: Formato instrucción R

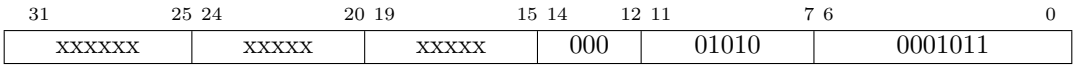

Figure 4.2: Formato Instrucción RVTM_BEGIN

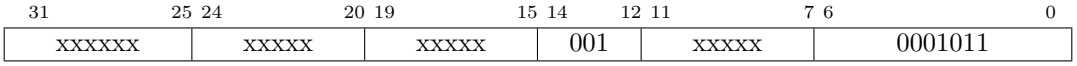

Figure 4.3: Formato Instrucción RVTM_COMMIT

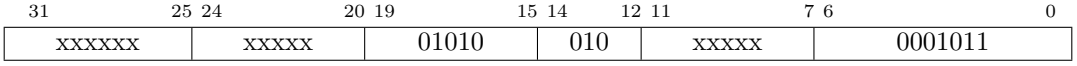

Figure 4.4: Formato Instrucción RVTM_CANCEL

Como se puede observar, tanto el registro destino en RVTM_BEGIN como un registro fuente en RVTM_CANCEL están indicados en la propia instrucción. Esto no es lo habitual, ya que es el propio compilador el que establece en que registro está cada variable en un determinado momento. En nuestro caso, las instrucciones añadidas no pertenecen al ISA que maneja el compilador de RISCV utilizado, por lo que se ha tenido que escoger un registro arbitrariamente.

A partir de estas instrucciones, definiremos una abstracción para cada una para facilitar el uso de estas instrucciones por parte del programador. Si no lo hiciéramos, el usuario tendría que codificar la instrucción en binario a través de ensamblador, ya que al no pertenecer las instrucciones al ISA oficial, no existe soporte por defecto. Además, el proceso de añadir las instrucciones al ensamblador resulta bastante complejo, y se sale del alcance de este trabajo.

Todas las abstracciones se definirán a través de macros de C. RVTM_BEGIN() encapsulará la instrucción de comienzo de transacción, permitiendo devolver el estado de la transacción. RVTM_COMMIT contendrá la instrucción de confirmación. Por último, RVTM_CANCEL(razón) permitirá indicarle un motivo de cancelación a la instrucción.

Para simplificar más la integración del modelo en las aplicaciones, se proporciona una librería que permite al usuario únicamente tener que indicar donde se inicia la sección crítica y donde termina. De esta forma, si no se puede ejecutar con Memoria Transaccional, se procederá a utilizar locks, para así evitar un livelock y garantizar la ejecución de la aplicación. Se propone lo mostrado en el algoritmo 1.

El método beginTransaction intenta ejecutar la sección crítica aplicando el modelo transaccional. Si aborta un determinado número de veces antes de completarse, o se produce un aborto por un motivo no manejable a través de HTM, como los abortos por capacidad, se procede a utilizar un lock global. Es importante determinar un número de intentos adecuado. Si es demasiado bajo, podríamos no estar aprovechando del todo las transacciones. Si por el contrario es demasiado alto, una transacción podría estar continuamente abortando hasta llegar a este límite, malgastando recursos. También hay que tener en cuenta que si una sección crítica se está ejecutando utilizando el lock, no se podrán ejecutar transacciones hasta que termine, ya que se estaría violando la atomicidad. Por último, se especifica que si se ha abortado por que se está utilizando el lock, no se contabilice en el número de intentos, ya que la transacción no es culpable del aborto. El método commitTransaction (Algoritmo 2) se encarga de confirmar una transacción, o de liberar el lock, dependiendo de como se haya ejecutado beginTransaction.

También se debe definir como manejar las escrituras especulativas, para así permitir una restauración en caso de aborto. Se optará por una aproximación perezosa. De esta forma, los valores escritos procedentes de

Algorithm 1 Fallback

```
1: procedure beginTransaction
2:     utilizarLock ← false
3:     intentos ← 0
4:     while utilizarLock == false do
5:         intentos ← intentos + 1
6:         TransaccionIniciada ← RVTM_BEGIN()
7:         if TransaccionIniciada then
8:             if LOCK_NO_ADQUIRIDO then return
9:             else
10:                RVTM_CANCEL(LOCK_ADQUIRIDO)
11:        if ABORTO_POR_CONFLICTO && LOCK_ADQUIRIDO then
12:            intentos ← intentos − 1
13:        ESPERAR_LIBERACION_LOCK()
14:        if ABORTO_EXPLICITO && LOCK_ADQUIRIDO then
15:            intentos ← intentos − 1
16:        if !PUEDE_TERMINAR_AL_REINICIAR then
17:            utilizarLock ← true
18:        if intentos == INTENTOS_MAXIMOS then
19:            utilizarLock ← true
20:    adquirirLock()
```

Algorithm 2 commitTransaction

```
1: procedure commitTransaction
2:     if lockAdquirido() == true then
3:         liberarLock()
4:     else
5:         RVTM_COMMIT()
```

una transacción no estarán presentes en memoria hasta que se confirme la transacción. Se almacenarán en la caché de datos L1, evitando la inclusión de un buffer específico para almacenar los valores especulativos. Es la solución más habitual entre los procesadores de propósito general, ya que implican el menor número de cambios en el hardware actual.

Además, se resolverán los conflictos siguiendo el modelo ansioso. Cuando se detecta un conflicto, se procede a tomar una acción. Esta acción será la de abortar la transacción que detecta el conflicto, siguiendo el modelo pasivo. Se aplicará en todo momento una metodología best effort, delegando el manejo del aborto de transacción al software. Esto por un lado, aportará flexibilidad al programador para proceder según el error, y simplificará los cambios necesarios al hardware, al no tener que implementar este procedimiento. Por último, se establece una granularidad de bloque de caché. Añadiendo un bit para indicar pertenencia a read set y otro para write set, podemos aprovechar el acceso a la caché para actualizar el estado del bloque en la transacción.

Respecto a la copia de los valores de los registros del procesador previos al inicio de la transacción, dependerá de la implementación de RISCV en un chip concreto, por lo que se decide no especificarlo. Algunos de los métodos utilizados habitualmente se muestran en la sección 3.1.3.

4.2 Implementación en gem5

Antes de iniciar el proceso de implementación, se comprueba que un binario que solo contenga instrucciones de las extensiones oficiales se ejecute correctamente. Probamos con el binario HelloWorld que se incluye con el simulador. El resultado es que la ejecución no termina, por lo que hay que resolver este problema antes de empezar con la implementación. Se ejecuta el simulador en modo debug para generar una traza que nos muestre el flujo de las instrucciones. Se observa que las instrucciones AMO no se ejecutan correctamente, al no leerse el valor que correspondería en función de las instrucciones previas. Se asocia el problema a la interacción de RISCV con Ruby ya que ejecutando el binario en RISCV con el modelo Classic si funciona correctamente.

Como primera alternativa, se decide intentar implementar soporte de Memoria Transaccional en el modelo de memoria Classic. Debido a que el diseño de todo el sistema desde cero llevaría bastante tiempo, se toma como referencia el implementado en Ruby. El principal elemento que conforma el modelo Classic es el objeto Cache, formado a su vez por varios bloques de cachés. Se tendrá que añadir a la Cache una variable que indique si se está ejecutando una transacción o no, otra que indique la causa de un posible aborto y otra que indique si ha habido un aborto. También habrá que añadir a los bloques dos variables que indiquen si pertenecen al conjunto de lectura y/o escritura, respectivamente. Habrá que incluir lo necesario para que una instrucción de inicio, confirmación o cancelación se detecte. A través de la interfaz Packet, se podrá determinar si una petición proveniente de CPU es de este tipo. Un inicio de transacción pondrá a True la variable correspondiente. Esta variable se utilizará para determinar que accesos pertenecen a una transacción. Una lectura o escritura añadirá el bloque de caché al conjunto de lectura y escritura, respectivamente. Esto implicará poner la variable asociada a cada bloque a True. Estas variables se utilizarán para poder determinar si una petición procedente de otro procesador, a través del protocolo de coherencia, generan un conflicto. De esta forma, si se detecta una lectura sobre un bloque que esté en el conjunto de escritura, o una escritura sobre un bloque que esté en cualquiera de los conjuntos, se pondrá a True la variable que indica un aborto. Esta variable se comprobará en el siguiente acceso a memoria, detectándose el aborto y procediendo a abortar. Cuando se produzca una escritura sobre un bloque modificado por primera vez en una transacción, habrá que mantener una copia realizando un writeback. El modelo Classic utiliza una lista donde incluir que bloques mandar al siguiente nivel. En caso de aborto, se almacenará el motivo en la variable correspondiente y se desactivará el modo transaccional. Será necesario ampliar el objeto Cache para que en caso de aborto, se reinicien los conjuntos de lectura/escritura poniendo las variables de cada bloque a False, e invalidando los bloques escritos durante la transacción.

A pesar de estos cambios, no se consigue que funcione correctamente, por lo que se intenta corregir el problema de Ruby anteriormente mencionado. Según la estructura de Ruby, el objeto Sequencer es el que se encarga de manejar los accesos atómicos a memoria. Comparándolo con el modelo Classic, se observa que el procedimiento está incompleto, en concreto, en el manejo de las operaciones Swap. Uno de los motivos posibles para que todavía no se haya implementado este soporte es que RISCV todavía está siendo añadido a gem5, avanzando poco a poco en cada versión, utilizando el modelo Classic para añadir un soporte inicial. Para completarlo, se debe comprobar si una operación Swap se realiza en modo atómico para tratarla como tal. Se compara el código fuente asociado a la interacción entre las instrucciones AMO y memoria en el modelo classic con el de Ruby. Se inserta un código similar al de Classic en Ruby y se vuelve a probar el binario. Ahora su ejecución es correcta, por lo que se da por resuelto el problema y se procede a implementar las instrucciones.

Una vez verificado el funcionamiento del entorno de simulación, pasamos a la fase de implementación. gem5 consigue simplificar el proceso de añadir nuevas instrucciones asociando cada una a una clase de C++. Hay definidas clases con funcionalidad básica que o pueden ser utilizadas directamente, o se pueden reutilizar y crear la propia heredando de una de ellas. Cada tipo de instrucción lleva asociada un conjunto de métodos que definen su comportamiento. En nuestro caso, las instrucciones que vamos a implementar se clasifican como accesos a memoria. Aunque en realidad no lo sean, nos permitirán acceder al sistema de Memoria Transaccional, el cual reside en memoria, reutilizando el canal de comunicación habitual entre CPU y memoria. Además el modo de simulación será timing. Este modo de acceso, como se ha visto en la sección 3.8, implica una fase inicial, donde se inicia la comunicación con memoria, y otra fase donde se analiza la respuesta por parte de memoria. A cada una de esas fases se les asocia las funciones initiateAcc y completeAcc, respectivamente.

Cada una de las instrucciones deberá pertenecer a una jerarquía de clases que permita reutilizar funcionalidad. En nuestro caso, La instrucciónes RVTM_BEGIN Y RVTM_CANCEL heredarán de la clase base de toda instrucción de RISCV, RiscvStatisInst. Proporcionará la funcionalidad mínima necesaria para que la clase funcione como instrucción.

Para implementar la instrucción RVTM_COMMIT, utilizaremos las denominadas macro instrucciones. Permitirán que una instrucción a nivel de código ejecute varias en tiempo de ejecución.

En este caso, la macroperación estará compuesta por dos microinstrucciones:

- MicroMfence : detiene el cauce hasta que se completen todos los accesos a memoria previos.
- MicroCommit : inicia la confirmación de la transacción.

Si no se definieran como microinstrucciones, heredando de RiscvMicroInst, podría haber problemas con el Program Counter. Hay que tener en cuenta que, aunque en realidad se ejecuten dos instrucciones, a nivel de código solo se ha ejecutado una, por lo que si se definieran como instrucciones normales, el Program Counter se incrementaría el doble de lo necesario. También es fundamental la ejecución de *mfence*, y así garantizar, en los procesadores fuera de orden, que se haya comprobado si han habido conflictos en todos los accesos. Si no lo hiciéramos, los accesos no completados podrían generar un conflicto y no detectarse al haber finalizado la transacción, incumpliendo la atomicidad al no tomarse medidas.

La jerarquía de las nuevas clases quedaría como se indican en las figuras 4.5 y 4.6.

gem5 ya incluye soporte para memoria transaccional por hardware en ARM, por lo que ofrece una interfaz común a través del método initiateHtmCmd de la clase ExecContext. A esta función habrá que indicar como parámetro que cambio vamos a realizar sobre la transacción. En la figura 4.7 se muestra como abstrae al ISA de la comunicación con memoria.

A continuación se muestra como cada instrucción hace uso de esta interfaz:

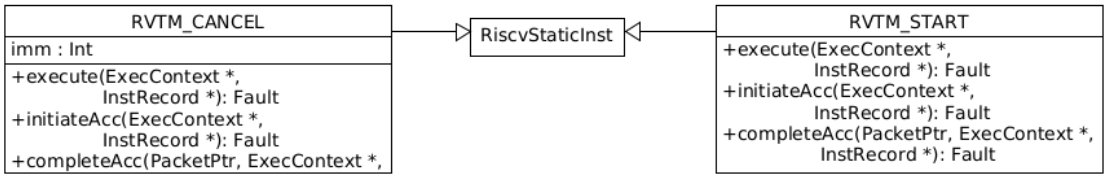

Figure 4.5: Diagrama instrucciones RVTM_START y RVTM_CANCEL

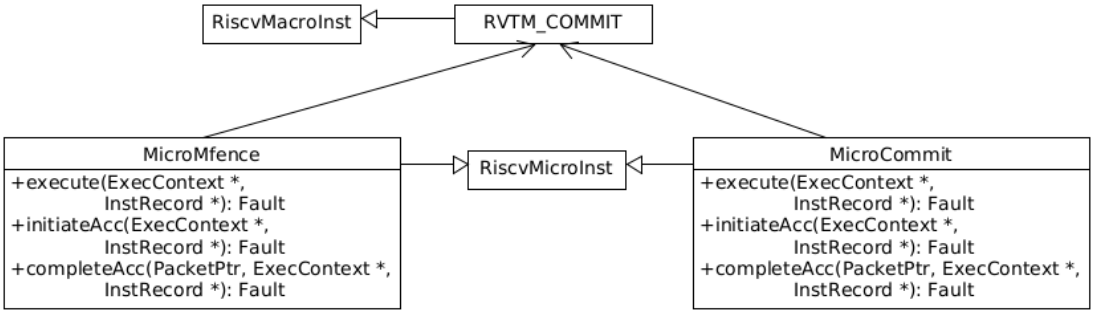

Figure 4.6: Diagrama instrucción RVTM_COMMIT

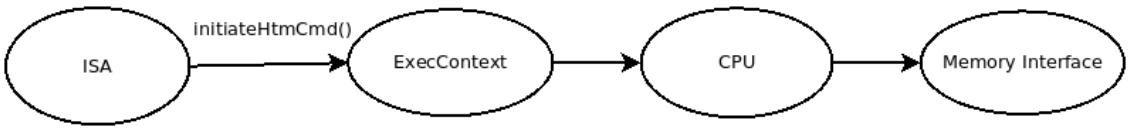

Figure 4.7: Diagrama interfaz ExecContext

- La instrucción RVTM_START indicará el inicio de una nueva transacción. El método initiateAcc será el que llame a initiateHtmCmd con la opción HTM_START. El método completeAcc será el que realice el Checkpoint del procesador. Esto solo sucede si la transacción se ha iniciado correctamente ya que solo se llama si initiateAcc no ha generado un error.
- La instrucción RVTM_CANCEL hará uso de initiateAcc para pasar el parámetro HTM_EXPLICIT a initiateHtmCmd para indicar a memoria la cancelación. Esta instrucción no aborta la transacción directamente. En lugar de eso fuerza un aborto de transacción. En completeAcc será donde se indique este aborto al procesador y éste proceda a abortar.
- La instrucción RVTM_COMMIT pasará a initiateHtmCmd el parámetro HTM_COMMIT en la función initiateAcc para iniciar la confirmación de la transacción activa. La funcionalidad de completeAcc es reiniciar el Checkpoint para la siguiente transacción.

En los algoritmos 3,4 y 5 se muestra la implementación de los métodos en pseudocódigo.

Algorithm 3 RVTM_COMMIT

```
1: procedure INITIATEACC
2:     flags ← HTM_COMMIT
3:     initiateHtmCmd(flags)
4: procedure COMPLETEACC
5:     checkpoint.reset()
```

Algorithm 4 RVTM_START

```
1: procedure INITIATEACC
2:     currentTransactionLen ← length of currentTransaction
3:     if currentTransactionLen > maxHtmDepth then return Failure::NEST
4:     flags ← HTM_START
5:     initiateHtmCmd(flags)
6: procedure COMPLETEACC
7:     currentTransactionLen ← length of currentTransaction
8:     if currentTransactionLen == 1 then
9:         checkpoint ← getHtmCheckpoint()
10:        checkpoint.save()
11:        checkpoint.destinationRegister(dest)
```

Algorithm 5 RVTM_CANCEL

```
1: procedure INITIATEACC
2:     flags ← HTM_CANCEL
3:     initiateHtmCmd(flags)
4: procedure COMPLETEACC
5:     checkpoint ← getHtmCheckpoint()
6:     checkpoint.cancelReason(reasonFailure) return Failure::EXPLICIT
```

También debemos definir los constructores, donde se inicializarán una serie de variables que determinarán el tratamiento que recibirá la instrucción. Por ejemplo, RVTM_START tendrá las flag isMemRef y HtmStart a true. Este conjunto de opciones permitirá que el simulador proceda adecuadamente hasta el final de la instrucción.

Una vez implementadas las instrucciones, hay que incluirlas en el Decodificador para que el simulador las identifique. En gem5 se define un decodificador por cada ISA, habitualmente en un fichero llamado decoder.isa. Este fichero tiene un formato específico. Como se observa en la figura 4.8, la palabra decoder

```
  decode QUADRANT {
    ...
    0x3: decode OPCODE {
        ...
        0x02: decode FUNCT3 {
            format RVTM_FORMAT {
                0x0: RVTM_START();
                0x1: RVTM_COMMIT();
                0x2: RVTM_CANCEL();
            }
        }
        ...
    }
}
```

Figure 4.8: Fichero decoder.isa

aparece varias veces. Sirve para indicar el inicio de un decode block, en el cual se indica un campo de la instrucción, cuyo valor se comparará con los valores indicados entre llaves. Estos valores pueden indicar el inicio de un nuevo decode block para comprobar el valor de otro campo, o directamente indicar la identificación de una instrucción.

Para añadir las nuevas instrucciones, se ha tenido que incluir el valor 2 previamente asignado dentro del segundo decode block mostrado, que corresponde con los 5 bits restantes del campo opcode. Se puede observar que este decode block está asociado al valor 3 del primer decode block, que corresponde con los dos primeros bits del campo opcode. También es necesario un nuevo decode block asociado al valor insertado, con el que se identificarán las nuevas instrucciones comprobando el valor del campo FUNCT3. En el nuevo decode block , antes de los valores aparece un bloque format. Un formato facilitará la definición de las clases que implementan las instrucciones. El formato indicado únicamente se utiliza para definir la llamada al constructor, ya que se ha decidido definir las clases a parte. Por último, acompañando a los valores está el nombre de las clases asociadas a las instrucciones.

Cuando una transacción aborta, es necesario volver al estado previo del núcleo y así poder continuar la ejecución sin que los cambios dentro de la transacción se reflejen. Para ello es necesario guardar una copia de los valores de todos los registros utilizados. gem5 implementa esto a través de una clase RiscvHTMCheckpoint en cuyas variables se almacenan esos valores. Necesitamos definir como se va a realizar el checkpoint en RISCV, ya que cada arquitectura nombra a los registros de forma distinta. gem5 también ofrece una interfaz para manejar los checkpoints a través de la clase BaseHTMCheckpoint, por lo que tendremos que crear una nueva clase que herede de ésta y que implemente estos métodos. El proceso es sencillo: se copian loa valores de las arrays que representan los registros en otras arrays asociadas al objeto RiscvHTMCheckpoint. De esta forma, cuando se quiera restaurar el estado previo a una transacción, únicamente habrá que volver a copiar estas arrays. Este objeto también se utilizará para almacenar el motivo del aborto. Cuando se resuelve el conflicto, se copia su valor al registro destino indicado en RVTM_BEGIN.

Una vez incluidas las instrucciones, definida la clase RiscvHTMCheckpoint y compilado el simulador sin errores, se realiza alguna prueba preliminar aplicando la metodología expuesta en la siguiente sección. En algunos benchmarks se observa un número desproporcionado de transacciones ejecutadas de forma no especulativa. En un primer momento, se asocia el problema al manejador de abortos. Se comprueba que los valores devueltos por el sistema de memoria transaccional concuerdan, y se observa que al manejar un aborto explícito, el código de error no es correcto. El objeto encargado de los códigos de error es RiscvHTMCheckpoint, por lo que en primera instancia, se hace una comprobación rápida del código fuente. Durante la comprobación, se observa que la llamada a la función bits() dentro del método restore no pasa los parámetros de forma correcta. Este error provocaba que el bit que indicaba si la transacción podía reiniciarse estuviera

siempre a cero. De esta forma, cualquier aborto explícito provocaba la adquisición del lock, algo que no es correcto. Para solucionarlo, únicamente se da la vuelta a los argumentos. Al realizar de nuevo las pruebas que mostraban un comportamiento erróneo, se comprueba que ahora si se obtienen unos datos más comprensibles. Aunque se trata de un error leve, puede llegar a provocar grandes problemas al utilizar memoria transaccional en gem5, por lo que se decide proponer la solución para incluirla en la versión oficial de gem5 [22]. En la fecha de la redacción del documento, el cambio propuesto está aprobado y listo para incluirlo en la siguiente versión de gem5.

Chapter 5

Metodología de evaluación

Dentro de cualquier proyecto, es fundamental comprobar que la solución funciona correctamente, y además, obtener datos que permitan compararla con otras alternativas más consolidadas. Para este trabajo, necesitaremos de una prueba que nos ayude a determinar si el sistema de Memoria Transaccional por Hardware que se propone ofrece unos resultados correctos y a determinar su desempeño. Para ello, utilizaremos una prueba de rendimiento, o benchmark. Consiste en la ejecución de un programa o varios con el objetivo de observar como interactúa con una máquina, en nuestro caso, con la máquina virtual que se forma a partir de los parámetros del simulador.

5.1 Stanford Transactional Applications for Multi-Processing(STAMP)

Aunque la Memoria Transaccional está emergiendo y no paran de aparecer nuevas implementaciones, no existen un gran número de herramientas que nos permitan evaluar de forma sencilla un determinado sistema.

El conjunto de benchmarks STAMP [23] destaca sobre el resto de opciones y está posicionado como referente en el presente de la evaluación de los sistemas MT. Utiliza aplicaciones reales para realizar la evaluación. Surge de la necesidad de disponer de un conjunto de herramientas lo suficientemente variadas para poder evaluar el sistema en el mayor número de situaciones posible, y poder obtener una conclusión lo más genérica posible. Ofrece una interfaz común para declarar las transacciones que facilita la portabilidad. Ofrece opciones tanto para STM, como para HTM y para utilizar locks. Consta de 8 benchmarks:

- bayes : esta aplicación implementa el aprendizaje de una estructura de una red bayesiana. Utiliza la estrategia ascenso de colinas con búsqueda local y global. Genera nuevas dependencias en función de los datos observados. La transacción se utiliza para proteger el calculo y suma de una nueva dependencia. Este proceso representa casi el total del tiempo de ejecución, de forma que se forman largas transacciones con grandes R/W sets.
- genome : combina distintos segmentos de ADN y reconstruye el original. Elimina los duplicados añadiendo cada segmento a un hash set y comprueba cuales encajan con el original utilizando el algoritmo de Rabin-Karp. Utiliza transacciones para las dos fases. Tanto la longitud como el tamaño de los R/W set son moderados. Hay poca contención.
- intruder : emula un sistema de detección de intrusos basado en firma. En la fase de captura se encolan los paquetes, en la de ensamblado se unifican los paquetes de una misma sesión en un diccionario y por

último, la captura. Se protegen con transacción las dos primeras fases. Contiene un número moderado de transacciones, pero tienen poca longitud. El nivel de contención va de moderado a alto.

- kmeans : agrupa objetos relacionados en K clusters. Las transacciones protegen la actualización de los clusters. Cuanto mayor es K, menor es el nivel de contención. El tamaño tanto de las transacciones como de los R/W sets es pequeño.
- labyrinth : calcula si existe un camino entre dos puntos del laberinto. El tamaño tanto de las transacción como de los R/W sets es muy grande, ya que casi todo el código se ejecuta en una transacción. La contención es muy alta.
- ssca2 : construye un grafo utilizando array adyacentes y auxiliares. Las transaccione protegen el acceso a las array adyacentes. El tamaño tanto de las transacciones como de los R/W sets son bajos. La contención también.
- vacation : emula un sistema de reserva de viajes. Los clientes puede reservar, cancelar y actualizar. Estos tres procesos están protegidos por transacciones. Tiene un tamaño medio de transacciones y R/W sets. La contención va de baja a moderada en función de los parámetros.
- yada : implementa el algoritmo de Ruppert para el refinamiento de mallas Delaunay. Las transacciones protegen el acceso a la cola de trabajo, lo cual implica casi toda la ejecución. Las transacciones y los R/W sets son grandes. La contención es moderada.

Para la evaluación, se utilizarán tamaños de entrada pequeños (sufijo small), que permitirá simular los benchmarks en un tiempo razonable. Además, para kmeans y vacation habrá dos versiones: una de baja contención (sufijo -l) y otra de alta (sufijo -h).

5.1.1 Modificación STAMP

El benchmark STAMP define un API a través de una serie de macros, los cuáles se utilizan en todos los benchmarks. Esto permite que si se quiere incluir soporte para un nuevo sistema de Memoria Transaccional, únicamente habrá que modificar con la nueva funcionalidad el fichero donde están definidos estos macros. Las funciones relevantes que se llaman en estos macros son beginTransaction, abortTransaction y commitTransaction. Estas funciones serán las que definan como se van a manejar las transacciones en los benchmarks. A su vez, estas funciones se apoyan en otras funciones auxiliares que definen unas acciones más concretas. Estas serán las que tengamos que sobrescribir en caso de que la arquitectura objetivo sea RISCV. En estas funciones es donde incluiremos los macros anteriormente definidos.

5.2 Parámetros

Para todas las simulaciones utilizaremos el modelo de CPU TimingSimpleCPU y el modelo Ruby para el sistema de memoria. Definiremos una jerarquía de cachés de tres niveles, siendo los dos primeros cachés privadas de cada procesador, y el último nivel compartido. El tamaño de la caché L1 será de 32KB, el de la L2 será de 256 KB con una asociatividad de 8 vías y, por último, la L3 tendrá 32MB y una asociatividad de 16 vías. También es necesario definir una red que una todos los elementos y permita su interacción. Utilizaremos la red Crossbar, que es la que se utiliza por defecto. La figura 5.1 muestra un ejemplo de esta estructura utilizando 8 procesadores. La tabla 5.1 resume los parámetros de simulación.

Una vez definidos los parámetros comunes, se proponen dos estudios sobre el rendimiento de las transacciones en función de dos parámetros:

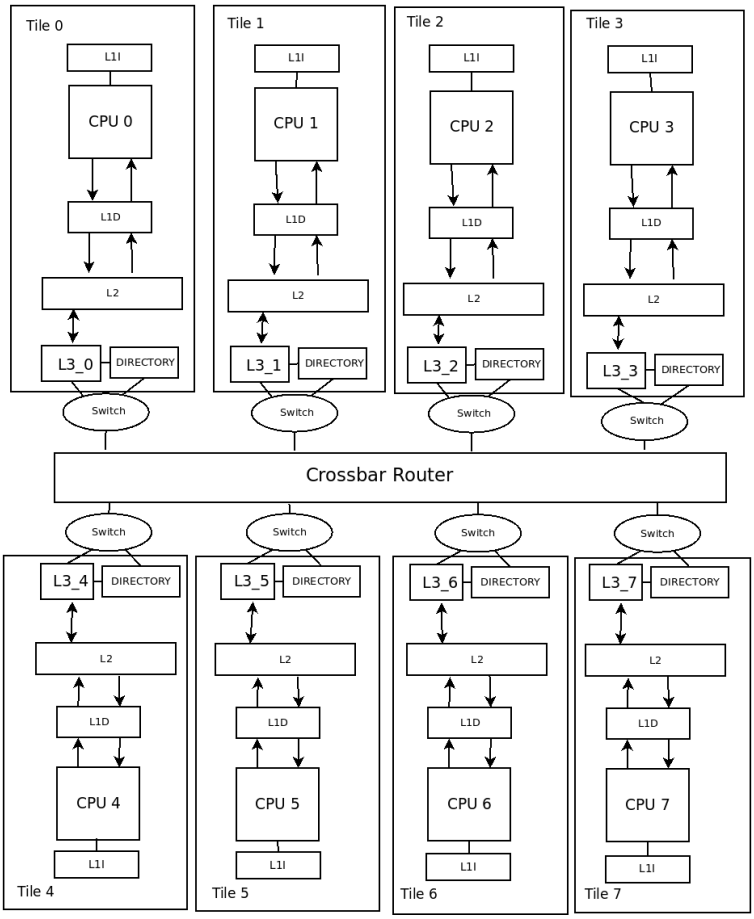

Figure 5.1: Diagrama CPU – Cache

Característica	Descrpción
Procesadores	1 a 16 núcleos RISCV, *in-order*,1GHz
Red	Crossbar
Cache L1	32KB,privada,1 a 8 vías,protocolo MESI
Cache L2	256KB,privada,8 vías
Cache L3	32MB,compartida,16 vías
Memoria	2GB,DDR3-1600

Table 5.1: Características entorno a simular

bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2 -t*threads*
genome	-g256 -s16 -n16384 -t*threads*
intruder	-a10 -l4 -n2038 -s1 -t*threads*
kmeans	-m40 -n40 -t0.05 -i inputs/random-n2048-d16-c16.txt -p*threads*
labyrinth	-i inputs/random-x32-y32-z3-n96.txt -t*threads*
ssca2	-s13 -i1.0 -u1.0 -l3 -p3 -t*threads*
vacation	-n4 -q60 -u90 -r16384 -t4096 -c*threads*
yada	-a20 -i inputs/633.2 -t*threads*

Table 5.2: Parámetros benchmarks

- Variación del número de hilos: variaremos el número de hilos de 1 a 16 en potencias de 2. Con esto evaluaremos la escalabilidad del sistema, comprobando el aumento que se produce en los abortos por conflicto de memoria. Utilizaremos una asociatividad de 8 y un tamaño de L1 de 32KB.
- Variación de la asociatividad: variaremos el nivel de asociatividad de la caché L1 de datos de 1 a 8 en potencias de 2. Así comprobaremos como varia los abortos por capacidad. Utilizaremos 4 hilos y el mismo tamaño de L1.

Cada uno de los benchmarks de STAMP requiere de una serie de parámetros para modificar su funcionamiento. Tenemos que ajustarlos al contexto de una simulación, es decir, valores suficientemente grandes para poder visualizar el comportamiento del benchmark, pero sin llegar a ciertos valores donde el tiempo de simulación sería poco practico. En la tabla 5.2 se muestran esos valores para cada uno de los benchmarks.

Chapter 6

Evaluación

A través del proceso de evaluación mencionado en la sección anterior, se busca concluir como se desenvuelve el sistema de Memoria Transaccional propuesto en un amplio número de situaciones. Se mostrarán los resultados obtenidos y una serie de gráficas que permitan observar de forma más sencilla estos resultados.

En la tabla 6.1 podemos observar que tipo de transacciones contiene cada benchmark. De esta forma, seleccionando un determinado benchmark podemos evaluar como afecta el incremento del número de hilos a una ejecución en función de la longitud de las transacciones. También podemos evaluar como afecta la asociatividad de L1 en función del R/W set.

La longitud indica cuantas instrucciones están incluidas en una transacción. El R/W set indica cuantas direcciones de memoria distintas se acceden en la transacción. EL tiempo indica cuanto tiempo respecto al total se ejecutan las transacciones. Por último, la contención indica cómo de probable es que se genere un conflicto al determinar cuantos de los accesos totales son a las mismas direcciones.

Las medidas utilizadas son relativas entre los propios benchmarks. Por ejemplo, si un benchmark tiene una longitud corta, implica que la longitud sea mucho menor que la que esté clasificada como media. Hay que tener en cuenta que estas propiedades son teóricas. Se han obtenido a partir del análisis del código fuente. Aunque es posible que exista una relación directa entre estas propiedades y el desempeño real de las aplicaciones, también habrá algunas aplicaciones que no se ejecuten tal como indican sus propiedades.

Benchmark	Longitud	R/W set	Tiempo	Contención
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Medium
yada	Long	Large	High	Medium

Table 6.1: Características benchmarks

6.1 Escalabilidad

En esta sección se muestran los resultados obtenidos de la ejecución de los benchmarks con la configuración indicada, y aplicando la variación de hilos. La gráfica 6.1 muestra la mejora obtenida en función del número de hilos respecto a la versión secuencial sin aplicar memoria transaccional. Así podremos observar si la mejora obtenida al paralelizar es mayor que la penalización obtenida por el manejo de las secciones críticas. A primera vista, se observa que los benchmarks bayes, yada y labyrinth escalan de forma pésima, incluso llegando a empeorar la versión secuencial. Otros, como intruder, vacation y genome, mejoran hasta los 8 hilos, pero empiezan a mostrar degradación a partir de los 16 hilos. Las dos versiones de kmeans son los únicos benchmarks que muestran un comportamiento excelente, mostrando que todavía pueden seguir mejorando. Los motivos de todos estos comportamientos se expondrán más adelante. Cabe destacar que del benchmark ssca2 y kmeans-h-small no se han podido obtener todos los resultados que se hubieran querido. El motivo es una implementación no del todo completa de algunas llamadas al sistema relacionadas con el manejo de hilos. La corrección de este fallo podría conllevar un tiempo del que no se dispone.

Las gráficas 6.2 y 6.4 servirán para explicar los resultados de la primera gráfica. La primera muestra como se reparten los abortos entre cuatro tipos distintos. Un aborto por conflicto indica que se ha realizado un acceso a memoria que viola la atomicidad. Un aborto por capacidad se produce cuando se remplaza un bloque de la L1 que esté en el conjunto de lectura/escritura. Un aborto explícito, en este caso, significa que se ha iniciado una transacción mientras el lock global está adquirido. Por último, un aborto por excepción ocurre habitualmente por un fallo de página. Hay que tener en cuenta que, aunque se representan en el mismo eje, cada tipo de aborto implica una penalización distinta. Por ejemplo, un aborto por explícito nunca será más costoso que un aborto por conflicto, por lo que no tendrán la misma influencia en el tiempo de ejecución. De esta gráfica cabe destacar como, en todos los benchmarks, se produce un aumento del número de abortos por conflicto conforme se aumentan el número de hilos. Este será el motivo principal que impida a determinados benchmarks escalar idóneamente. La segunda gráfica muestra cuantas transacciones se han abortado, y confirmado, además de cuantas se han tenido que ejecutar con el lock global. Nos permitirá saber que benchmarks se ven más afectados por tener que secuencializar las transacciones a través del lock global. Cuanto más grandes sean los conjuntos de lectura/escritura, mayor número de transacciones se tendrán que ejecutar a través del lock. Podemos observar como esto se cumple con yada, bayes y labyrinth.

A continuación se van a razonar los resultados de cada benchmark.

- bayes : este benchmark no muestra mejora. Se puede observar como el aumento considerable de abortos provoca un aumento en el número de transacciones que se ejecutan con el lock (Figura 6.4). La gráfica 6.2 muestra como este aumento es debido a los abortos por conflicto, fomentados por la alta contención del benchmark. Su gran tamaño de conjuntos de lectura/escritura puede llegar a provocar un número importante de abortos por capacidad, que junto con los abortos por excepción provocan el uso del lock en una parte destacable de las transacciones. Esto también provoca un incremento del número de abortos explícitos por adquisición del lock. Teniendo en cuenta que la mayor parte de la aplicación se ejecuta con transacciones, el mal desempeño del sistema de MT provocará una degradación de rendimiento considerable.
- genome : muestra una mejora ascendente hasta los 8 hilos, estancándose en los 16. Muy probablemente el rendimiento empeore más allá de estos 16 hilos. Se observa como el número de abortos se mantiene siempre por debajo de las transacciones confirmadas (Figura 6.4), permitiendo a bastantes transacciones ejecutarse de forma paralela. Sin embargo, aunque la contención sea relativamente baja y en las pruebas realizadas los abortos nunca superan las confirmaciones , se observa que el número de abortos por conflicto se dobla por cada incremento en el número de hilos (Figura 6.2). Por lo tanto, es bastante probable que con 32 hilos los abortos ya superen las confirmaciones. Además, con este aumento se observa un leve incremento en el número de transacciones que utilizan el lock. Hay que tener en cuenta que genome se ejecuta en gran parte en modo transaccional, llevará al benchmark a empeorar el rendimiento. Su tamaño relativamente mediano del conjunto de lectura/escritura evita en gran medida la aparición de abortos por capacidad.

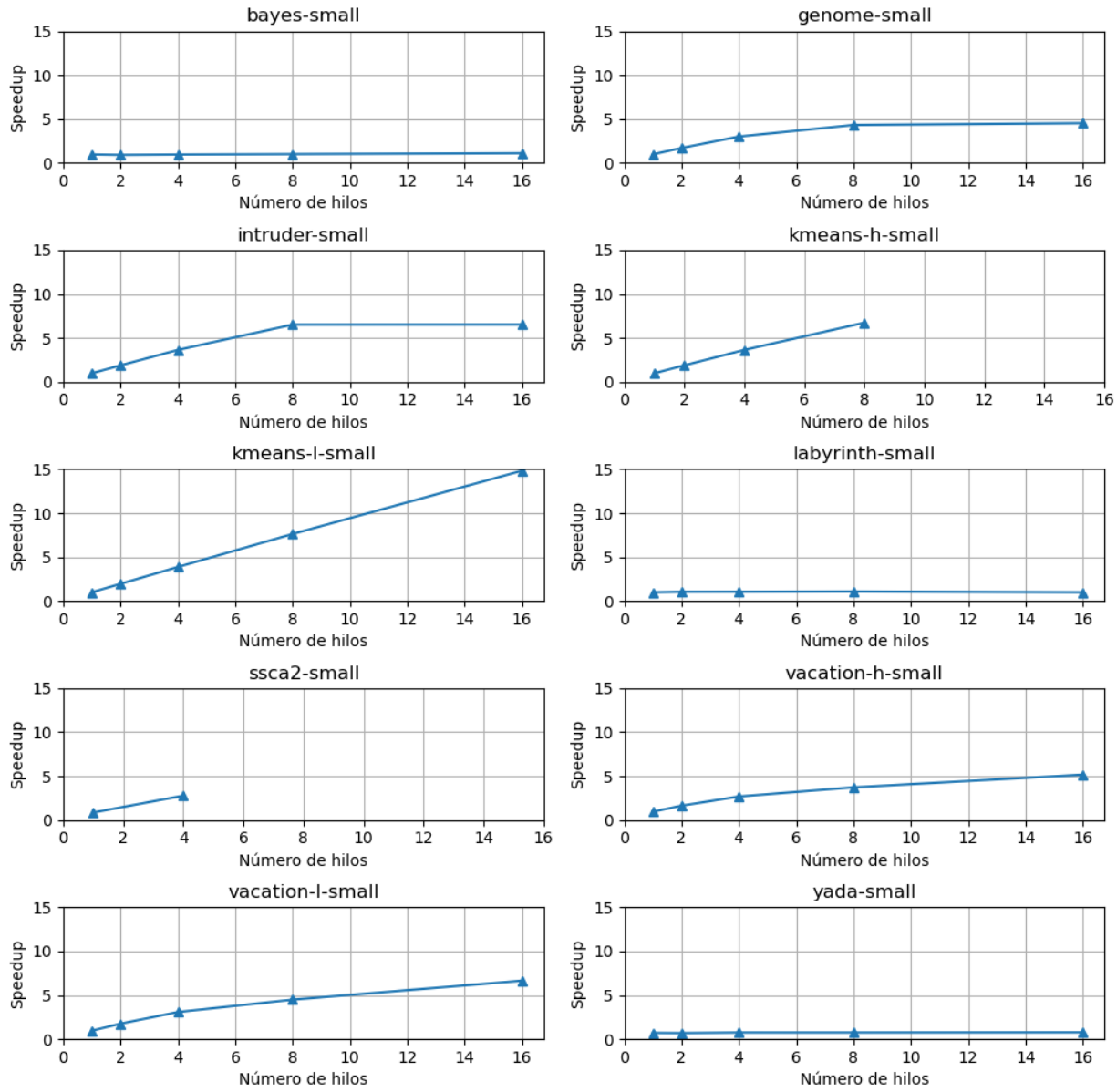

Figure 6.1: Gráfica que representa la mejora obtenida respecto a la versión secuencial sin memoria transaccional

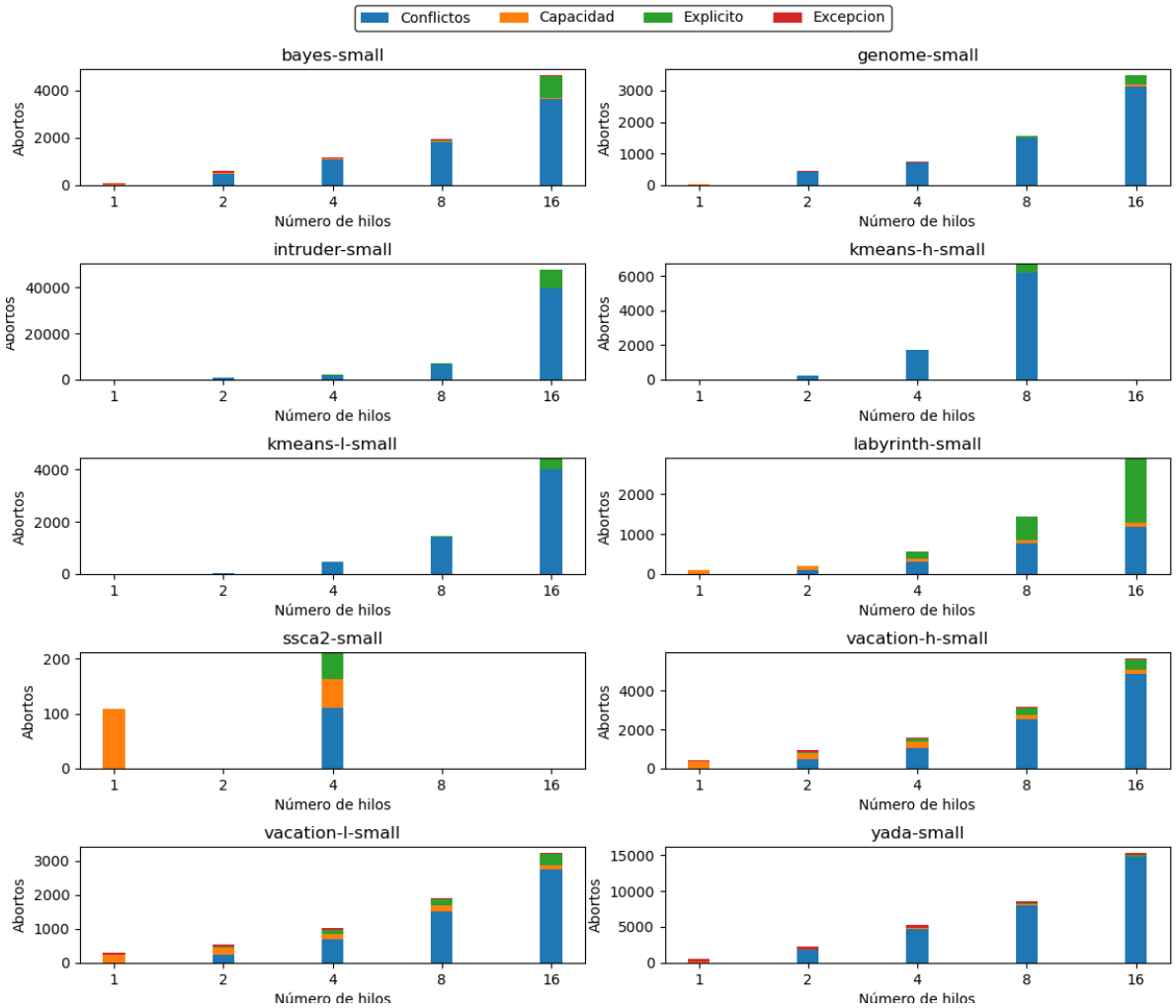

Figure 6.2: Gráfica que representa los resultados obtenidos de ejecutar los benchamrks con los parámetros indicados y aplicando la variación de hilos descrita. En el eje X se indican el número de hilos. En el eje Y, cuántos abortos de cada tipo.

bayes-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	498	1084	1834	3670
Capacidad	29	28	27	18	12
Explicito	0	2	9	69	938
Excepcion	79	83	65	46	41

genome-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	414	706	1487	3134
Capacidad	32	6	1	18	65
Explicito	0	4	18	56	283
Excepcion	16	16	14	13	10

intruder-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	662	1986	6614	39763
Capacidad	18	7	4	1	0
Explicito	0	11	73	571	8047
Excepcion	47	47	46	46	44

kmeans-h-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	238	1704	6220	0
Capacidad	0	0	0	0	0
Explicito	0	0	48	495	0
Excepcion	0	0	0	0	0

kmeans-l-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	24	453	1414	4025
Capacidad	0	0	0	0	0
Explicito	0	0	3	33	409
Excepcion	0	0	0	0	0

labyrinth-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	106	300	764	1180
Capacidad	96	96	96	93	96
Explicito	0	4	163	598	1629
Excepcion	0	0	0	0	0

ssca2-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	0	110	0	0
Capacidad	108	0	52	0	0
Explicito	0	0	49	0	0
Excepcion	0	0	0	0	0

vacation-h-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	468	1030	2507	4873
Capacidad	372	323	318	249	173
Explicito	0	64	155	332	546
Excepcion	65	69	68	67	67

vacation-l-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	238	679	1509	2760
Capacidad	230	199	175	170	112
Explicito	0	41	100	180	316
Excepcion	58	58	57	54	54

yada-small

	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos
Conflictos	0	1793	4738	8067	14684
Capacidad	130	122	130	119	94
Explicito	0	0	25	66	300
Excepcion	499	412	359	338	280

Figure 6.3: Tablas que muestran los abortos más en detalle.

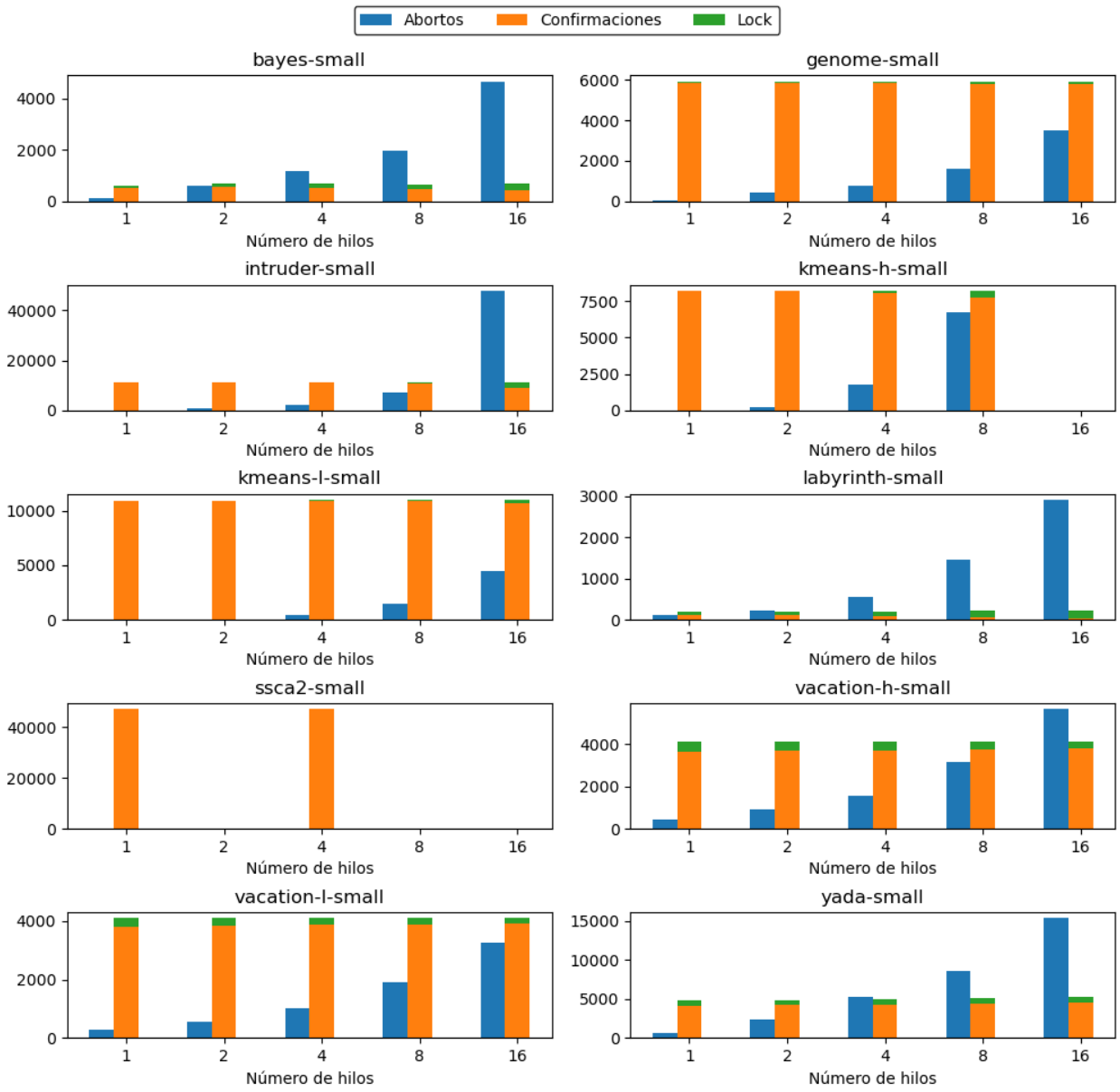

Figure 6.4: Gráfica que muestra las transacciones iniciadas, confirmadas, abortadas y ejecutadas con el lock global.

- intruder : este benchmark muestra una mejora ascendente hasta los 8 hilos. A partir de ese punto, la mejora se estanca, siendo bastante probable que si incrementamos aún más el número de hilos la mejora obtenida disminuya. En la figura 6.4 se observa un incremento considerable de abortos al cambiar de 8 a 16 hilos, lo que provoca una relación bastante elevada de abortos por transacción confirmadas coincidiendo además con la bajada de rendimiento. Además, un porcentaje considerable de transacciones se ejecutan a través del lock global. Esto se debe principalmente a que al tener una contención alta, el incremento del número de hilos conllevará un incremento considerable de conflictos(Figura 6.2). Aunque las transacciones sean cortas y no representen un porcentaje alto de la aplicación, el número de conflictos generados al utilizar 16 hilos es destacable.

- kmeans : este benchmark es el que mejor responde al aumento del número de hilos. El motivo principal es que el número de abortos por confirmación se mantiene bajo en todas las pruebas. En la ejecución de baja contención, se observa un incremento relevante en el número de abortos (Figura 6.4), mayoritariamente por conflicto (Figura 6.2), a partir de los 16 hilos, pero al tener transacciones de corta longitud, no afectarán en exceso al rendimiento. Sin embargo, también con 16 hilos, se observa un incremento de transacciones que se ejecutan con lock, lo cual podemos asociar al incremento de los abortos por conflicto. Por lo tanto, aunque los abortos en sí no penalicen demasiado, un número elevado provocará la adquisición del lock y serializando las transacciones. En la ejecución con alta contención se observa un mayor número de abortos por conflicto que provocan una menor mejora respecto a la versión secuencial. También se observa que ese incremento implica que de 8 a 16 hilos se muestren indicios de que un mayor número de hilos podría obtener peores resultados que utilizando menos hilos.

- labyrinth : este benchmark no muestra mejora con ningún número de hilos. Sus conjuntos de lectura/escritura de gran tamaño provocan un gran número de abortos por capacidad. Estos abortos provocan que, como mínimo, se ejecuten con el lock el 50% de las transacciones (Figura 6.4), impidiendo la paralelización de toda la aplicación, ya que gran parte se ejecuta con transacciones. Además, se produce un incremento de los abortos proporcional al número de hilos, que provoca también un incremento del uso del lock. . Esto a su vez provocará una gran cantidad de abortos explícitos, provocados por la adquisición del lock. Su alta contención provocará también un incremento del número de abortos por conflicto que seguirá aumentando conforme añadamos nuevos hilos. Si a todos estos hechos le sumamos que una gran parte de la sección paralelizada se ejecuta a través de transacciones, encontramos una situación por la que la aplicación no obtendrá mejora alguna.

- ssca2 : por desgracia, este benchmark únicamente se ha podido ejecutar con 1 y 4 hilos debido a problemas puntuales con el simulador. Sin embargo, al tener propiedades idénticas que kmeans y gracias a los resultados que se han podido obtener, podríamos decir que su comportamiento será similar a kmeans. Un conjunto de lectura/escritura reducido impedirá la aparición de abortos por conflicto, sobre todo conforme se aumenta el número de hilos. Sus transacciones cortas y su poca incidencia sobre el total de la aplicación, además de una baja contención, permitirá a ssca2 escalar bastante bien con el número de hilos.

- vacation : la mejora del tiempo de ejecución se mantiene al alza hasta los 16 hilos, aunque la gráfica ofrece muestras de aplanamiento, lo que podría generar un estancamiento o incluso una disminución de la mejora a partir de 32 hilos. El motivo principal se muestra en la figura 6.4, donde el número de abortos crece de forma casi proporcional al número de hilos. Con este aumento, se producen más abortos por transacción confirmada, lo que implica un mayor esfuerzo medio para confirmar una transacción. Si usáramos 32 hilos, los abortos superarían a las transacciones confirmadas, y se observaría como la mejora respecto a la versión secuencial sería menor que con 16 hilos. También se observa como el porcentaje de transacciones que se ejecutan con el lock es destacable respecto a las que se confirman en modo transaccional. El motivo es un número importante de abortos por capacidad. La figura 6.2 muestra como con el incremento del número de hilos, aumenta la relevancia de los abortos por conflicto, siendo éste el principal motivo de la bajada de rendimiento. Por lo tanto, si además tenemos en cuenta que la mayor parte de la aplicación se ejecuta con transacciones, esta bajada de rendimiento en la ejecución de estas secciones provocará una bajada casi proporcional en toda la aplicación. Con mayor contención, este problema se acrecienta, obteniendo menor mejora que con baja contención.

- yada : se comporta de forma muy pobre, empeorando en todo momento la versión secuencial y sin vistas a una posible mejora. Se observa (Figura 6.4) como una parte relevante de las transacciones deben ejecutarse con el lock, impidiendo la paralelización de estas secciones. Además, se incrementa el número de abortos con cada incremento de hilos. A partir de los 4 hilos los abortos ya superan a las transacciones confirmadas. A partir de este punto, con cada aumento de hilos esta diferencia será mayor. En la figura 6.2 se observa como el causante del aumento del número de abortos son los conflictos. Hay que recordar que yada tiene una contención alta, y por ello seguirán aumentando de forma considerable. También hay que tener en cuenta los abortos por excepción, provocados por fallos de página, junto con los abortos por capacidad, generados por el tamaño de los conjuntos de lectura/escritura. Provocarán el uso del lock en bastantes transacciones. Al ejecutarse casi toda la aplicación en modo transaccional, todos estos problemas tienen un gran impacto sobre el rendimiento general de la aplicación, obteniendo los tiempos mencionados.

6.2 Asociatividad

El utilizar la caché de datos L1 como buffer de escritura especulativa, obliga a tener en cuenta varias características de estas cachés a la hora de observar el comportamiento del sistema de Memoria Transaccional. Una de estas características es la asociatividad. Una caché asociativa permite almacenar un bloque entre los bloques de uno de los conjuntos en los que se divida la caché. Esto permitirá reducir los fallos de caché al tener mayor flexibilidad a la hora de colocar el bloque. Una asociatividad mayor implica más conjuntos, y por lo tanto, un reparto más equilibrado de los bloques. En Memoria Transaccional, el remplazo de un bloque en el conjunto de lectura/escritura provocará un aborto por capacidad. Este tipo de aborto es habitual que provoque la serialización de la transacción, produciendo una bajada de rendimiento. Aumentar la asociatividad permitirá, en principio, reducir estos abortos. Se utilizarán las gráficas 6.5 y 6.7 para mostrar el comportamiento de los benchmarks variando la asociatividad de la caché L1.

Como podemos observar en la gráfica 6.5, todos los benchmarks excepto labyrinth reducen los abortos por capacidad con aumento de la asociatividad. La razón más que probable de que labyrinth no se aproveche de estos cambios es que los conjuntos de lectura/escritura tienen un tamaño bastante grande. En el resto de benchmarks se observa que la asociatividad 4 tiene un rendimiento similar a la 8, excepto en yada. Incluso con asociatividad 2, intruder y kmeans tienen un desempeño similar que con asociatividad 4. Además, los benchmarks en los que se observa una menor reducción son los que mayores conjuntos de lectura/escritura tienen, a excepción de bayes, el cual se beneficia mucho más. Esto indica que el tamaño utilizado para la L1 podría resultar no suficiente para grandes conjuntos de lectura/escritura.

En determinados benchmarks, como genome o vacation, se da la circunstancia de que la disminución de los abortos por capacidad provoca una reducción considerable de los abortos por conflicto. La razón más probable es que al ejecutarse una transacción con el lock global, esta abortará todas las transacciones con las que entre en conflicto, pero ninguna podrá abortarla a ella. Si se ejecutará como transacción, otras transacciones también podrían invalidarla. Este intercambio de abortos puede generar un orden en la ejecución de las transacciones tal que se reduzcan los conflictos. Como hemos dicho, esto no ocurre en todos los benchmarks, por lo que este suceso depende de cómo se realicen los accesos a memoria, es decir, depende de la aplicación. El resto de benchmarks mantienen unos abortos por conflictos similares a lo largo del cambio de asociatividad, que podría suponerse como el comportamiento habitual, ya que en principio, las causas del aborto por conflictos son distintas del aborto por capacidad.

Para saber porque disminuyen los abortos por capacidad, se va a utilizar la figura 6.6. En ella se muestran los tamaños medios de los conjuntos de lectura/escrituran tanto al abortar como al confirmar, por nivel de asociatividad.

Un indicio fundamental de como afecta la asociatividad es como la media de tamaño de transacciones confirmadas aumenta, indicando que han podido confirmar transacciones de mayor tamaño, presumiblemente

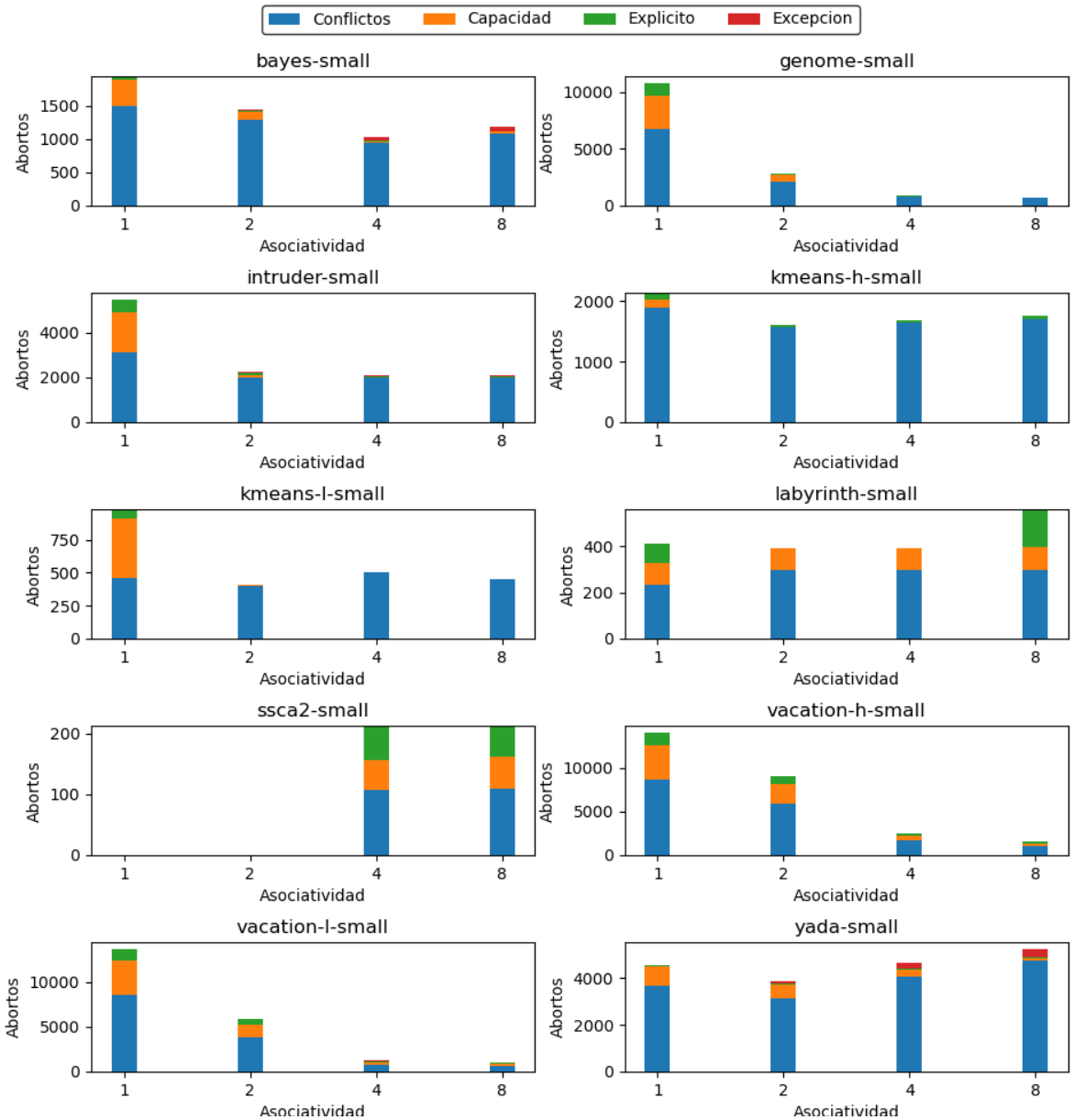

Figure 6.5: Gráfica que representa el número de abortos en función del nivel de asociatividad de la caché L1.

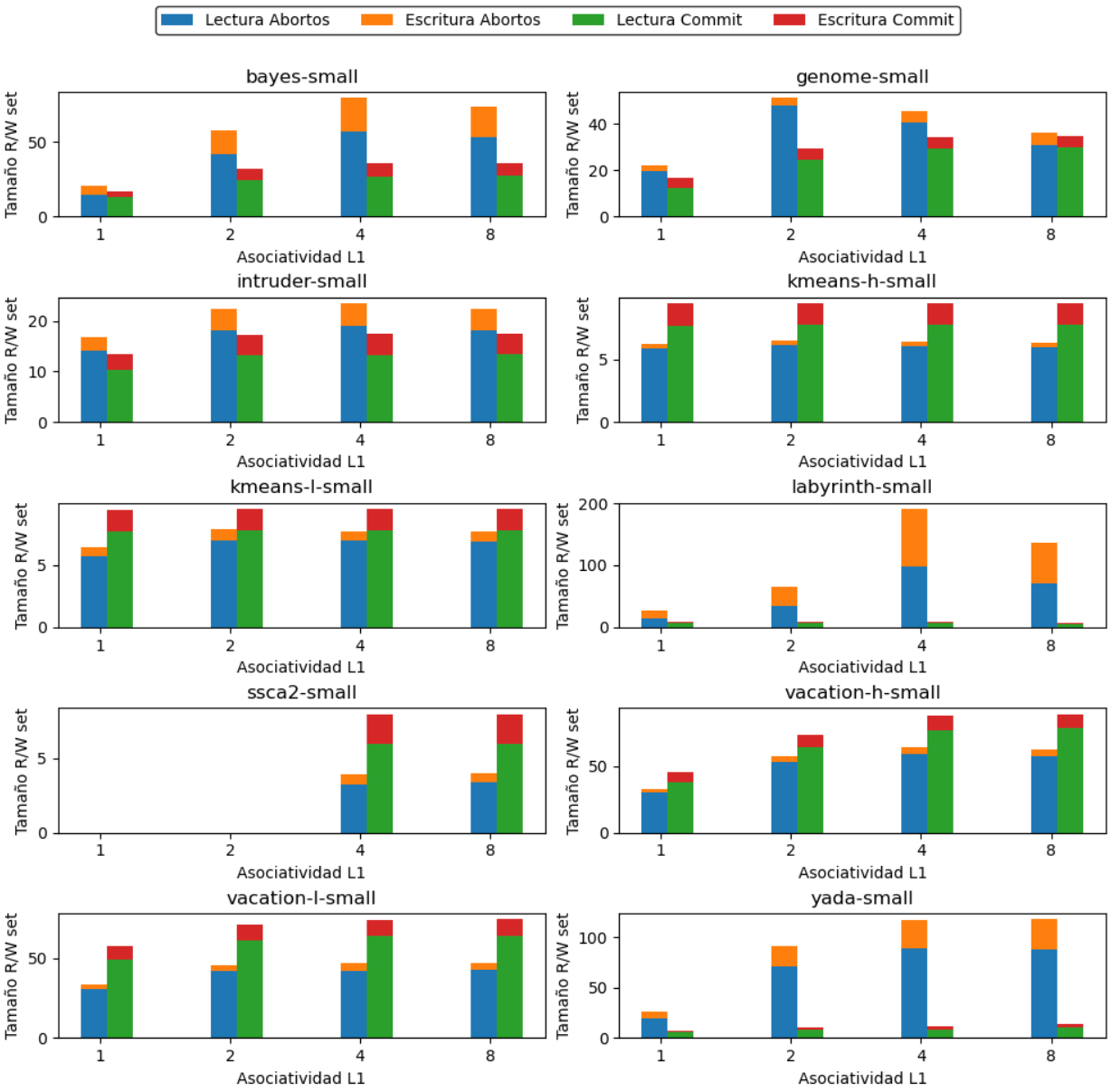

Figure 6.6: Gráfica que muestra tamaños medios de los conjuntos de lectura/escritura tanto de las transacciones abortadas como confirmadas.

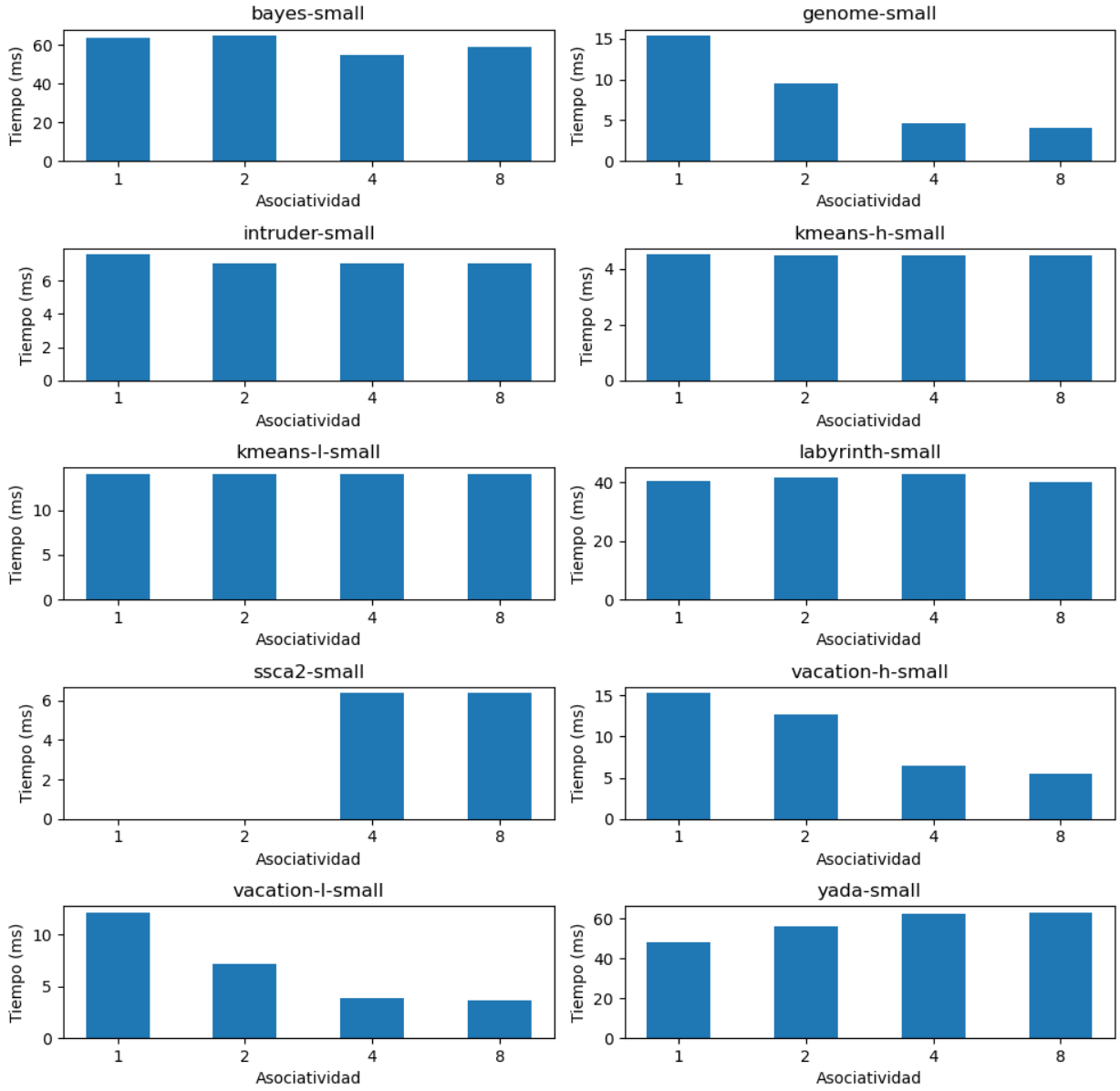

Figure 6.7: Gráfica que muestra los tiempos de ejecución en función del nivel de asociatividad de la caché L1.

por el aumento de la asociatividad. Loa benchmarks con conjuntos pequeños no muestran variación en las medias de tamaños. Esto indica que en el benchmark predominan los abortos por conflicto, de ahí que aunque se reduzcan los abortos por capacidad, la media por abortos no varíe. Además, la de confirmación tampoco lo hace por ser los abortos por capacidad muy pequeños respecto al total de transacciones que se confirman. Otros benchmarks si que muestran un aumento de la media de los tamaños. Esto indica que los abortos por capacidad representan una parte relevante de los abortos totales, y que el aumentar la asociatividad ha permitido que determinados abortos por capacidad ya no ocurran. En la mayoría de benchmarks, a partir de cierto punto no se observan variaciones. Esto indica que los abortos por capacidad se han reducido lo máximo posible, y el aumento de la asociatividad solo implicará mayor complejidad en el procesador. En casos de un conjunto de lectura/escritura grande, como yada o labyrinth, se puede observar claramente como, aunque la media de tamaño de las transacciones abortadas aumenta, no lo hace apenas la de las confirmadas. Esto indica que el tamaño del buffer de escritura especulativa puede no ser suficiente para benchmarks con conjuntos de lectura/escritura excesivamente grandes.

Una vez sabemos que los abortos por capacidad se pueden reducir con una mayor asociatividad, resulta interesante saber como afecta esto al tiempo de ejecución. Como podemos observar, los únicos que se benefician de forma clara del aumento de la asociatividad reduciendo sus tiempos de ejecución son los benchmarks genome y vacation. Son los que obtienen una bajada más significativa de abortos por capacidad, y además, los únicos benchmarks en los que también se reducen de forma considerable los abortos por conflicto, como se ha descrito anteriormente. Los benchmarks con conjuntos de lectura/escritura pequeños, como kmeans y ssca2, no mejoran en exceso al tener los abortos por capacidad poca relevancia. En bayes se observa como, aunque los abortos por capacidad se reducen a un número bajo, la alta contención provoca un gran número de abortos por conflicto que limitan el margen de mejora del benchmark al reducir los abortos por capacidad. En yada el tiempo de ejecución va empeorando conforme se aumenta la asociatividad. Esto se debe a que al reducir los abortos por capacidad se ejecutarán más transacciones sin el lock, por lo que los conflictos que generen pueden conllevar una mayor penalización. El benchmark labyrinth, como cabía de esperar, no varía su tiempo de ejecución. El no reducir los abortos por capacidad mantiene un rendimiento constante. Por último, intruder se mantiene en tiempos similares. Aunque se observa una reducción considerable de abortos por capacidad al pasar el nivel de asociatividad de 1 a 2, la mejora de tiempo es relativamente baja. Un posible motivo es que intruder tiene una longitud de transacción corta, por lo que aunque se tenga que utilizar el lock por aborto por capacidad para una transacción, el resto de transacciones tendrá que esperar poco tiempo.

Chapter 7

Conclusiones y vías futuras

Los sistemas de Memoria Transaccional (MT) permiten a los programadores reducir la complejidad de nuevas aplicaciones que intenten aprovechar al máximo el paralelismo. Permiten garantizar la atomicidad de las secciones críticas a través de una interfaz muy sencilla. Su alternativa más directa son los locks, que a gran escala requerirán de una gran destreza por parte del programador y una gran cantidad de tiempo.

Cada sistema de MT tendrá unas propiedades concretas, que le harán despuntar en la ejecución de determinadas aplicaciones, pero que lastrarán la ejecución de otras. En este trabajo se ha propuesto una implementación concreta de MT y se ha ampliado el ISA RISCV para poder utilizarlo. Aunque existen distintas alternativas, se ha escogido el simulador gem5 para facilitar el prototipado del nuevo sistema de MT y obtener estadísticas de su rendimiento para su posterior evaluación.

Primero se ha determinado qué instrucciones eran necesarias para añadir soporte del sistema MT a RISCV. Una instrucción que inicie la transacción y otra que la confirma serán fundamentales para implementar una transacción. Aunque con estas dos instrucciones hubiera sido suficiente, se ha decidido añadir una tercera que permita cancelar transacciones por un motivo distinto a los conflictos. Se les ha asignado una codificación y un comportamiento compatible con la especificación oficial de RISCV, por lo que puede considerarse como un prototipo de la extensión T de RISCV.

A partir de esta especificación, se han implementado estas instrucciones en el simulador gem5. Realizando una comparativa entre sistemas de Memoria Transaccional incluidos en distintos ISAs, se ha decidido utilizar las características más utilizadas. Se aplicará una detección de conflictos ansiosa, permitiendo resolver el conflicto lo antes posible. El control de versiones se aplicará de forma perezosa, utilizando la caché L1 como buffer de escritura especulativa. Por último, se seguirá un modelo *best effort* para el manejo de los abortos. Un sistema de MT de estas características ya se encuentra implementado en gem5, por lo que se ha podido reutilizar.

Por último, se ha evaluado el sistema de Memoria Transaccional utilizándolo en los benchmarks de STAMP. Por un lado, se ha realizado un estudio de escalabilidad en el que se comprueba como responde el sistema conforme se aumentan el número de hilos. Por otro lado, se ha comprobado como afecta la asociatividad de las cachés en la tasa de abortos por capacidad. Previamente, se han tenido que realizar algunas modificaciones en STAMP para poder utilizar nuestro sistema de MT.

El primer estudio muestra una relación directa entre el número de hilos y los abortos por conflicto. Un mayor número de hilos provocará un aumento de los abortos por conflicto. En los casos que el aumento sea destacable respecto al total de transacciones, se producirá una degradación del rendimiento. El segundo estudio permite observar que, a mayor asociatividad de la caché L1, menores serán los abortos por capacidad. Dependerá de la aplicación si la reducción de los abortos por capacidad permitirán mejorar el rendimiento.

Durante el desarrollo del trabajo, se han encontrado una serie de errores en el simulador que han afectado al desarrollo del trabajo. El más relevante impedía la correcta ejecución de las instrucciones atómicas de RISCV cuando se utilizaba Ruby como modelo de memoria. También se ha detectado un error en el procesamiento de los códigos de aborto. Se ha solucionado y se ha propuesto esta solución en el repositorio oficial de gem5, siendo aceptada e incluida en la versión 21.1 del simulador. También se han detectado otros fallos, pero que no han podido solucionarse, y que han evitado obtener algunos resultados de la evaluación.

Una de las posibles vías que se abren a partir de este trabajo es ampliar el soporte de Memoria Transaccional de gem5 añadiendo distintas implementaciones de cada una de las propiedades de un sistema de MT. Por ejemplo, resultaría interesante implementar una detección de conflictos perezosa. De esta forma, se podrán comparar la aproximación ansiosa ya implementada con la perezosa sobre un mismo conjunto de aplicaciones, y determinar cuál es el adecuado.

Otra de las posibles vías es incluir la detección de conflictos en otro protocolo de coherencia. Así, se podrá evaluar de forma más precisa la ejecución de aplicaciones paralelas sobre memoria transaccional en un procesador concreto. Puede ser necesario incluir el protocolo, ya que gem5 no implementa todos los protocolos existentes.

Se podrá también incluir nuevas instrucciones al sistema de MT que flexibilicen su utilización. Por ejemplo, las instrucciones Early Release permitirán liberar un recurso de los conjuntos de lectura/escritura antes de la confirmación, permitiendo evitar determinados conflictos. Una vez incluidas las instrucciones, resultaría interesante evaluar su impacto sobre el rendimiento de las aplicaciones.

Por último, otra posible vía sería ampliar los benchmark de STAMP. El trabajo consistiría en encontrar aplicaciones que cubran el mayor número de ámbitos posibles, y incluir memoria transaccional a través de la API que define STAMP. Esto brindará a STAMP de un mayor abanico de posibilidades a la hora de obtener una evaluación aproximada de otra aplicación real.

Bibliography

[1] Electronics Weekly : Intel accelerates dual core, cancels Tejas and Jayhawk. ONLINE : https://www.electronicsweekly.com/news/products/micros/intel-accelerates-dual-core-cancels-tejas-and-jayhawk-2004-05/

[2] Tim Harris, James Larus, and Ravi Rajwar : Transactional Memory, 2nd edition. ISBN: 9781608452354 (paperback)

[3] Schindewolf, Martin & Cohen, Albert & Karl, Wolfgang & Marongiu, Andrea & Benini, Luca. (2009). Towards Transactional Memory Support for GCC.

[4] TinySTM. ONLINE : https://github.com/patrickmarlier/tinystm

[5] Knight, Tom. (1986). An architecture for mostly functional languages. 105-112. 10.1145/319838.319854.

[6] Herlihy, Maurice & Eliot, J. & Moss, Eliot. (1993). Transactional Memory: Architectural Support For Lock-free Data Structures. ACM SIGARCH Computer Architecture News. 21. 289-300. 10.1109/ISCA.1993.698569.

[7] Chaudhry, Shailender & Cypher, Robert & Ekman, Magnus & Karlsson, Martin & Landin, Anders & Yip, Sherman & Zeffer, Håkan & Tremblay, Marc. (2009). Rock: A high-performance sparc CMT processor. Micro, IEEE. 29. 6 - 16. 10.1109/MM.2009.34.

[8] Wang, Amy & Gaudet, Matthew & Wu, Peng & Amaral, José & Ohmacht, Martin & Barton, Christopher & Silvera, Raúl & Michael, Maged. (2012). Evaluation of Blue Gene/Q hardware support for Transactional Memories. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT. 127-136. 10.1145/2370816.2370836.

[9] Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8

[10] Intel. ONLINE : https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-overview.html

[11] Intel. ONLINE : https://www.intel.es/content/www/es/es/support/articles/000055673/processors.html

[12] ARM Transactional Memory Extension (ARM). ONLINE : https://community.arm.com/developer/research/b/articles/posts/arms-transactional-memory-extension-support-

[13] Advanced Micro Devices. Advanced Synchronization Facility: Proposed Architectural Specification. ONLINE : http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf

[14] Damron, Peter & Fedorova, Alexandra & Lev, Yossi & Luchangco, Victor & Moir, Mark & Nussbaum, Daniel. (2006). Hybrid Transactional Memory. ACM SIGPLAN Notices. 41. 336-346. 10.1145/1168857.1168900.

[15] Waterman, A. S. (2016). Design of the RISC-V Instruction Set Architecture. UC Berkeley. ProQuest ID: Waterman_berkeley_0028E_15908. Merritt ID: ark:/13030/m50c9hkd. Retrieved from https://escholarship.org/uc/item/7zj0b3m7

[16] NVIDIA RISC-V Story. ONLINE : https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf

[17] Western Digital (WD). RISC-V: Accelerating Next-Generation Compute Requirements. ONLINE : https://www.westerndigital.com/company/innovations/risc-v

[18] Repositorio GitHub RISCV. ONLINE : https://github.com/riscv

[19] RISCV Rocket chip. Berkeley : https://github.com/chipsalliance/rocket-chip

[20] Lowe-Power, Jason & Ahmad, Abdul & Akram, Ayaz & Alian, Mohammad & Amslinger, Rico & Andreozzi, Matteo Maria & Armejach, Adrià & Asmussen, Nils & Bharadwaj, Srikant & Black, Gabe & Bloom, Gedare & Bruce, Bobby & Carvalho, Daniel & Castrillón, Jerónimo & Chen, Lizhong & Derumigny, Nicolas & Diestelhorst, Stephan & Elsasser, Wendy & Fariborz, Marjan & Zulian, Éder. (2020). The gem5 Simulator: Version 20.0+.

[21] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovi´c, RISC-V Foundation, December 2019

[22] Gerrit : https://gem5-review.googlesource.com/c/public/gem5/+/47939

[23] Minh, Cao & Chung, JaeWoong & Kozyrakis, Christos & Olukotun, Kunle. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. 35-46. 10.1109/IISWC.2008.4636089.