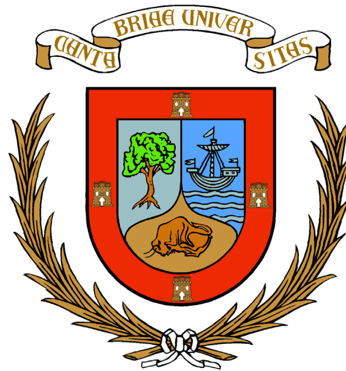
Universidad de Cantabria

Departamento de Electrónica y Computadores



Tesis Doctoral

SOPORTE ARQUITECTÓNICO A LA SINCRONIZACIÓN IMPARCIAL DE LECTORES Y ESCRITORES EN COMPUTADORES PARALELOS

Enrique Vallejo Gutiérrez

Santander, Marzo de 2010

Universidad de Cantabria

Departamento de Electrónica y Computadores

Dr. Ramón Beivide Palacio

Catedrático de Universidad

Y

Dr. Fernando Vallejo Alonso

Profesor Titular de Universidad

Como directores de la Tesis Doctoral:

ARCHITECTURAL SUPPORT FOR PARALLEL COMPUTERS WITH FAIR READER/WRITER SYNCHRONIZATION

realizada por el doctorando Don Enrique Vallejo Gutiérrez, Ingeniero de Telecomunicación, en el Departamento de Electrónica y Computadores de la Universidad de Cantabria, autorizan la presentación de la citada Tesis Doctoral dado que reúne las condiciones necesarias para su defensa.

Santander, Marzo de 2010

Fdo: Ramón Beivide Palacio

Fdo. Fernando Vallejo Alonso

Agradecimientos

Son tantas las personas a las que tengo que agradecer su ayuda y apoyo que esta sección debiera ser la más larga de la tesis. Por ello, no pretendo convertirla en una lista exhaustiva, que además, siempre se olvidaría de alguien importante. Ahí van, por tanto, mis agradecimientos condensados.

A Mónica, por su apoyo incondicional, culpable última de que tengas esta tesis en tus manos.

A mi familia, especialmente a mis padres, por el soporte y apoyo recibido.

A mis directores de tesis, Mon y Fernando. Aunque esto sea una carrera en solitario, es necesaria una buena guía para llegar al final.

A los revisores de la tesis, tanto los revisores externos oficiales, Rubén González y Tim Harris, como todos aquellos que se molestaron en leer el trabajo y ayudaron con sugerencias y posibles mejoras.

A los compañeros del Grupo de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria, por la ayuda y colaboración en el desarrollo del trabajo, en especial a Carmen Martínez.

A la gente del BSC-MSR Center y de la UPC, en especial a Mateo Valero y Adrián Cristal. Sin su trabajo previo en Kilo-Instruction Processors esta tesis hubiera sido imposible; pero también lo hubiera sido sin su guía, ayuda y apoyo y dirección durante su desarrollo. També voldria agrair l'ajuda i col·laboració dels companys del Departament d'Arquitectura de Computadors de la UPC, especialment a Miquel Moretó, i del BSC. You all know who you are, locos.

I meet a lot of bright and helpful people during my internship in the Systems and Networking Group in Microsoft Research Cambridge, and they all deserve a sincere acknowledgement. But specially, I had the luck to work under the supervision of Tim Harris. He deserves a double acknowledgement. First, for his help, support and encouraging during my internship and the subsequent work. Second, because most of the work presented here would have been impossible without his prior pioneering work on nonblocking synchronization and Transactional Memory. Thanks, Tim.

A todos mis amigos que tuvieron que sufrir de vez en cuando alguna aburrida explicación (culpa suya, por preguntar…) y supieron recordarme que de vez en cuando que existe un mundo real al otro lado de la pantalla.

This work has been partially supported by the Spanish Ministry of Education and Science / Ministry of Science and Innovation, under contracts TIN2004-07440-C02-01, TIN2007-68023-C02-01, CONSOLIDER Project CSD2007-00050 and grant AP-2004-6907, by Microsoft Research Cambridge and by the joint BSC-Microsoft Research Centre. The author would also like to thank the support of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

Para Mónica

Resumen

Resumen

La evolución tecnológica en el campo de los microprocesadores nos ha llevado a sistemas paralelos con múltiples hilos de ejecución simultánea, o *threads*. Estos sistemas son más difíciles de programar y presentan sobrecargas en la ejecución mayores que los sistemas uniprocesadores tradicionales, lo que puede limitar su rendimiento y escalabilidad. Estas sobrecargas se deben principalmente a los mecanismos de sincronización, el protocolo de coherencia y el modelo de consistencia y otros detalles requeridos para garantizar una ejecución correcta.

El Capítulo 1 de esta tesis hace una introducción a los mecanismos necesarios para diseñar y hacer funcionar tal sistema paralelo, detallando en gran medida el trabajo reciente del área. En primer lugar, el apartado 1.1 estudia los mecanismos que hacen posible la construcción de sistemas de memoria común: El protocolo de coherencia y el modelo de consistencia del sistema. Estos dos mecanismos garantizan que los diferentes procesadores tienen una visión uniforme de la memoria del sistema, y especifican cómo puede comportarse el sistema cuando hay carreras en el acceso a datos compartidos. A continuación, en el apartado 1.2 se exponen los fundamentos de la programación paralela en tal modelo de memoria común, con énfasis en los mecanismos de sincronización: barreras y secciones críticas protegidas por *locks*. Se discuten las principales dificultades que presenta esta programación paralela con respecto a la programación *single-threaded*: la ley de Ahmdal, que limita la escalabilidad del sistema por la componente secuencial de los programas; el compromiso en cuanto al nivel de granularidad del paralelismo, en que una granularidad más fina permite más paralelismo pero presenta más *overhead* en la ejecución; los problemas de *deadlock* (interbloqueo), inversión de prioridad y *starvation* (inanición), inherentes a sistemas basados en *locks*; y finalmente, se analiza la complejidad del diseño de estructuras de datos concurrentes, especialmente por dos problemas: la dificultad de ampliar una estructura de datos con nuevos métodos sin conocer en detalle la implementación del resto de métodos, y la falta de *composibilidad*, es decir, que la construcción de una estructura de datos de un nivel superior, a partir de estructuras basadas en *locks* más sencillas, puede dar lugar a *deadlock*.

Existen múltiples implementaciones alternativas de *locks*, que se revisan en el apartado 1.3. En esencia, un *lock* es una abstracción que protege una cierta zona de memoria o parte del

código, de forma que sólo un *thread* pueda acceder a la zona protegida. Alternativamente, existen *locks* de lectura/escritura, que permiten un único escritor que modifique los datos, o múltiples lectores que accedan a la vez a los datos compartidos sin modificarlos. Hay implementaciones centralizadas, en que el *lock* es una única palabra de memoria a la que se accede mediante operaciones atómicas del procesador. Una implementación así genera contienda: todos los *threads* que quieren tomar el *lock* acceden a la misma posición de memoria, lo que implica múltiples invalidaciones a nivel de coherencia, empeorando el rendimiento del sistema. Alternativamente, existen implementaciones distribuidas, como los *locks* basados en colas: cada *thread* que quiera tomar el *lock* reserva una estructura denominada "nodo" en memoria, y los diferentes nodos de cada *thread* se enlazan en forma de una cola de solicitantes. En estos diseños el *lock* se pasa en orden de un nodo al siguiente, lo que evita la contienda, y además garantiza el *fairness*: existe una política que garantiza que todos los *threads* que pretenden tomar un *lock* lo consiguen hacer en un tiempo razonable, sin producirse *starvation*. En este apartado 1.3 se discuten también múltiples alternativas de *locks*, como aquellos que abortan si el *lock* no se toma en un cierto periodo de tiempo (*trylocks*), las implementaciones jerárquicas, o *locks* diseñados para comportarse especialmente bien en caso de existir un comportamiento patológico, como ser adquirido siempre por el mismo *thread* (*biased locks*).

Una propuesta relativamente reciente que aborda los problemas de programación paralela es la Memoria Transaccional (*Transactional Memory*, TM), que se introduce en el apartado 1.4. Un sistema de Memoria Transaccional proporciona al programador la abstracción de una transacción con la que puede realizar modificaciones en memoria compartida. Las transacciones en memoria son análogas a las transacciones de las bases de datos, garantizando las propiedades de atomicidad (se ejecuta toda la transacción, o se aborta toda la transacción, pero no parte de ella) y aislamiento (ningún *thread* es capaz de ver el estado intermedio de otra transacción) al código que corre dentro de la transacción. Esto permite eliminar la gestión de *locks* de la tarea de programación al sustituir secciones críticas por transacciones, evitar el problema de *deadlock*, simplificar la modificación de código existente y proporcionar *composibilidad*.

Para garantizar una ejecución correcta, el sistema de Memoria Transaccional debe ejecutar el código transaccional de manera especulativa, almacenando los cambios que realiza la transacción (el *write-set*) en algún tipo de buffer, y utilizar un mecanismo de validación (*commit*) que garantice que los cambios se hacen visibles a nivel global (al resto de *threads*) de forma atómica, sin que hayan ocurrido cambios en las posiciones de memoria leídas (el *read-set*). Además, el sistema debe detectar las violaciones: si dos transacciones acceden de manera concurrente a los mismos datos en memoria (solapándose parcialmente el *write-set* de una de ellas con el *read-set* o el *write-set* de la otra) una de las transacciones debe pararse y esperar al *commit* de la otra, o abortar y reiniciarse. El mecanismo que decide la acción cuando se detecta un conflicto se denomina *contention manager*.

Existen múltiples implementaciones de Memoria Transaccional, tanto en software (STM), como propuestas Hardware (HTM) o propuestas tanto híbridas como aceleradas por *hardware* (HyTM o HaTM), que ejecutan parte de las operaciones (como el versionado de los datos) en software, y otra parte (como la detección de conflictos) mediante hardware específico. Los sistemas hardware proporcionan un rendimiento mucho mayor, al eliminar ciertas tareas que pueden requerir un tiempo de ejecución lineal con el tamaño del *read/write-set*. Sin embargo, en el momento de escribir este texto, no existe todavía ninguna implementación comercial de sistemas HTM; El procesador Rock de Sun [22] pretendía soportar un sistema híbrido, pero su producción ha sido cancelada.

En el apartado 1.4 se discuten múltiples detalles que condicionan o limitan una implementación de Memoria Transaccional. Uno de estos detalles es el problema de la privatización: Los datos compartidos pueden disponer de alguna variable que permita hacerlos privados y acceder a ellos sin necesidad de utilizar transacciones. Sin embargo, en múltiples implementaciones de STM pueden darse carreras de datos que hacen que los datos se modifiquen simultáneamente desde una transacción y desde el código externo, sin que se advierta la colisión. En el apartado 1.4.6 se discuten el problema y diferentes soluciones. Otro detalle es la política utilizada en el *contention manager*. En general, en sistemas HTM el *contention manager* es muy simple, utilizando una política sencilla que elige una transacción u otra en función de algún parámetro, como pueda ser el uso de un *timestamp*. Otra dificultad puede ser la ejecución de operaciones irrevocables, es decir, aquellas que no pueden deshacerse en caso de tener que abortar la transacción. Ejemplos de estas operaciones son la entrada/salida o las operaciones del sistema que operan sobre procesos. Para gestionarlo, el sistema debe soportar las transacciones irrevocables, es decir, transacciones que se garantiza que van a finalizar sin abortar, y que el *contention manager* siempre elige con preferencia.

Aunque la Memoria Transaccional libera al programador de la complejidad de la gestión de *locks*, muchos de los sistemas STM actuales están internamente basados en *locks* de escritura [36, 46, 124]. Una implementación así, aunque puede bloquearse puntualmente (cuando un *thread* que ha tomado un *lock* es sacado temporalmente de ejecución), es más sencilla y flexible que una implementación no bloqueante [42]. Además, Dice y Shavit discuten [34] las ventajas de utilizar una implementación basada en *locks* de lectura/escritura, básicamente: es inmune al problema de la privatización; presenta una implementación interna sencilla, sin costosos procesos de validación; mejor *fairness* al implementar lectores visibles, especialmente en transacciones largas; y soporte sencillo para operaciones irrevocables. Sin embargo, el rendimiento de un sistema así está limitado por el alto coste de tomar *locks* de lectura en todas las posiciones de memoria leídas en la transacción.

Finalmente, el apartado 1.5 se centra en la evolución del diseño de los microprocesadores. La tecnología actual permite niveles de integración que no merece la pena explotar con un único hilo de ejecución, ya que las mejoras de la arquitectura según aumenta el número de

transistores proporcionan beneficios marginales. El principal problema es que las implementaciones actuales incorporan múltiples elementos que no se pueden escalar sin afectar al tiempo de ciclo, como las memorias direccionables por contenido (*Content-Adressable Memory*, CAM) utilizadas en las colas de Load-Store (*Load-Store Queue*, LSQ) o el buffer de reorden (*Reorder Buffer,* RoB), o el tamaño del banco de registros. Esto hace que no pueda aumentarse el tamaño de la ventana de instrucciones (número de instrucciones en vuelo en el procesador) de forma acorde con el aumento del tiempo de acceso a memoria.

En este apartado 1.5 se citan un conjunto de técnicas recientes que pretenden proporcionar una arquitectura con un tamaño de ventana de instrucciones muy grande, del orden del millar de instrucciones, para atacar el aumento en el tiempo de acceso a memoria principal. Conjuntamente, una arquitectura con estas técnicas se ha denominado *Kilo-Instruction Processor* [30]. Específicamente, se proponen arquitecturas basadas en *checkpoints* que eliminan el RoB y realizan la validación de todas las instrucciones correspondientes al mismo *checkpoint* conjuntamente; Mecanismos de reducción del número de registros físicos necesarios para un cierto tamaño de ventana; implementaciones jerárquicas de la LSQ, acordes con la arquitectura de *checkpoints*; y otras técnicas que permiten la escalabilidad de los recursos del procesador y su ventana de instrucción para que soporte latencias más grandes. Conjuntamente, estas técnicas permiten explotar el paralelismo a nivel de instrucción en el procesador, a pesar de la existencia de grandes latencias en la memoria principal.

Todos los aspectos citados condicionan el rendimiento y el diseño de un sistema paralelo de memoria común. El objetivo de esta tesis es introducir una serie de técnicas, principalmente en *hardware*, para acelerar la ejecución de estos programas paralelos.

El Capítulo 2 introduce un sistema de Memoria Transaccional híbrido basado en *locks* de lectura/escritura. Se ha comentado las ventajas de un sistema basado en estos *locks*, así como el problema de rendimiento que presentan en un STM. Para abordar este aspecto, se propone una modificación al *runtime* de un STM tal que, en caso de disponer de un sistema HTM genérico, la mayoría de la ejecución se aproveche de sus capacidades. El STM base utiliza la versión de lectura/escritura del *lock* MCS [107]. En general, se mantiene un *lock* por cada objeto transaccional y se permite la ejecución de transacciones *software* o *hardware* concurrentemente. Para ello, se añaden los siguientes cambios:

- Si el soporte *hardware* está presente, se comienza una transacción *hardware* junto con la transacción *software*.

- En cada acceso a un objeto desde una transacción *hardware*, se verifica el estado del *lock*, para detectar conflictos con transacciones que se ejecuten en *software*, y por tanto que tomen los *locks* en el modo correspondiente. Para ello, la cabecera del *lock* se modifica de forma que la transacción hardware pueda detectar conflictos con lectores o escritores sin necesidad de seguir la cola de punteros del *lock*. Si se detecta

un conflicto con una transacción *software*, la transacción *hardware* aborta, ya que no puede esperar a que la otra llegue al *commit*.

- En cada acceso a un objeto transaccional, el versionado y la detección de conflictos se pueden delegar al HTM. En el apartado 2.3 se discuten diferentes alternativas que descargan diferente cantidad del trabajo en el HTM, y proporcionan diferente rendimiento, junto con las implicaciones sobre los campos a modificar de las cabeceras.

El sistema presentado se evalúa con el simulador GEMS [101] y mediante diferentes *microbenchmarks* transaccionales. Los resultados muestran que la aceleración *hardware* mejora el rendimiento por varios motivos. En primer lugar, la cantidad de trabajo del *runtime* que no se ejecuta gracias al soporte *hardware* proporciona un *speedup* entre 1.68× y 3.45× en el mejor de los modos. Además, el soporte hardware evita la congestión de lectura en la raíz de ciertas estructuras de datos como árboles o listas. Específicamente, la raíz de estas estructuras es leída en todas las transacciones, lo que genera una importante contienda a nivel de coherencia en el acceso al *lock* correspondiente en modo lectura. En cambio, con el soporte *hardware* estos accesos se reducen a una simple comprobación, que no genera invalidaciones de coherencia. Esto hace que la escalabilidad del sistema sea mucho mayor con el soporte *hardware* que en el STM base. Finalmente, se estudia el número de reintentos en modo *hardware* antes de revertir al modo más lento y seguro, en software. Los resultados indican que, a falta de algún tipo de indicación por parte del *hardware* (por ejemplo, del motivo por el que aborta una transacción), el número óptimo de reintentos crece con el número de *threads* en el sistema. Los resultados también muestran, sin embargo, que el rendimiento del sistema *hardware* sufre cuando el nivel de congestión es grande. Este problema tiene una doble raíz: Por una parte, al aumentar el número de transacciones *hardware* que abortan y pasan al modo *software*, la ejecución es más lenta. Por otra parte, la ejecución concurrente de transacciones *hardware* y *software*, con diferentes políticas de control de congestión, puede introducir *starvation*, lo que penaliza el rendimiento. Este aspecto se estudia en el apartado siguiente.

El Capítulo 3 estudia los problemas de *starvation* causados por falta de *fairness* en el acceso a los datos entre transacciones *hardware* y *software*. En concreto, las transacciones software requieren tomar un lock en cada objeto accedido en el momento de hacer el *commit*. Las transacciones *hardware*, en cambio, solo requieren comprobar el estado de dicho *lock* en el momento de acceder al objeto. El aislamiento del sistema HTM garantiza que los accesos realizados (por ejemplo el acceso al *lock*) se mantienen inalterados hasta el final de la transacción. Esto se traduce en que cualquier intento por parte de otro *thread* de modificar la variable leída (por ejemplo, una transacción software que pretenda tomar el *lock*) se verá retrasado hasta después del *commit* de la transacción *hardware*. En concreto, con el modelo HTM utilizado, LogTM [110], el aislamiento se garantiza en base a una extensión del protocolo

de coherencia que envía confirmaciones negativas (NACK) para evitar accesos que entrarían en conflicto con una transacción existente.

Claramente, el sistema HyTM propuesto favorece a las transacciones *hardware*. Esto, de por sí, no es problemático, ya que éstas son el caso más frecuente y el más rápido. Sin embargo, el sistema puede generar *starvation* en las transacciones *software* si un objeto es leído frecuentemente por transacciones *hardware* y pretende ser modificado por una transacción software. En este caso, la transacción *software* debería esperar a que no hubiera ninguna transacción *hardware* accediendo al objeto; sin embargo, como múltiples transacciones *hardware* pueden leer el objeto en paralelo (diferentes lecturas a nivel de coherencia no constituyen una violación), puede darse el caso en que siempre haya alguna transacción *hardware* accediendo en modo lectura a un objeto. Esto puede parar indefinidamente a una transacción *software* que pretenda escribirlo, generando *starvation*. Este problema ocurre frecuentemente en nuestros benchmarks en el caso de estructuras de datos con un único punto de entrada, como una lista o un árbol binario.

Para abordar este problema, en el apartado 3.2 se propone un mecanismo *hardware* para reservar ciertas líneas en el directorio, basado en una estructura denominada *Reservation Table* (RT). La RT es una pequeña tabla, con unas 4 entradas, colocada junto al controlador de memoria. Cuando un procesador recibe un NACK a una petición de coherencia, envía una petición de reserva a la RT. A partir de ese punto, se registra ese bloque concreto como reservado para un cierto procesador, y no se atiende (a nivel de coherencia) a ninguna petición de otro procesador. Además, se proporciona un mecanismo de notificación, que garantiza que las transacciones HTM, una vez finalizada su ejecución, liberan el bloque de sus caches, evitando accesos consecutivos. Finalmente, se estudia el problema del *deadlock* en este sistema, mostrando cómo la implementación más sencilla de la RT puede generarlo, y se proporciona una política que es compatible con el sistema de *timestamps* del HTM base, LogTM.

El sistema propuesto se ha implementado en la misma infraestructura de simulación indicada antes. Las evaluaciones muestran un *speedup* considerable en aquellas aplicaciones con mucha congestión, mientras que el rendimiento en los casos no congestionados no sufre. Especialmente, se compara el rendimiento obtenido con nuestro mecanismo, frente a una política que pretende abordar el mismo problema en sistemas HTM propuesta en [16]. En el caso de la política para sistemas HTM, la solución es abortar las transacciones hardware que generan el *starvation*. En el caso de nuestro sistema híbrido, esto aumenta el número de transacciones abortadas, y la proporción de transacciones que se ejecutan en software, reduciendo el rendimiento. Por el contrario, la RT minimiza el número de abortos y por tanto el paso al modo software, lo que proporciona *speedups* de hasta 2× en casos congestionados.

Vista la importancia de los *locks* de lectura/escritura, el Capítulo 4 se centra en mecanismos *hardware* de aceleración de estas operaciones de sincronización, para reducir los *overheads*

asociados. En concreto, existen diferentes propuestas para la implementación en hardware de operaciones de sincronización, que se citan en el apartado 4.1. Diferentes implementaciones asocian un cierto estado a cada palabra de memoria [6, 8, 32, 76, 80], construyen colas hardware de solicitantes para conseguir una transferencia del *lock* rápida [55], o asignan *locks* de lectura/escritura dinámicamente a cualquier posición de memoria arbitraria [159]. Sin embargo, todas estas propuestas tienen diferentes limitaciones que, a nuestro entender, pueden restringir su utilización a un punto de vista puramente académico o de investigación. En concreto, algunas de las limitaciones que encontramos en las propuestas existentes, que se resumen en la Tabla 4-1, son:

- En múltiples propuestas resulta necesario añadir *tags* a todas las líneas de memoria o palabras del sistema, lo que aumenta considerablemente los recursos utilizados.

- La mayoría de las propuestas no permiten utilizar *locks* de lectura/escritura.

- La escalabilidad está restringida en múltiples propuestas a un sistema basado en un bus central con *snoopy*, o a un único CMP.

- En la mayoría de los casos el sistema es inflexible; por ejemplo, por requerir una asociación estática de *threads* a procesadores, y fallando en el caso de migración de un procesador a otro; por fallar en el caso de realizarse operaciones migración de páginas en aquellas con *locks* activos; por limitar el número de *locks* que se pueden tomar en un momento dado; o por no permitir el uso de trylocks que abortan si no es tomado en un intervalo dado.

En el apartado 4.2 se introduce el mecanismo del *Lock Control Unit* (LCU). Este sistema busca proporcionar un sistema *hardware* flexible y escalable para gestionar *locks*, con un *overhead* pequeño y un tiempo de transferencia del *lock* bajo. En concreto, nuestra propuesta utiliza una unidad hardware, la *Lock Control Unit* (que a su vez da nombre a todo el sistema) asociada a cada procesador. La LCU recibe las peticiones del procesador, y está compuesta de una tabla con múltiples entradas, utilizándose una por cada *lock* a gestionar, así como la lógica de control necesaria. El procesador utiliza nuevas instrucciones para tomar y liberar un *lock* en cualquier posición de memoria arbitraria, en modo lectura o escritura. Las entradas de la LCU se utilizan como interfaz con el procesador, que itera en la consulta de la entrada local correspondiente hasta que el *lock* se obtiene, y como nodos de una cola de solicitantes.

A nivel global, existe una o varias estructuras denominadas *Lock Reservation Table* (LRT). Esta estructura es la que se encarga de orquestar el acceso a un mismo *lock* por parte de diferentes LCUs. Cada *lock* se asocia a una LRT de forma unívoca según su dirección física, lo que permite disponer de múltiples LRTs, por ejemplo una por cada controlador de memoria. Cuando una LCU recibe una petición para un *lock* en una nueva posición de memoria, envía una solicitud a la LRT correspondiente, en función de la dirección física del *lock*. Si el *lock* no existía

previamente, la LRT asigna una nueva entrada en su tabla, y concede el *lock* a la LCU solicitante. En cambio, si el *lock* ya existía con una entrada en la LRT correspondiente, se reenvía la solicitud al último LCU solicitante. Éste guarda un puntero del nuevo solicitante, generándose así una cola enlazada que se utilizará para la transferencia directa del *lock* entre las diferentes LCUs. Cuando una LCU recibe la concesión del *lock*, el procesador local, que se encuentra iterando sin éxito intentando adquirirlo, lo consigue al fin y accede a la sección crítica. La transferencia del *lock* de una LCU a otra implica una notificación a la LRT correspondiente, para garantizar que se mantienen correctos los punteros de cabeza y final de la cola. Sin embargo, estas notificaciones están fuera del camino crítico de la transferencia del *lock*, lo que evita interferencias en el traspaso del mismo.

El sistema de la LCU permite tomar los *locks* en dos modos. En primer lugar, está el modo ordinario, basado en la construcción de una cola, que se ha indicado antes. Además, existe un modo denominado *overflowed locking*, que permite eliminar la entrada de la LCU. Cuando no hay más solicitantes para un *lock*, se puede eliminar la entrada de manera segura, ya que los metadatos correspondientes están registrados en la LRT por si fuera necesario comenzar una cola al recibir nuevas peticiones. De esta forma, en un sistema que utilice *fine-grain locking* un mismo procesador puede tomar múltiples *locks* sin que el número de entradas de la LCU constituya un problema, ya que pocos de estos *locks* tendrán una cola asociada.

El sistema propuesto también permite el uso de *locks* de lectura/escritura. Múltiples nodos consecutivos de la cola pueden recibir la concesión de un *lock* en modo lectura, y ésta concesión no se transmite al siguiente solicitante hasta que todos los lectores han liberado su *lock*, en cualquier orden. Finalmente, se propone una estructura opcional adicional, denominada la *Free Lock Table* (FLT). En los casos en que un *lock* es tomado repetidamente por un mismo *thread*, el *lock* puede guardarse en la FLT local al ser liberado, lo que permite un acceso más rápido en la siguiente operación y evita las notificaciones externas innecesarias.

Existen propuestas *hardware* que asignan los *locks* al procesador solicitante. El sistema propuesto, por contra, asigna cada *lock* a un *thread*, no a un procesador, pero registrando el número de procesador desde el que realizó la petición para tareas de direccionamiento. Esto permite que, mediante un mecanismo de punteros apropiado, se soporten los casos en que un *thread* migra de un procesador a otro, tanto con el *lock* tomado, como cuando es parte de la cola. Especialmente, la LCU implementa un temporizador que, tras recibir un *lock*, lo transfiere al siguiente solicitante tras un periodo de tiempo si éste no ha sido tomado, evitando *starvation* temporal o *deadlock* en los casos de suspensión o migración del *thread* solicitante.

El apartado 4.2.3 incluye un estudio detallado de la implementación del mecanismo propuesto. En este apartado se proponen una serie de invariantes que debe cumplir el sistema. También se estudia cómo se evitan las carreras de datos, por ejemplo en las notificaciones a la LRT cuando se transfiere un *lock*. Se analiza la máquina de estados a implementar en cada entrada de la LCU. Se estudian los casos de *overflow* de recursos, es

decir, cómo se comporta el sistema cuando no quedan disponibles entradas en la LCU local, o en la LRT; El primer caso se resuelve mediante el uso de entradas especiales que solo pueden tomar los *locks* en el modo de *overflow*, y con políticas que garantizan que éstos *locks* queden libres para ser tomados sin necesidad de una cola; El segundo caso, se gestiona desbordando las entradas de la LRT a una estructura de datos en memoria principal. También se analiza cómo puede el mecanismo soportar los eventos de paginación o sistemas virtualizados, aunque en estos casos no se propone un mecanismo detallado al ser altamente dependiente de la implementación concreta. Finalmente, este apartado también propone dos optimizaciones del sistema base, por una parte para optimizar la patología en que un *lock* es tomado casi exclusivamente en modo de lectura (como la raíz de una estructura de datos) y por otra parte, para mejorar el rendimiento en sistemas con una estructura jerárquica, como pueda ser un sistema compuesto por múltiples CMPs.

El mecanismo de la LCU ha sido evaluado en la misma plataforma de simulación, y los resultados se muestran en el apartado 4.3. En primer lugar, se estudia cómo se minimiza el número de mensajes en el camino crítico de la transferencia del *lock*, al requerirse una única notificación. Especialmente, este resultado mejora mucho el valor obtenible mediante *locks* software. Éstos dependen del protocolo de coherencia, e incluso la mejor implementación basada en colas requiere 4 ó 6 mensajes en función del protocolo, considerando las invalidaciones de coherencia involucradas. A continuación, se utiliza un microbenchmark para medir el tiempo de transferencia del *lock* con múltiples *threads* accediendo a la misma sección crítica. La latencia obtenida supera con creces cualquier *lock software*, y también propuestas *hardware* que no construyen una cola de solicitantes. El sistema STM basado en *locks* de lectura/escritura, utilizado como base en el Capítulo 2, se utiliza aquí como *benchmark* que utiliza *fine-grain locking*. Los resultados muestran que la LCU evita la congestión de lectura de los *locks* MCS originales, y que su menor *overhead* proporciona *speedups* entre 1.46× y 2.97× según la aplicación, con 16 *threads*. Finalmente, se utilizan aplicaciones paralelas tradicionales de las suites SPLASH-2 [157] y PARSEC [13]. Los resultados en este caso son más modestos, ya que la proporción del tiempo de ejecución destinado a tareas de sincronización es mucho menor en estos *benchmarks*. En promedio, el sistema consigue un *speedup* de 2.9% utilizando en cada caso la configuración óptima, con o sin FLT.

El Capítulo 5 pasa a considerar la microarquitectura del procesador, especialmente en lo relativo a las tareas de sincronización basadas en *locks*. Este Capítulo no evalúa ninguna propuesta mediante simulación, sino que discute las posibilidades de organización del sistema considerando una arquitectura basada en *Kilo-Instruction Processors*. En primer lugar, se introduce la idea de *transacciones implícitas* como modelo de organización de un procesador basado en *checkpoints*, como los estudiados en la Introducción. Se muestra cómo la validación atómica de un *checkpoint* a nivel global proporciona un comportamiento similar al caso de la memoria transaccional. También se estudia cómo esto permite simplificar el modelo de

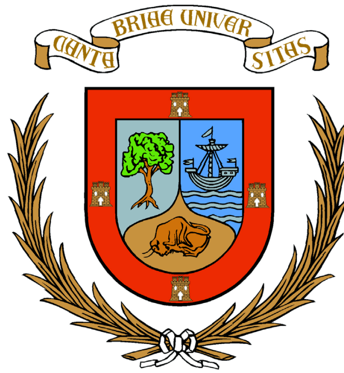
consistencia (proporcionando un modelo de Consistencia Secuencial) al tiempo que se persigue un rendimiento elevado al explotarse el ILP en el resto de *checkpoints* en vuelo.

Este Capítulo también estudia diferentes posibilidades para la especulación en secciones críticas gracias a la citada capacidad transaccional. En primer lugar, se muestra cómo un mecanismo de detección y eliminación de *silent stores* temporales permite la especulación en *locks software*. Además, se argumenta cómo esa implementación es factible, gracias a la implementación jerárquica de la cola de load/stores propuesta para los *Kilo-instruction Processors*. Después, se discute la implementación del mecanismo de la LCU y su interfaz con la microarquitectura. En concreto, es necesaria una cola llamada *LCU buffer* que almacena las operaciones de sincronización de cada *checkpoint*. Además, es necesario adelantar las adquisiciones de *locks* antes del *commit* del *checkpoint* en que se ejecutan, con la consiguiente acción de compensación (la liberación del mismo *lock*) en caso de abortarse la transacción. Finalmente, se argumenta cómo el mecanismo de la LCU propuesto en el Capítulo 4 no permite la especulación en secciones críticas, y se esboza una modificación con un modo de adquisición especulativo que sí que lo permitiera.

Como resumen, esta tesis propone una serie de mecanismos *hardware* y *software* para simplificar la programación de sistemas paralelos de memoria común y aumentar su rendimiento, centradas en torno a la programación con *locks* de lectura/escritura con garantías de *fairness*. Estas propuestas se han realizado en diferentes ámbitos: diseñando un sistema híbrido de Memoria Transaccional basado en estos *locks*; proporcionando mecanismos que garantizan el *fairness* en dicho sistema entre transacciones *hardware* y *software*; proponiendo un mecanismo *hardware* de aceleración de los *locks fair* de lectura/escritura; y estudiando la relación entre estos mecanismos de sincronización y la microarquitectura del procesador, centrándose en arquitecturas basadas en *checkpoints*. Adicionalmente, se han abierto varias vías de investigación, principalmente en lo relativo a implementaciones *hardware* de *locks* de de lectura/escritura.

University of Cantabria

Department of Electronics and Computers



Doctoral Thesis

ARCHITECTURAL SUPPORT FOR PARALLEL COMPUTERS WITH FAIR READER/WRITER SYNCHRONIZATION

Enrique Vallejo Gutiérrez

Santander, March 2010

Abstract

Abstract

Technological evolution in microprocessor design has led to parallel systems with multiple execution threads. These systems are more difficult to program and present higher performance overheads than the traditional uniprocessor systems, what may limit their performance and scalability. These overheads are due to the synchronization, coherence, consistency and other mechanisms required to guarantee a correct execution.

Parallel systems require a deeper knowledge of the system from the programmer in order to achieve good performance and scalability. Traditional parallel programming has been based on synchronization primitives such as barriers, critical sections and reader/writer locks, highly prone to programming errors. Transactional Memory (TM) is a relatively recent proposal that seeks to remove the synchronization problems from the programmer. However, many TM systems still rely on reader/writer locks, and would get benefited from an efficient implementation.

This thesis presents new hardware techniques to accelerate the execution of such parallel programs. We propose a Hybrid TM system based on reader/writer locks, which minimizes the software overheads when acceleration hardware is present, but still allows for correct software-only execution. The fairness of the system is studied, and a mechanism to guarantee fairness between hardware and software transactions is provided. We introduce a low-cost distributed mechanism named the Lock Control Unit to handle fine-grain reader-writer locks. Finally, we propose an organization of a parallel architecture based on Kilo-Instruction Processors, which helps to simplify the consistency model while allowing for high performance thanks to the speculative large instruction window.

Table of Contents

ABSTRACT I

TABLE OF CONTENTS V

LIST OF FIGURES IX

LIST OF TABLES XI

CHAPTER 1. INTRODUCTION 1

1.1. Shared-memory architectures 4
- 1.1.1. Memory coherence 5
- 1.1.2. Memory consistency 6

1.2. Fundamentals of shared-memory parallel programming 8
- 1.2.1. Synchronization mechanisms 8
  - 1.2.1.1. Barriers 8
  - 1.2.1.2. Locks and critical sections 9
- 1.2.2. Difficulties of parallel programming 9
  - 1.2.2.1. Scalability and critical sections 9
  - 1.2.2.2. Granularity 10
  - 1.2.2.3. Deadlock 11
  - 1.2.2.4. Priority inversion 11
  - 1.2.2.5. Starvation 12
  - 1.2.2.6. Complexity of concurrent data structures 12

1.3. Locking mechanisms 14
  - 1.3.1.1. Centralized vs. queue-based locking 14
  - 1.3.1.2. Spin-locks vs. try-locks vs. blocking locks 15
  - 1.3.1.3. Reader-writer locking 16
  - 1.3.1.4. Fairness issues 17
  - 1.3.1.5. Hierarchical lock implementations 18
  - 1.3.1.6. Pathologies of locks 19
    - 1.3.1.6.1. Single-thread locking and biased locks 19
    - 1.3.1.6.2. Waiting thread suspension in FIFO locks 20
    - 1.3.1.6.3. Mostly Reader locking 20

1.4. Fundamentals of Transactional Memory 21
- 1.4.1. Atomic sections 21
- 1.4.2. The ACID properties 22
  - 1.4.2.1. Atomicity 22
  - 1.4.2.2. Consistency 23
  - 1.4.2.3. Isolation 23
  - 1.4.2.4. Durability 24
- 1.4.3. Programmability and composability 24
- 1.4.4. Update mechanism: In-place vs. deferred 25

1.4.5. Conflict detection mechanism: Lazy vs. Eager, visible vs. invisible readers 25
1.4.6. Data validation and the privatization problem 26
1.4.7. Contention management 28
1.4.7.1. Abort-always policy 29
1.4.7.2. Requestor-wins policy 29
1.4.7.3. Requestor-waits policy 29
1.4.7.4. Software policies 30
1.4.8. Progress conditions 30
1.4.8.1. Wait-free, lock-free, obstruction-free and blocking progress conditions 31
1.4.8.2. Non-blocking TM 31
1.4.8.3. Lock-based TM 32
1.4.9. Nested transactions 32
1.4.10. Irrevocable transactions 33
1.4.11. Hardware Transactional Memory 34
1.4.11.1. Transactional Memory by Herlihy and Moss 34
1.4.11.2. Transactional Coherence and Consistency (TCC) 35
1.4.11.3. Bulk Transactional Memory 35
1.4.11.4. Log-based Transactional Memory (LogTM) 36
1.4.12. Software Transactional Memory 36
1.4.12.1. Software Transactional Memory by Shavit and Touitou 37
1.4.12.2. Software Transactional Memory for Dynamic-Sized Data Structures 37
1.4.12.3. Fraser's OSTM 38
1.4.12.4. Lock-based STMs 39
1.4.13. Hardware-software Transactional Memory 40
1.4.13.1. Hardware-accelerated TM 40
1.4.13.1.1. Hardware-Accelerated TM (HATM) 40
1.4.13.1.2. Signature-accelerated TM (SigTM) 41
1.4.13.1.3. Flexible TM (FlexTM) 41
1.4.13.2. Hybrid TM 41
1.4.13.2.1. Intel's Hybrid TM 41
1.4.13.2.2. Sun's Hybrid TM and Rock 42
1.5. Processor microarchitecture and performance aspects 43
1.5.1. Evolution of ILP-driven microarchitectural designs 43
1.5.2. Thread-level parallelism 45
1.5.3. Kilo-instruction processor overview 46
1.5.3.1. Checkpointing mechanism and early release 46
1.5.3.2. Bi-level issue queue 47
1.5.3.3. Ephemeral registers 47
1.5.3.4. Load/store queue handling 48
1.6. Contributions of this thesis 48
1.6.1. Lock-based Hybrid TM 48
1.6.2. Fairness among software and hardware transactions 49
1.6.3. HW acceleration of locking mechanisms 49
1.6.4. Microarchitectural improvements for performance, locking and consistency 49
CHAPTER 2. LOCK-BASED HYBRID TRANSACTIONAL MEMORY 51

2.1. Advantages of reader-writer blocking TM **53**
2.2. Base STM overview **53**
2.3. Acceleration opportunities with a generic HTM **57**
2.3.1. Avoid locking 58
2.3.2. Read set removal 61
2.3.3. Write set removal and in-place update 62
2.4. Evaluation **63**
2.4.1. Evaluation infrastructure 63
2.4.2. Simulated models 64
2.4.3. Benchmarks 65
2.4.3.1. Red-black tree 65
2.4.3.2. Skip-list 66
2.4.3.3. Hash table 67
2.4.4. Performance results 67
2.4.4.1. Single-thread performance 67
2.4.4.2. Read-only transactions 68
2.4.4.3. Reader-writer transactions 70
2.4.4.4. Number of HW retries in HW transactions 72
2.5. Summary **73**
CHAPTER 3. FAIRNESS IN HYBRID TRANSACTIONAL MEMORY **75**
3.1. Writer starvation in Hybrid TM **77**
3.2. Directory reservations **79**
3.2.1. Issues with LogTM transactions 80
3.2.2. Reservation Table and fair queuing 82
3.2.3. Thread de-scheduling and migration 83
3.3. Evaluation **84**
3.3.1. Performance results 85
3.4. Summary **88**
CHAPTER 4. HW ACCELERATION OF LOCKING MECHANISMS **89**
4.1. HW mechanisms for locking acceleration **91**
4.2. The Lock Control Unit mechanism **94**
4.2.1. Hardware components 95
4.2.1.1. The Lock Control Unit 95
4.2.1.2. The Lock Reservation Table 97
4.2.1.3. The Free Lock Table 97
4.2.2. General overview 98
4.2.2.1. Programming interface 98
4.2.2.2. Two locking modes 99
4.2.2.2.1. Queue-based locking 100
4.2.2.2.2. Overflowed locking 100
4.2.2.3. Write-lock acquisition, transfer and release 100
4.2.2.4. Reader locking 103

4.2.2.5. The Free Lock Table 105
4.2.3. Detailed overview 106
4.2.3.1. Design invariants 106
4.2.3.2. LRT detailed overview 107
4.2.3.3. LCU state machine 109
4.2.3.4. Communication primitives 110
4.2.3.5. Thread suspension and migration 111
4.2.3.6. Types of LCU entries and forward progress 112
4.2.3.7. Management of LCU overflow 114
4.2.3.8. LRT overflow 116
4.2.3.9. Read-only locking 116
4.2.3.10. Hierarchical Locking 118
4.2.3.11. Paging, virtualization and process faulting issues 119
4.3. Evaluation 122
4.3.1. Number of messages in the critical path of lock transfer 123
4.3.2. Lock transfer time 124
4.3.3. Fine-grain locking: STM benchmarks 127
4.3.4. Traditional parallel benchmarks 129
4.4. Summary 130
CHAPTER 5. IMPLICIT TRANSACTIONAL MEMORY 133
5.1. Atomic sections and processor checkpointing 135
5.1.1. Implementation details of implicit transactions 136
5.1.2. Implicit transaction length and performance 139
5.2. Sequential consistency with implicit transactions 141
5.3. Lock speculation with implicit transactions 143
5.3.1. Dependencies through locks and critical section speculation 144
5.3.2. Critical section speculation with software locks 145
5.3.3. Critical section speculation with the LCU model 147
5.3.3.1. Implicit transactions and ordinary LCU access 147
5.3.3.2. Speculation support in the LCU 148
5.3.3.3. Implicit transactions and speculative LCU access 150
5.3.3.4. Speculative LCU accesses and HyTM 151
5.4. Speculation beyond flags and barriers 151
5.5. Summary 152
CHAPTER 6. CONCLUSIONS 155
6.1. Contributions 157
6.2. Future work 159
6.3. Publications 159
6.3.1. HyTM, kilo-instruction architectures and reader/writer synchronization 160
6.3.2. Interconnection Networks 160
CHAPTER 7. REFERENCES 163

List of Figures

Figure 1-1: Example of a consistency problem. Initially A = B = 0. Can P_2 load 0 from *A*? 7
Figure 1-2: Example of data race 8
Figure 1-3: Example of the usage of a lock 9
Figure 1-4: Coarse (left) vs fine (right) grain locking 10
Figure 1-5: Necessity of acquiring multiple locks to avoid concurrency problems, from [71] ... 13
Figure 1-6: Example of the MCS lock structure 15
Figure 1-7: Example of the MCS reader-writer lock structure. Grey fields represent readers ... 16
Figure 1-8: Example of the Krieger *et al.* reader-writer lock. Grey fields are readers 17
Figure 1-9: Incorrect implementation of a barrier using reader-writer locks 18
Figure 1-10: Example of code that fails in the privatization problem 27
Figure 1-11: Example of code that fails in the publication problem 28
Figure 1-12: Object structure in DSTM 38
Figure 1-13: Object structure (left) and object handle in the write-set list (right) of Fraser's OSTM 38
Figure 1-14: Orec organization in the word-based STM by Harris and Fraser, taken from [64]. 39
Figure 1-15: Orec table in the HyTM system by Damron *et al.*, taken from [33] 42
Figure 1-16: Block diagram of an out-of-order superscalar processor 44
Figure 2-1: STM programmer interface 54
Figure 2-2: STM Object structure (left) and read/write set list element (right) 54
Figure 2-3: Three steps in a commit of a binary tree data structure. The intermediate step b) contains a cycle, what can lead to zombie transactions that never commit 56
Figure 2-4: Modified STM lock structure for HT Txs to detect conflicts with writing SW Txs..... 59
Figure 2-5: Lock implementation in the HyTM model 60
Figure 2-6: Modified STM Object with a version field allowing for in-place updates................ 62
Figure 2-7: Left, Red-Black tree example. Right, status after the addition of node 20 65
Figure 2-8: Skip-list example, taken from [116] 66
Figure 2-9: RB, skip and hash speedup with read-only transactions, k = 8, in linear (left) and logarithmic (right) scales 69
Figure 2-10: Left: RB cycle dissection, p=0, k=8. Right: Cycles in lock accesses......................... 70
Figure 2-11: Skip-list performance under low (left) and high (right) contention, p = 10%......... 71
Figure 2-12: Transactions aborted in HW and SW modes. Skip list with k=8, p=10% 71
Figure 2-13: Hash-table speedup and abort rate with k=8, p=25% (high contention) 72
Figure 2-14: Performance with different number of HW retries of aborted transactions 72
Figure 3-1: Example of NACK messages in LogTM. Proc. B requests the memory location *a*, which has been read by processors A and C 78
Figure 3-2: Pathological example of transactional code that stalls due to writer starvation..... 78

Figure 3-3: Message transfer with the Reservation Table mechanism...................................... 79
Figure 3-4: Reservation Table structure.. 80
Figure 3-5: Message transfer with the Reservation Table mechanism...................................... 81
Figure 3-6: Message transfer with the Reservation Table mechanism...................................... 83
Figure 3-7: Normalized execution time of the RB benchmark, k=11, p=10% 86
Figure 3-8: Normalized execution time of Hash and skip, k=11... 87
Figure 4-1: Lock Control Unit architecture.. 95
Figure 4-2: Contents of each LCU entry 96
Figure 4-3: Contents of each LRT entry. The $Tail_{id}$ field's composition is equal to $Head_{id}$ 97
Figure 4-4: Contents of each FLT entry 97
Figure 4-5: Functions for lock acquisition and release... 99
Figure 4-6: Lock acquisition when the lock is free 101
Figure 4-7: Enqueue when the taken lock is uncontended. LCU_0 is the current owner 101
Figure 4-8: Lock transfer 102
Figure 4-9: Possible data race in the notification mechanism 103
Figure 4-10: Example of concurrent read locking 104
Figure 4-11: Races in the enqueue process 108
Figure 4-12: Simplified state machine of the LCU entries.. 110
Figure 4-13: Detailed state machine of the LCU entries 110
Figure 4-14: Example of migration. Thread t_2 migrates from processor 2 to 9 while waiting.. 112
Figure 4-15: Example of read-only locking... 117
Figure 4-16: Sketch of a hierarchical implementation 119
Figure 4-17: CS execution time including lock transfers. SSB vs LCU....................................... 125
Figure 4-18: CS execution time including lock transfers. LRT vs software locks 126
Figure 4-19: Transaction cycle dissection of the RB benchmark with 2^8 max. nodes............... 127
Figure 4-20: Transaction execution time, 16 threads and 75% of read-only transactions 128
Figure 4-21: Application execution time 129
Figure 5-1: Execution flow example with 4 processors.. 137
Figure 5-2: Estimation of the proportion of rollbacks in SPLASH.. 140
Figure 5-3: Sequentially consistent reordering of memory operations from 2 processors...... 142
Figure 5-4: Sequentially consistent reordering of checkpoints from 2 processors 142
Figure 5-5: Different checkpointing schemes with critical sections 144
Figure 5-6: Examples of critical sections that often admit parallel execution, from [118]....... 145
Figure 5-7: Example of speculative access to a lock 149
Figure 5-8: Deallocation of speculative entries in the lock queue... 150

List of Tables

Table 2-1: Summary of operations when conflicts occur in the HyTM 61
Table 2-2: Single thread normalized performance (inverse of the transaction run time) 68
Table 4-1: Comparative of SW and HW locking mechanisms 94
Table 4-2: Possible states of a LCU entry 96
Table 4-3: Communication primitives 111

Chapter 1. Introduction

Chapter 1

Introduction

The free lunch is over [148]. The traditional evolution that processor microarchitectures had followed during the last 20 or more years, based on higher processor frequencies, wider instruction windows and deeper instruction pipelines, is over. The current technological constraints, mainly the power wall and the memory wall, make such design lead to diminishing returns. Therefore, most processor architects have moved to Chip Multiprocessor (CMP) designs that rely on an increase of the number of processing cores, but not on their individual performance. This makes that traditional, single threaded programs, do not perform better with new generations of processors. Programmers cannot rely on the technological advances to make their simple code run faster; they must use a parallel programming model to exploit the system capabilities.

Parallel programming makes use of multiple tasks running simultaneously on multiple processors, making a joint effort to complete the execution faster. Such design involves many difficulties, both in the hardware side (how to design a parallel machine) and in the software side (how to efficiently program such parallel machine). Even more, both hardware and software interact with each other, so the design of a given layer is dependent on the design of the rest of the system. This introductory Chapter will detail the main problems of parallel programming and the issues of designing parallel machines, along with the related work that targets these problems. Specifically, it introduces the main difficulties of parallel programming, with a special focus on synchronization mechanisms; recent proposals on Transactional Memory that try to simplify the programming models; and technological and architectural aspects than condition the design of current parallel systems. All these aspects are the base for the subsequent work developed in this thesis.

There are two basic models for parallel programming. The message-passing paradigm, for example implemented in MPI [151], requires the programmer to explicitly declare each communication primitive used to communicate the processors. The shared-memory paradigm, by contrast, provides a shared address space that can be accessed by all of the processes, or “threads”, which are implemented for example using the POSIX standard [1]. The communication between threads occurs through shared variables, which in turn rely on the

underlying architecture providing the necessary coherence and consistency mechanisms to send the required messages. However, in this case the communication burden is hidden from the programmer.

Parallel programming is inherently more difficult than serial programming. In a parallel program, the different tasks have to synchronize with each other to access the data to be processed (data partitioning) and communicate intermediate results (data sharing), all of it synchronizing the different execution phases (synchronization, involving task creation and termination). Debugging is also more complex than in serial programming, since the execution is not deterministic, depending on the relative speed of the different tasks. Finally, the architectural implementation can impose additional restrictions that the programmer or compiler must consider. Shared-memory machines often implement relaxed memory models that provide higher performance but require of explicit fences and synchronization instructions. Section 1.1 details the design issues and performance implications of different coherence and consistency models. Section 1.2 gives deeper insight into the problems and difficulties of parallel programming on a shared-memory architecture. One of the basic tools used for mutual exclusion are locks, which are further studied in section 1.3.

Transactional Memory (TM) [70] pretends to simplify some of the difficulties of parallel programming. By providing the abstraction of a database *transaction* to the programmer, the problem of mutual exclusion is highly simplified. Multiple hardware TM (HTM) and software TM (STM) systems have been proposed, with different benefits and weaknesses. Also, Hybrid TM (HyTM) systems try to get an efficient implementation with low hardware costs. They are introduced in detail in section 1.4.

This Chapter will also deal with the microarchitectural issues that arise when designing a high performance parallel system. The evolution in microarchitecture design leading to diminishing returns in single-processor designs, and the current performance issues are discussed in section 1.5. This section also presents an introduction of the Kilo-Instruction Processor architecture that is the base for the last Chapter of this thesis.

Finally, this introductory Chapter is concluded with a summary of the contributions presented in this thesis, in section 1.6.

1.1. Shared-memory architectures

Shared-memory parallel machines provide a single physical view of the memory for all the processors. On these machines, different concurrent tasks of the same process, named threads, can communicate with each other by merely reading and writing memory locations. By contrast, on a distributed-memory machine, each processor has its own memory and different tasks working on the same program must communicate to each other by explicitly sending messages with data o synchronization. This thesis focuses on shared-memory machines.

Shared-memory architectures must provide the same view of the memory for all the processors in the system. This might be simple in a model that does not use data caches, in which all processors access the same shared memory controller. However, most systems have two or more levels of caches to reduce the performance penalty of the distant memory (in terms of processor cycles to access data). This allows for the concurrent existence of multiple copies of the same data block, what imposes coherence and consistency problems, detailed next.

1.1.1. Memory coherence

In a shared-memory system multiple processors can access the same block with read and write operations. This will generate multiple copies of the block in the different data caches, with multiple versions of the data as the writes occur. Informally, a coherence mechanism deals with the delivery of a valid version of the data to each request.

A formal definition can be taken from [68]. A memory system is coherent if:

1. A read by a processor P_1 to a location *A* that follows a write by P_1 to *A*, with no writes of *A* by another processor occurring between the write and the read by P_1, always returns the value written by P_1.

2. A read by a processor to location *A* that follows a write by another processor to *A* returns the written value if the read and write are sufficiently separated in time and no other writes to *A* occur between the two accesses.

3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

Multiple coherence protocols have been developed, both in hardware and in software. A good survey can be found in [146]. Most machines nowadays implement hardware coherence, but others, such as the Cell processor [73], implement coherence between the main processor and the SPE units in software, requiring the programmer to reason about what is shared in each processing element.

According to their implementation, there are typically two classes of coherence protocols: bus based or directory based. Bus-based systems can rely on a *snoopy* protocol. All the processors "snoop" a centralized bus and detect other processors' requests and updates. This allows them to respond to memory requests if they have the block of data in their local caches, rather than waiting for the long-latency main memory access. Similarly, it allows them to detect conflicts with the memory writes from other processors, updating or invalidating their local caches. By contrast, directory based protocols can be implemented in a distributed manner, such as in the

DASH multiprocessor [91]. The directory contains metadata about the sharers of each block, and possibly about an owner who contains an updated version. Directory-based protocols are implemented with coherence messages, which are sent from one element to another to invalidate sharers upon a write, propagate updated versions of a block upon a read, and so on.

The caches in a coherent system implement a state machine with several possible states, typically all or part of the MOESI states [149]. These states differ on their exclusiveness and ownership, as follows:

a) Modified (M), the cache contains the only valid copy of the data, which is dirty: the values in main memory have not been updated yet.

b) Owned (O), the cache contains a dirty copy of the data, with the main memory not yet updated. However, this copy is not exclusive; other caches can have the same block in shared state. The only owner is responsible for the block, meaning that any request in the directory will be forwarded to the owner, who serves the request.

c) Exclusive (E), the cache is the only one in the system with a valid and clean copy of the block.

d) Shared (S), the cache contains a valid copy of the block, which can be also present in other caches.

e) Invalid (I), not present in the cache or invalidated by a coherence request, such as a remote write.

Different implementations are typically identified with the subset of these states that are used, for example, MESI, MOSI or MSI. Additionally, a real protocol requires multiple transient states, to support the races that occur when multiple processors request a block concurrently with different access modes. This makes coherence protocols very complex and difficult to validate. Recent proposals such as [102] focus on simplifying the verifiability of the coherence mechanism, while they preserve the presented valid states.

1.1.2. Memory consistency

The memory consistency model (or, simply, the memory model) manages the correct ordering of memory operations to different memory locations. While the coherence protocol has to propagate the updates performed on memory locations to all of the required processors, the consistency model specifies the timing of such update.

Let us consider the example case of two processors P_1 and P_2 in a coherent system, accessing a memory location A with an initial value of a_0. A write from P_1 updates A from a_0 to a_1. What happens with subsequent reads to A? By the first condition of the coherence, P_1 must read the updated value a_1. However, P_2 can continue accessing its locally cached entry with the old

value a_0 for a certain time before the updated value is propagated, and still obey the second condition of the coherence mechanism. This case would be a coherent system, with a relaxed consistency model.

The different observed data can be especially important when multiple memory locations are involved and P_1 and P_2 use one of them to synchronize with each other. Consider the example in Figure 1-1 in which processor P_1 updates the data in *A* (instruction 1) and then sets a flag in *B* to notify processor P_2 that the data is ready (instruction 2). What happens if the update of the flag in *B* propagates to P_2 before the update of the data in *A*? P_2 will access the old data 0 in *A*, considering it as the new one, what can lead to an unexpected program behavior. Observe that the problem in this case is not a coherence or architectural problem, but the programmer (or the compiler) not obeying to the consistency model restrictions, which would typically require of an especial *fence* instruction between instructions 1 and 2 from P_1 to make sure that all data are correctly propagated. Several examples of possible problems with a relaxed model, along with a detailed study of multiple consistency models, can be found in [5].

	Processor P_1	*Processor* P_2
1:	st &*A*, 10	beq &*B*, 0, -4
2:	st &*B*, 1	ld &*A*, r1

Figure 1-1: Example of a consistency problem. Initially A = B = 0. Can P_2 load 0 from *A*?

Sequential Consistency (SC) [87] is the most desirable model since it provides the most intuitive programming model. SC requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order, and the accesses among different processors were arbitrarily interleaved. This is what any programmer would intuitively consider to be the 'common' behavior of a multiprocessor system if not aware of what a memory model is. The problem with SC is that it most often requires that memory operations from each program appear to be executed in-order, what may limit the system performance. Write buffers, multiple memory controllers or an unordered interconnect raise issues with possible races and might not be used in a system obeying SC.

Other consistency models such as Total Store Order (TSO, [141], used in Sparc machines among others) or Processor Consistency (PC, [54], used in the x86 architecture and others) achieve a higher performance by relaxing these constraints, at the cost of a more complex programming framework. These models typically allow a processor to observe its own writes before they are sent to the rest of the processors, what allows for significant optimizations while still providing a quite intuitive memory model. More relaxed models, such as Relaxed Consistency (RC, [51]) allow further interleaving of writes from different processors.

Contrary to coherence, the consistency model is visible to the programmer. Hill [72] argues for simple consistency models such as SC, despite its lower performance, to prevent difficult-to-

find consistency problems. Besides, speculative execution [52] has been proven to allow for simple consistency models with a similar performance to the relaxed ones.

1.2. Fundamentals of shared-memory parallel programming

Shared-memory parallel programs make use of multiple concurrent threads, provided by a system library. Typically, the implementation will be specific to the operating system (such as Linux, Solaris or Windows threads) or follow the portable standard POSIX [1]. Such implementation will define, among others, the tasks of creating new threads, synchronizing their operations and terminating them. This section is focused on the synchronization mechanisms.

1.2.1. Synchronization mechanisms

The access to shared data must be protected to prevent different threads from updating the same value, overwriting each other's modifications, without notice. An example is presented in Figure 1-2: The increase of the variable *a* in C code is translated to three ISA instructions. In the example execution presented on the right, *a* is initially 0. Two threads run the same increase function, but *a* finishes being 1, not 2. The problem is that the update of the memory location is not atomic; instead, both threads read the same initial value 0 (at time 1 and 2), and write the same updated value 1 (time 4 and 5).

C code

```
void increase(&a){
  a = a + 1;
}
```

Assembler code

```
ld &a, l1
addi l1,1,l2
st l2, &a
```

Execution

	Thread 1	*Thread 2*
1:	ld &a, l1	
2:		ld &a, l1
3:	addi l1,1,l2	addi l1,1,l2
4:	st l2, &a	
5:		st l2, &a

Figure 1-2: Example of data race

To overcome such problems, a proper synchronization mechanism among threads is required. There are two classic means of synchronization to overcome such problems: First, barriers can divide execution into different phases of executions in which each thread only accesses an assigned, individual data set. Second, locks are used to block a memory location or code section, reserving it for a given thread. They are presented in the next sections.

1.2.1.1. Barriers

Barriers provide a single synchronization point for many threads. Once a thread arrives at a barrier, it starts waiting (typically, using busy wait) for the remaining threads to reach the same point. Once all threads reach the barrier, it becomes "open" and all threads proceed,

synchronized from the same point. Barriers are typically used in iterative parallel programs, to divide sections. Within each section, a thread can access its assigned data without restriction, for example, a block of a matrix.

1.2.1.2. Locks and critical sections

Locks are abstractions used to block a given data or code section. A lock provides with two basic operations, namely *lock* and *unlock*, that reserve or release the right to access the code section protected by the lock, called *critical section* (CS). Only one thread can *take* or *hold* a lock concurrently; any other call to *lock* will have to wait for the first thread from *unlocking*. Locks will be studied in detail in section 1.3.

The problem presented in Figure 1-2 could be solved trivially using a lock, as presented in Figure 1-3. The acquisition of *lock1* prevents any other thread from executing the protected line 3 when calling the function *increase*. Note that correctness must be manually ensured by the programmer: accesses to the protected shared variable from outside any critical section are allowed by the system, but incorrect.

```
1: void increase(int &a){
2:     lock(&lock1);
3:     a = a + 1;
4:     unlock(&lock1);
5: }
```

Figure 1-3: Example of the usage of a lock

1.2.2. Difficulties of parallel programming

Parallel programming is essentially more difficult than traditional, single-thread programming. Ideally, one would expect that a system running *N* threads should finish its work *N* times faster than the same system using a single processor. However, in this section we will cite different effects that limit performance and makes the task of parallel programming more complex. These include the overhead of the parallelization mechanisms and the effect of critical sections, the granularity of the data partitioning and the forward progress issues that can occur in parallel systems.

1.2.2.1. Scalability and critical sections

Amdahl's Law studies the scalability of a system as the number of processors increases. Gene Amdahl [9] stated that programs contain serial sections (typically, those critical sections protected by global locks) that cannot be parallelizable and that limit the maximum parallelism available in the system. The formula derived from his argument is presented in [68]:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

Where *n* is the number of processors, $r_s + r_p = 1$ and r_s represents the ratio of the sequential portion in one program. From the formula it can be derived that the maximum speedup achievable with an unlimited number of processors is $1/r_s$. For example, a program with a 95% of its execution in a parallel section ($r_s = 0.05$) will have a maximum speedup of 20×.

Some authors, such as John L. Gustafson [57], discuss this formula. They argue that, while the formula is correct, larger systems are typically used to execute programs with larger datasets, what provides a lower rate of serial execution r_s and superior scalability. In any case, it is clear that critical sections are a limiting factor of the performance of parallel programs.

1.2.2.2. Granularity

Also related with Amdahl's Law is the effect of the *granularity* of the locking mechanism. Consider the example codes in Figure 1-4. Both functions increment the 1000 elements of a fixed-size shared vector, passed as a parameter. The structure *vector* and the code of the function *increase* differ on each case. The left code implements a coarse-grain strategy: the vector is protected by a single lock, taken during the whole operation. By contrast, the right code makes use of a fine-grain locking mechanism: The vector contains two arrays, one of values and one of locks. Each value of the vector is assigned with its corresponding lock. The program acquires and releases each individual lock on each update.

```
1:  void increase_coarse(vector &v){          void increase_fine(vector &v){
2:      lock(&(v.lock));                          for (i=0;i<1000;i++){
3:      for (i=0;i<1000;i++){                       lock(&(v.lock[i]));
4:        v.a[i] = v.a[i] + 1;                      v.a[i] = v.a[i] + 1;
5:      }                                           unlock(&(v.lock[i]));
4:      unlock(&(v.lock));                        }
5:  }                                         }
```

Figure 1-4: Coarse (left) vs fine (right) grain locking

The tradeoff is clear. In the coarse-grain approach, locks are assigned to large blocks containing multiple entries. The update of any of the entries requires the acquisition of the lock of the block. In the example, a single lock is acquired in line 2, protecting the whole vector. This simplifies the programming, but provides lower performance due to Amdahl's Law: Different values of the vector cannot be accessed concurrently, despite being independent, increasing the effective size of the "serial" accesses. By contrast, the fine-grain locking approach allows for different threads to access different sections of the same vector concurrently, increasing the available parallelism. However, there is a twofold cost: First, the

increased memory required to store the variables required for the locks. Second, the increased execution time in acquiring and releasing locks. Which approach is better will depend on many factors, including the lock implementation, the behavior of the program, the complexity of the code and other details.

1.2.2.3. Deadlock

The use of locks introduces a dependency between threads: The execution of the waiting threads depends on the forward progress of the lock holder. Using multiple locks to protect different objects or sections of the code lead to more complex designs, in which a thread can depend on a second thread which, in turn, is waiting for a third thread to release a lock. If a cyclic dependency is ever generated (the last threads requires a lock taken by the first thread on the chain), the system will block, since no thread can make forward progress. This situation is defined as deadlock.

There are four concurrent conditions required for deadlock to occur, as studied in [136]

1. Mutual exclusion: Multiple threads cannot access (acquire) the same contended resource (lock) concurrently.
2. Hold and wait: A waiting thread must be holding a resource (lock) while waiting for another one.
3. No preemption: Once a thread acquires a lock, it cannot be externally forced to release it.
4. Circular wait: A dependency cycle exists between different threads acquiring locks.

Non-trivial parallel programs must take special care with deadlock problems. Two approaches exist: Either deadlock is prevented (typically, by requiring a certain order in the access to locks, what prevents the circular wait) or deadlock must be detected when it occurs and solved (by typically detecting the cyclic dependency between threads and forcing some thread to release a taken lock in a preemptive way).

Preventing deadlock problems while providing good performance leads to complex designs, difficult to maintain and reason about. For example, Rossbach *et al.* [121] present the case of a Linux kernel file (`mm/filemap.c`) containing a 50 line comment only to describe the lock ordering used in the file.

1.2.2.4. Priority inversion

Soft or hard real-time systems require the use of priority-based scheduling to obey to the different deadlines of each task. However, locking can negatively affect the priority mechanism. If a low-priority thread acquires a lock, it can delay the execution of a high-priority thread that needs access to the same lock. Even more, the priority-based scheduler will assign

more resources to the higher priority task that is waiting, preventing the faster execution of the blocker task with lower priority. Such effect is called priority inversion, since the priority of the threads is effectively inverted by the access to the lock.

The simplest solution, implemented in many operating systems, is the use of a priority inheritance protocol [131]. A priority value is assigned to each lock. This value will be set to the maximum priorities of the threads waiting to acquire the lock. Whenever a lock priority value is higher than the priority of the thread that acquired it, the thread's priority is increased to that given by the lock. This ensures that more resources are assigned to low-priority threads that are blocking high-priority threads, so they can finish their critical section faster and release the lock.

1.2.2.5. Starvation

Starvation is the situation in which a thread or a group of threads stall while waiting for a resource, such as acquiring a lock, because of the interference of different threads. The paradigmatic example occurs with unfair reader-writer locks where writers starve. Reader-writer locks will be studied with more detail in section 1.3.1.3, and fairness issues will be presented in section 1.3.1.4. The reader is referred to those sections to understand the starvation problem, and how it can affect both the thread-local progress and the global progress.

1.2.2.6. Complexity of concurrent data structures

Apart from the previous difficulties, concurrent data structures incur in additional complexities in terms of design and composability. To illustrate these problems, we will study a simple, fine-grained lock-based implementation of a set based on a linked list, as presented in [71].

"Sets" are data structures that contain multiple objects of a given type T, identified by an individual key, and that provide three self-defining method calls: *add()*, *remove()* and *contains()*. "List-based sets" are sets with an internal implementation of a linked-list: Objects are referenced by pointers in order, and the list of objects is ordered according to each individual key. Concurrent access to a shared set of this type implies that multiple threads will issue operations of the three types that will overlap their executions. In our example we consider a lock-based implementation for simplicity, though different implementations exist, blocking or lock-free. Fine-grain locking implies that a single lock is associated to each object on the list, instead of a single lock protecting the access to the whole list.

The *contains()* method checks for the existence of a value, iterating on the list from the first element. To prevent following invalid pointers from objects being concurrently updated (*removed* from the set), the method acquires the locks of the intermediate elements of the list using *lock coupling* [12]: the lock of an object is taken before its link is followed; then, the

second object is locked before the first lock is released. In this way, only locked pointers are followed, preventing invalid indirections.

Similarly, the *remove()* method has to concurrently lock the object to be removed and its predecessor in the list, to prevent an unexpected behavior when a concurrent removal of the predecessor or a concurrent *addition* in between occur. Figure 1-5, taken from [71], shows an example. The removal of *a* requires locking the predecessor, the head node. Similarly, the removal of *b* requires locking the predecessor node, *a.* Failing to do so could lead to *b* updating the *next* pointer in *a* to target *c*. That should make *b* non-reachable, but the concurrent update of the *next* pointer in the head node (from the *remove()* of *a*) could overwrite the action, making *b* reachable again.

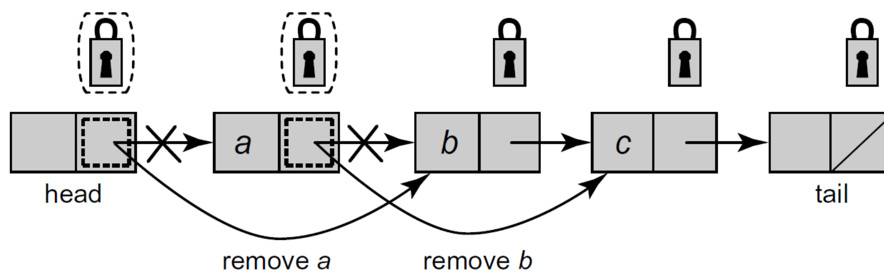


Figure 1-5: Necessity of acquiring multiple locks to avoid concurrency problems, from [71]

This example illustrates why concurrent data structures are complex to design: the implementation of a method depends on the implementation of the remaining methods in the object, as occurs with the locks requirements between *add(), remove()* and *contains()*. Proving the correctness of a concurrent object is nontrivial, since the interaction of different threads can occur in multiple ways. Also, trying to add a new method to a concurrent data structure might require rewriting all the previous methods to adapt to the new requirements.

But modifying concurrent objects is not the only problem. Lock-based objects do not *compose*, this is, one cannot build a higher-level function based on lower-level objects and methods, without a deep knowledge of their implementation. For example, consider the case of two set objects with the previous implementation, and the idea of building a function that *moves* one object from one set to the other. This could be easily implemented by calling *remove()* in the first set, and *add()* in the second; however, this leads to two problems:

1. Other threads can see an intermediate invalid state, in which the object is not present: it has been removed from one set, but not yet added to the second.

2. The *move()* function can lead to deadlock if one thread intends to *move* an object from set A to set B, while other thread calls *move* from B to A, if the objects keys are consecutive.

These two problems show the lack of composability of lock-based programs: correct fragments of code may fail when combined [65].

1.3. Locking mechanisms

Despite the difficulties and problems presented in section 1.2.2, locks are heavily used as a main tool in parallel programs. Multiple implementations of locks have been developed, with different levels of complexity and different characteristics, from the simplest test-and-set lock to the complex, reader-writer queue-based implementations that support the detection of suspended threads. More complex locks try to solve specific performance problems, at the cost of more complex implementations and possibly lower performance in the common case. Herlihy and Shavit [71] cover in detail multiple implementations of locks. In this section we will review the main alternatives when implementing locks and the performance pathologies – uncommon behavior that differs from the expected case – that determines their performance in certain cases.

In general, coherence-based software-only locks make use of at least a memory location to identify the status of the lock. Locks can be “free” or “acquired”, depending on the status of the lock variables. To modify these values without data races, the architectural ISA typically provides with atomic instructions such as test-and-set, or the more flexible load linked and store conditional (LL/SC) in Alpha [137], PowerPC [113], MIPS [78] or ARM [130], or compare and swap (CAS) in x86 [75], Itanium [74], SPARC [141] and other architectures. Threads atomically change the lock status to prevent data races that would occur if ordinary load and store instructions were used.

1.3.1.1. Centralized vs. queue-based locking

Centralized locks are those that share a fixed group of memory locations indicating if the lock is taken by some thread, or free. Multiple implementations follow this idea, such as Test-And-Set (TAS) and Test-And-Test-And-Set (TATAS) locks [84] or ticket locks [106]. These locks have a simple implementation and fast access, but perform poorly when multiple threads contend trying to acquire the same lock. In such case, all waiting threads will spin-wait accessing the same coherence location, waiting for the lock owner to release the lock. However, the lock owner will update the lock line when releasing it, what invalidates all the sharers at the coherence level. After that, a coherence burst is generated to reload the data on the requestors. Such coherence traffic reduces the performance, increasing the lock transfer time in presence of contention.

Queue-based locks (first proposed in [11]) prevent these performance problems. In such model, each requesting thread allocates a “queue node” in shared memory. These nodes are connected forming a queue, with the lock owner being the queue head. The lock release involves an update of the next node in the queue, so the next thread becomes the new lock owner and gains access to the critical section. With this idea, threads spin on their private

queue node, minimizing the number of coherence misses to two (one when the lock is received, and other when it is transferred to the next thread). Different approaches have been designed with this idea, such as MCS [106] or CLH [28] locks.

The MCS lock [106] is presented in Figure 1-6. The lock object contains a single field, a **tail** pointer, being *null* when there is no lock requestor, or pointing to the last requestor *qnode* when there are any. Thread-private qnodes contain the pointer to the next element in the queue, and a status word that indicates the reception of the lock when modified. The first node in the queue, called the head of the queue, is the only lock owner. Whenever this thread releases the lock, it will modify the status word of the next qnode, if present, so the lock is transferred. Otherwise, the **tail** pointer is set to *null*, releasing the lock.

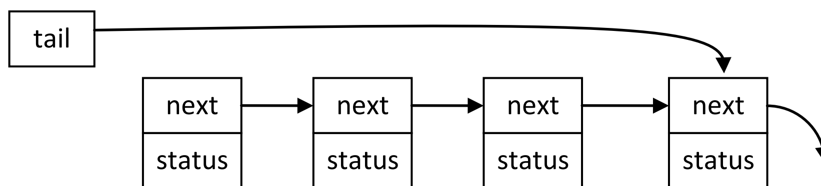


Figure 1-6: Example of the MCS lock structure

1.3.1.2. Spin-locks vs. try-locks vs. blocking locks

Spin-locks, as considered above, are those that keep threads spinning on the iterative request code until the lock is finally acquired. This can hurt performance, since the execution of the waiting thread could prevent the execution of the lock owner thread due to lack of resources. They are also called busy-locks.

Blocking[1] locks are implementations in which, after a certain spinning time, the thread is suspended. In such case, the kernel of the OS is responsible to wake up the thread again when the lock is released and ready to be acquired. This is the implementation of many common OS locks, such as Solaris adaptive locks [104].

Another alternative is the use of *trylocks*. A thread calling *trylock* will acquire the lock if it is free or, optionally, spin-wait for a given time if it is acquired. If the lock is not released within the timeout, the function returns without acquiring the lock, letting the programmer decide the next step. Such an implementation can be used to resolve deadlocks, using the trylock timeout as a deadlock indicator, as implemented in many databases [41] and lock-based STMs [36, 46, 124].

[1] Note that "blocking" does not refer here to the forward progress condition of the lock code as discussed in section 1.4.8.1. Essentially, when considering the progress condition all locks are blocking, since a lock-free or wait-free implementation would require finding the lock always free.

1.3.1.3. Reader-writer locking

Locks protect concurrent access to shared locations to prevent update problems. However, in some cases, we might be interested in a read-only operation on a shared object. While such operation must be protected from concurrent writers, multiple threads can read the same object concurrently since the state is not modified. Reader-writer locks [27] implement such policy, allowing for multiple threads to hold the lock in a read-only mode, or a single thread in the write mode.

Workloads using reader-writer locks are typically dominated by readers, this is, reading the values of an object without modifying it is much more frequent than updating the object. However, while these locks allow for multiple readers to proceed in parallel, their performance is generally low. The problem arises from multiple readers having to update a single common field in the lock word, with the coherence invalidations that it entails.

In [107] Mellor-Crumley and Scott propose a reader-writer version of their MCS lock. An example of their lock organization is shown in Figure 1-7. Their implementation includes a **reader_cnt** counter, which has to be increased on each *lock()* and decreased on each *unlock()* called by readers. When a releasing reader decreases this value to zero, it notifies the next waiting writer (if present), using the **next_writer** pointer in the lock object. The use of the reader counter generates coherence invalidations that make readers conflict with each other at the coherence level (in the sense that they invalidate each other's shared block), despite they are logically allowed to proceed in parallel. However, it is required since the acquire and the release order of concurrent readers is not necessarily the same.

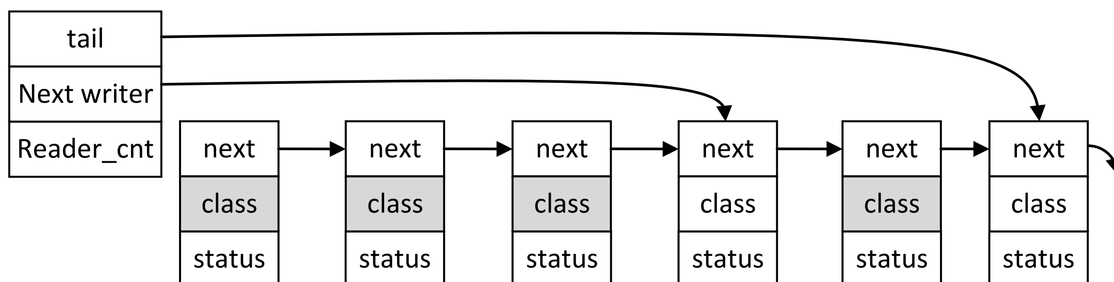


Figure 1-7: Example of the MCS reader-writer lock structure. Grey fields represent readers

Krieger *et al.* propose in [82] a reader-writer version of the MCS lock that does not require a reader counter. Instead, it requires a double-linked list to allow for out of order readers release. In their queue, readers will modify the queue on release (using their two pointers in the qnode) to remove their node from the queue. While this prevents the coherence congestion in the reader counter, it still requires a field which is updated on all lock accesses (a **tail** *pointer*) and a more complex queue management. Their implementation is depicted in Figure 1-8.

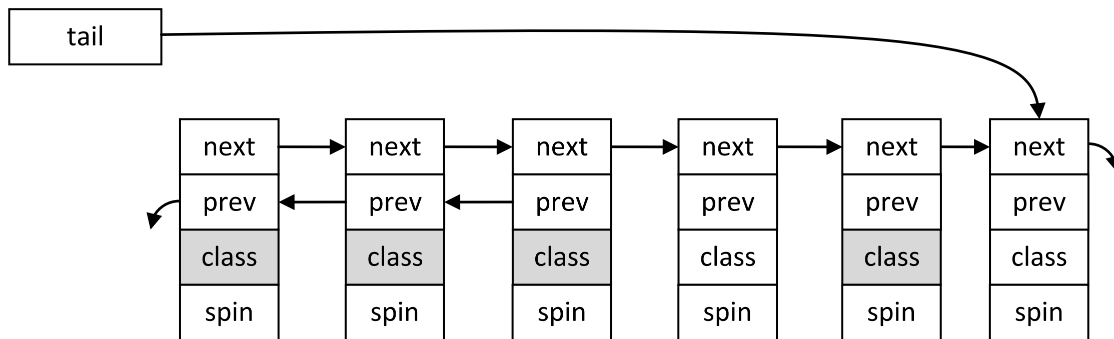


Figure 1-8: Example of the Krieger *et al.* reader-writer lock. Grey fields are readers

Lev *et al.* propose different implementations for scalable reader-writer locks in [95]. Their idea is to replace the reader counter with a variation of the Scalable Non-Zero Indicator (SNZI) proposed in [40]. SNZI is a tree-based structure that is used to determine with a single read of the root node if there is *any* reader, independently of the actual number. Each thread is assigned a leaf of the binary tree in the SNZI. Whenever a thread acquires the lock in read mode, it sets its leaf value from FALSE to TRUE, and propagates this value towards the tree root. The propagation proceeds until it finds a tree node being already true. Similarly, lock releases set the leaf to FALSE and propagate the value only while all children are false. This mechanism updates the root node only when the number of readers changes from 0 to 1, or 1 to 0, which are the interesting state changes for writers. This increases the throughput when the number of readers is high, since it prevents the problems of coherence congestion on a single location, at the cost of higher requirements (much higher memory requirements and longer execution time for locking and unlocking operations without contention).

1.3.1.4. Fairness issues

Fair implementations of locks ensure that all threads trying to access a resource eventually do so; unfair implementations favor a certain group, what can degenerate on starvation in a highly contended case. An example occurs with hierarchical locks, in which the threads are grouped in blocks (for example, to favor the lock transfer between threads in the same chip, in a multi-CMP system). If such policy is taken to the edge, a chip with multiple consecutive accesses to the lock would prevent any access from threads in remote chips. Those remote threads would stall in their spin wait, generating starvation. Another paradigmatic example is the case of reader-writer locks with reader preference. In such implementations, when the lock is in read mode, new readers can access the lock in the same mode, but writers must wait for all the readers to finish. If a lock is frequently accessed by readers, writers can become blocked, waiting for the (inexistent) case of all readers finishing their read-lock.

An example of starvation with unfair reader-writer locks is presented in Figure 1-9. The code implements a barrier based on a reader-writer lock. When a thread reaches the barrier, it will first increase the counter of threads in the barrier (acquiring the barrier lock in write mode, lines 2-4). Then, it will iterate waiting for the arrived threads count to equal the required

number of threads (acquiring the lock in read mode for that, lines 5-11). When the number of arrived threads is high, they will all iterate in the second section, maintaining the lock in read mode and preventing any incoming thread from modifying the counter field. This example is highly artificial, since the access to **b.counter** is atomic and does not require the read lock, but it illustrates the starvation problem, and how it can affect the global progress.

```
1:  void barrier(barrier &b){
2:      unfair_writelock(&(b.lock));
3:      b.counter++;
4:      unfair_writeunlock(&(b.lock));
5:      finished = 0;
6:      while (finished == 0){
7:        unfair_readlock(&(b.lock));
8:        if (b.counter == b.threads)
9:            finished = 1;
10:       unfair_readunlock(&(b.lock));
11:     }
12: }
```

```
struct {
    long counter;
    long threads;
    lock lock;
} barrier;
```

Figure 1-9: Incorrect implementation of a barrier using reader-writer locks

The solution typically passes by providing a *fair* access to the resource, for example, implementing a FIFO queue for requestors (independent of their read or write access mode).

Starvation directly affects the forward progress of the starved threads. Indirectly, it can also affect the overall program forward progress, if it depends on the forward progress of the starved thread, leading to a blocking situation.

Fairness does not necessarily imply the use of a queue, for example ticket locks use a counter to determine the next thread to acquire a lock in FIFO order but with a contended, centralized mechanism. Similarly, fairness does not require strict FIFO order; different levels of fairness could be considered. For example, a lock without FIFO order, but guaranteeing starvation freedom of all requestors within a large timescale would be “more fair” that the simplest reader-preference lock that starves writers.

1.3.1.5. Hierarchical lock implementations

Hierarchical locks [117] improve the performance in systems with different latencies between processors, such as a CC-NUMA model [91] or a modern system comprising multiple CMP chips. The fundamental idea of a hierarchical lock is to group requestors in clusters, so that communication between members of the same cluster is fast (such as different cores in the same chip), while the communication between cores in different clusters is slow (cores in different chips). Then, the implementation tries to favor the lock transfer to a member of the same cluster on the release operation.

The specific details depend on the lock implementation. A hierarchical backoff lock [117] can vary the exponential wait between requests, with lower wait times for members of the cluster that owns the lock. This favors the re-acquisition of the lock by a member of the same cluster. A hierarchical CLH queue lock [97] uses a single global queue comprised of multiple local queues, or 'splices'. Neighbor requests are appended in the same local queue. The first requestor from each cluster appends its local queue into the global queue. Once this happens, subsequent local requests build a new local queue, which will be appended to the global queue in a later position. Since this happens before the first requestor acquires the lock, the maximum number of requests from a given cluster equals the number of threads in the cluster. This favors the transfer of locks within the same cluster, without starving other clusters.

In general, any implementation of a hierarchical lock would trade the fairness of the lock with the locality achieved: In one end, the lock is transferred in strict FIFO order; in the other end, the lock is always transferred to a member of the same cluster, if requested.

1.3.1.6. Pathologies of locks

Pathology represents a behavior that deviates from the ordinary or expected one. When a pathological behavior occurs, the performance degrades because the lock implementation is not designed to handle such case. In this section, different pathological cases are presented and analyzed.

1.3.1.6.1. Single-thread locking and biased locks

Locks are designed by definition to be taken by multiple threads. If a lock is always taken by the same thread, it means that the object protected by the lock is not effectively shared and the lock is redundant. In such case, the lock can be removed to improve performance and reduce memory usage.

However, in some cases, locks are private to a single thread, but this cannot be determined at compile time. The paradigmatic example comes from Java locks and the *synchronized* methods. Such methods require a lock to ensure thread-safety, but in many cases, they are private to a single thread. Even more important is the fact that this privacy cannot be determined at compile or runtime, so the locks must remain.

Biased locking [79, 122] is a technique that assigns an owner to a given lock. While the same owner keeps acquiring and releasing the lock, its operations are fast. This is obtained by providing a fastpath in the code that does not require atomic operations. Other threads that want to access the same lock will require unbiasing it, which is a costly operation.

1.3.1.6.2. Waiting thread suspension in FIFO locks

Queue-based locks provide FIFO order, what guarantees strict fairness in the access to a lock. However, in some cases this can be counterproductive. When the lock is released, it is transferred to the next requestor in the lock queue. If that specific lock requestor is not active, the lock will block, without any thread accessing the critical section. This can happen if the lock has been evicted by the Operating System because a higher priority task is present. Similarly, if there are more threads than processors and all the threads compete for a lock, many waiting threads will be evicted at a given moment. This generates temporary starvation, since no thread accesses the critical section protected by the lock and no forward progress is obtained.

To overcome this problem, different versions of queue-based locks with eviction detection have been developed in [128, 127, 67]. These are queue based locks with a timestamp on each queue node. Waiting threads periodically update their timestamp, and check the timestamp of the previous node. If a given node is not updated for a long time, it means that the waiting thread has been evicted. In that case, the next node in the queue will detect the eviction and remove the useless node. This prevents the problem of granting the lock to an evicted thread, and also provides support for abortable locks (the *trylock* construct) with FIFO priority.

1.3.1.6.3. Mostly Reader locking

A final pathology that can be considered is the case of a reader-writer lock in which the lock is taken in read mode most of the time. This is the case for an object frequently read by multiple threads, but hardly ever updated by any of them.

For example, consider a binary tree protected with fine-grain reader-writer locks. Each node in the tree contains an associated lock. This lock needs to be read-locked when the node is read and write-locked on a node update. For example, during a search on the tree multiple nodes are traversed and read-locked until the sought node is found. If such node is to be updated, it (and some of its predecessors, depending on the algorithm) will be write-locked. In this structure, the root node is read-locked on all accesses (tree *lookups*, *updates* and *removals*), but hardly ever write-locked, only when the root node or one of its immediate successors needs to be modified.

Additionally, different situations might lead to read-only locking. If the protected object is really never accessed in write mode, the lock is redundant, since the only point in read-locking is to protect the object from concurrent updates. Such behavior should be detected at compile or run time, and the lock should be removed. However, in some cases, this issue might go undetected and the lock be present in the executed code. Alternatively, and highly related, a read/write lock might be assigned to an object which is initialized once at the beginning of the execution and accessed always in read mode. While these are examples of wrong coding, they can go unnoticed and lead to the same “mostly reader locking” pathology as discussed in the previous paragraph.

1.4. Fundamentals of Transactional Memory

Transactional Memory aims to simplify the complexity of parallel programming. It does so by importing the concept of *transactions* from databases to shared-memory programming. Essentially, transactions are atomic blocks that run as if they were isolated in the system, with a runtime mechanism that serializes their execution and commits or aborts them as required. The idea was first suggested by Lomet in [96], and the first feasible implementation was proposed by Herlihy and Moss in [70]. A recent survey on Transactional Memory can be found in [88].

There are multiple proposals to implement TM using specific hardware, known as Hardware Transactional Memory (HTM). The most relevant HTM proposals are Stanford's Transactional Coherence and Consistency (TCC, [61]), Wisconsin's Log-based Transactional Memory (LogTM, [110]) and Bulk ([18]). These systems typically extend the coherence protocol to detect isolation violations and to guarantee that transaction commit is seen as atomic. They will be detailed in section 1.4.11.

Software Transactional Memory (STM) systems perform the necessary tasks in a software runtime. They typically have much higher overhead than HTM proposals since they need to handle the versioning and conflict detection in software, but they can run in any generic platform. Multiple mechanisms have been developed, to study different aspects, such as contention management, forward progress guarantees, update mechanism, etc. They will be studied in section 1.4.12.

Finally, certain TM proposals offload some tasks to specific hardware, for example, the conflict detection, while other parts of the runtime run in the software-only system. There are two variants: Hardware-accelerated TMs (HaTM) offload some part of the runtime to a specific hardware, so it cannot run without such hardware. Some examples are SigTM [17] or FlexTM [134], discussed in section 1.4.13.1. By contrast, Hybrid Transactional Memory (HyTM) allows for concurrent hardware and software-only transactions, with some mechanism required to detect conflicts between each other. Some examples of this variant are HyTM [33] or the HyTM model proposed for the Rock processor [22], presented in section 1.4.13.2.

This section introduces the main characteristics of Transactional Memory systems. Later, the main STM, HTM and hybrid proposals are detailed.

1.4.1. Atomic sections

The key idea of Transactional Memory is to provide an *atomic{}* abstraction to the programmer. The code inside the brackets of an atomic block is executed transactionally[2],

[2] Some authors discuss the difference between atomic blocks and memory transactions [63]. For example, a critical section protected by a lock is atomic with respect to the rest of the code. Along this text we consider them equivalent when referring to TM systems.

similarly to transactions in databases. This means that all of the atomic code behaves as if it would run in a single step, without any possible interference from other threads in the system.

With this tool, the programmer does not need to lock and unlock objects to preserve correctness. Instead, all of the accesses to shared data that need to obtain a consistent view of the memory are performed inside a transaction. The underlying hardware or runtime will ensure that the transaction runs correctly.

Internally, a transaction typically has the following phases:

- Transaction start, which prepares the necessary data structures in the runtime or hardware substrate.
- Transaction execution, which runs the code inside the atomic block. This execution will be, in many systems, speculative, without modifying the shared data in main memory. Alternatively, this execution can modify the shared memory, but maintain an "undo-log" that allows to recover the initial state in case a violation is detected and the transaction has to abort.
- Transaction commit. In this phase, the speculative execution is validated to make sure that it does not interfere with other concurrent transactions. If no violation is found, the transaction is validated (committed) and the changes are made global. Otherwise, the transaction is aborted, all of the changes are discarded, and the execution returns to the first phase.

The set of data that a transaction modifies during its execution is called the transaction's write-set. Similarly, the group of memory locations read by the transaction is called the transaction's read-set. Of course, both sets can overlap, and frequently do, since objects are typically read before being modified.

1.4.2. The ACID properties

The previous section states that the atomic code behaves "as if it would run in a single step". Obviously, running multiple instructions in a single step is impossible; this section details the properties that the code execution has to guarantee to provide the expected correct behavior. These properties have been deeply studied in database theory, and are collectively know with the acronym ACID: Atomicity, Consistency, Isolation and Durability. Transactional Memory typically obeys only two, but all of them are presented for completion.

1.4.2.1. Atomicity

The code inside a transaction is indivisible. This means that it cannot be partially executed; instead, it must run with an "all or nothing" policy. If the transaction commits, all of its changes are made globally visible. If, by contrast, the transaction aborts, all of its changes must be

reverted, with no side effects. This conditions the execution of transactions with “undoable” operations, such as Input/Output operations.

1.4.2.2. Consistency

Transactions move the shared memory state from one consistent state to another. While internally transactions could operate with inconsistent memory states, both the initial and final states must be consistent.

This property is dependent on the specific application, since the meaning of a consistent state of the memory is not initially known. As explained in [88], it typically depends on a collection of invariants on data structures. For example, asserting that **numCustomers** contains the number of items in the **Customer** table, or that the **Customer** table does not contain duplicate entries. Transactional Memory does not typically implement consistency, since it is the programmer’s responsibility to define what a consistent state is. Harris and Peython-Jones incorporate this functionality in Haskell STM in [66].

1.4.2.3. Isolation

Isolation requires that different concurrent transactions do not interfere with each other during their execution. This could happen if two transactions pretend to modify the same memory location, or one transaction tries to read a memory location that has been speculatively modified by another transaction. If this happens, a violation is detected, and one of the concurrent transactions is forced to wait or abort to preserve the isolation property.

While isolation is a property that comes from the database world, there are certain differences in transactional memory. While all database accesses are considered part of a transaction (or a transaction composed of a single operation), in TM transactional and non-transactional code can coexist. In this case, two different approaches have been considered, regarding the interaction of transactional and non-transactional code [100]:

- Weak isolation: non-transactional code can access intermediate data from a transaction. Similarly, memory updates coming from non-transactional code are not detected as isolation violations.

- Strong isolation: transactions are guaranteed to be isolated from both transactional and non-transactional code. Therefore, non-transactional code is not allowed to update the read or write set of a transaction until it commits, or forces an abort if it does so.

Martin *et al*. [100] explain how the programming languages and software verification communities have long used the term “atomic” to mean “in isolation”, and the transactional memory community has largely adopted this usage. Therefore, it is common to find the expressions “weak atomicity” and “strong atomicity” to refer to the properties just explained.

We will use both manners equivalently in this text. They also show how simply translating lock-based code to transactional memory by replacing locks by transactions is incorrect and can deadlock under any of the two isolation models.

Typically, weak atomicity is maintained by software TM systems, while strong atomicity is provided by hardware TM systems that extend the coherence protocol for both transactional and non-transactional accesses. It is the programmer's responsibility to guarantee a correct execution if both transactional and non-transactional code accesses the same shared data in a weakly atomic model. Providing strong atomicity in STM systems require complex mechanisms such as augmenting non-transactional memory accesses to detect conflicts [10], what impacts the overall performance.

1.4.2.4. Durability

Durability is the property of surviving system crashes. Such property is crucial for databases, where the system state must be recoverable after an event that crashes the database server, such as a power failure or hardware failure. To guarantee durability, a transaction log must be maintained and recorded in real time, so the system state can be recovered after an unexpected crash.

Since memory transactions are part of a program execution, and this program will not survive a system crash, durability is not a property maintained by TM systems.

1.4.3. Programmability and composability

Section 1.2.2.6 discussed some difficulties of lock-based parallel programming. This section shows how Transactional Memory targets them and simplifies parallel programming.

The first problem is a complex programmability. Specifically, it is difficult to modify existent lock-based shared data structures, for example, adding a new method to an existent lock-based *set* structure. The problem in such case is that concurrent calls to different methods of the same shared object can concurrently modify it. Therefore, a given method cannot expect to find the object in a consistent view, but can find intermediate changes from other method. This is prevented by the *isolation* and *consistency* properties of Transactional Memory. Since transactions are isolated, there is an underlying mechanism that guarantees that each method call behaves as if it was the only one running, without concurrent calls. Since all transactions are consistent, they can expect to find the shared object in a valid state, independently of the concurrent ongoing transactions. This removes the problems of locking races to validate the correctness of a method in presence of concurrent methods.

The second problem is composability: correct fragments of code may fail when combined. This problem does not arise with Transactional Memory: transactions do *compose*, this is, they can be safely nested one into another. The transactional properties (atomicity, isolation,

consistency) are guaranteed at the outermost level, what prevents any incorrect execution between different threads running concurrent transactions. Deadlock is inexistent, since the transactional runtime detects violations and aborts transactions if needed. Forward progress should be provided, either naturally by the underlying mechanisms, or by using a specific contention manager.

1.4.4. Update mechanism: In-place vs. deferred

Since multiple transactions can try to update the same memory location, there must be a mechanism to guarantee that only one of them succeeds and the others abort (if the location is part of the read sets) or are serialized in a proper order. Different mechanisms for safe memory update have been developed, which can be broadly grouped into two versions: in-place update (or eager update), or deferred update (or lazy update).

In the first case, in-place update, transactions write their memory changes to the actual memory location when they modify it. This requires that transactions record the old values (in some sort of undo-log) to be able to recover the original state if the transaction aborts. Also, a protection mechanism is required to prevent other transactions from reading or modifying the updated memory location. This has the advantage of providing fast commits, since the transaction has already updated the main memory.

By contrast, deferred update mechanisms save the changes contained in the transaction's write set in a write buffer. If the transaction successfully commits, the write buffer contents are moved to the corresponding memory locations. This requires a copy on commit, so the commit operation takes longer.

1.4.5. Conflict detection mechanism: Lazy vs. Eager, visible vs. invisible readers

Two transactions conflict when the write set of one of them overlaps with the read or write set of the other one. Read-set overlapping is not a problem since multiple transactions are allowed to read the same data concurrently.

To be able to detect conflicts, the TM system must add some metadata to the memory contents. In the case of STM systems, these metadata are typically object locks (such as [36, 46, 124]) or *orecs* (ownership records, such as those in [64, 33]) in the object header. If the STM detects conflicts with a word-level granularity, these locks or orecs will be typically contained in a hash table, accessed using the memory address as the hash key. HTM systems typically extend the coherence hardware with Read/Write bits, use specific buffers to contain the read and write sets, or use some signature-based mechanism to detect the conflicts.

Lazy conflict detection mechanisms (generally called *optimistic* mechanisms) search for data conflicts at commit time. Under this model the transaction runs speculatively and, at commit

time, the object metadata are checked to be correct (either by acquiring a lock, validating a timestamp or propagating some coherence updates, for example). Optimistic mechanisms do not typically require to read-lock objects; instead, the read set is validated with some different mechanism, such as checking an update counter or an object pointer that is changes on each update. This mechanism has the benefit of additional concurrency, and also progress guarantees: at least the committing transaction is guaranteed to progress. However, under this model, starvation can happen, especially for long transactions.

By contrast, eager conflict detection (generally named *pessimistic*) checks for data conflicts earlier, typically on each data access. Pessimistic concurrency control typically requires the acquisition of object reader-writer locks at runtime, what is known as encounter-time locking. A conflict is detected when a thread finds an object lock taken. This implies that the object is being modified by another transaction, known as the blocker transaction. These mechanisms allow for less concurrency, since the eager data blocking prevents other accesses, but the blocker transaction can fail and have to abort. However, if conflicts occur they are detected and handled earlier, preventing the aborted transaction from making more useless work. In general, pessimistic concurrency control mechanisms are preferred when the conflict rate is high, since they provide a better performance due to the eager conflict detection and resolution.

Regarding object readers, there are two possible implementations. *Invisible readers* do not allow determining if a given transaction is reading an object. Instead, the reader transaction will have to validate its read set at commit time to determine if it is still valid. This is the most common approach in STM systems. In these systems, the writer has to update the object's metadata to allow readers to detect a conflict. Such update can be, for example, in the form of a version counter (like the global clock version used in TL2 [36] or tinySTM [120]) or an updated pointer in the object header to the new data block after writing (used in systems such as DSTM [69] or Fraser and Harris' OSTM [47, 48]). By contrast, in a system with *visible readers*, any transaction can determine at a given moment if there is a reader for a memory location. This happens typically in HTM models that leverage the coherence protocol (such as LogTM [110]), and in STM systems with per object reader-writer locks (such as TLRW [34]) or lists of readers.

1.4.6. Data validation and the privatization problem

As presented in section 1.4.5, transactions need to detect conflicts between their own read and write sets and other transactions' write set. However, there are cases in which the validation mechanism can fail to detect a real logical conflict, and lead to unexpected program behavior. This happens in systems with invisible readers in which transactions convert shared objects to private. The problem is therefore known as the 'privatization problem', and it is explained next with the example in Figure 1-10.

```
// Initially x=0, x_shared=true

//Transaction Tx1              //Transaction Tx2

1:  atomic{                    1:  atomic{
2:      x_shared = false;      2:      if(x_shared)
3:  }                          3:          x = 42;
4:  x++;                       4:  }
```

Figure 1-10: Example of code that fails in the privatization problem

In the example, two threads run concurrent transactions Tx1 and Tx2. The object **x** is initially 0 and shared. Tx1 privatizes the object (this would be equivalent, for example, to removing the object from a list or queue) and later works in the object (line 4). By contrast, Tx2 modifies the object, but only if it is still shared (line 2). Intuitively, one would expect to find a final result of **x** = 1 (if Tx1 serializes before Tx2) or **x** = 43 (if Tx1 serializes after Tx2). However, we will consider now two cases that can lead to **x** = 0 or **x** = 42.

In a system with lazy update, the transaction Tx2 can execute before Tx1. At commit, the validation phase finds that **x_shared** (part of the read set) is correct. However, before committing its write set updates, thread 1 runs lines 1-4. The update of **x_shared** in line 2 does not conflict with Tx2, which uses invisible readers and, moreover, has already validated the read set. The later update of **x** in line 4 does not conflict either, since it is a non-transactional access. Finally, Tx2 updates shared memory with its write set changes, overwriting the previous private update by thread 1, and leading to **x** = 42.

The second case occurs if the system uses early update. In such case, Tx2 can run lines 1-3 updating **x** to 42 and recording the old value in its undo-log. Before Tx2 attempts to commit, thread 1 runs lines 1-4 and updates **x** to 43. The commit of Tx2 detects a conflict in **x_shared** and restores the original value from the undo-log, leading to **x** = 0.

An additional problem can occur in any of both cases. Since thread 1 modifies the object outside any transaction, it can temporarily move it to an inconsistent state (This does not happen in the example in Figure 1-10 where **x** is a single variable, but could occur if we considered more complex objects). In such case, Tx2 could read an inconsistent view of the memory status, leading to a 'zombie' transaction with an unexpected result, which is doomed to fail, but could not reach the validation phase if the execution flow depends on the accessed data.

Multiple solutions to the privatization problem have been studied in different sources. Spears *et al.* identify in [142] multiple techniques:

1. Data can be statically partitioned between transacted and non-transacted parts of the heap, with explicit marshalling between them. This is the approach in Haskell STM.

2. STM-aware libraries can be used for private accesses (like line 4 in Figure 1-10) with the overhead that that it entails. This approach is taken in the Java STM in [133] that uses whole-program analyses and JIT optimizations to reduce the performance penalty of memory barriers in non-transactional code.

3. Synchronization barriers can be added such that data is not used both in private mode and in shared mode between barriers.

4. Explicit fences can be required after transactions that make parts of the heap private. This is used, for example, in McRT-STM [124].

5. Pessimistic concurrency control can be used so that if Tx1 is serialized after Tx2 then Tx1 will not commit until Tx2 has committed, or aborted and cleaned up. This option will be studied later in Chapter 2.

The dual of the privatization problem is known as the *publication* problem [3, 108]. The publication problem typically arises with compiler optimizations: an object is accessed before validating if it is private or not. Although making these operations inside a transaction should provide a correct behavior thanks to the isolation property, data races similar to those in the privatization problem can occur, leading to an erroneous behavior. An example is presented in Figure 1-11, in which the variable **x** is initially private. Intuitively, both **r2** and **r3** should finish with values of 0 or 43, depending on the ordering of transactions Tx1, Tx2 and Tx3. However, the explicit software prefetch in Tx2, or even a possible memory reordering in Tx3 (since there is no memory fence between the accesses to **x_shared** and **x**, this case depends on the consistency model) can lead to a result of 42.

```
// Initially x=42, x_shared=false

//Transaction Tx1             //Transaction Tx2          //Transaction Tx3

1:  x++;                      1:  atomic{                1:  atomic{
2:  atomic{                   2:     tmp = x;            2:     r3 = 0;
3:     x_shared = true;       3:     r2 = 0;             3:     if(x_shared)
4:  }                         4:     if(x_shared)        4:         r3 = x;
                              5:         r2 = tmp;       5:  }
                              6:  }
```

Figure 1-11: Example of code that fails in the publication problem

1.4.7. Contention management

The contention management policy refers to the mechanism employed when a conflict is detected. These conflicts occur between two transactions, with at least one of them being a writer. If a writer transaction conflicts with multiple other readers, each pair of conflicts is considered individually.

To handle the conflict, there are two possibilities. The first one is to make one of the transactions wait. For example, a transaction has write-locked an object, when other transaction pretends to read the same object. In this case the second transaction can wait for the object to be unlocked, and then access it. The second option is to abort one of the transactions involved. In the previous example, the reader might abort the writer transaction that has locked the object; when the abort finishes, the reader can access the object.

Typically, HTM systems employ simple policies. In HTM, conflicts are typically detected when a coherence request arrives, so the action to be taken should be fast. STM systems, by contrast, tend to use more complex contention managers. In this section we will consider three simple policies, followed by a study of some managers typically used in STM.

1.4.7.1. Abort-always policy

This policy is used in systems such as TCC [61], a HTM with deferred update and lazy conflict detection. When a transaction commits, its updates are broadcasted on the system bus, and remote processors snoop this bus to detect conflicts with their read or write sets. Once the commit starts, it is guaranteed to succeed. Then, if a conflict is detected, the processor that detects the conflict by snooping the bus aborts its transaction.

This policy guarantees forward progress, since there is always at least one committing transaction. However, it is unfair, favoring shorter transactions that commit faster, and can lead to starvation of threads running long transactions.

1.4.7.2. Requestor-wins policy

The ‘requestor-wins’ policy is used for systems with eager conflict detection, but is very similar to the previous one. In these systems, a requestor sends a coherence request for a block that has been accessed by another transaction. When the conflict is determined, the coherence requestor wins and receives the coherence grants for the block. The receiver of the request is forced to abort the transaction and restart again.

This policy prioritizes non-transactional accesses in a strongly-atomic model, making transactional code to abort when a non-transactional thread requests access to some address. Also, this is the simplest policy to use when a write-buffer is used, where transactional stores are kept private. However, it can introduce livelock when two or more transactions pretend to modify the same data: Each transaction is restarted, retries the conflicting update and makes the other transaction restart again. The solution requires some contention mechanism, such as using an exponential backoff between transactions retries.

1.4.7.3. Requestor-waits policy

This policy is used in systems such as LogTM [110] with eager update and eager conflict detection. A conflict is detected when a requestor processor pretends to perform an access

that conflicts with a remote transaction. The remote processor that detects the conflict sends a **NACK** message that forces the requestor to retry. This stops the requestor, preventing the access to the contended data. Eventually, the transaction commits, and subsequent requests are satisfied.

The benefit of this policy is that it reduces the number of required aborts, stopping threads instead of repeating their work. The problem with this policy is the possibility of introducing deadlock if a cyclic wait is formed (involving different memory addresses). Some mechanism is required to preventively abort some transactions to prevent such deadlock. The original LogTM model, for example, makes use of a timestamp based on the clock of each processor. This timestamp is piggybacked on each request. When a transaction makes an *older* transaction wait, and receives a **NACK** from an *older* thread, it is aborted, guaranteeing that a cycle is never formed.

1.4.7.4. Software policies

Software contention managers tend to be more elaborate since the software runtime is more flexible and can decide the action to take with more data. Multiple implementations have been proposed for different STM systems. Some of the most commonly used advanced policies are:

- Polite [69]: A transaction that detects a conflict retries the conflicting operation multiple times with an exponential backoff wait. If the conflict persists after a threshold, the transaction aborts.

- Karma [126]: The transaction that has accessed a larger number of objects receives priority when deciding which one should abort.

- Greedy [56]: Uses visible reads and favors transactions with earlier start time to provide livelock and starvation freedom.

1.4.8. Progress conditions

As discussed before, concurrent programs contain multiple threads that interact with each other. Therefore, these threads can obstruct the execution of each other. One example has been presented in Figure 1-5, where one thread has to lock part of a data structure to guarantee correctness; concurrent threads that need to access such data will find the lock taken and will have to wait for the unlock call. Similarly, in lock-free data structures, different methods can modify concurrently an object's data, modifying each other's execution path and leading to livelock.

Each method of an object will provide a given progress condition, that defines the worst-case behavior in presence of concurrent accesses to the same object. If all of an object's methods satisfy a given condition (for example, they are all wait-free) the object is also said to

guarantee such progress condition. Transactional memory systems also have progress conditions, defining the worst-case behavior when multiple transactions conflict with each other.

In section 1.4.8.1 different progress conditions are presented. Subsequent sections deal with progress conditions present in different TM implementations.

1.4.8.1. Wait-free, lock-free, obstruction-free and blocking progress conditions

Wait-free is the most restrictive progress condition. It requires that a method call finishes the execution in a finite number of steps. Therefore, the implementation cannot depend on the object state, nor obviously can rely on locks. This does not mean that the execution finishes in a fixed amount of time, since cache misses, page faults or thread de-scheduling can delay the finalization.

Lock-free objects guarantee that *some* method call finishes infinitely often in a finite number of steps. In this case different concurrent methods may disturb each other, but the implementation must be such that at least one of the methods manages to progress. Informally, lock-free algorithms guarantee system-wide progress, while wait-free ones guarantee per-thread progress. Obviously, implementations with locks are not lock-free, since the eviction of a lock owner can starve all of the remaining threads. However, algorithms that do not use locks are not necessarily *lock-free*: consider the classical example of Dekker synchronization [38] between two threads, in which a thread accesses a critical section and then passes the turn to the other thread. If one thread is temporarily halted, the other thread will not be allowed to enter the critical section *infinitely often*.

A more relaxed condition is *obstruction-freedom*. A method is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps. This means that no method can move the object data into a state in which exclusively it can progress, since all methods should be able to progress if they run in isolation. Also, with this progress condition, different active threads can cause livelock if they interfere with each other.

Finally, *lock-based* (or *blocking*) implementations are the less restrictive ones. They are the simplest to program, but they can only guarantee forward progress for the thread that acquires a lock. If such thread is suspended, it can block any other thread that requires access to the same lock.

1.4.8.2. Non-blocking TM

Initial efforts on Transactional Memory aimed towards non-blocking runtimes. This guarantees that, even if two transactions conflict in their data access, one of them will proceed in a finite amount of time. This is the most restrictive model, since wait-free implementations are

obviously impossible: two conflicting transactions cannot concurrently proceed in a finite amount of steps.

Commonly, many lock-free TM implementations will record object ownership at commit time. This means that the transaction is about to modify such object if the transaction commits successfully. If a given transaction acquires ownership but it is later evicted, a non-blocking implementation allows for other committing transactions to “help” the evicted one to commit or abort. This ensures that the active transactions can proceed, at the cost of aborting other evicted committing transactions.

Multiple early TM systems are lock-free. For example, DSTM [69], Rochester STM [99] and Harris and Fraser’s STM [64] are all nonblocking. Most hardware proposals that use deferred update and lazy conflict detection are nonblocking (since speculative data are kept private until commit, what cannot affect other transactions), such as TCC.

1.4.8.3. Lock-based TM

Ennals [42] argued that lock-free TM implementations add an unnecessary overhead and complexity to the system, since the progress conditions they pursue are unnecessary in most cases. In a TM system, the OS scheduler should be aware of the status of different transactions to prevent the eviction of transactions in a critical section. Moreover, such eviction would be typically brief, so starvation would be temporary. Furthermore, lock-based implementations can provide higher performance, what amortizes the penalty of the possible temporary starvation.

Multiple lock-based TM systems have been developed. All of these systems use a two-phase locking mechanism [43] to guarantee correctness: In the first *grow* phase locks are taken, and in the second *shrink* phase locks are released, guaranteeing that no lock is released before another one is taken. Transactions can be linearized in the mid point when all locks are taken. For example, TL2 [36], TinySTM [46] or the Intel STM compiler [124] are internally based on write-locks. Other proposals argue for the benefit of reader-writer locks, such TLRW [34]. Additionally, there are hardware proposals such as LogTM [110] that are also blocking, since they extend the coherence protocol to “block” the access to memory using **NACK** messages.

1.4.9. Nested transactions

Nested transactions are transactions called inside another transactional section. Supporting transaction nesting is required for the modularity of the TM system, since otherwise transactions would not be allowed to call other functions without knowing if they start a transaction or not.

There are two possible variants of nested transactions, derived from database transactions. *Closed nesting* is the most intuitive version. It maintains atomicity and isolation at the

outermost level. By contrast, *open nesting* [156] releases the atomicity and isolation of internal transactions when they commit. Therefore, the changes of an internal transaction are made visible *before* the external transaction commits. This allows for greater concurrency and performance, at the cost of a much more complex implementation. Open-nested transactions commonly require compensating actions, triggered if the outermost transaction aborts. For example, an open-nested internal transaction could be used to atomically allocate memory using *malloc*; the compensating action would be to release such memory calling *free* if the external transaction aborts. In this example, multiple concurrent transactions might call *malloc* using open nesting without conflicting with each other, increasing the parallelism. However, reasoning about compensating actions is a very subtle task, highly prone to errors, what removes the alleged simplicity of programming of Transactional Memory

Regarding the implementation, most TM systems only support closed-nesting. The simplest implementation uses flat-nesting, making no difference between different levels of nesting. In this case, conflict detection and aborts are performed at the external level, requiring a complete restart even if conflicts occur in an intermediate internal transaction. More elaborate implementations can maintain different status checkpoints for the different nesting levels. This allows for partial aborts, preventing a complete abort if a conflict is detected in an internal transaction.

1.4.10. Irrevocable transactions

Transactional execution is by its own nature, speculative. If a conflict is detected between two or more transactions, one of them will be selected as a victim and delayed, or aborted and restarted. Aborts are unavoidable when two transactions speculatively modify a common memory location. However, some operations cannot be undone in the case of an abort, because their effects are globally visible at the time of being performed. Especially, this is the case of IO operations when accessing to disk, GPU, network and other devices. Such operations are called *irrevocable*. Other operations must be globally visible because they are related with the OS state, such as memory handling with *malloc* and *free* or process termination. A naïve TM implementation would simple forbid these operations inside transactions, but this reduces the effectiveness of the transactional block.

One simple solution is to delay the irrevocable operations until the transaction is safe to commit. Commit handlers for HTM are proposed in [105] as a mechanism to register the irrevocable operations that have to be performed, and delay them until commit. Other systems also employ a similar mechanism [111, 160]. Although these mechanisms suffice for many simple cases, they are not general: Composed operations, such as an output which depends on a previous input, cannot be performed, since the control flow of the transactional execution cannot depend on the irrevocable actions.

An alternative mechanism consists of running irrevocable operations eagerly, but adding compensation actions in case of a transaction abort [62, 105, 111, 160, 138]. These compensation actions undo the effect of the globally-visible irrevocable operation; for example, a *malloc* syscall can be undone with the corresponding *free*. However, several issues arise with this implementation. First, the design of an irrevocable action is not always possible, since some operations such as IO cannot be undone. Second, the compensation action might have unexpected interactions with other transactional code, leading to unexpected results. Finally, derived from the previous case, the use of compensation actions makes the programming model more complex, what is against the simplicity idea that drives Transactional Memory.

Finally, other solutions [61, 143] propose the notion of irrevocable (or *inevitable*) transactions. Irrevocable transactions are guaranteed to commit, and can therefore omit the versioning task. However, a single irrevocable transaction can exist at a given moment; otherwise, a conflict between two irrevocable transactions would not be solved, since none of them can abort. Simple implementations such as TCC [61] permit execution of the irrevocable transaction only, if it exists. More elaborate mechanisms [143], allow irrevocable and ordinary transactions to run in parallel, but the conflict resolution mechanism always favors the irrevocable one in case of conflict. The clear problem with this implementation is that it precludes any parallelism within irrevocable transactions, even when they do not conflict with each other.

1.4.11. Hardware Transactional Memory

Hardware Transactional Memory (HTM) implementations rely on specific hardware to provide the atomicity and isolation properties of transactions. Since such hardware always has a limited size, HTM systems must take care of resource overflow. This section reviews the most influential HTM proposals.

HTM typically relies on a checkpointing mechanism in order to restore the processor architectural state. Such mechanism has been largely used in microarchitectural designs, in order to tolerate branch misspredictions and speculation failures. Most HTM designs leverage such checkpoints save the architectural status at the beginning of a transaction and restore it after a transaction abort. With the checkpoint restoration (and the memory, if required), the processor returns to the exact status that it had at the beginning of the transaction. Checkpoint mechanisms are further discussed in section 1.5.3.1.

1.4.11.1. Transactional Memory by Herlihy and Moss

The first viable proposal of Hardware Transactional Memory was introduced by Herlihy and Moss in 1993 [70]. They suggested the use of a small cache for speculative transactional accesses. If the transaction commits, the values of the transactional cache are made globally visible and other processors can snoop them. If it aborts, these values are discarded. The

transaction size is limited by the buffer size, with overflow not allowed. This means that the programmer must be aware of the buffer size, to prevent exceeding the capacity. A variant of this original model is being considered by AMD for inclusion in their hardware, as presented in [37].

1.4.11.2. Transactional Coherence and Consistency (TCC)

Transactional Coherence and Consistency [59] is a system that leverages memory transactions to simplify the coherence and consistency mechanisms. In this model, all the code is transactional, with speculative updates being buffered in a per-processor transactional write buffer. The system makes use of a centralized broadcast bus, where memory updates are lazily broadcast on transaction commit. Remote processors snoop this bus to detect transaction conflicts and abort. When the transaction size exceeds the capacity of the transactional write buffer, the overflowing processor acquires exclusive ownership of the bus in a form of "stop the world" mechanism. This halts any other remote transaction that cannot access main memory, until it is released at commit time.

The scalability problem derived from the centralized bus was attacked in [20]. This work extends the same idea to a directory-based system where multiple directory controllers are present. Transaction commits "reserve" those directories controller that are written on the transaction and, when all of them are reserved, the transaction is safe to commit. Directory controllers update the memory changes to the coherence sharers, what allows for conflict detection and transaction abort.

1.4.11.3. Bulk Transactional Memory

Bulk [18] is a HTM proposal using a similar hardware to the one in TCC. The bulk system is both applicable to HTM or other systems that use atomic commit of multiple operations, such as Thread-Level Speculation (TLS, [140, 145, 60]) or checkpointed multiprocessors (detailed later in section 1.5.3). Bulk introduces the idea of using a Bloom filter [14] to encode the read and write-set information. Two filters named *signatures* are used to record the read and write set contents. Each signature represents a superset of the accessed memory locations, with a fixed length of some Kbits. Instead of detecting conflicts at line-level, Bulk looks for conflicts in the transaction signatures. This simplifies the implementation, since it removes the need for tagging with read/write bits L1 cache and broadcasting invalidations of the transaction, and removes the overflow problem. Instead, the write signature is broadcast on transaction commit. Remote processors check it against their local read or write signatures to detect conflicts and abort if required. This mechanism has some possibility of false transaction aborts, but reduces the network traffic and simplifies the implementation. The evaluations show that such false abort is negligible, with global performance being competitive to other proposals.

1.4.11.4. Log-based Transactional Memory (LogTM)

Log-based Transactional Memory [110] is a HTM system with eager versioning and eager conflict detection. The L1 caches are augmented with Read/Write bits, which are set when a transactional operation accesses or modifies a memory line. Transactional memory updates are made in-place, and the old memory values are recorded in a per-thread undo-log located in shared memory. The transaction conflict detection is eager, adding **NACK** messages to the coherence mechanism as described in section 1.4.7.3. These **NACK** messages are also used to maintain transaction isolation. The transaction commit is fast, since the transactional memory updates are not speculative: The only task in the transaction commit is to flash-clear the R/W bits in the L1. By contrast, transaction abort requires stepping through the log, restoring each modified memory update before isolation is released.

Resource overflow is handled in a smart way in LogTM. There is no problem with the per-thread undo log, since it is stored in main memory. When a "transactional writer" has to evict transactionally modified data due to cache resource overflow, it sends the block to main memory in a special "sticky" state. In this state, the memory contains the updated data, but the directory controller still records the "transactional writer" as the block owner. Thus, when a request for the block is received, it is forwarded to the "transactional writer", who can defer the request (with a corresponding **NACK** message) or decide to abort.

Different versions of the LogTM system have been proposed. Multiple pairs of R/W bits can be used to support different levels of nested transactions, with both open and closed nesting, as presented in [111]. LogTM-SE (Signature-edition) [158] replaces the R/W bits in the L1 with read and write signatures, as detailed in section 1.4.11.3 for the Bulk proposal. LogTM-VSE [150] proposes additional hardware to virtualize the transactions allowing for migration (copying the transaction signatures to the new processor) and supporting context switching and paging.

1.4.12. Software Transactional Memory

Software Transactional Memory (STM) systems employ a runtime that performs the required actions to provide the correct behavior for user transactions. This means that data versioning, conflict detection, contention management, atomicity and isolation requirements and any other required task are handled in software-only functions, similar to database implementations.

This section reviews the most important STM proposals. While there have been a multitude of systems proposed, we review here those that have obtained a larger visibility and more affect this work. Thorough reviews can be found in [88] and [15].

1.4.12.1. Software Transactional Memory by Shavit and Touitou

Shavit and Touitou proposed in 1995 the first software version of a TM, coining the term Software Transactional Memory [132]. They propose a nonblocking system which requires that transactions declare in advance the locations that will be accessed. This allows for transactions to acquire all required locks in advance, in a given order (according to their increasing memory addresses) to prevent deadlock. Once a transaction has acquired all locks, it is guaranteed to commit.

To provide nonblocking properties, the system must guarantee that transactions proceed once they have acquired their locks. If a transaction is preempted once it has the locks acquired, it can block other transactions requiring the same locks. In this case, the blocked transactions use the "helping" mechanism described in section 1.4.8.2, running the code of the blocker transaction that already holds the locks. Once the blocker transaction commits, locks are released and other transactions can proceed.

This system does not provide much flexibility, since it requires the programmer to know the accessed locations in advance. Subsequent sections focus on dynamic systems in which transactions access any location.

1.4.12.2. Software Transactional Memory for Dynamic-Sized Data Structures

Dynamic-STM (DSTM, [69]) is a deferred update obstruction-free system that does not require the programmer to declare accessed locations in advance. It introduces an explicit selectable congestion manager. It also introduces the use of "early release", this is, the possibility of a transaction to remove unnecessary accessed objects from the read set, to decrease the conflict rate. While this characteristic can be used to increase performance, it is very risky since it can lead to incorrectly synchronized programs, and removes the claimed simplicity of Transactional Memory.

DSTM is available for Java and C++. Read objects are tracked on a read-only table that records the pairs of object-version identifiers. Transactional memory updates are handled by cloning: when an object is open for reading, a private copy of the object is allocated with the new content. The different versions are accessed by means of a double indirection in the object header, which follows the structure presented in Figure 1-12. Atomicity in transaction commit is handled by atomically modifying the single transaction status word from ACTIVE to COMMITTED, after validating that all read objects remain unmodified. Obstruction freedom is obtained by transactions being able abort the owner transaction and to duplicate the object locator in case of a conflict.

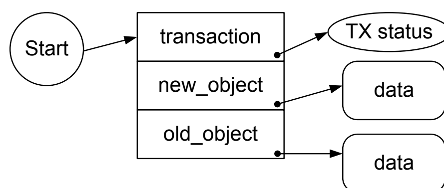


Figure 1-12: Object structure in DSTM

1.4.12.3. Fraser's OSTM

Keir Fraser presented in his PhD [47] an object-based STM providing lock-free progress guarantees. Each object contains a header with a pointer to the current data, as presented in Figure 1-13 on the left. Each transaction maintains two lists for read and written objects. These lists record the locations of the object header and the original data field, and, in the case of the write list, the new data field. The structure of the object handles of the write-set lists is presented in Figure 1-13, on the right. The read-set list is analogous, but both old and new object pointers point to the same data block.

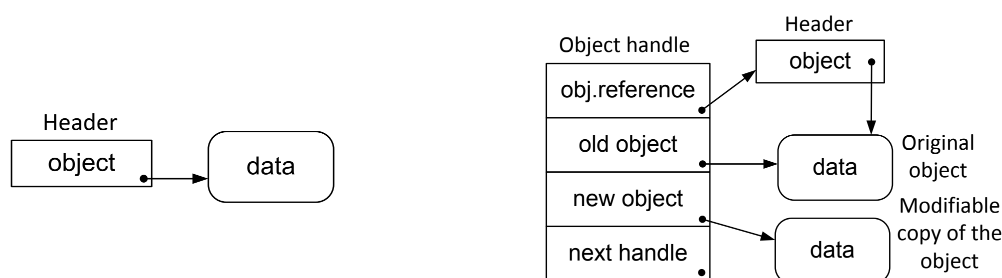


Figure 1-13: Object structure (left) and object handle in the write-set list (right) of Fraser's OSTM

Transaction commit is performed by "reserving" each of the modified objects in the write set. To do this, "bit stealing" is used in the object header pointer. Since valid pointers are always odd (due to the machine word alignment requirements), the last bit is used to indicate if a given object is reserved. Reserved objects contain a pointer to the owner's transaction descriptor, with the last bit set to 1. Such bit has to be checked on every object access to prevent data races. Once all the write set is reserved in this manner, the read set is validated (checking that both old and new pointers are the same) and transaction commits, updating each object's header pointer to the new actual content.

Transaction commit is lock-free, since it employs the helping mechanism introduced in section 1.4.8.2. Threads that find an object reserved (determined by last bit of the header pointer) access the owner's transaction descriptor and help it proceed with the commit. While this can make that multiple threads perform the commit of a single transaction in parallel, it provides the desired lock-free properties.

Two variants of this system have been proposed. First, the word-based WSTM system proposed in [64] by Harris and Fraser has a similar behavior, but the versioning and conflict

detection mechanisms are applied at the memory word-level. To avoid tagging each memory word with an additional header, there is a single ownership record (*orec*) table. This table is accessed for each location by means of a hash function. Each location in the table contains a version number to determine if there have been changes in the associated memory words, instead of using the data pointers to detect changes. The system structure is presented in Figure 1-14, obtained from [64]. The same bit stealing mechanism is used in orecs to distinguish between transaction descriptor pointers and object version numbers.

The second derived system is a reader-writer lock-based version of the initial OSTM. This system contains an additional lock field in the object header. This last version is the one used as the base model for our proposal, and it is detailed in section 2.2.

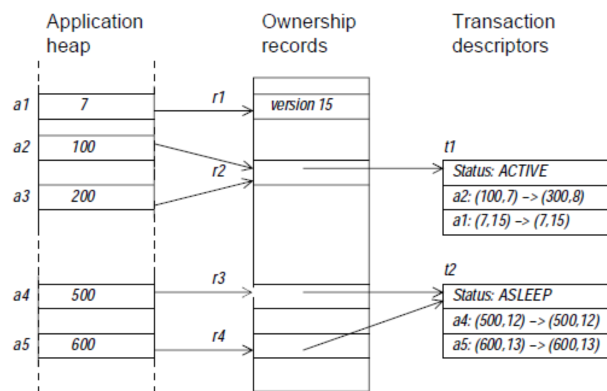


Figure 1-14: Orec organization in the word-based STM by Harris and Fraser, taken from [64]

1.4.12.4. Lock-based STMs

As discussed in section 1.4.8.3, multiple STM systems are nowadays being designed with a lock-based runtime. The management of locks is made in the internal runtime, what hides the complexity from the programmer. Deadlock is either prevented by acquiring the locks in order, or avoided by aborting transactions if a try-lock operation timeouts.

Different systems that rely on a mutual exclusion locking mechanism are Transactional Locking II (TL2, [36]), tinySTM [46] or the Intel C++ STM [124]. These systems use invisible readers and suffer from the 'privatization problem' what requires some explicit mechanism for the programmer to handle it.

Additionally, other systems have been developed by using reader-writer locks. Dice and Shavit discuss many benefits for such design in [34]. Among them, they specify a simpler validation of the read and write set ("what you lock is what you get"), stronger progress properties (especially for long transactions), implicit privatization and support for irrevocable transactions [143]. Their proposal relies on a simplified single-line reader/writer lock that assigns one word of the line to each possible reader.

1.4.13. Hardware-software Transactional Memory

Hardware Transactional Memory systems can be limited in many ways – such as running certain OS operations inside a transaction or not allowing unbounded transactions due to virtualization constraints. Additionally, their hardware requirements are in many cases excessive, or impose overheads in non-transactional code, such as adding new metadata to every coherence line. On the other hand, STM systems are inherently slower than HTM, due to the overheads of bookkeeping, validation and commit in software-only systems, but still are more flexible than HTM systems.

With these concerns, multiple systems have been developed to use both hardware supported operations and STM code [17, 33, 85, 135, 109] seeking for a cost-effective solution. We can distinguish several characteristics in these systems:

- The *hardware-dependency* determines if the system can run transactions in a software-only mode if the hardware is not available. Hardware-dependant systems require the hardware to execute part of the transactional runtime. Therefore, the code compiled for such systems cannot be run in ordinary systems, what restricts their portability. Hardware-independent systems allow for correct but slower execution if the required hardware is not present.

- The *hardware-specificity* determines if the system requires some specific hardware (such as [125, 135, 17]) or it relies on a generic, bounded HTM for the hardware acceleration (such as [33] or the hybrid proposal for the Rock processor [22] detailed in [35]). Generic systems rely on the bounded HTM, and revert back to a software-only solution if the hardware capabilities are exceeded.

In general, we will refer to Hardware-Accelerated TM systems (HaTM) when the system runs a modified STM that relies on a specific hardware to offload some tasks, and Hybrid TM systems (HyTM) when the system is hardware-independent and allows for HTM and STM transactions running concurrently. Both options are detailed in the next sections.

1.4.13.1. Hardware-accelerated TM

Hardware-accelerated Transactional Memory systems run the required operations in an ordinary STM, but offload some tasks to a specific hardware. In this section we consider the most influential proposals.

1.4.13.1.1. Hardware-Accelerated TM (HATM)

Hardware-Accelerated TM [125] introduces “mark bits”, new per-thread metadata added to each memory blocks (for example, for each 16-byte block on each line). These bits are set on transactional reads, and cleared when coherence events occur with the given line. If the bits remain set at commit time, they prevent much of the validation operations. Otherwise, the

original STM mechanisms must be used. The system is oriented towards providing a small legacy for future generations of processors: if the mechanism is not present (equivalent to reading always 0 in the mark bits), the system still behaves correctly.

1.4.13.1.2. Signature-accelerated TM (SigTM)

In [17] the authors propose SigTM, a HaTM system that uses signatures for conflict detection. Any other task is run in software, allowing for flexible implementations such as different contention management policies. The system is focused in providing strong isolation, what is granted by the coherence extensions: All coherence requests are checked against the local signature, what allows for fast conflict detection between transactional and non-transactional code.

1.4.13.1.3. Flexible TM (FlexTM)

Rochester's FlexTM [134] is a HaTM that accelerates the Rochester STM [99]. It introduces several possible hardware accelerations to offload different tasks of the STM runtime, some of them already introduced in previous works:

1. A signature mechanism for the read or write sets, similar to SigTM.
2. A per-thread Conflict Summary Table (CST) that records the identifiers of conflicting transactions.
3. Programmable Data Isolation, a coherence extension that provides isolation by hiding transactional updates to remote threads. It is based on a MESI protocol, and includes new coherence states and transitions, specific for transactional status of cache lines.
4. Alert-on-Update, a conflict-detection mechanism that allows to "mark" memory locations and to receive an asynchronous exception when they are modified by a remote thread.

Depending on which of these mechanisms are used, FlexTM can implement different conflict detection mechanisms (eager or lazy), conflict managers and commit protocols.

1.4.13.2. Hybrid TM

Hybrid Transactional Memory systems allow for hardware-accelerated transactions or software-only transactions to run concurrently in the system. To do this, hardware transactions must check the software metadata in order to detect conflicts. The two main hybrid systems that have been developed are described next.

1.4.13.2.1. Intel's Hybrid TM

The Hybrid model developed by Kumar *et al.* in [85] accelerates the execution of the object-based DSTM system, discussed earlier in section 1.4.12.2. It makes use of a bounded HTM that

cannot run long transactions, given the size limitations of the buffer that keeps transactional data. The ISA contains the ordinary load/store instructions, plus the transactional versions of these. The transactional objects in DSTM are extended so hardware transactions can detect conflicts with software ones.

The system can run each transaction in the accelerated HW mode, or the original SW one. Typically, transactions are started in HW, and those that exceed the resources fall back to the slow SW mechanism. The system allows for SW and HW transactions to run concurrently, each one with a different update policy. HW transactions update the objects in place, while SW transactions use the original deferred update mechanism. New fields are added to the object metadata for HW transactions to detect conflicts with SW ones, and vice versa. When a conflict is detected, one of the involved transactions is aborted. In this case, ordinary load/store instructions are used even in the HW mode, to prevent that the atomicity does not allow the aborted SW transaction read its own status word.

1.4.13.2.2. Sun's Hybrid TM and Rock

The hybrid TM system proposed in [33] implements a word-level hybrid system that can use any generic HTM as the hardware support. The transactional runtime begins a hardware transaction and relies on the strong atomicity of the HTM to provide correct execution and prevent software transactions from reading invalid metadata. Each memory word in the system is associated to an ownership record (orec) in a single table, addressed with a hash function, as depicted in Figure 1-15. Software transactions must acquire these orecs in read or write mode. Hardware transactions only need to validate that there is no conflict with ongoing software transactions, by checking the corresponding orec field.

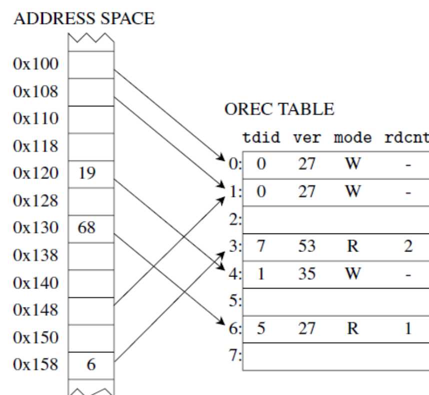


Figure 1-15: Orec table in the HyTM system by Damron *et al.*, taken from [33]

The check of conflicts against software transactions introduces an overhead in Hardware transactions. If all of the transactions are running in Hardware, such check is unnecessary. In [94] the authors proposed an improved system, PhTM, which considers different phases of execution: HARDWARE, HYBRID, SOFTWARE, SEQUENTIAL and SEQUENTIAL-NOABORT. The HARDWARE mode allows only for HW transactions and removes the need of checking the orec

metadata; if most transactions are short and can be run in the HTM, it allows for the maximum performance. The SEQUENTIAL and SEQUENTIAL-NOABORT modes allow for a single transaction at the time, what allows for irrevocable operations inside the transaction.

This HyTM model has been proposed for the Rock architecture [22]. The Rock chip contains 16 cores that implement a checkpoint-based architecture with Hardware scouting, described in [21]. In this architecture, the checkpointing mechanism and the write buffer used for the hardware scout is also employed for the HTM. The limited size of the write buffer makes the HTM bounded. The performance of a hybrid TM model with this CMP is studied in [35].

1.5. Processor microarchitecture and performance aspects

The historical improvements in processor performance have been largely due to two different reasons. On the one hand, the technological improvements have led to faster circuits and a larger scale of integration. On the other hand, the microarchitectural changes (allowed by these technological improvements) have improved the internal behavior of the processor, exploiting growing rates of parallelism. A good historical survey can be found in [68].

1.5.1. Evolution of ILP-driven microarchitectural designs

Early processors would sequentially step through all the phases of each instruction execution, typically some of fetch, decode, read of registers, ALU operation, and register or memory writeback. A pipelined architecture improves the performance by executing multiple consecutive instructions, each of them in a consecutive phase, increasing the maximum utilization of each of the processor's units. A pipelined (or 'scalar') processor can access the register file to read operands of an instruction, while the next instruction is being decoded, and the previous instruction is accessing the ALU. This mechanism achieves a performance increase thanks to exploiting the so-called Instruction-Level-Parallelism (ILP).

However, ILP imposes several challenges in the processor design. The main challenges are data dependencies and branch prediction. Data dependencies occur when any of the input (operands) registers of an instruction are the output (result) of a recent previous instruction, which is yet in execution. This causes a 'stall' in the processor pipeline, since the execution cannot be performed until the operands are ready. Branches suffer a similar problem. First, the jump address is not known until it is calculated, what is typically solved with a Branch-Target-Buffer (BTB). Additionally, the processor needs to use a predictor of whether to jump or not on a conditional branch, since the fetch stage of the instruction following the branch is executed before the condition is evaluated.

Superscalar processors further exploit ILP by allowing multiple instructions to be executed in parallel in each phase of the execution. This implies that the processor has multiple execution units (fetch, ALUs, etc) in order to allow for parallel processing. A good survey of such

architectures can be found in [139]. However, such a design increases the impact of data dependencies between near instructions.

To target the data dependencies problem, an out-of-order (OOO) execution model allows executing instructions when their operands are ready, rather than following the fetch order. Figure 1-16 shows a simplified block diagram of an OOO superscalar processor. Since multiple versions (old and new) of a given register might need to be accessed concurrently, OOO architectures require of a virtual-to-physical register renaming mechanism, such as Tomasulo's algorithm [152]. This implies that the actual number of physical registers will largely exceed the number of virtual registers, visible to the programmer. A "Re-Order Buffer" (ROB) tracks the instructions being executed and their relative order, and is used to forward data dependencies between in-flight instructions. The number of instructions that can be concurrently in-flight, recorded in the ROB, is known as the "instruction window" size. While instructions are executed out of order, the ROB follows a strict FIFO discipline. The processor does not retire (commit) an instruction until all the previous ones have completed. This in-order commit of instructions preserves the notion of sequential execution to the programmer.

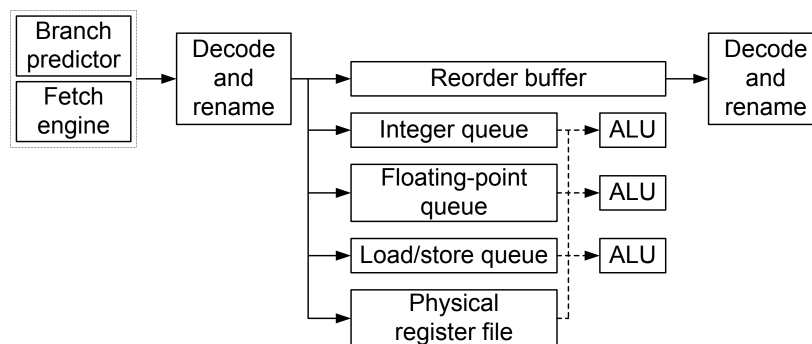


Figure 1-16: Block diagram of an out-of-order superscalar processor

In an out-of-order design some instructions can get 'stuck' in the pipeline while other non-dependent instructions proceed. These instructions are typically loads that suffer long-latency misses (such as L2-cache misses). The OOO model tolerates such case, as long as the ROB does not get filled. In such case, the instruction fetch must halt until the long-latency miss is served, with an associated performance drop.

The performance drop that is caused by a large latency of main memory accesses is known as the "memory wall", since it severely limits the achievable performance of a single processor and becomes a "wall" in the increase of processor performance. Multiple techniques have been proposed to deal with this performance limitation, among them, instruction prefetching techniques, runahead execution [112], scout threads and kilo-instruction processors [30].

Other current limitations of uniprocessor microarchitecture design are the "power wall" and fault-tolerance. The former limits the maximum chip frequency (since the dynamic power in CMOS technologies grows quadratically with the frequency) and the number of active

transistors in a chip, due to leakage currents that increase power. The latter is caused by the large current scale of integration, which is prone to soft and hard errors. While really important, these aspects are not part of the focus of this thesis.

1.5.2. Thread-level parallelism

The beginning of this Chapter discussed that current technologies limit the performance of the single-processor approach. The current scale of integration allows for 32 nm transistor half-pitch size, what allows for more than a billion (10^9) transistors in a single chip. The ITRS predictions [25] believe that Moore's Law will continue at its current pace, doubling the transistors budget every two years, at least until 2013, and continue from that point with a slower pace.

With such a large transistor count, a naïve resizing of previously existent processor designs leads to diminishing performance increases. Simply redimensioning the structures in the processor is each time more costly and less efficient. For example, increasing the maximum fetch from 4 to 8 instructions per cycle requires at least doubling the resources of the processor, but has a small impact in the achieved performance. Increasing the instruction window size requires augmenting the ROB and the load/store queues, which are Content-Addressable Memories (CAM) that typically grow quadratically in power consumption and area. Increasing the size of the processor caches leads to higher access times, what can negatively hit performance.

With such a scenario, cost-efficient designs rely on exploiting Thread-Level Parallelism (TLP): Running multiple concurrent threads in the same chip. Two general TLP techniques have been applied: Simultaneous Multi-Threading (SMT) and Chip-Multi-Processors (CMP).

Simultaneous Multi-Threading (SMT) [153] shares the processor resources between multiple hardware thread contexts. Each context contains its own state (private register file, for example), but the execution units (fetch, branch predictors, ALUs, Load/Store queues, etc.) are shared among all the threads. The processor fetches instructions from the instruction flow of all of the running threads, and executes them concurrently, depending on the availability of execution units. When one of the threads stalls due to cache misses, other threads can proceed with their execution. This allows for higher utilization of the processor resources, which would be otherwise stopped when execution stalls due to long-latency misses.

On the other hand, Chip Multi-Processors (CMPs) integrate multiple processors or "cores" in the same chip. Each of these cores has its own execution resources and local caches. Typically, a second or third-level cache can be shared between all the processors in the same chip.

Finally, it is also common to find designs that combine both approaches, such as the Niagara [81] processor with 8 cores and 4 hardware threads on each core, 32 threads overall, or its successor Niagara 2, with 8 cores and 8 hardware threads on each core, 64 threads overall.

While these techniques are more cost-efficient than the single-processor approach, they incur two problems. First, they impose the burden of programming multi-threaded parallel applications in order to exploit the chip performance. The problems and complexities of parallel programming have been deeply discussed in previous sections. It is widely agreed that simplified mechanisms to program these architectures are desirable for widespread adoption. Secondly, the performance increase that can be expected from a multithreaded application is limited by the amount of serial code present in the application, as studied by Ahmdal's Law and presented in section 1.2.2.1. Parallel applications with highly-independent threads, such as web or database servers that use different threads to attend different requests, will exploit nicely this so-called throughput-computing; other applications with higher interaction between the different threads will have a limited scalability. For this reason, implementing high performance processor architectures, even in highly-parallel systems, is still a concern. The next section details the implementation of kilo-instruction processors, which target the memory wall with a large instruction window.

1.5.3. Kilo-instruction processor overview

Kilo-instruction processors (KiP) [30, 31] target the memory wall by implementing a large instruction window, of one thousand or more in-flight instructions. As discussed in section 1.5.2, simply scaling the resources of an ordinary out-of-order processor is not affordable due to nonlinear increases in area and power. The architecture of Kilo-instruction processors introduces efficient mechanisms to reduce the resource utilization in order to handle large instruction windows efficiently. The main mechanisms are detailed next.

1.5.3.1. Checkpointing mechanism and early release

The CAM implementation of the ROB becomes a scalability limit. In the KiP architecture, the ROB is removed and substituted by a checkpointing mechanism. The processor takes periodical checkpoints of the architectural state of the processor, typically on long-latency instructions such as L2 misses or hard-to-predict branches. These checkpoints typically consist of a copy of the live physical registers, this is, those that can be accessed by future instructions. On an exception, interrupt and branch or data missprediction, the processor restores the status to the checkpoint before the faulty instruction, discarding all of the instructions fetched after such checkpoint was taken.

With this checkpointed architecture, the processor does not require a ROB to reorder the instructions in the execution unit. Each instruction has an associated checkpoint. The checkpoints implement counters for the number of pending instructions. Instructions commit out-of-order [29] and update the register file or save stores in the Load/Store Queue (LSQ). The instruction commit is not globally visible until checkpoint commit: the updated physical registers would be deallocated if the checkpoint was restored, and the LSQ retains the updates until the checkpoint commits. When a checkpoint counter decreases to 0, all of the associated

instructions have successfully committed and the checkpoint itself can commit, making all the associated memory updates in the LSQ visible. Therefore, all of the instructions in a checkpoint commit out-of-order, but their architectural changes are made visible at the checkpoint commit. Of course, different checkpoints must commit in order, to maintain the image of sequential execution to the programmer.

Finally, a performance improvement is the use of a pseudo-ROB structure, similar to a ROB but only contains the most recently fetched instructions. This structure allows for fast recovery of instructions that fail very quickly. It allows to simplify both the recovery process in such early fails and the implementation of the issue queue, as discussed next.

1.5.3.2. Bi-level issue queue

The issue queue holds an instruction while the instruction is waiting for its input data operands and execution resources. The issue queue is a very expensive resource: It is on the critical path for instruction execution and is one of the major energy consumers in the processor core. When a load instruction suffers a long-latency data miss, that instruction and all of the instructions that depend on the loaded value will occupy the issue queue slots for a long time.

To prevent such occupation, KiP architectures implement a two-level issue queue. Long-latency instructions, detected when they exit the pseudo-ROB without input data, are removed from the issue queue and moved to a secondary FIFO queue, called Slow-Lane Instruction Queue (SLIQ). Also, any instruction that depends on a prior long-latency instruction is moved to the SLIQ. When the long-latency data arrives, the processor starts to issue those instructions that had been deferred to the SLIQ in FIFO order. Other similar approaches that exploit two queues for fast and slow instructions are the Waiting Instruction Buffer [90] and the Slice Data Buffer [144].

1.5.3.3. Ephemeral registers

As presented in section 1.5.1, a large instruction window requires a large physical register file, in order to support the required register renaming mechanism for all the in-flight instructions. The larger the instruction window, the larger the register file. The register file is a complex structure whose upsizing largely increases the power and area requirements.

Kilo-Instruction Processors rely on a mechanism named ephemeral registers to reduce the number of concurrently used physical registers. First, the register renaming phase does not allocate a physical register for a given instruction, but a virtual tag. Each virtual tag contains a counter, which is increased on each subsequent use of the logical register, and decremented on commit of an instruction that accesses it. After the corresponding logical register is redefined, the virtual register and its associated physical register can be released if the counter reaches zero. Altogether, these techniques allow for a late allocation of physical registers, after

the issue stage, and an early release, before the renaming instruction commits, effectively reducing the required register file size.

1.5.3.4. Load/store queue handling

The load/store queue (LSQ) is a critical element when addressing the instruction window size. Each load has to search the store queue for older stores matching the load address, and if found, the load should use the value from the store queue instead of the cache. As a consequence, each issued store must check whether a younger load with a matching address has already executed, potentially violating program semantics. Instruction commit must be performed in order, maintaining program semantics. Finally, the LSQ must be searched for conflict when coherence invalidations are received in the local cache. For these reasons, LDQs are typically implemented as Content-Addressed Memories (CAMs), with limited scalability.

Multiple proposals target the lack of scalability of the LSQ. Pericàs *et al.* propose an *epoch-based* LSQ [115] that exploits execution locality in a distributed implementation of Kilo-Instruction processors [114]. This queue is split in two sections, a High Locality Queue (HLQ) in the out-of-order core, and a Low Locality Queue (LLQ), which is divided into multiple sections in the slow in-order cores. Loads and stores are moved from the HLQ to a LLQ when they are known to depend on a long-latency operation. Most LSQ hits occur between recent instructions, so the small HLQ is implemented as a CAM, and suffices most of the requests. The large LLQ is implemented with multiple banks, and relies on a filtering (hashing) mechanism to reduce the required amount of searches on each access. Similarly, most requests from each section of the LLQ are served locally.

There are other related hierarchical implementations proposed. The store queue in CPR [7] introduced the idea of hierarchical implementation. It implements a small L1-CAM queue for recent stores and a large and slow L2 CAM store queue, which uses a direct-mapped structure to filter most requests. The proposal in [50] uses a hierarchical implementation that breaks the L2 queue into two pieces: A FIFO queue for ordering, and a cache for forwarding data.

1.6. Contributions of this thesis

The work in this thesis derives from the limitations of reader/writer locks and STM systems based on those locks, and the microarchitectural implications of such locking mechanisms. The main contributions of this thesis are described next.

1.6.1. Lock-based Hybrid TM

Chapter 2 introduces a mechanism to build a hybrid TM system that relies on a reader-writer lock-based STM, which can be accelerated using a generic HTM. The base system provides interesting properties, such as immunity to the privatization problem, but its performance is poor due to the overhead of reader locking. The necessary changes to implement a hybrid TM system are discussed. The hybrid model presented in this Chapter removes the overheads of

the STM, maintains the STM mode for long transactions or transactions with forbidden operations, and allows for concurrent STM/HTM transactions committing in parallel.

1.6.2. Fairness among software and hardware transactions

The HyTM introduced in Chapter 2 presents different fairness properties for different types of transactions. Specifically, the use of different contention management policies leads to uneven progress conditions. In the studied model, the STM transactions employ a per-lock FIFO queue to ensure that starvation never happens. However, the HTM model relies on the coherence mechanism, what can lead to writer starvation.

Chapter 3 introduces the Reservation Table (RT), a low cost distributed mechanism that prevents the fairness problems in the HyTM model. Evaluations show that this model improves the performance up to 2.7× in a highly congested skip-list benchmark, improving the base mechanism and previously proposed models.

1.6.3. HW acceleration of locking mechanisms

The locking mechanism necessary in parallel programming (including the STM and HyTM mechanisms discussed earlier) imposes ultimately an overhead on the execution time. Chapter 4 introduces the Lock Control Unit (LCU), a distributed mechanism loosely based on the Reservation Table. The LCU is a distributed mechanism that achieves a very low lock transfer time, supports reader/writer locking, and does not fail in many of the common corner cases that make other hardware schemes proposed for locking unfeasible. Evaluations with different benchmarks show that the LCU outperforms previous proposals, while still providing more flexibility.

1.6.4. Microarchitectural improvements for performance, locking and consistency

High-performance parallel architectures rely on some form of large-instruction window mechanism, such as the kilo-instruction processors presented in section 1.5.3. Chapter 5 discusses how parallel machines based on kilo-instruction processors can leverage the checkpointing mechanism to increase the performance and simplify the consistency and locking operations. Additionally, it is discussed how the presented LCU mechanism can be adapted to a checkpointed microarchitecture with implicit transactions, and an alternative implementation with speculative locking capabilities.

Chapter 2. Lock-based Hybrid Transactional Memory

Chapter 2

Lock-based Hybrid Transactional Memory

This Chapter will discuss the advantages of a reader-writer lock-based implementation of Software Transactional Memory, and propose a Hybrid TM model based on such approach.

2.1. Advantages of reader-writer blocking TM

Software Transactional Memory has been initially developed by people coming from the field of parallel programming and data structures. Therefore, it is not surprising that most of the initial STM proposals ([132, 69, 64]) focus on lock-free or obstruction-free mechanisms. Ennals [42] argues that a blocking implementation of TM (as discussed in section 1.4.8.1) can be effective, with a simpler design that pays back the cost of the lock-free mechanism.

Besides the benefits of simplicity in the implementation, reader-writer locking mechanisms introduce additional benefits, as discussed in [34]. Systems based on reader-writer locks are immune to the privatization and publication problems. Specifically, the races presented in section 1.4.6 cannot happen if the reading transaction locks the object in read mode. In the example in Figure 1-10, both transactions Tx1 and Tx2 would acquire a lock on **x_shared** with different access mode during their commit. Therefore, there can be no race during the commit operation, and both transactions are completely serialized.

There are more benefits on a system based on reader-writer locks. Irrevocable transactions [143] can be easily implemented with these locks and encounter-time locking. The read locks acquired by the irrevocable transaction guarantee that any conflict with another transaction is detected eagerly, what is required to guarantee forward progress. Additionally, such system provides stronger progress properties, especially for long transactions, which are likely to abort in a system with lazy validation mechanisms based on data versioning.

The main problem associated with reader-writer locking is the low performance compared with other mechanisms, as discussed in [124].

2.2. Base STM overview

The base STM is a lock-based version of Fraser's Object-based STM, introduced earlier in section 1.4.12.3. The system is programmed as a C library that has to be compiled along with

the user code. The API is shown in Figure 2-1. The source code of the base STM is available online [26]. Running transactions are represented by **tx_id** transaction records.

```
Begin transaction:          stm_tx *new_stm_tx(tx_id *tx, stm *mem, sigjmp_buf *penv);
Commit Transaction:         bool commit_stm_tx(tx_id *tx);
Validate transaction:       bool validate_stm_tx(tx_id *tx);
Abort transaction:          void abort_stm_tx(tx_id *tx);

Read STM block b:           void *read_stm_blk(tx_id *tx, stm_blk *b);
Write STM block b:          void *write_stm_blk(tx_id *tx, stm_blk *b);

Allocate STM block:         stm_blk *new_stm_blk(tx_id *tx, stm *mem);
De-allocate STM block b:    void free_stm_blk(tx_id *tx, stm *mem, stm_blk *b);
```

Figure 2-1: STM programmer interface

This STM requires that all accessed objects within a transaction are declared as transactional, with the type **stm_blk**. Such work would be typically part of the compiler (as proposed in other systems, such as the Tanger C compiler [45] or the prototype C++ compilers from Intel and Sun), but the used system still requires the programmer to do it manually. Each object contains a header with a lock and a pointer to the current **data** of the object, as presented in Figure 2-2. MCS reader-writer locks (presented in section 1.3.1.3) are used to provide concurrent reader or exclusive writer access and fairness guarantees. Lock fairness is useful to prevent starvation, since otherwise readers or writers could stop each other as will be studied later on.

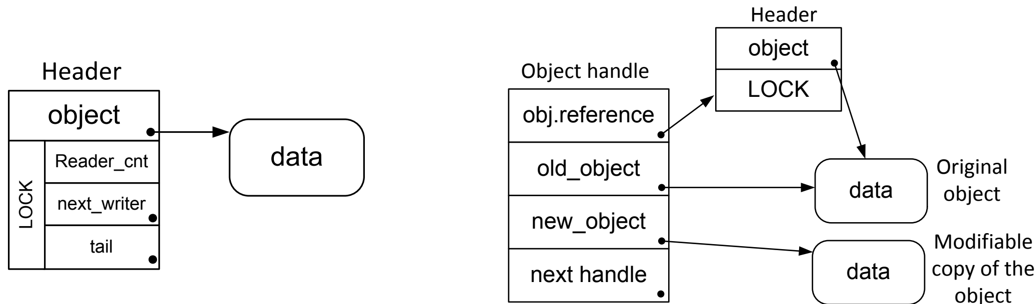


Figure 2-2: STM Object structure (left) and read/write set list element (right)

The STM makes use of a cloning mechanism. On each write access, a new per-thread private **data** block is allocated with the original content copied. All updates are made on the private data version of the object, leaving the original data block in place, accessible for other transactions. The read and write sets are maintained as a single unified ordered list of accessed objects. The elements of this list are as depicted in Figure 2-2 right, similar to the original OSTM structure. Different elements are linked with the **next** pointer. Both **old** and **new** pointers are the same when the object is in read mode, and they differ when the object is modified. The list is ordered according to the object header virtual address.

Two alternative implementations for lock acquisition exist. A strictly pessimistic model acquires all object locks in their corresponding mode the first time that an object is accessed.

Conflicts are detected in the lock acquisition, and a timeout mechanism (using a **trylock**) determines when to abort. This implementation introduces serialization dependencies between transactions that can limit the performance: If a transaction accesses an object in write mode, followed by a read access by another transaction, the reading transaction must wait for the commit of the writing one before proceeding. However, if the reading transaction was going to finish before, it could be better to use the current **data** content and proceed with the execution. If the reading transaction serializes before the writing transaction, the execution is correct without involving any wait. To prevent those dependencies between readers and writers, the second locking mode delays the lock acquisition to the commit phase. This is the alternative used by the base STM. Even more, locks are taken in order thanks to the ordered list in the read/write set, what prevents any deadlock possibility.

Conflict detection remains as eager as possible, considering the acquisition of locks at commit time. If an object is accessed, the current **data** pointer is recorded as part of the read/write set. If the same object is accessed again, the read/write set list is searched and the original object is retrieved, along with a check of the **data** pointer. If the object pointer differs from the recorded pointer, the object has been updated in the meanwhile, and the transaction aborts. Finding a lock taken does not imply a conflict nor aborts the transaction in case of a timeout, since the order of acquisition prevents any deadlock problem.

At commit time, a two-phase locking protocol is used. In the first *grow* phase, all the locks in the read/write set list are acquired, along with a validation of the data pointers of all accessed objects. This guarantees the consistency of the transaction. Then, all the modified objects are updated by upgrading the **object** pointer to the new data block. Finally, in the shrink phase, all locks are released.

Each thread contains a transaction descriptor. This descriptor contains the status of the transaction, a pointer to the read/write set list and a pointer to jump back to the start of the transaction using a call to **siglongjmp** in case of an abort. The pointer to be used in such call is recorded at the start of the transaction. The descriptor also contains data regarding memory allocation and garbage collection, along with a pool of pre-allocated **data** chunks. This system might fail in presence of the ABA problem: A memory location is changed from its original value A to B, and then back to A; an external observer, based solely on the object value, might erroneously believe that the object has never been changed if it only observes the extreme values, A. In our case, if a **data** block is modified and removed while other transactions still point to it in their read/write set lists, the reuse of the same memory for the same object (on a subsequent modification) would lead to incorrect execution. In such case, the validation phase of the other transactions would suffer a false positive since both pointers in the object header and the read/write list are equal, and the conflict would go undetected. To prevent this ABA problem, the STM uses a generational garbage collector (GC) which does not reuse a block until it is safe to do so.

The base STM system allows for 'zombie' transactions, this is, transactions that have read inconsistent values and are doomed to abort, but that have not yet detected so. In managed languages like C# or Java all such failures could be detected by the runtime system. The problem comes from the early access to the object data, but not to the lock status, what can lead to accessing inconsistent objects, breaking the isolation of other transactions. While these transactions fail the validation at commit time, the inconsistency of the read values could lead to accesses to deallocated memory locations (what is allowed by the used GC implementation) or start a cyclic execution that follows loops in the data structures, which never reaches commit. It is the responsibility of the programmer to detect such cases and abort the transaction, for example, with periodic calls to **validate_stm_tx**.

In any case, the STM runtime includes two characteristics to mitigate the problem of zombie transactions. Firstly, a signal handler to detect the cases in which an invalid pointer is followed to a deallocated memory area. In such case, a SIGSEGV exception is triggered, and the associated handler performs a validation of the ongoing transaction (which must fail, since it has read intermediate data of another transaction). Secondly, on each read or write access, the object is recorded in the read/write set list. Subsequent accesses to the same object find it in the list and check that the recorded **old** pointer is still valid. If the inconsistent state comes from a partial commit from another transaction, eventually such transaction would finish commit and one of the objects will be updated, what makes the inconsistency visible.

An example is presented in Figure 2-3 with a binary tree structure. Each transactional object comprises one of the tree nodes, along with the two pointers to the next objects. In the example, a transaction moves the node C to the head of the tree, what involves adding a pointer from C to A, and removing the pointer from B to C. The commit phase must write-lock both B and C. The intermediate step b) contains a cycle, what is inconsistent since trees do not contain cycles by definition. Another transaction that makes a search on the same tree can observe the state in b), and never finish. However, the modifying transaction will eventually update the object B, breaking the cycle (state in c), or abort, restoring the initial state in a). This update is detected in the next access to B in the cycle, and the zombie transaction aborts. With these two mechanisms, the benchmarks presented in section 2.4.2 do not require any validation from the user.

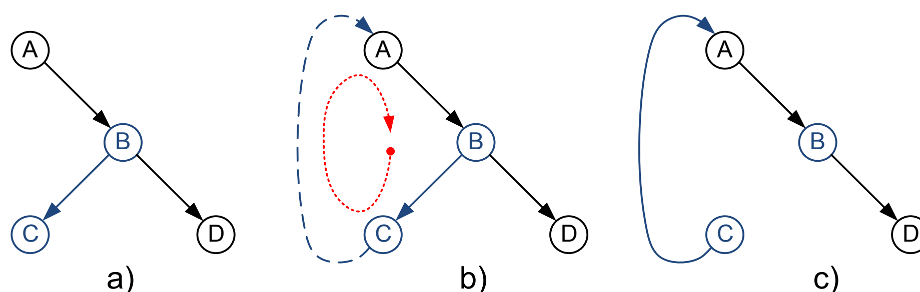


Figure 2-3: Three steps in a commit of a binary tree data structure. The intermediate step b) contains a cycle, what can lead to zombie transactions that never commit

This version of the STM clearly differs with the other ones presented in section 1.4.12.3. With respect to the lock-free OSTM, this version does not fail in the privatization problem discussed in section 1.4.6. With respect to the WSTM, this model has a different granularity and does not introduce aliasing in the access to the metadata. Such aliasing has been studied to increase the number of false conflicts due to the “birthday paradox” in [161].

2.3. Acceleration opportunities with a generic HTM

The performance of the base STM with pessimistic concurrency control is poor, as will be detailed in section 2.4. The present section discusses the performance penalty when compared with a pure hardware TM system. Then, the different acceleration opportunities that can be achieved using a generic, bounded HTM are studied.

A lock-based STM adds four main overheads when compared with running the same transactions on a native HTM (as discussed in [17]). First, the locking mechanism itself is not necessary in a HTM system. Second, transactions need to maintain the read-set and write-set lists. This introduces a list-search for each object accessed, and an increase in the used memory. In HTM systems the hardware itself tracks the objects accessed in the transaction (with read and write bits, signatures or other mechanisms). Third, on commit, the lists have to be traversed to lock and validate the objects. Fourth, the indirection-based object structure makes it necessary to copy entire objects when opening them for update even if only a single field is going to be touched. In HTM these copies are implicitly managed and at a finer granularity.

Next, it is proposed the construction of a hybrid TM system that removes these costs by combining the base STM with a generic HTM system. A progressive approach is used: initially, the STM **new_stm_tx** and **commit_stm_tx** functions are modified to start a ‘sympathetic’ HTM transaction when each transaction is started. Transactions are initially attempted in this ‘hardware mode’, being called hardware-transactions or HW-Tx. The HW-Tx will follow the same execution path as the original code: It invokes the same STM-library operations as normal, preventing double compilation of the transaction’s implementation. If the transaction aborts a given number of times, it will be retried in the original, slow SW-only mode without the wrapped HTM transaction. Both HW and SW transactions will be allowed to run concurrently in the system.

Many runtime operations are unneeded when running in HW-accelerated mode. Instead of providing two different execution paths for HW-Txs and SW-Txs (as proposed in [33, 85]) we will progressively analyze the different steps that can be dynamically removed from the HW-accelerated execution path. In any case, if HW resources are exceeded then the HW transaction is aborted and fallen back to SW execution. This progressive approach allows us to observe the implications of every architectural change and its effect on the global system

performance. In addition, we will compare the system with a faster ordinary STM which fails in the privatization problem.

The design uses an ordinary HTM supporting strong atomicity between transacted and non-transacted accesses. It is assumed that all memory accesses are implicitly transacted when running inside a transaction, this is, no 'escape code' is required or ever used. It is also assumed that the ISA provides a new instruction (named **InHWTx** in this work) to determine whether or not execution is inside a HW transaction. These requirements are already satisfied by most HTM proposals. The HTM does not need to be unbounded, this is, it is not required to support any corner execution case. Instead, the HTM can fail in reasonable cases, such as long transactions that exceed the capacity of a write buffer, or when transactions call certain OS services such as memory allocation or I/O. In such case, the transaction will be aborted and restarted, until the abort threshold is reached and it is moved to the software-only mode. Ideally, in these cases the STM should be used directly; however, this implies that the HTM notifies the software about the abort reason (a transaction conflict or an unsupported operation). While such capability has been proposed for some systems [21], it is absent in most HTM proposals [18, 59, 110] so it is not required in the design.

When making performance-related decisions, the design assumes that per-cache line conflict detection is used and that updates are eager. These latest assumptions affect performance, not correctness. The evaluations will be performed using the LogTM HTM, whose characteristics have been discussed in section 1.4.11.4. Next, sections 2.3.1 to 2.3.3 progressively introduce the possible improvements of the base STM when using a HTM acceleration mechanism.

2.3.1. Avoid locking

As the underlying HTM mechanisms provide transaction atomicity and data collision detection, locking is un-needed when running transactions in HW. However, since the system allows for concurrent HW and SW transactions, locking cannot be removed. SW transactions require locks for correctness, and SW-locked objects must be respected by HW-Txs. Also, SW-Txs should not acquire a lock if this conflicts with a HW-Tx accessing the object. Therefore, it is clear that SW transactions must still use the lock to protect the data, and that HW transactions must check the lock status to detect conflicts with SW transactions.

As discussed earlier, the design approach is to leave the same STM interface and modify the internal implementation to make use of the HTM capabilities. The locks are maintained in the object header, but modified **lock** and **unlock** operations are used, along with a modified lock structure. The problem with the original MCS lock is that the lock header fields depicted in Figure 2-2 allow a HW transaction to determine if there are active readers (depending on the **reader_count** value) but it is not known whether the lock is acquired in write mode or not. The absence of readers with a non-null **next_writer** pointer does not guarantee that the lock is

taken in write mode, since it can have been already released, but the pointer not having been updated yet. Therefore, the HyTM will use a modified version of the lock that includes a **writing** field as depicted in Figure 2-4. This flag is set when the lock is acquired in write mode in software transactions, in their commit phase.

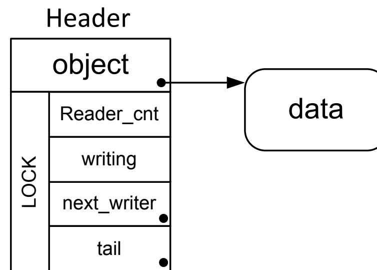


Figure 2-4: Modified STM lock structure for HT Txs to detect conflicts with writing SW Txs

With this configuration, HW transactions must just check for collisions with SW ones: a HW **read_lock** operation must conflict with an active SW writer, and a HW **write_lock** operation must conflict with any active SW reader or writer. Similarly, SW transactions must conflict with existent HW writers when trying to acquire a read lock, and with existent HW readers or writers when acquiring a write lock. To this extent, we modify the lock acquisition functions as depicted in Figure 2-5. The code will first determine if a HW transaction is running (lines 3 and 17). In such case, it will check the corresponding fields of the lock (**writing** in any case, and **reader_cnt** if it is a write lock). If a conflict is detected, the HW transaction is aborted and restarted (lines 8, 21). Waiting for the lock to be released is meaningless, since the update of the **writing** or **reader_cnt** fields in the lock release would constitute a violation of the isolation of the transaction, and cause a HW abort. If no conflict is detected, the execution returns without acquiring the lock: as we will detail later, the strong atomicity of the HTM will prevent any further problem. Since HW Txs do not update the lock, the unlock operations (lines 29, 38) do not need to perform any action.

The strong atomicity of the HW transaction will ensure that the checked value is maintained during the transaction, preventing any update from SW transactions or aborting the HW transaction. The exact behaviour depends on the details of the HTM. In a direct-update, early-detection HTM system (such as LogTM [110], used in our evaluations), the coherence extensions providing strong atomicity will prevent any SW-Tx from acquiring a lock in a conflicting mode once it has been checked by the HW-Tx. In this case, the STM transaction would have to wait for the HTM to commit and release the protected locations, specifically, the conflicting field of the lock header. A HTM with lazy detection (like TCC [59] or Bulk [18]) would favour SW-Txs, aborting the HW-Tx once the lock fields are remotely updated by the SW-Tx. In any case, conflicting transactions are never allowed to proceed in parallel.

```
 1: void rd_lock(mrsw_lock_t *lock, mrsw_qnode_t *qn) {
 2:
 3:    if (InHWTX) {
 4:       if (!(lock->writing)) {
 5:          // Nothing else to check, just return
 6:       }
 7:       else {
 8:          ABORT_HW_TRANSACTION();
 9:       }
10:    }else {
11:       [...] //The ordinary "SW-only" code from [107]
12:    }
13: }
14:
15: void wr_lock(mrsw_lock_t *lock, mrsw_qnode_t *qn) {
16:
17:    if (InHWTX) {
18:       if (!(lock->writing) && (lock->reader_count == 0)) {
19:          // Nothing else to check, just return
20:       } else {
21:          ABORT_HW_TRANSACTION();
22:       }
23:    }else {
24:       [...] //The ordinary "SW-only" code from [107]
25:       lock->writing = 1;
26:    }
27: }
28:
29: void rd_unlock(mrsw_lock_t *lock, mrsw_qnode_t *qn) {
30:
31:    if (InHWTX) {
32:       // Nothing to check, just return
33:    }else {
34:       [...] //The ordinary "SW-only" code from [107]
35:    }
36: }
37:
38: void wr_unlock(mrsw_lock_t *lock, mrsw_qnode_t *qn) {
39:
40:    if (InHWTX) {
41:       // Nothing to check, just return
42:    }else {
43:       lock->writing = 0;
44:       [...] //The ordinary "SW-only" code from [107]
45:    }
46: }
```

Figure 2-5: Lock implementation in the HyTM model

Table 2-1 summarizes the actions involved when the lock is held by a thread in reader or writer mode and a new thread arrives generating a collision. The references to HTM “aborts” could be delays, depending on the HTM implementation. When two transactions try to lock the same object in read mode, they can proceed in parallel: SW-Txs just increase the **reader_count** field, while HW-Txs check the **writing** field which is null.

Since HTM typically detect conflicts with a cache line granularity, it is important that the different fields accessed stay on different cache lines. This implies that cache-padding must be used to ensure that **writing**, **reader_count**, and the rest of the header are allocated in different lines. While this involves a significant overhead, it is also a requirement of other HyTM models such as [33].

	Current lock holder:	HW writer	SW writer
	Lock status:	reader_count=0 writing=false	reader_count=0 writing=false
HW writer	Check:	writing== false & reader_count ==0	
	Action:	HTM aborts on real data collisions	Explicit abort after check of writing
SW Writer	Action:	HTM coherence extensions prevent SW-Tx from modifying writing	Use of the ordinary lock queue system

a) Writer-writer conflict

	Current lock holder:	HW reader	SW reader
	Lock status:	reader_count=0 writing=false	reader_count=1 (+) writing=false
HW writer	Check:	writing== false & reader_count ==0	
	Action:	HTM aborts on real data collisions	Explicit abort after check of reader_count
SW Writer	Action:	HTM coherence extensions prevent SW-Tx from modifying writing	Use of the ordinary lock queue system

b) Reader-writer conflict

	Current lock holder:	HW writer	SW writer
	Lock status:	reader_count=0 writing=false	reader_count=0 writing=true
HW reader	Check:	writing== false	
	Action:	HTM aborts on real data collisions	Explicit abort after check of writing
SW reader	Action:	HTM coherence extensions prevent SW-Tx from modifying reader_count	Use of the ordinary lock queue system

c) Writer-reader conflict

Table 2-1: Summary of operations when conflicts occur in the HyTM

The use of HTM transactions introduces some complexity in the management of zombie transactions. Consider again the example presented in Figure 2-3 in page 56, with a reading zombie transaction caused by a writing transaction that temporarily generates a cycle in a shared tree structure. In the original STM, the committing transaction eventually finishes and the cycle is broken. However, if the reading transaction is a HW-Tx and the HTM is eager (such as LogTM, the model in the evaluations), the committer can be delayed, since the update to node B conflicts with the ongoing transaction. In this case, a deadlock occurs: The SW-Tx cannot complete commit because it is prevented by the strong isolation of the HTM. Conversely, the HW-Tx never commits because it is in a cycle caused by the SW-Tx. To prevent these situations, the runtime in this mode automatically validates the HW-Txs. This occurs when the read/write set is searched with a hit each certain number of times (we used 25 hits in our evaluations).

2.3.2. Read set removal

The second acceleration technique comes from the fact that the explicit read-set list can be elided in HW transactions. Such transactions still need to check the locks of read objects for two reasons: to avoid reading write-locked objects and to prevent any further SW-Tx write-locking the object. Therefore, instead of building up a read-set list, in HW-TXs the **read_stm_blk** call will check that the lock is in the appropriate status (**writing = false**) and return the **data**

pointer to the shared object. This prevents any further software writer committing changes to the block before the HW commit, thanks to the strong atomicity.

This operation decreases overhead for two reasons:

- The transaction log is reduced to the write set. Though a search on each access is still needed, it is much faster.

- The same applies to the validation: the number of validation steps is reduced to the modified objects count only.

2.3.3. Write set removal and in-place update

Applying the idea of removing the private log to the write set would prevent the object copy on the first access and would provide updates in place. However, as detailed in section 2.2, SW transactions rely on the update of the **data** pointer to detect conflicts on the validation step. HW transactions do not need to allocate a new **data** block, but modifying the object without updating the **data** pointer would not allow SW transactions detect HW updates.

To overcome this drawback, we add an additional **version** counter in the object header, as depicted in Figure 2-6. This field is set to 0 when the object is first instantiated (in any HW or SW transaction) and increased on every call to **write_stm_blk** by a HW-Tx. SW-Txs record the value of this word in their entry on the read/write set. The validation process in software transactions implies now checking both the **data** pointer and the **version** field. Special care is needed to handle version overflows, what could lead to the ABA problem in software transactions. A simple solution is to abort HW-Txs on counter overflow and clear the counter on SW updates (to minimize overflows).

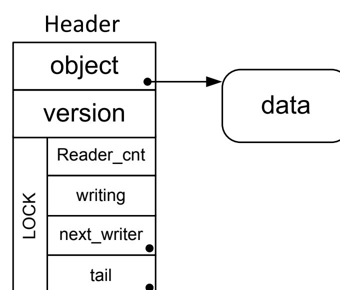


Figure 2-6: Modified STM Object with a version field allowing for in-place updates

With this optimization, HW transactions make their updates in place, do not maintain a software copy of their read or write sets, and consequently avoid any search on these sets. As our experimental results will show later, this last system provides the highest performance since it removes most of the overheads of the original STM.

2.4. Evaluation

This section details the infrastructure and benchmarks used to validate the correctness and performance of the used models. Then, it presents the results that show how the hybrid model removes most of the STM overheads, while still preserving its original code as a “fallback” option in case of frequent rollbacks.

2.4.1. Evaluation infrastructure

The proposed system has been implemented and simulated with GEMS [101]. GEMS is a simulator that implements a detailed model of the memory hierarchy and interconnection network, in a module called Ruby, and a detailed model of an out-of-order processor, in another module called Opal. Ruby implements a specific language to describe coherence operations, SLICC, what simplifies the development and validation of coherence protocols. Coherence operations are defined in SLICC, what is later translated to C++ by the corresponding parser, prior to the module compilation.

GEMS is based in the full-system simulator Simics [98]. Simics alone provides fast and accurate execution of a full system based on different architectures. This allows for the booting and execution of an unmodified OS, such as running Solaris on a simulated Sun Sparc system. Simics is completely accurate, this is, it implements all the necessary interrupts, I/O and disk interaction, etc, so that the simulated code is not aware of the simulation infrastructure and no modification is required. By contrast, Simics focuses on speed, not providing any detailed model of the memory hierarchy, processor microarchitecture and other internal system details. GEMS is a timing-first model: Its internal model simulates all the details of the processors microarchitecture, memory hierarchy and other details. Once an operation is ready to commit in GEMS (such as an instruction commit in the processor pipeline), Simics is instructed to proceed with the execution. Therefore, GEMS makes use of the accurate model of Simics, being allowed to elide some operations which are complex to simulate and irrelevant for our study, such as DMA operation, booting and switching off the machine, disk operation, etc.

GEMS includes a model of the LogTM transactional memory protocol, based on a MESI coherence protocol. The LogTM model does not make use of the Opal module. Therefore, a complex out-of-order processor is not simulated. Such model resembles a simple in-order core, what has been already used by many works on HTM and HyTM such as [61, 110, 24, 33]. Additionally, a simple “network multiplier” was used to consider the different relative speed between the processor cycle and the network cycle. With this network multiplier set to 4, the processor is capable to process 4 instructions per cycle when no L1 miss occurs, halting in such case until the miss is satisfied. Each processor has private 64KB L1 and 1 MB L2 caches, with 64-bytes per cache line. Cache latencies and network parameters have been set to resemble those of the Sunfire E25K [2]. The system contains 16 processors, except for cases in which a

larger model is evaluated with 32 processors. The number of threads varies, leaving at least one processor empty for OS tasks. A distributed directory is used, with one directory controller per processor.

The STM implementation was extended to use the LogTM ISA, modifying only the STM library and not the applications using it. The STM library was modified so that HW-Txs never call the system **malloc** and **free** functions present in the garbage collector. Instead, HW transactions are aborted if the local pool of pre-allocated chunks is exhausted. Three reasons support this:

- First, it prevents some limitations on the base simulator. Although the HTM simulates unbounded transactions, there are various low level operations (related with OS locking, as discussed in [155]) which cannot occur transactionally.
- Second, it prevents congestion problems in the garbage collector (GC), and eventually in the OS, when a HW transaction modifies some global structure. Specifically, the GC should run as escape code to prevent dependencies among different transactions, with the corresponding compensation actions. However, this was not implemented in the used version of the simulator and requires tight coupling between the STM and the HTM, what is against the idea of a HyTM based on a general HTM.
- Finally, it models a HTM system more restricted than the original, unbounded, LogTM model. Ongoing work, such as the Rock processor [22, 21], suggests that commercial HTM will be bounded and will not allow certain operations inside HW-Txs.

2.4.2. Simulated models

The proposed hybrid system was implemented in the base STM library. Conditional compilation directives allow for the evaluation of the different intermediate steps that have been discussed in section 2.3. Then, the evaluation considers the following versions:

- The original, software only, STM system (labelled **sw** in the plots), which is expected to be limited by the cost of read-locking.
- A version eliding the locks in HW, but maintaining both read and write sets (**rw**), as discussed in section 2.3.1.
- A version that avoids entries in the read set (**noread**), as discussed in section 2.3.2.
- A full accelerated version, without read and write sets (**nowrite**), as presented in section 2.3.3.

A real HyTM system would only use the last version, but we simulate all of them to evaluate the effect introduced by each change. The original OSTM presented in section 1.4.12.3 is also profiled (labelled **fraser**). This OSTM provides higher performance than the SW-only lock-based

STM, at the cost of failing in the privatization problem. Its comparison points out the performance costs of the privatization safety property based on pessimistic concurrency control.

2.4.3. Benchmarks

The proposed model has been evaluated with three microbenchmarks: red-black tree (**RB**), skip-list (**skip**) and hash-table (**hash**) data structures. The problem sizes are specified by a key *k*, similarly to other works in the area [36, 48, 47]. Each thread executes writing transactions with probability *p,* or read-only transactions that search and read a given key with probability (1-*p*). Next sections detail these benchmarks.

2.4.3.1. Red-black tree

A red-black tree [12] is a data structure organized as a binary tree, in which each node has a 'colour' attribute which can be either red or black. The tree is 'self-balancing', which means that after an insertion or deletion, an appropriate rotation can be required to ensure that the tree is balanced. The specific rules for the balancing operations, which depend on the relative colour of the different nodes, are out of the scope of this work, but can be found in [12]. The advantage of the balancing operations is that the worst-case execution time remains logarithmic, $\mathrm{O}(\log(N))$ with *N* being the cardinality of the tree.

The required rotations imply that updating operations (insertions or deletions) can modify a significant number of nodes in the tree, as depicted in Figure 2-7. In the left case, the tree contains 12 nodes. The right figure shows the situation after the addition of a new node with index 20. The process is as follows. First, a search is performed from the root node 5 to the appropriate position, being the right child of node 18. Then, a series of rotations (the group of nodes 8-9-13 modify their relative location) and colour changes (nodes 8, 13, 15, 16 and 18 change their colour) are applied to guarantee the balancing rules. Therefore, a single addition can modify more than half of the nodes in the shared structure due to the rebalancing tasks.

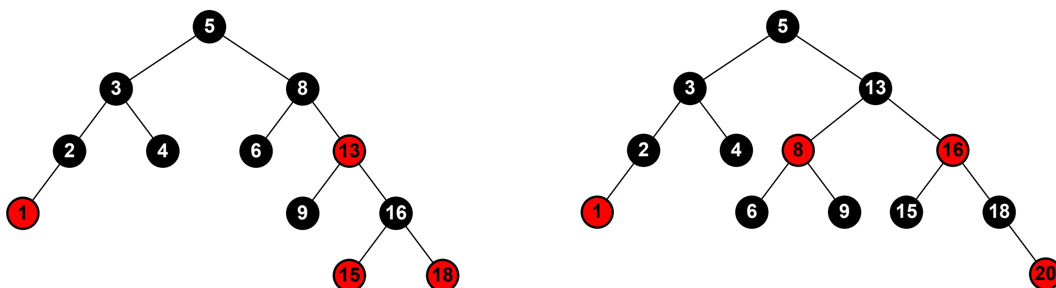


Figure 2-7: Left, Red-Black tree example. Right, status after the addition of node 20

The benchmark consists of multiple threads accessing concurrently the shared structure, which contains keys in the range $\left[1 \ldots 2^{k}\right]$. Each access by a given thread will perform a *search* for a given random key with probability $(1-p)$, what is a read-only operation. With probability

$p/2$, the access will *add* a given node, if it is not present, or with the same probability, $p/2$, the access will *remove* the given node if it is found, with the required balancing operations in both cases. Therefore, the number of threads and the parameters *k* and *p* will determine the size and congestion in the red-black tree benchmark.

Each node of the red-black tree (including its two pointers) is declared as a transactional object. Each access by a given thread (search, addition or removal) is contained inside a single transaction. In the benchmark, all threads iterate accessing transactionally the red-black tree. After a sufficient warm-up period, the number of simulated cycles per transaction is measured, executed for a period long enough to converge to a fixed value, and averaged across nine simulation runs. Performance is reported as the inverse of this value. In most cases, our results are normalized against the single-processor, software-only performance (**sw**).

2.4.3.2. Skip-list

A skip-list [116] is a data structure which provides $O(\log(N))$ average access time and implements a simpler algorithm than a red-black tree, at the cost of a higher, linear, worst-case access time. The data structure is organized as a hierarchy of parallel linked lists that connect increasingly sparse subsequences of the items. The lower level is an ordinary ordered linked list, and each higher layer acts as an "express lane" for the lists below. An example taken from [116] is presented in Figure 2-8. The top part shows a skip-list with 4 levels, and the required operations to search for a given node with key 17. The bottom part shows the required changes to add such node 17 to the skip list, involving a maximum of 4 pointers update. The allowed operations in the microbenchmark are the same as in the red-black tree. The methodology is the same as in that case, and the parameters *k*, *p* and the number of threads will determine again the congestion in the shared data structure.

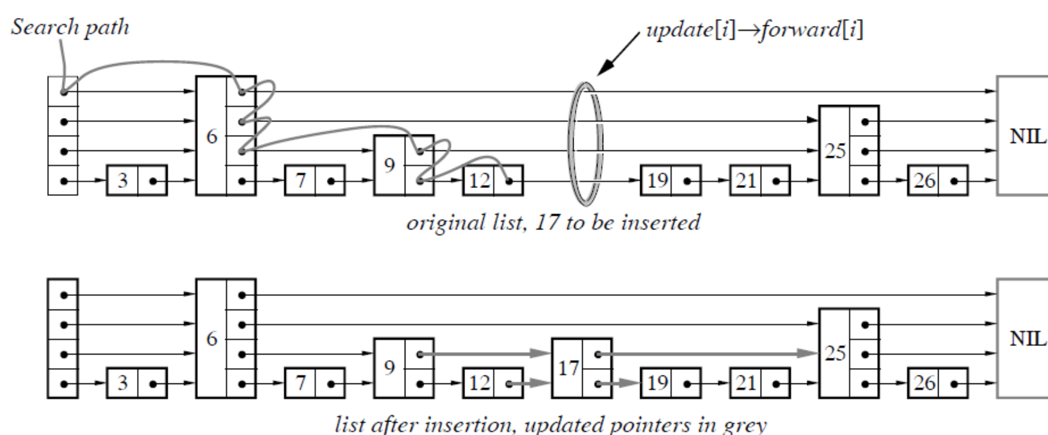


Figure 2-8: Skip-list example, taken from [116]

2.4.3.3. Hash table

This benchmark comprises two hash-table data structures using direct chaining when multiple elements map to the same bucket of the table. Each table has a different size (using 1024 and 512 in the evaluations), and each element (also identified with a key in the range $\left[1...2^k\right]$) can be present in at much one of the lists. Each thread will iterate, performing one of three random operations on each transaction:

- *Search,* with probability $(1-p)$: A given random key is searched. One of the tables is looked up first; if it is not found, the other one is looked up. The first table is randomly chosen.

- *Add/remove*, with probability $p/2$: The given key is searched in both tables. If present, it is removed; otherwise, it is added in one of the tables chosen randomly.

- *Move*, with probability $p/2$: The given key is searched in both tables. If it is found, it is removed from its location and added to the other table.

This benchmark is an example of the composability problem that transactional memory solves. Without TM, even with correct implementations of hash-tables, one cannot program this code without modifying the internals of the hash-table. A lock-based version is prone to deadlock, if one bucket of each list needs to be locked by two different transactions in the opposite order, for example in two *move* operations that map into the same buckets. A lock-free version will probably allow for remote operations to observe intermediate states with an object present in both tables (in the case of *move* operations or multiple *add/removes*) or an object being *moved* not present in either table.

This benchmark has been also developed as a contrast with the previous ones, to evaluate the effects of reader-locking congestion. In this benchmark, the data structures do not contain a single “point of entry”, as occurs with the previous benchmarks in the tree root or the first element of the list. This prevents any performance bottleneck associated with read-locking such element, as will be studied in the next section. Therefore, its performance and scalability will be much better than the previous benchmarks.

2.4.4. Performance results

This section presents the single-threaded evaluation first, to observe the raw effect of the elided sections on the code. The multithreaded case is evaluated in the following sections. All the evaluations have been performed in a simulated machine with 16 processors and the parameters specified in section 2.4.1.

2.4.4.1. Single-thread performance

The first test shows the performance improvement obtained in the single thread case, which reflects the sequential work removed by the HW-support. Table 2-2 shows these values for

different problem sizes (from 2^8 to 2^{15} maximum elements) normalized to the software-only (**sw**) case. The benchmarks are run with read-only transactions ($p = 0\%$), and with $p = 10\%$. In this evaluation, the locks are present as discussed for the base STM, despite not being required for single-threaded execution. They are preserved to be able to compare the base speedup in this and further evaluations.

	RB *p*=0			RB *p*=10%			Skip *p* = 0			Skip *p* = 10%			Hash *p* = 0		Hash *p* = 10%	
Key *k*	8	11	15	8	11	15	8	11	15	8	11	15	8	11	8	11
sw	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
rw	1.43	1.41	1.38	1.41	1.38	1.33	1.03	1.08	1.13	1.04	1.05	1.17	1.23	1.23	1.23	1.33
fraser	1.70	1.69	1.65	2.19	1.23	1.22	1.14	1.23	1.25	1.15	1.18	1.26	1.38	1.39	1.03	1.49
noread	2.50	2.78	2.92	2.22	2.27	2.35	1.93	2.24	2.37	1.94	2.12	2.58	1.71	1.71	1.52	1.75
nowrite	2.84	3.20	3.36	1.92	2.53	2.68	2.29	2.67	2.84	2.83	3.16	3.45	1.88	1.87	1.68	2.00

Table 2-2: Single thread normalized performance (inverse of the transaction run time)

Fraser's performance improves the base **sw** system in a 20-60%, since there is no reader locking overhead. The lock elision (**rw**) provides a similar but slightly lower improvement, given that the **writing** field of the lock still has to be checked by readers and the associated memory overheads of the lock padding. The remaining two improvements provide higher benefits (up to 3.45× in the best case) by offloading the log management and conflict detection functions to the HTM. This improvement grows with the problem size (2^{11}, 2^{15}) in **RB** and **Skip**, since the transaction log size is also increased.

2.4.4.2. Read-only transactions

Figure 2-9 shows the performance obtained with read-only transactions and multiple threads in the **RB**, **skip** and **hash** benchmarks with key size $k = 8$ and $p = 0$. The data structure is pre-initialized so it contains valid data for the read-only transactions. Plots are normalized to the performance of the single-threaded, base STM system (**sw**). Therefore, the performance of different models with 1 thread does not coincide; instead, they preserve the relative performance presented in Table 2-2.

The left figures use a linear scale in the vertical performance axis, and the right figures use a logarithmic scale. Apparently, with the linear scale, the three workloads seem to show a superlinear speedup on most modes (**fraser**, **rw**, **noread** and **nowrite**), since the speedup exceeds the thread count. This effect is caused by the normalization to the single-threaded **sw** case: the individual speedup of each mode with respect to the single-threaded performance in the same mode, presented in Table 2-2, does not present superlinear speedup.

A closer look at the logarithmic plots on the right shows that all these models show an approximately linear speedup: performance lines parallel to the reference linear speedup line. These linear speedups maintain the relative performance presented in Table 2-2 among the different modes. This is the expected performance for this benchmark, since all read-only transaction should proceed in parallel. However, the base STM (**sw**) suffers significant

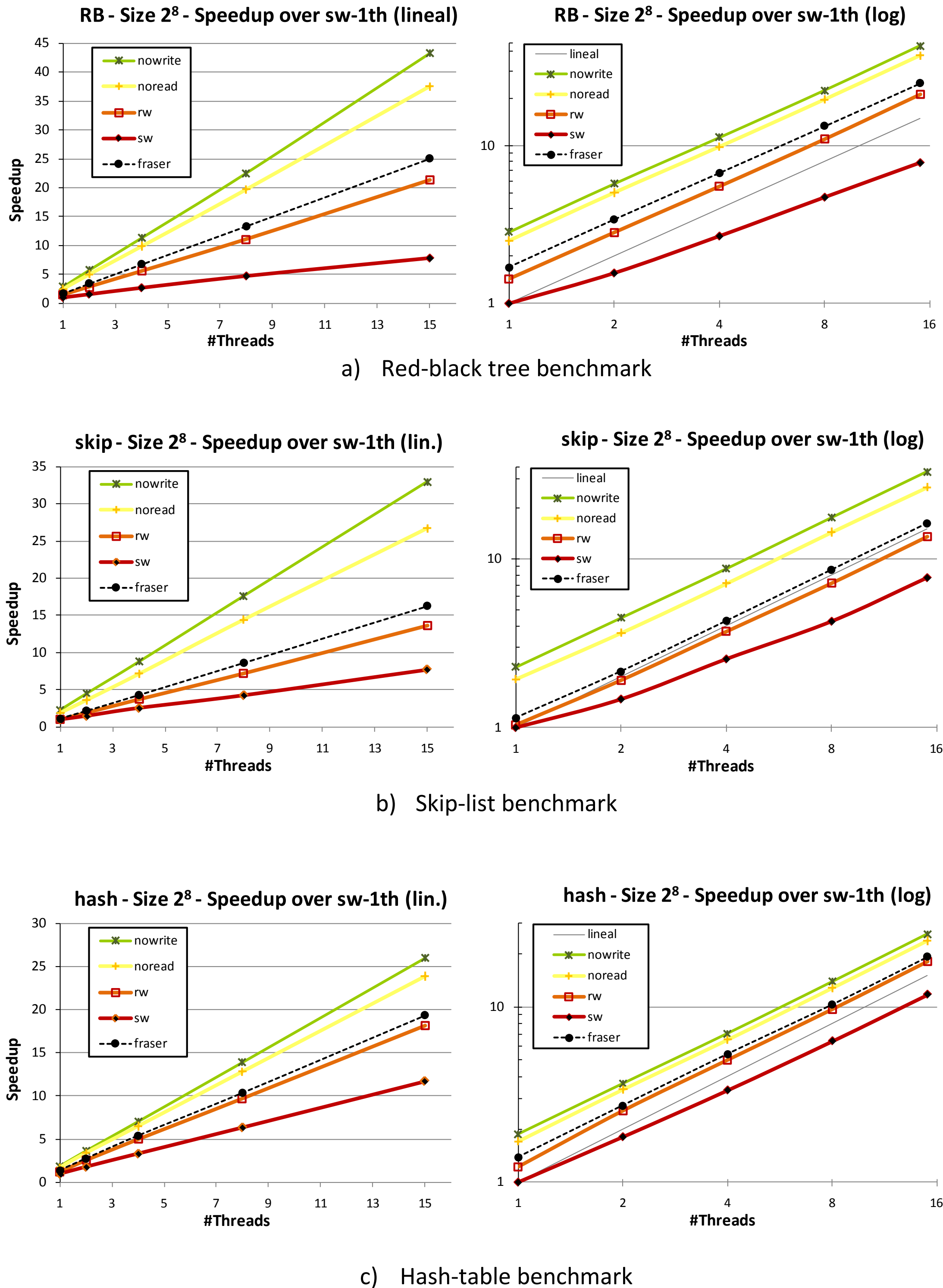


Figure 2-9: RB, skip and hash speedup with read-only transactions, k = 8, in linear (left) and logarithmic (right) scales

performance degradation in the **RB** and **skip** benchmarks as the number of threads increase. The degradation also occurs in the **hash** benchmark, but much more moderately. This makes that the performance improvements of the hybrid model over the base system increase with the thread count. For example, in **RB** (case a) with 15 threads, lock elision (**rw**), gains a factor

of 2.72×, while **noread** (that saves the read set storage and validation) and **nowrite** (that saves any logging and commit steps) provide speedups of 4.77× and 5.49×.

The poor scalability of the base STM system (**sw**) in the **RB** and **skip** benchmarks can be explained by profiling the **sw** and **nowrite** cases in Figure 2-10. The left plot dissects the cycles spent on the different execution phases of the STM. It shows how the commit phase of the **sw** design is almost removed in the **nowrite** approach since the validation and update steps are unnecessary. Also, this commit phase grows in **sw** with the number of threads due to the cycles spent manipulating the locks of the read-set objects. To verify this, the right plot studies the lock handling time. This time grows with the thread count due to the contention on the lock **reader_count** field, what occurs especially toward the root of the shared structure. In the case of **nowrite**, the lock action is reduced to a simple check that does not introduce coherence contention and remains constant with the thread count.

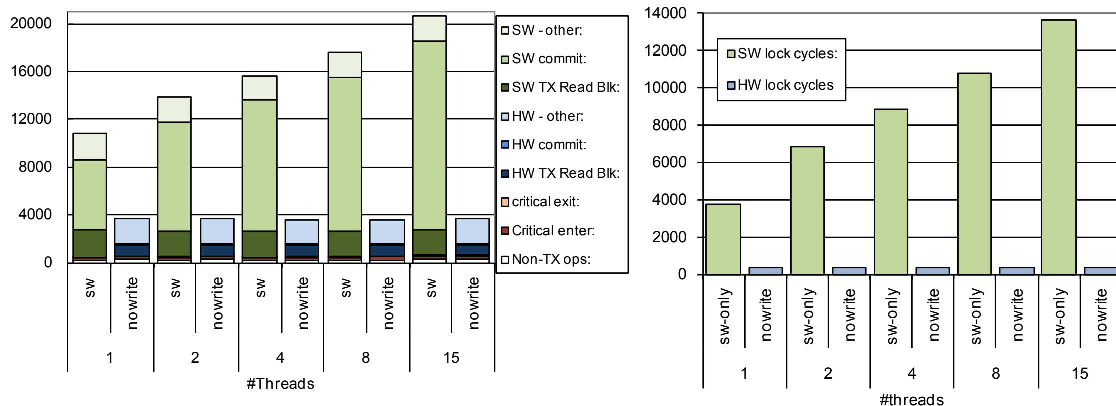


Figure 2-10: Left: RB cycle dissection, p=0, k=8. Right: Cycles in lock accesses

2.4.4.3. Reader-writer transactions

This section deals with the common case of concurrent read-only and writing transactions. Since both RB and skip have shown to perform similarly (as seen in Figure 2-9 and on simulation results not included here) the evaluation will be restricted to only one of them. Evaluations will use writing transactions with $p = 10\%$, and vary the size of the data structure to modify the contention between multiple threads. Each HW transaction is retried 3 times before it is reverted to the software mode. Figure 2-11 shows the performance obtained with the skip-list benchmark under low (left, $k = 15$ and $p = 10\%$) and high (right, $k = 8$ and $p = 10\%$) contention.

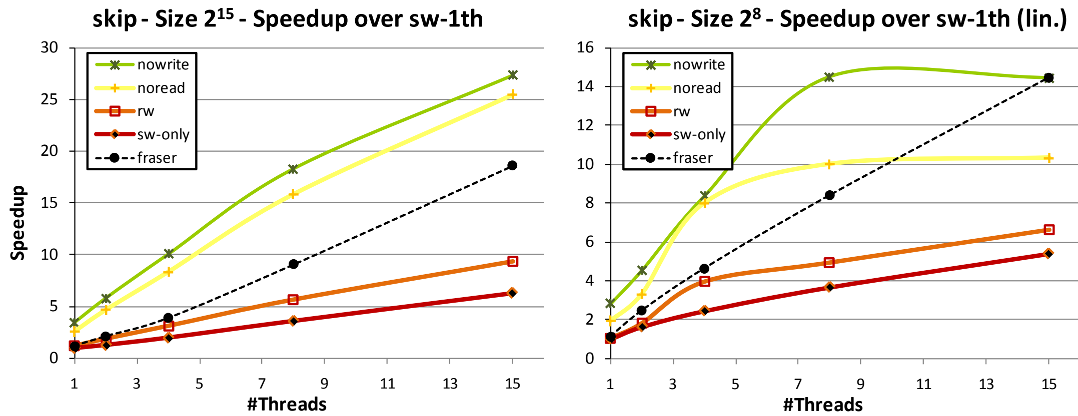


Figure 2-11: Skip-list performance under low (left) and high (right) contention, p = 10%

With low contention (left) both Fraser's OSTM and the hybrid models scale well. It seems that the hybrid versions' performance slightly degrades with the number of threads, but that effect is minimal. The right plot shows the performance under high contention, with a slight super-speedup observed with 4 threads in some cases. Such effect corresponds to the increased cache capacity when multiple processors are used. In this contended case the performance of all models suffers as the thread count increases, as expected. However, this performance penalty is much higher in the hybrid models than in **fraser**; in fact, the performance of the best hybrid model **nowrite** with 15 threads drops to almost resemble that of the **fraser** model.

This degradation comes from two reasons. First, it is due to the higher proportion of slower SW transactions triggered by the increase of the rate of HW transactions aborted, as shown in Figure 2-12. Though the actual HW abort rate is not very high, LogTM makes processors wait on conflicts rather than abort, leading to a significant impact on the overall performance with a small rate of aborts. The second reason is the lack of fairness that will be studied in Chapter 3.

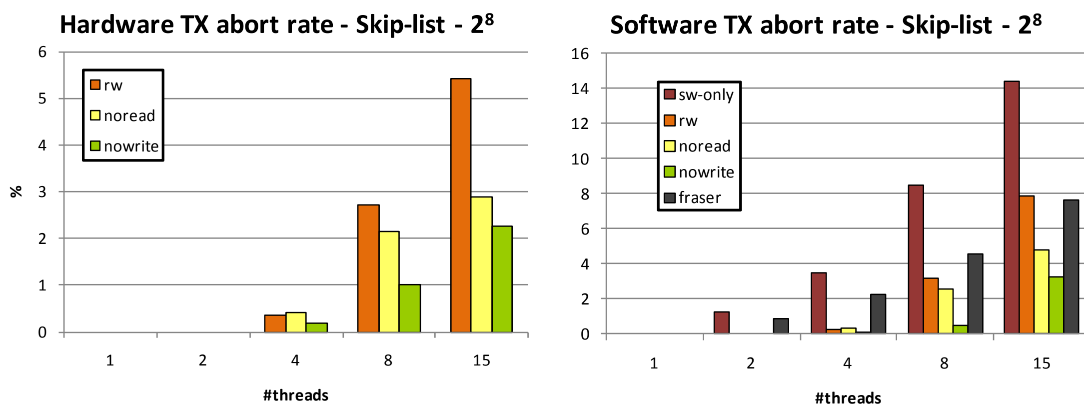


Figure 2-12: Transactions aborted in HW and SW modes. Skip list with k=8, p=10%

Finally, Figure 2-13 shows the performance in a contended case of the hash-table benchmark ($k=8$, $p=25\%$). The base STM performs better than before, since there is no root-node reader congestion. The right plot shows that, as collisions are more infrequent in this data structure, the abort rate is much lower and hence, the scalability higher. The system basically preserves the relative speedups of the different execution modes presented in Table 2-2 for the different thread counts.

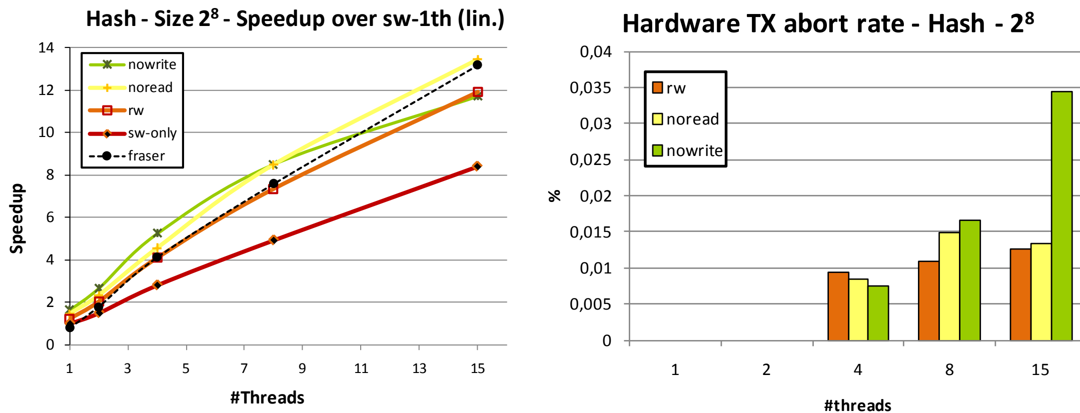


Figure 2-13: Hash-table speedup and abort rate with k=8, p=25% (high contention)

2.4.4.4. Number of HW retries in HW transactions

The analysis in section 2.4.4.3 retried each HW transaction for 3 times before reverting to the sw-only mode. To validate the suitability of this threshold, the contended tests presented in Figure 2-11 (on the right) have been repeated with different thresholds, only for the optimal **nowrite** mode, with up to 32 threads. The results are normalized to the case of not retrying the HW transactions, directly switching to the sw-only mode.

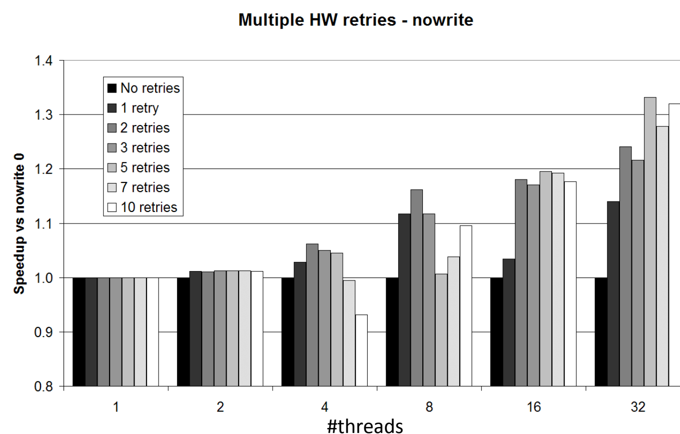


Figure 2-14: Performance with different number of HW retries of aborted transactions

As seen, the performance is generally improved when transactions are retried multiple times, up to a maximum of 33% in the case of 32 threads. However, the optimal value for the number

of retries is low (about 2) when the number of threads is low, from 4 to 8. This optimal threshold increases for higher thread counts, ranging from 5 to 10. A similar behavior can be observed in **rw** and **noread**. This suggests that a fixed re-execution policy is not ideal. When the number of threads is low, the conflicts will be lower, and a small number of transaction retries will suffice. However, when HW transactions abort due to some HW limitation (such as accessing the GC inside a transaction), or conflicts with a SW transaction, the conflict will likely occur again in the HW retry. Therefore, a threshold too large causes too many unnecessary retries in HW before switching to the safe and slow SW mode. As discussed earlier in the beginning of section 2.3, the Rock processor contains a **cause** field in a status register that informs about the reason of a transaction abort, so the runtime can decide if it should be restarted in HW or SW. Our design does not consider it for its loss of generality, but would clearly benefit from such information.

2.5. Summary

This Chapter has introduced a novel Hybrid Transactional Memory whose conflict detection mechanism is based on locks. The base STM provides fair access to the transactional resources thanks to the use of MCS fair reader/writer locks. The base lock structure is modified for HW and SW transactions to correctly interact with each other. All the changes to the system are performed in the STM library, so the programmer does not need to be aware of the hardware support, and single path of execution is required at compilation and execution time for both hardware and software transactions.

Evaluations have been performed in a simulation tool, with different microbenchmarks that stress the system with varying parameters to control the congestion. The results show that hardware acceleration elides a significant part of the runtime, what gives a speedup of more than 3× in single-threaded tests. In multithreaded cases, the hardware support removes the reader-locking congestion and allows for nice scalability of the system, except for highly congested cases with a high number of transaction aborts. Such cases cause a large number of transactions in software mode, which are penalized when they encounter a conflict with hardware transactions.

The next Chapter will focus on the problem of the different priorization policies for hardware or software transactions, proposing a low-cost mechanism, the Reservation Table, that restores fairness and increases performance.

Chapter 3. Fairness in Hybrid Transactional Memory

Chapter 3

Fairness in Hybrid Transactional Memory

The lack of fairness in a Transactional Memory system can lead to starvation pathologies. Specifically, writers can starve when updating a location if it is frequently accessed by readers, as discussed in [16]. Such case happens in the RB and skip benchmarks used previously: when a writer pretends to update a node which is close to the root, there are often multiple concurrent readers of the same node. However, in the original STM the use of fair reader-writer locks provides fair access between different transactions, eventually allowing the writer to proceed. The use of fair reader-writer locks is a simple mechanism; while many other STMs make use of more complex contention managers, the obtained effect is the same.

This section studies the effect of an unfair HTM applied to the HyTM model presented in Chapter 2. Specifically, it is detailed how the use of the HTM removes the initial fairness properties and allows for starvation scenarios, which can degenerate to deadlock in corner cases. This effect is similar to the starvation problem that was presented in Figure 1-9 for reader-writer locks. A new HW mechanism is proposed, named Directory Reservations, to overcome these problems. Evaluations show that such mechanism performs appropriately, providing more performance than other designs for HTM-only systems.

3.1. Writer starvation in Hybrid TM

The fairness between different transactions in the base STM is provided by the use of per-object queue-based FIFO locks. These locks ensure that any transaction will eventually manage to access the object. Although the transaction could abort in the validation step after acquiring the lock and have to retry, no starvation is caused by the locking mechanism.

However, on the HyTM system this fairness guaranty is lost when using LogTM as the base HTM (or, in general, any eager conflict detection HTM that stalls the requestor of conflicting addresses). The problem comes from the way that LogTM extends the coherence mechanism to provide atomicity, presented in section 1.4.11.4 and detailed with an example in Figure 3-1: Coherence requests (step 1, exclusive request from processor **B**) are forwarded to the transactional readers (step 2, processors **A** and **C** are sharers), which detect the conflict with R/W bits or a hashing mechanism. If a conflict is detected, a **NACK** (“Negative

Acknowledgement") reply is sent to the coherence requestor, temporarily denying access to the line (step 3). This prevents the requestor from modifying any line that has been transactionally read or written. Similarly, it prevents any reader from accessing transactionally modified lines.

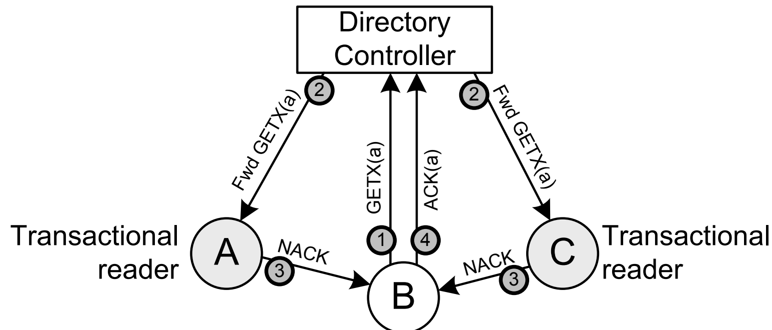


Figure 3-1: Example of NACK messages in LogTM. Proc. B requests the memory location *a*, which has been read by processors A and C

This NACK'ing mechanism effectively constitutes a hardware-based lock on the lines read or modified during the transaction. It can be read-locking when a transaction reads a line, in which case several transactions can concurrently access the shared line since no **NACK** is sent to **GETS** ("GET Shared") coherence requests. Alternatively, it constitutes a write-locking mechanism if the line has been modified by a transaction and is kept with exclusive coherence permissions, sending **NACK** replies to both **GETS** and **GETX** ("GET eXclusive") coherence requests.

This kind of multiple-reader, single-writer locking can lead to writer starvation on frequently read lines. The problem occurs if two or more processors are continuously running transactions that hold the same cache line in their read set, while a third processor is waiting to make a transactional write to that line. The putative writer will be continually NACK'ed while the readers continue executing transactions. A pathological example code is shown in Figure 3-2, where **a** is a shared variable initially set to 0, **N** is the thread count and **th_id** is a per-thread id. When there are enough threads running this code, execution never ends due to writer starvation. Without special contention management, performed simulations show that even four threads are enough to block the Hybrid TM system, due to starvation of some writer transaction. A more realistic example could be a shared queue in which different consumers check a "type" field in the head object to decide whether removing it or not.

```
1: while (a < 1000){
2:    atomic{
3:        if ((a %N)==th_id) a++;
4:    }
5: }
```

Figure 3-2: Pathological example of transactional code that stalls due to writer starvation

Similar problems occur for example in the red-black tree microbenchmark. With 31 threads (in a machine with 32 processors), after starting a couple of thousand transactions, and depending on the program run, from 4 to 12 processors are stalled trying to modify some node which is frequently read because it is located close to the root. This starvation anomaly does not happen in the base STM. Next section introduces the idea of Directory Reservations, a novel, low cost HW mechanism that builds a "semi-fair" queue to prevent this starvation problems.

3.2. Directory reservations

The general idea of Directory Reservations is that NACK'ed processors will issue a request to the directory controller to *reserve* the line for them. This request is piggybacked in the coherence response, so it incurs in no additional traffic. This case is depicted in Figure 3-3, where the **ACK** message (step 4) contains a **reservation** bit set, so it is labelled **RESERV** in the figure. The figure also shows the new behaviour: Whenever any other processor (**D** in the Figure) issues a **GETX** or **GETS** request for the same line (step 5 in Figure 3-3), the directory controller determines if the current requestor is the one that reserved the line. Since it is a different requestor, it sends a **NACK** message (step 6) to **D** without any need to forward the request to the current sharers. Only requests from the processor that reserved the line (processor **B** in the example) will be forwarded to the corresponding owner or sharers. Eventually, in a general case, the blocking processors (**A** and **C**) will commit their transaction. When this happens, no new **NACK** will be issued to the coherence requestor, so **C** will receive the valid data with the valid permissions.

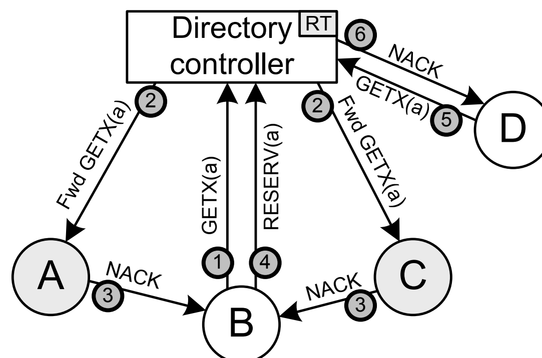


Figure 3-3: Message transfer with the Reservation Table mechanism

To provide this behaviour, new hardware must be added to the directory controller. The directory is extended with a new **R** (`Reserved') bit per line, and an attached Reservation Table (RT) to support the new functionality, as presented in Figure 3-4. A 'Reserved' bit **R** is added to each directory line, but the Reservation Table size is small, in the example containing only 3 entries, since the number of concurrent reservations (which depend on the transactional conflicts) should be small. If the reservation requests exceed the capacity of the RT, the request is not attended until an entry is released. When a reservation request arrives, if there

is an available entry in the RT, the **R** bit is set and the corresponding **address** and **requestor** fields are recorded in the RT. The rest of the fields of the RT will be explained later. Subsequent requests on the same line find the **R** bit set and check the **requestor** field in the RT to determine if they can proceed or not. The reservation is maintained until the requestor obtains the coherence permission on the block; the final **ACK** of such request triggers the RT entry deallocation and **R** bit clearance.

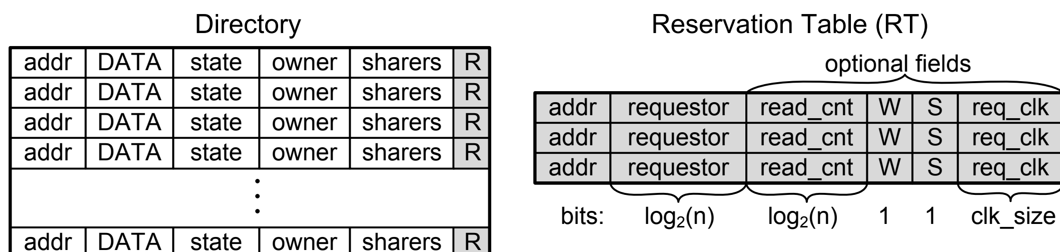


Figure 3-4: Reservation Table structure

The use of a per-line reservation bit **R** implies a relative overhead, since all the memory in the system must be tagged. Two alternative implementations are discussed next. Firstly, not using any indication bit removes completely the memory overhead. However, this implies that any request to any memory location that arrives at the directory must be checked against the RT entries. Although the RT is small and the RT lookup could be probably overlapped with the directory access, this approach unnecessarily increases the energy consumption, and the RT access can become a bottleneck in the directory controller. An alternative implementation could be to use a single reservation bit per directory controller. Such bit would be set whenever there is any reservation active in that given memory controller. Only when the bit is set, all the requests are checked against the local RT entries.

Next sections deal with different issues that arise when the RT hardware is used, including how to deal with different potential deadlock situations and how to provide fairness guarantees with this model.

3.2.1. Issues with LogTM transactions

Depending on the details of the HTM used, the Reservation Table mechanism has to be adapted to guarantee forward progress and prevent deadlock. This section covers the specific changes required to integrate this mechanism with the LogTM HTM model.

First of all, the processors that are sharing a line and NACK'ing incoming requests (**A** and **C** in the example of Figure 3-3) need to invalidate their local copy of the line after commit. Otherwise, they might start a new transaction after commit that also reads the same line. To this end, we add a new **requested** flag to the L1 cache lines, which is set by a directory indication when the reserved block is requested (step 2 in Figure 3-3). When the transaction commits, all of the **requested** lines in the local cache are invalidated, or sent back to the directory if they have been modified.

Additionally, it has to be considered that the proposed model generates a dependency between the threads running in different processors. In the example presented in Figure 3-3, the forward progress of the thread in **D** depends on the progress of **B**, which eventually depends on **A** and **C**. The dependency occurs at different levels: **D** depends on **B** because of the reservation mechanism, while **B** depends on **A** and **C** because of the transactional conflict. In this case, a problem arises if the processors **A** or **C** incur a transactional conflict with a transaction running in **D**. Suppose for simplicity that such conflict occurs in line **b**, different from the one which is reserved, **a**. This would cause a deadlock, e.g.: **A** waiting for **D** (transactional dependency), **D** waiting for **B** (reservation), and **B** waiting for **A** (transactional dependency). This case is depicted in Figure 3-5.

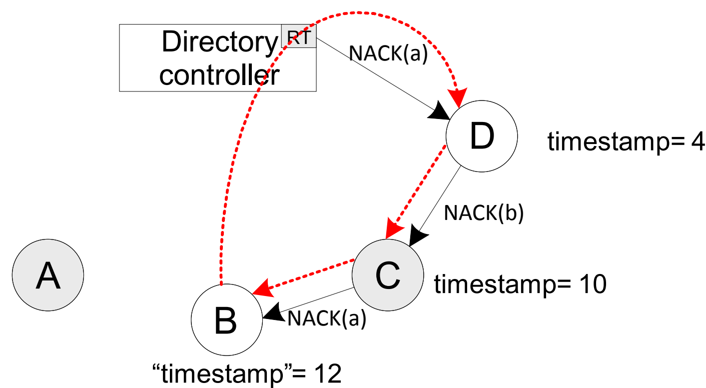


Figure 3-5: Message transfer with the Reservation Table mechanism

The solution has to consider the detailed deadlock avoidance mechanism in LogTM, which was introduced in section 1.4.7.3: Each transaction has an associated unique timestamp, and transactions abort when a possible cycle is detected. To detect this case, each processor contains a **possible_cycle** flag which is set when the transaction sends a **NACK** to an older transaction (in terms of timestamp order) and cleared when the transaction aborts or commits. Transactions abort when they have the **possible_cycle** bit set and receive a **NACK** from an older processor. This solves the deadlock problem, without aborting all of the transactions in the cycle, and ensures that the oldest transaction never aborts. However, there can be "false positives", in the sense that processors can abort in absence of any cycle in the system, but this rate is low.

In the example presented in Figure 3-5 each processor has its own timestamp. The processor B does not necessarily have to be running a transaction, but a timestamp is assigned in any case, which refers to the cycle of the request if it is non-transactional. With the given configuration, B is the "youngest" processor, but since it does not send a **NACK** reply to anyone, it never sets its **possible_cycle** flag and never aborts. Instead, the **NACK** is sent from the RT.

To brave deadlock in this case, we modify the RT structure to include an additional **timestamp** field (not depicted in Figure 3-4). When the reservation is set, it records the timestamp of the request, 12 in the example. Subsequent requests from different processors (such as the

request from **D** in the case depicted in Figure 3-5) are NACK'ed only if their timestamp is higher (younger) than the recorded timestamp. This prevents a cyclic dependency and allows the transactional conflicts to be directly solved between the involved processors. In this example, the request from **D** conflicts with the transaction in **C**, so given their relative timestamps, **C** will have to abort its transaction and **D** will proceed.

3.2.2. Reservation Table and fair queuing

The reservation mechanism proposed in the previous sections enables any writer to proceed with its execution after the current sharers of the line commit their transactions. However, it does not implement any queue for the remaining readers or writers. Once the reservation is cancelled, the rest of the requests will race for the line. This can lead to undesired effects, such as lack of the initial fairness, reader starvation (if the implementation did not allow NACK'ed readers to reserve a blocked line, and transactional writers constantly modify it) or LIFO accesses (if NACK'ed retries apply an exponential backoff wait between consecutive coherence retries). This section shows how to implement a policy that ensures certain fairness, but without the requirements of a FIFO queue.

This mechanism makes use of the optional **read_cnt** field (count of pending readers) and **W** ("write-requested") and **S** ("served") flags in the RT. Initially, these fields are all 0. The idea is to record the number of readers that request the line between a pair of writers. Once the reservation has been set, any GETS request for the same line will be NACK'ed as explained previously, and it will increase **read_cnt**. To prevent counting the same read request twice, every request message includes a **nacked** flag, which is set on every retry. Only requests with **nacked** = 0 increase the **read_cnt** in the RT. Whenever a new GETX request (not coming from the saved requestor) arrives, the **W** flag is set, and **read_cnt** is no longer incremented.

Once the original reservation is served (what sets **S** = 1), any further GETX request is NACK'ed if there are pending readers. The directory controller will decrease **read_cnt** on every GETS request served. When **read_cnt** reaches 0 and the **W** bit is set, only a GETX request will success (and, in case of a new conflict, generate a new reservation). Meanwhile, exclusive requests are NACK'ed by the directory. The specific actions of the directory controller when it receives a request for a reserved block are specified in the diagram of Figure 3-6. This case covers the timestamp check discussed in the previous section, the queue formation (the bottom left area) and the queue shrinking (the bottom right area). Note that "serve request" typically means that the request is forwarded to the owner or sharers, what does not imply that it will be acknowledged.

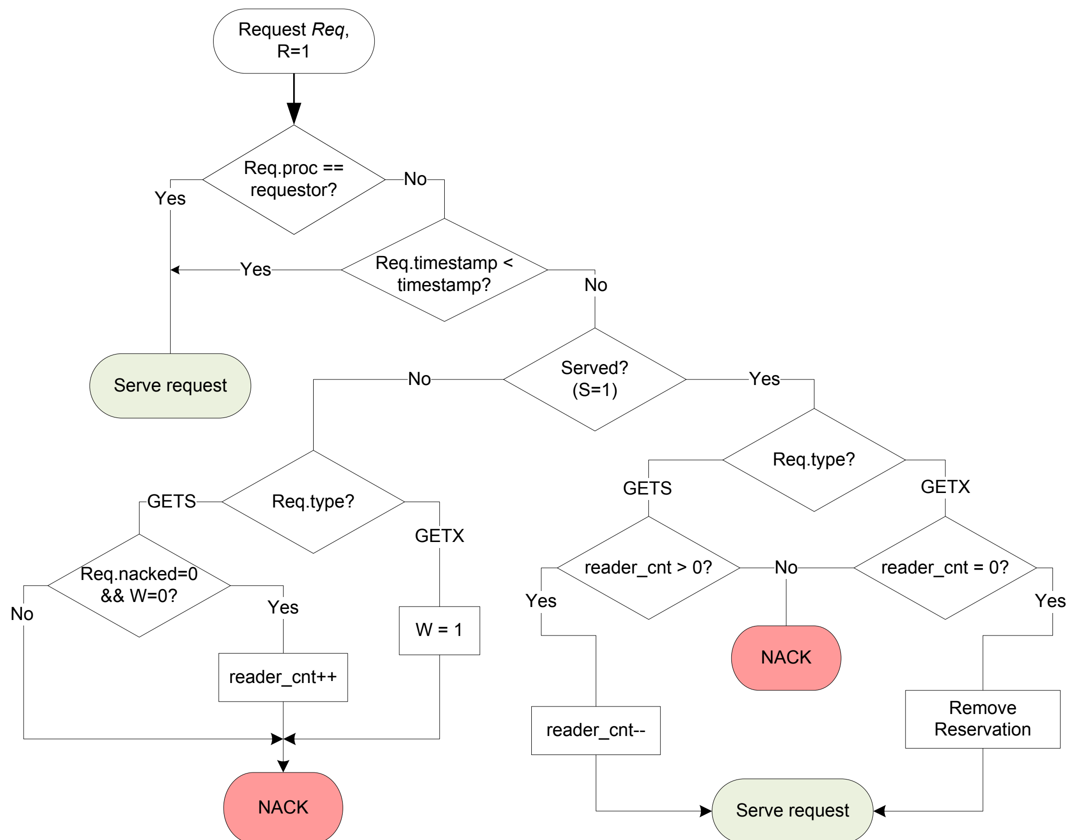


Figure 3-6: Message transfer with the Reservation Table mechanism

This design does not implement a real queue, given that the directory is not aware of the identity of read and write requestors. Once the original reservation is served, only the amount of read requests before any write request will be preserved. If new readers try to access the line, nothing prevents them from doing so before the next writer succeeds. If a new writer comes and wins the request race, its request will be satisfied. However, this is enough to make sure that the proportion of sharers and writers in the queue is satisfied.

3.2.3. Thread de-scheduling and migration

If the reservation owner get de-scheduled there would be no request for the reserved line when the resource became free. This would block any other threads trying to access the line. This case does not generate a deadlock, but a temporal starvation in the same manner as de-scheduling a thread which is waiting in a lock queue, as discussed in section 1.3.1.6.2. As described in that section, this problem has been covered in multiple works [128, 127, 67] with timestamp mechanisms that detect requestor evictions and remove them from the queue . Waiting threads periodically “publish evidence” that they are still iterating, in the form of an increase of their timestamp. If other thread finds a timestamp not increased in a long time, it can “jump ahead” the queue.

The Reservation Table includes a clock field (**req_clk** in Figure 3-4) that records the clock of the last served request (green fields in the diagram of Figure 3-6). This field serves as the evidence

of the active iteration. On every new GETX request received from the reservation holder, the counter **req_clk** is updated with the current clock. When a conflicting request is received (that should be NACK'ed by the directory), the counter is checked against the current clock. If the difference exceeds a threshold, a timeout is triggered and the reservation is removed to prevent starvation. Such threshold will be set to several times (2 or 3) the maximum delay between requests, including the exponential backoff if used, to consider the case in which network congestion delays a request.

By contrast, after the request has been served, the **req_clk** field must be updated each time a reader is served. If no reader retries its original request in a long time, the reservation is also removed after the timeout to prevent starvation of waiting writers.

This model also covers the case of a thread migration. Specifically, if the reservation owner thread migrates to a different processor, subsequent coherence requests will find the line reserved for the original processor. The problem is that the reservation mechanism records the physical processor id, not the logical thread number. With the timeout mechanism, the reservation will eventually expire and the new processor will access the line or reserve it again if required. This case is highly uncommon, but its support is required to ensure that the reservation mechanism does not introduce any deadlock possibility in the system.

3.3. Evaluation

In the multiple-reader, single writer problem, prioritizing readers over writers provides the highest throughput, as a single writer never stops multiple concurrent readers. Thus, any fairness mechanism will reduce throughput, as long as it does not introduce any of the discussed starvation problems. Similarly, the use of the Reservation Table can locally introduce more contention in the system as the reservation owner stops furthers threads that could proceed with their execution. On the other hand, starved threads do not contribute to the global throughput, so temporarily blocking other threads to guarantee writer progress might lead to higher throughput. Thus, the goal of the implementation would be to produce the lowest performance degradation while still ensuring starvation freedom.

The proposed model has been implemented in GEMS, using a Reservation Table with 3 entries. The model implemented the necessary compatibility with the base LogTM protocol to prevent any possible deadlock. The optional semi-fair queuing and eviction detection mechanisms were also used. The obtained performance has been compared against the base hybrid **nowrite** model with no Reservation Table (labelled **HyTM** on the plots). The same benchmarks presented in section 2.4.3 have been used. Three versions of the RT mechanism with different reservation policies have been studied:

- **RT-always**: The base mechanism, NACK'ed requests always make the reservation.

- **RT-delayed**: the NACK'ed request is retried several times before it makes the reservation. Specifically, a value of 100 was used in the experiments.
- **RT-oldest**: the NACK'ed request only performs the reservation when the NACK'ers have higher (younger) HTM timestamps. This prevents a writer from stopping transactions that started before.

The RT proposal was also compared with a different conflict resolution policy: the Hybrid model presented in [16] (labelled **Abort** policy in the following figures). Such mechanism targets the same problem, but considering dependencies between HTM transactions only. With this policy, write requests abort readers that have a higher (younger) timestamp, rather than waiting for them to commit.

The performance results are normalized against the base HyTM model. The evaluation plots will show the execution time, divided among the different execution phases of the transactional memory runtime.

3.3.1. Performance results

Figure 3-7 presents the cycle dissection of the average transaction execution time of the RB-tree benchmark with size $k=11$ and $p=10\%$. The execution time corresponding to HW transactions uses different tones of blue, while the time corresponding to SW ones (after the corresponding HW aborts) is coloured in shades of green. Aborted execution uses a striped pattern. The figure shows performance results running with 2 to 31 threads (simulating a 32 processor machine in this last case).

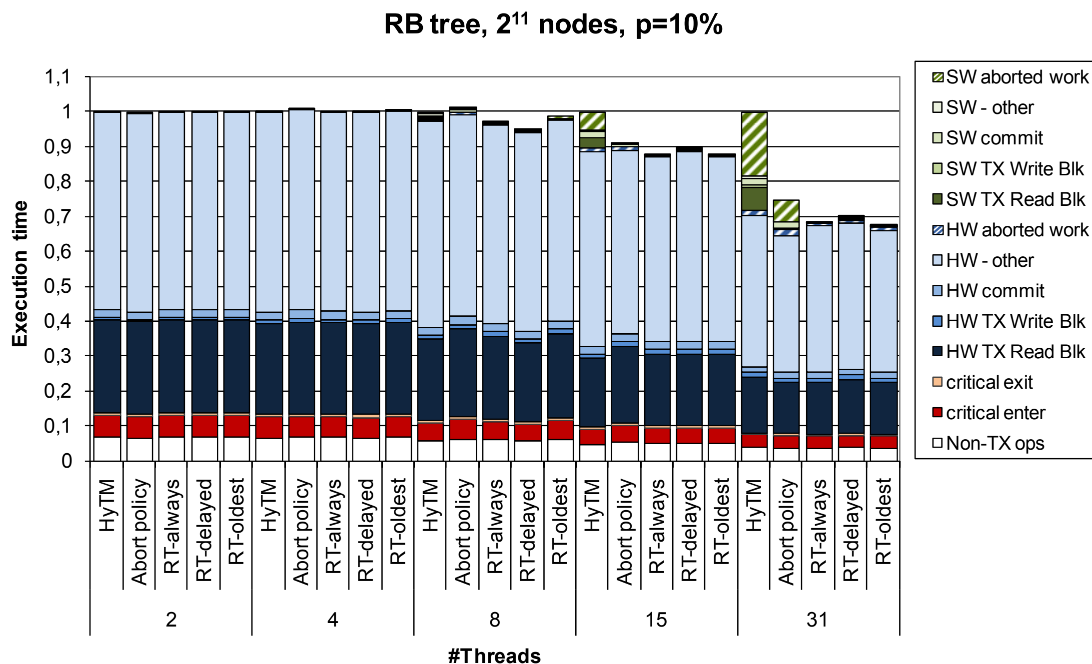
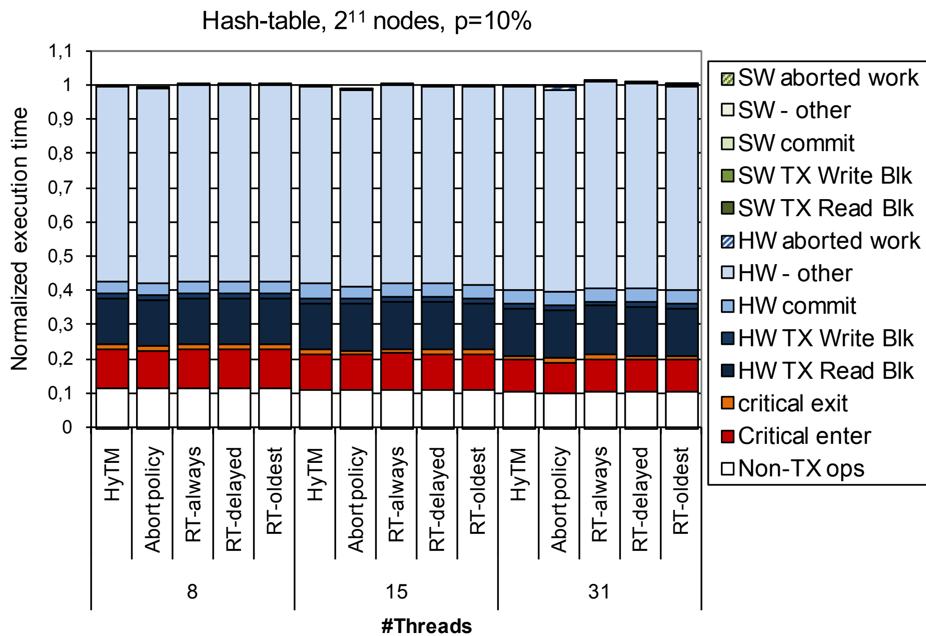


Figure 3-7: Normalized execution time of the RB benchmark, k=11, p=10%

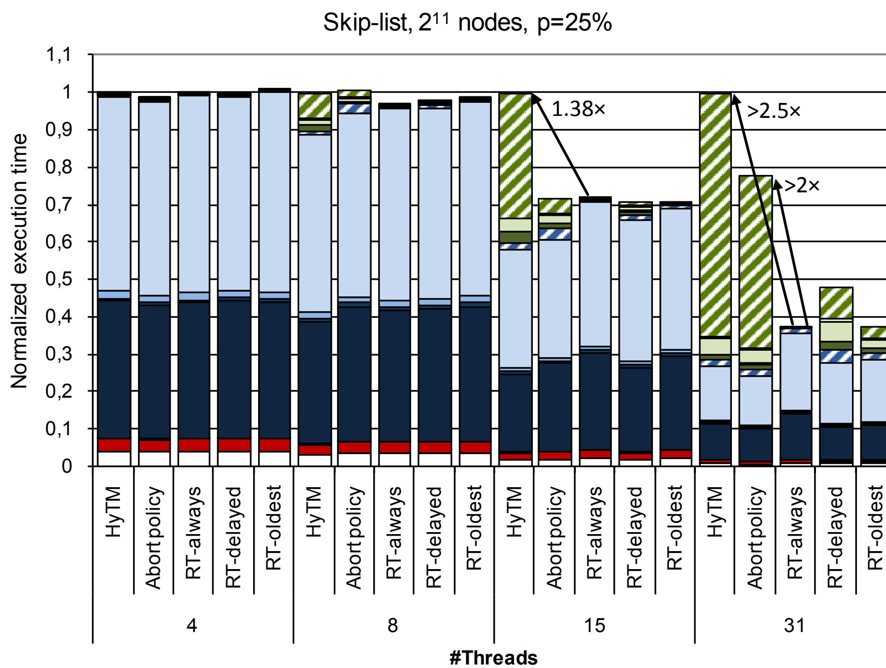
This experiment provides a reasonable congestion level, which increases with the thread count. With a small thread count (2 to 4 threads), the system does not suffer from writer starvation, since collisions are uncommon. It can be observed that the RT mechanism does not penalize execution time, what should be obvious since it is hardly ever used. The contention increases with the thread count. The results with 15 and 31 threads in the base HyTM model show a higher amount of time wasted in SW and HW transaction aborts (dashed sections) and a significant amount of SW transaction time (greenish sections). The RT mechanisms stop such congestion halting new readers and the performance is significantly increased up to 30.8% with the best policy, which happens to be **RT-oldest**. There are no large differences between the three evaluated reservation policies. By contrast, the **Abort** Policy model is based on aborting running transactions, which increases the abort count and fastens the switch to the slower STM mode, more prone to aborts. The maximum performance increase obtained with this policy is only a 25.03% with respect to the base model.

Figure 3-8 shows performance results for other two cases. The top case shows the execution time of the hash-table benchmark. Since this benchmark presents almost no congestion, the Reservation Table is almost never used, and even with 31 threads the performance of all models is similar. By contrast, Figure 3-8 b shows the execution time of the skip-list benchmark with $p = 25\%$. In this case, the congestion is much higher, what is reflected in a much larger number of HW aborts and switches to SW (green blocks) in the base HyTM model. With 15 threads, any of the evaluated mechanisms provide a similar performance improvement. However, with 31 threads, the **Abort** policy provides a much lower performance, since it

increases the number of aborts, and therefore the switches to SW transactions. Both **RT-always** and **RT-oldest** provide a similar performance, reducing the original execution time in more than a 60%, leading to a speedup larger than 2.5×.



a) Hash-table tree benchmark, p=10%



b) Skip-list benchmark, p=25% (same legend as case a)

Figure 3-8: Normalized execution time of Hash and skip, k=11

3.4. Summary

This Chapter has studied the unfairness problems that occur when different congestion management policies are employed for hardware and software transactions in a Hybrid Transactional Memory system. It has been shown how these problems generate temporary starvation, which penalizes performance and can degenerate to deadlock.

The Reservation Table has been proposed as a low-cost mechanism that targets these unfairness problems in HyTM. Specifically, the table records some addresses which are “reserved” for the starved thread, and prevents accesses from different threads. The mechanism is designed to minimize the number of transactional aborts, what allows for a high performance since it reduces the amount of wasted work. Finally, it has been shown how the Reservation Table management could introduce deadlock if the deadlock detection mechanism of the HTM was not considered in the design, and a valid policy for the LogTM model has been proposed.

The proposal has been modeled and evaluated with the same tools as the work in the previous Chapter. Evaluations show that the system does not harm performance when contention is low, and it increases performance when contention grows. The performance increase mainly comes from the lack of starved threads and a lower rate of aborted transactions that switch to software. Specifically, in highly congested workloads the system obtains speedups higher than 2.5× from the base model, and higher than 2× when compared to other mechanism designed for HTM-only systems.

Chapter 4. HW acceleration of locking mechanisms

Chapter 4

HW acceleration of locking mechanisms

Previous sections have discussed the importance of locking mechanisms in parallel programming. Traditional parallel programs in shared-memory machines typically rely on locks to protect the access to shared data. Also, Chapter 2 has presented the benefits of lock-based software and hybrid transactional memory. As discussed in section 1.3, different implementations of the locking mechanism provide different runtime overheads, memory requirements, fairness guarantees and others.

Being the locking mechanism so important in a parallel system, it is interesting to note that most commercial architectures simply rely on the shared memory coherence mechanism for software locks, without providing any specific hardware support. Multiple hardware mechanisms have been developed to aid locking. These mechanisms allow for a fast and low-cost implementation of a locking mechanism, which in many cases allows the use of a fine-grain synchronization: locking mechanisms in which very small pieces of data are protected by individual locks, relying on the low overhead of the locking implementation. However, these proposals are too specific or limited to be widely deployed and completely replace software locks. Section 4.1 will consider these previous proposals, and discuss why they have not been implemented in other than research systems.

This Chapter will introduce a HW-supported locking mechanism called the Lock Control Unit (LCU). This mechanism, which slightly resembles the Reservation Table introduced in Chapter 3, builds hardware queues for fast and efficient lock transfer between the different lock requestors, supporting reader/writer locking and thread suspension and migration without significant performance degradation. The mechanism will be detailed in section 4.2, and evaluated in section 4.3.

4.1. HW mechanisms for locking acceleration

Fine-grain synchronization has been supported in hardware with word-level tags (such as Full/Empty bits) in many research machines such as HEP [76], Tera [8], the J- and M-machines [32, 80], or the Alewife Machine [6]. These architectures associate a synchronization bit with each word, which acts as a lock. While these bits provide a simple architectural support for

synchronization, they incur a significant area overhead, and do not support reader-writer locking mechanisms. The changes that would be required to support RW-locking, such as a reader counter per memory word, would imply unaffordable area costs.

QOLB [55] extends the performance of synchronization bits by building a hardware queue between requestors. QOLB allocates two cache lines for every locked address: the valid one, and a *shadow* line. One of the lines is used to maintain a pointer to the next node in the queue, if present, and the other one is used for the actual data and local spinning on the syncbit. When the **unlock** function is invoked, the pointer in the *shadow* line is used to notify the next processor in the queue, allowing for fast direct cache-to-cache transfer. As with previous designs, QOLB is designed to support mutual exclusion and does not support RW locks. The QOLB primitive has been implemented as part of the IEEE SCI standard for communication in shared-memory multiprocessors [58]. The implementation, which minimally differs from the original QOLB proposal as presented in [4], leverages the pointer-based coherence mechanism proposed for SCI. This paper acknowledges some problems of the implementation, such as the noticeable performance degradation when evicted threads receive a lock, and the issues that arise with process migration. As far as we know, the SCI standard has had a low industrial acceptance, and QOLB-compliant products have not been developed.

The previous designs were not tied to a specific architecture. Both the System on a Chip Lock Cache [123], and the Lock Table [23], propose a centralized system to control the state of several locks in a bus-based parallel machine. Acquired locks are recorded by leveraging the bus-based coherence mechanism, so the scalability of these mechanisms is limited. Moreover, overflow and thread migration are difficult to handle.

Different hardware mechanisms directly support locking on distributed systems. The Stanford Dash [91] leverages the directory sharing bits for this purpose. When a lock is released, the directory propagates the coherence update to a single lock requestor, preventing contention. After a while, all the remaining requestors receive the update, to prevent starvation if the original requestor does not acquire the lock. Other systems use remote atomic operations executed in the memory controller instead of the processor. This is the case of the **fetch-and-θ** instruction of the Memory Atomic Operations (MAO) in the MIPS-based SGI Origin [89] and Cray T3E [129]. These systems allow for remote updates in main memory, removing the problem of coherence lines bouncing on lock accesses. All remote updates take the same time, which is typically the memory access latency. While they do not support direct cache-to-cache transfer of locks, they do not use L1 memory at all. More elaborate proposals such as Active Memory Operations (AMO, [44]) or Processor-in-Memory architectures (PiM, [53]) do not further optimize the lock handling, but mainly focus on moving part of the operations to the memory side to reduce the performance penalty caused by the interconnection mechanism.

The Synchronization State Buffer (SSB, [159]) was proposed as a hardware structure to accelerate the fine-grain lock handling mechanisms in the Cyclops CMP system avoiding whole-memory tagging. When a synchronization operation (such as lock acquire) is invoked on a given data memory address, the shared-L2 memory controller allocates an entry in the SSB to control the locking state of such address. Same as in MAOs or AMOs, all lock-related operations are remote (in this case, in the on-chip L2 controller). SSB supports fine-grain reader/writer locks. However, these locks are unfair and can starve writers.

The lock box [154] adds support for fast lock transfer between different SMT threads in the same processor. The lock box is a shared CAM with one register per SMT thread. A valid entry indicates the address of the lock for which the corresponding SMT threads is waiting. Each time a thread releases a lock, it checks the shared lock box and directly notifies another SMT thread if the lock address is found, without accessing the L1D cache, minimizing the communication overheads. Additionally, this mechanism allows controlling the SMT priorization mechanism, so waiting threads receive less processor resources. While highly interesting, this mechanism works in a different level as the mechanism studied in this Chapter, with a much lower scalability, since it requires shared resources inside the same processor.

Finally, multiple proposals focus on accelerating the critical section execution with different means. Speculative Lock Elision [118] removes unnecessary locking by executing the critical section speculatively, and reverts to taking the lock if a conflict is detected. This allows for some parallelism in the critical section, especially when coarse-grain or highly conservative locks are used. Related mechanisms are Speculative Synchronization [103] and TLR [119], which improve the mechanism with timestamps to guarantee forward progress. Speculation in critical sections is not part of the proposal presented in this Chapter, but it will be considered in section 5.3. Accelerated Critical Sections [147] moves the execution of the critical section to the fastest core in an asymmetric CMP. While somehow related, such approach is orthogonal to this work.

The main characteristics of multiple locking mechanisms are presented in Table 4-1. The SW mechanisms on top of the table have been already introduced in section 1.3. The meaning of some of the columns of the Table is presented next. *Local spin* refers to implementations that wait for the lock iterating on a per-thread private location or structure; it is typical in queue-based locks and has the benefit of not sending remote messages while spinning. *Queue eviction* detection refers to the capability of detecting evicted threads in the queue before they receive the lock, so they can be removed to prevent temporal starvation. *Scalability* refers to the system behavior as the number of requestors or processors increase. Single-line locks present coherent contention and scale poorly, while queue-based approaches remove this problem and scale very well. Regarding hardware proposals, they can be limited to a single-bus or single-chip, what can restrict their scalability for larger systems. Some proposals can fail if

the *number of threads exceeds the number of processors*, or if one thread migrates. This happens typically because locks are assigned to hardware cores instead of software threads. *Memory/area overhead* refers to the memory required for the lock, or to the area cost in hardware implementations. Queue-based locks require O(*n*) memory locations per lock for *n* concurrent requestors. Some hardware proposals can have high area requirements (for example, tagging the whole memory) or just require the addition of limited structures per core or processor. The *number messages for lock transfer* represents the minimum number of messages in the critical path of the lock transfer operation. Only lock transfer has been considered, not lock acquisition, which occurs when the lock is free. This number will limit the base transfer latency of the system. It can depend on the specific coherence protocol, but in general, hardware proposals outperform software-only locks. Finally, requiring *changes to the L1 design* such as adding new bits or coherence states is undesirable, since it implies modifying a highly optimized structure, and possibly, the coherence protocol. The Lock Control Unit, as will be presented in the next section, satisfies all these desirable requirements.

		Performance				Flexibility					Implementation overhead		
		Reader-writer	Local spin	Fair	Queue-based	Queue Eviction detection	Trylock support	Scalability	Migration	threads > cores?	Memory/ area overhead	#messages lock transfer (critical path)	L1 changes
SW-only	TAS, contended RW locks	NO/YES	NO	opt	NO		YES	poor	OK	OK	1 cache line	6	NO
SW-only	MCS locks [106]	NO	YES	YES	YES	NO	NO	good	OK	slow	O(n) cache lines	6	NO
SW-only	MCS-RW, Krieger, Lev [82,95,107]	YES	YES	YES	YES	NO	NO	good	OK	slow	O(n) cache lines	6	NO
SW-only	Biased locks [79,122]	NO	opt	opt	opt	NO	YES	good	OK	slow	O(n) cache lines	6	NO
SW-only	Time-published locks [67,127]	NO	YES	YES	YES	YES	YES	good	OK	OK	O(n) cache lines	6	NO
HW-supported	fetch&op, AMOs [44, 89, 129]	NO	NO	NO	NO		YES	good	OK	OK	1 cache line	2	NO
HW-supported	SoC Lock Cache [123]	NO	NO	YES	NO		NO	bus-based	fails	fails	low	2	NO
HW-supported	Lock Table [23]	NO	NO	NO	NO		YES	bus-based	OK	OK	low	2	NO
HW-supported	Tagged memory [6, 8, 32, 76, 80]	NO	NO	NO	NO		YES	good	OK	OK	Memory tags	6	YES
HW-supported	QOLB [55]	NO	YES	YES	YES	NO	NO	good	fails	slow	Memory tags	1	YES
HW-supported	SSB [159]	YES	NO	NO	NO		YES	single-chip	fails	fails	low	2	NO
HW-supported	**Lock Control Unit**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**good**	**OK**	**OK**	**low**	**1**	**NO**

Table 4-1: Comparative of SW and HW locking mechanisms

4.2. The Lock Control Unit mechanism

This section introduces the Lock Control Unit hardware and analyzes its behavior. It is a distributed mechanism that handles reader-writer locking fast and efficiently. The system relies on two new architectural blocks: The Lock Control Unit (LCU) for exploiting locality and fast transfer time (which gives name to the whole mechanism), and the Lock Reservation Table (LRT) to manage lock queues. Each processor or core implements a LCU, which is responsible for receiving the thread's requests and building the queues. The LCU is a table whose entries are dynamically allocated for the requested locks. The LRT is responsible for the allocation and control of new lock queues. LCUs and LRTs communicate to each other with specific messages for requests, transfers and releases of reader-writer locks, while threads only query the local LCU. To minimize transfer time, lock transfers are direct from one LCU to the next.

In our system, locks are dynamically assigned to memory addresses on request. While the lock transfer occurs from one physical LCU to another, locks are associated with virtual threads by using a thread identifier. This allows the system to support thread suspension and migration

with graceful performance degradation. The LCU also implements a mechanism to remove uncontended LCU entries to decrease its occupation, what reduces the probability of entry exhaustion. Even so, the unavoidable case of resource overflow is handled: The LRT is backed to main memory, while the LCU contains specific entries to guarantee forward progress.

This Chapter is organized as follows. Section 4.2.1 details the different hardware units that have been just sketched. Section 4.2.2 provides a general overview of the system behavior. Section 4.2.3 will cover the detailed implementation with the required mechanisms to prevent deadlock, support different locking pathologies and others.

4.2.1. Hardware components

The architecture of the system is presented in Figure 4-1. It has been conceived for distributed shared-memory systems with multiple cores and multiple chips; this text will indistinctly refer to "cores" or "processors". Each core incorporates a Lock Control Unit (LCU) for implementing distributed lock queues. There is one Lock Reservation Table (LRT) per memory controller. Alternative organizations can be considered, as long as each lock is associated with a single LRT depending on its physical address. For example, in a single-chip system with shared static L2 banks, each LRT might be associated with a L2 bank, as occurs in SSB [159]. Finally, there is an optional Free Lock Table (FLT) per core, which is used to obtain biasing in the locking mechanism. Next sections will detail the elements recorded in each entry of these tables.

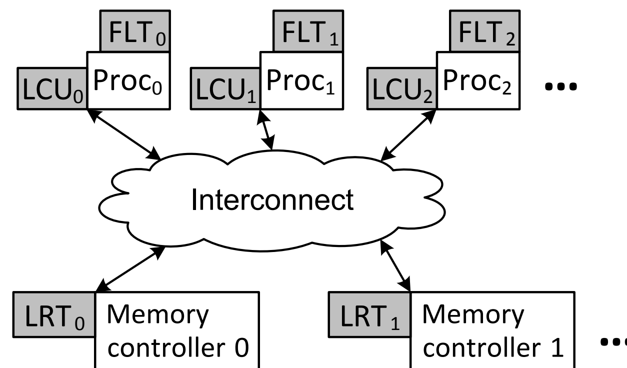


Figure 4-1: Lock Control Unit architecture

4.2.1.1. The Lock Control Unit

The LCU is composed of a table whose entries record the locking state of a given memory address and its associated logic. The composition of each entry is shown in Figure 4-2. The LCU is addressed with the tuple {Addr; $thread_{id}$}, so multiple logical threads on the same core can issue requests for the same lock. LCU entries are dynamically allocated on request, and persist until the lock has been released or the lock is taken without further requestors. The locks can be requested in read or write mode, according to the flag **R/W**. The **Head** flag indicates if the LCU is the first one in the lock queue, and the transfer count **TC** indicates the number of times that the Head flag has been transferred from one LCU to another. The function of this counter

will be further detailed in section 4.2.2.3. The **Next** field is used to build a queue of requestors, containing some details of the next requestor in the queue. Overall, with 64-bit addresses and 16-bit LCU identifiers, each LCU entry requires 157 bits, or less than 20 bytes of added storage.

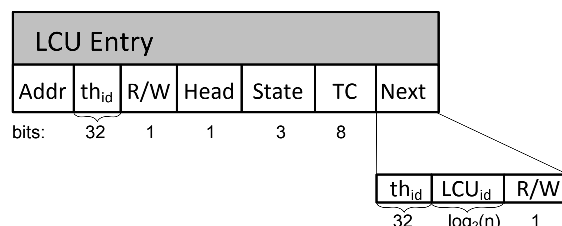


Figure 4-2: Contents of each LCU entry

Valid **states** for LCU entries are presented in Table 4-2. **ISSUED** implies that the local LCU has sent a request for a lock in the corresponding address. **WAIT** means that the LCU has become part of the lock wait queue, and will eventually receive the lock. When the lock is received in the LCU, the state is moved to **RCV**, and when the local thread actually acquires the lock, it is moved to **ACQ**. A released lock can be in state **REL** or **RD_REL** (read-released) before the entry is removed; these two cases will be covered later in section 4.2.2.4. The state **REM** is optional; it depends on the number and type of virtual channels used in the interconnection network. Its behavior and requirements are detailed in section 4.2.3.2.

Status	Meaning
ISSUED	Request issued
WAIT	Request enqueued
RCV	Lock received; not acquired yet
ACQ	Lock acquired
REL	Lock released
RD_REL	Read-lock released, Head=false
REM	Removed; RETRY forward pending

Table 4-2: Possible states of a LCU entry

The LCU will differentiate *contended* and *uncontended* locks. The former are locks required by multiple threads that concurrently try to acquire them, and the latter are locks taken by a single thread without further requestors. Contended locks use the **Next** field to build the queue of requestors, as depicted before. Uncontended entries do not have any other requestor, and hence do not use this field. When an uncontended lock is taken (in state **ACQ**), there is no need to maintain the LCU entry allocated: The required information is saved in the LRT as explained in the next section, and there is no queue to maintain. Therefore, in order to reduce the resource usage, uncontended taken locks are removed to minimize LCU occupation. Hence, LCU entries have a double function. First, they are used as queue nodes for contended locks. Second, they serve as the interface that is polled by the processor to interact with the locking mechanism, slightly similar to how Miss Status Holding Registers [83] provide an interface between the L1 cache and processor, and the rest of the memory hierarchy.

4.2.1.2. The Lock Reservation Table

The Lock Reservation Table (LRT) tracks the state of each hardware lock in the system. Since it will be typically associated with a memory controller, each LRT has control over the locks whose address belongs to the given memory controller. Each LRT contains multiple entries, one for each lock, as depicted in Figure 4-3.

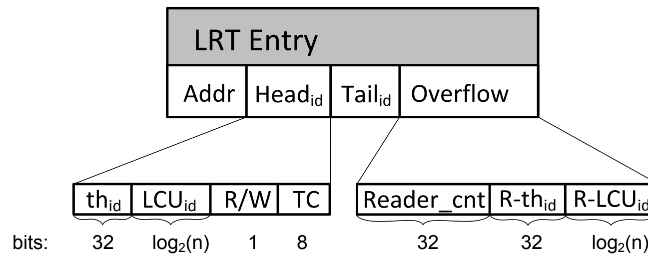


Figure 4-3: Contents of each LRT entry. The $Tail_{id}$ field's composition is equal to $Head_{id}$

Each lock is identified in the LRT by its physical address. The LRT entry contains two pointers, for the head and the tail of the lock queue. Each pointer, **$Head_{id}$** and **$Tail_{id}$**, identifies the lock owner by its thread identifier **th_{id}** and the LCU from which it requested the lock, **LCU_{id}**. Additionally, it records their corresponding **R/W** mode flag and transfer counter **TC**. This counter is used to prevent race conditions, and will be detailed in section 4.2.2.3. Finally, the LRT contains a set of fields to support a so-called *overflow* mode, including a *reservation* mechanism. This lock granting mode and the function of the corresponding fields are detailed in section 4.2.2.2.2. Overall, with 64-bit addresses and 16-bit LCU identifiers, each LRT entry requires 258 bits, or about 32 bytes of added storage.

4.2.1.3. The Free Lock Table

The Free Lock Table (FLT) is an optional component that is used to provide biasing capabilities to the locking mechanism. An entry of this table is depicted in Figure 4-4.

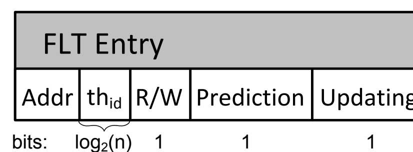


Figure 4-4: Contents of each FLT entry

When a thread releases a lock which is likely to be benefited from biasing, it can be added to the local FLT, instead of sending the release message to the LRT. In this case, the lock's address, **R/W** mode and the owner's **$thread_{id}$** are recorded in the FLT. Further requests for the same lock can find it available in the local FLT, without the requirement for a remote request to the corresponding LRT. The flags **Prediction** and **Updating** will be detailed in section 4.2.2.5.

4.2.2. General overview

This section will show the general behavior of the Lock Control Unit mechanism. It introduces the programmer interface, the different locking modes, lock acquisition, transfer and release operations, and the FLT behavior.

4.2.2.1. Programming interface

The processor communicates with the local LCU by means of two new ISA primitives: **Acquire** and **Release**. Threads issue lock requests to their local LCU, offloading the tasks of requesting the lock grant, forming a queue of waiting threads and transferring the lock to the next requestor.

Acquire and **Release** instructions behave in the following way:

- **Acquire** (**acq**): If the LCU entry for the given address is not present, a new entry is allocated and a request message is sent to the corresponding LRT. If the LCU entry is present with a valid State (**RCV** for write requests, or **RCV** or **RD_REL** for read requests, as explained later), the lock is acquired, returning TRUE. Otherwise, no action takes place.

- **Release** (**rel**): releases the lock, transferring it to the next requestor in the queue, or releasing it sending a message to the LRT.

The **Acquire** and **Release** synchronization primitives use three arguments:

- The (virtual) address to lock. The LCU makes use of the TLB to provide virtual-to-physical mapping. This implies that all the remaining operations work on physical addresses, and special care must be taken in case of memory page migrations.

- A process-wide thread identifier. This value can be provided by the user or the hardware. For example, the Alpha architecture contains a register with the current thread identifier, and the convention in the Sparc architecture says that the global register %g7 should contain a pointer to the current thread structure in main memory.

- The read or write mode, R/W. Alternatively, different ISA instructions might be used for read-locking and write-locking, in which this parameter would not be required.

Figure 4-5 shows the simplest implementation of the **lock**, **trylock** (based on a fixed number of retries) and **unlock** functions, with the trylock based on a fixed number of retries. Alternative implementations can consider the use of timeouts in trylocks, or periodically yielding the processor to prevent owner starvation, as presented in [67].

```
 1: void LCU_lock(addr, th_id, read){
 2:    while(!acq(addr, th_id, read)) {;}
 3: }
 4:
 5: bool LCU_trylock(addr, th_id, read, retries){
 6:    for (int i=0; i<retries; i++)
 7:       if(acq(addr, th_id, read)) return true;
 8:    return false;
 9: }
10:
11: void LCU_unlock(addr, th_id, read){
12:    while(!rel(addr, th_id, read)) {;}
13: }
```

Figure 4-5: Functions for lock acquisition and release

It is interesting that the proposed design considers nonblocking **acquire** and **release** instructions, returning immediately its result of success/failure. That is why the thread must iterate on the call to **acq** until success. A blocking mechanism could be considered, delaying the commit of the **acq** and **rel** instructions until they succeed. This poses two issues. First, the lock acquisition time can be extremely long. A memory request, for example, is satisfied in a variable amount of time, but longer events such as a page miss generate an exception to handle the page migration. Therefore, load/store instructions commit when the operation completes, or raise an exception if required. By contrast, if a lock is taken, the amount of time required for the acquisition to complete depends on the progress of the lock owner, without any possible exception to accelerate or process the acquisition. The second issue is that this operation couples the lock request and the lock acquisition phases, and can make the design of the LCU more complex if the request instruction is squashed in the pipeline, for example, due to a branch missprediction.

Some alternative configurations to the base design can be considered. First, a prefetch mechanism can be used to better tolerate the latency between the LRT and the LCU: Similar to the QOLB instruction in [77], an additional **Enqueue** (**enq**) instruction can be used to add a processor to a queue in advance of the acquisition moment. Secondly, different instructions could be used for the read or write modes of acquiring the lock. This increases the number of ISA instructions to 4, what is clearly affordable, and removes the number of parameters to two, similar to many common arithmetic instructions.

4.2.2.2. Two locking modes

The LRT responds to LCU lock requests by assigning a lock to the requested memory address and granting the lock to the requestor. The LRT supports two different modes of granting a lock. The queue-based mode depicted before is the common case. Alternatively, an overflow mode can be used to prevent starvation when there are no LCU entries available.

4.2.2.2.1. Queue-based locking

Queue-based locking is the ordinary lock granting mechanism. The LRT records the identity of the first requestor in the **$Head_{id}$** field, starting with a transfer counter **TC** value of 0. Any subsequent request is recorded as the queue $Tail_{id}$, and forwarded to the previous queue tail. A queue of requestors is built, where each queue node is a different LCU entry, chained with their **Next** pointers.

4.2.2.2.2. Overflowed locking

Alternatively, locks can be granted in an **overflow** mode, used to prevent starvation problems when no LCU entries are available in the requestor. In such mode, multiple readers can receive a lock grant without being added to the lock queue. In this case, the LCUs receive the lock grant but are not aware of the rest of requestors. Therefore, LCU entries receiving overflowed lock grants consider the lock to be uncontended, and remove the entry when the thread acquires the lock.

This mode is used for read requests only. In the case of write requests, a request that cannot be enqueued will use the reservation mechanism that will be explained in 4.2.3.7. With that mechanism, similar to the reservation mechanism in the Reservation Table from section 3.2, the requestor will eventually become the only requestor in the system and the lock will be granted in the ordinary, uncontended mode.

4.2.2.3. Write-lock acquisition, transfer and release

This section introduces the basic mechanism used to request, grant, acquire and release locks. The whole process is started with a thread requesting a lock **acquire** on a given virtual memory address **v_a**. The LCU resolves the physical address **a** corresponding to **v_a**, allocates an ISSUED entry for the lock and sends a lock REQUEST message to the corresponding LRT. This correspondence depends on the physical address **a**. Since the primitives are nonblocking, the processor must iterate until eventually acquiring the lock, as presented in Figure 4-5. Subsequent iterative requests from the local processor to the LCU that find the line in ISSUED state will do nothing and return a FALSE result. There are three possible cases, depending on the locking status of **a**.

a) The address **a** is not locked, so the LRT does not contain an entry for address **a**. The LRT allocates a new entry, recording the requestor's data in both **$Head_{id}$** and **$Tail_{id}$** pointers and returns a GRANT message with a **Head** flag set. This is depicted in Figure 4-6. When this message is received, the LCU switches the entry state to RCV and sets its **Head** flag. On the next iterative **acq** the lock will be taken and the **acq** instruction will return TRUE allowing the thread to proceed with the execution. The lock is now uncontended, as discussed in section 4.2.1.1, so the LCU entry is automatically removed once the lock is acquired. The LRT still records the locking data for new locking requests, not being aware of the release of the LCU entry.

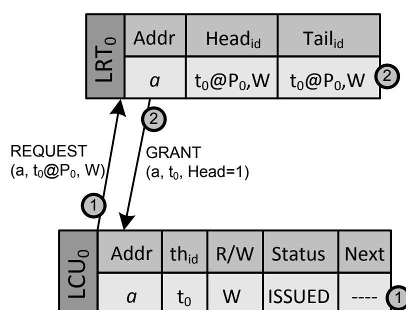


Figure 4-6: Lock acquisition when the lock is free

b) The address **a** is locked in uncontended mode. The LRT forwards the lock request, including the values in the **Head$_{id}$** tuple, to the owner LCU. With such information, the owner LCU re-allocates the evicted entry in state ACQ (acquired), records the requestor information in the **Next** tuple and sends a WAIT message to the requestor. The process is depicted in Figure 4-7, with the numbers indicating the order of events. Grayed fields represent those being updated, apart from the obvious entry re-allocation in the owner's LCU. Irrelevant fields are omitted for simplicity.

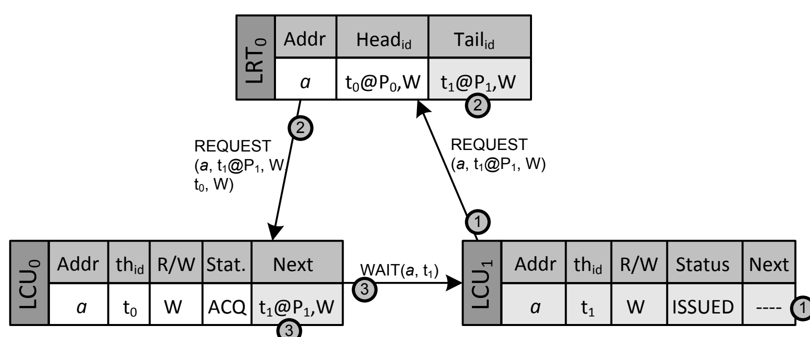


Figure 4-7: Enqueue when the taken lock is uncontended. LCU$_0$ is the current owner

c) The address is locked in contended mode with an associated queue. This case is similar to b), except that the existent tail LCU does not need to re-allocate the LCU entry.

At the end, the requestor acquires the lock if it is uncontended, or gets enqueued if it is already acquired. It can be observed that the FLT (or the previous LCU) *grants* the lock to the requesting LCU entry, while it is the local thread which *acquires* the lock with a new call to **acq**. Temporarily, the lock can be granted without being actually taken by any thread, this is, no thread can access the critical section or data protected by this lock. This is especially important for trylocks and corner cases of thread eviction, and it will be discussed in section 4.2.3.5.

Lock release is triggered by the owner thread invoking the **rel** instruction. Its behavior depends on the existence of requestors. If the lock is uncontended, the corresponding LCU entry will not be allocated when calling **rel**. However, invoking the **rel** implies that the lock has been previously acquired and it can be re-allocated; otherwise, the program would be incorrectly synchronized. To this end, the LCU re-allocates the entry with the parameters from the **rel**

instruction and sends a RELEASE message to the LRT. The LRT receives the lock release, removes its entry and sends an acknowledgement message back, allowing the LCU to remove its entry. The corner case of not having any free entry to re-allocate in the LCU is covered in section 4.2.3.6.

By contrast, if there is a queue, the lock is transferred to the **Next** requestor. An example is depicted in Figure 4-8. The receiver switches to RCV and notifies the reception to the LRT (step 2) in order to maintain the **Head$_{id}$** field valid. Finally, the LRT records the new owner and sends an acknowledgement message to the previous owner, allowing it to deallocate the entry.

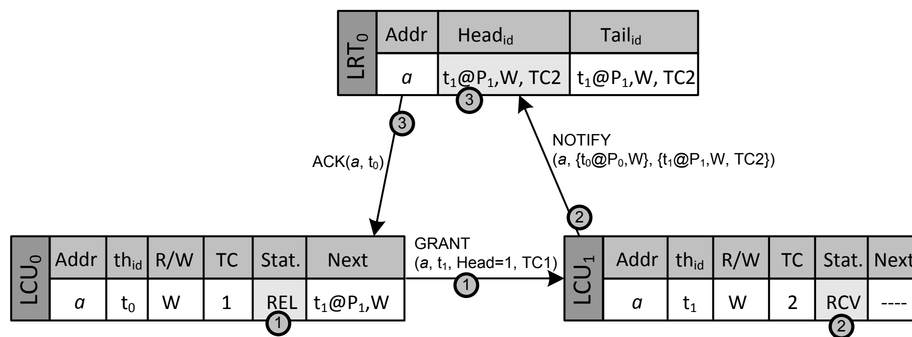


Figure 4-8: Lock transfer

The notification message is sent by the lock receiver for two reasons: First, it takes the remote message off the critical path, minimizing the transfer time. Second, it delays the deallocation of the LCU entry of the releaser until the LRT sends an acknowledgement back. Again, this ensures that the head in the LRT always points to a valid LCU entry, as it is updated when the lock has been already received. This is required to support thread migration, and will be discussed in section 4.2.3.5.

The transfer counter **TC** is used to prevent ordering problems when multiple notification messages race. Suppose three lock requestors t_1-t_3 in the queue, with the lock being transferred twice very quickly, from t_1 to t_2 and from t_2 to t_3, as depicted in Figure 4-9. The notification messages (labeled 1 and 2 in the Figure) are sent in the same order as the transfer, but they might arrive in the opposite order to the LRT (labels 3 and 4). Without an index of the number of transfers, there would be two alternatives, both of them undesired:

- Update the **Head$_{id}$** field with each notification message received.
- Send an acknowledge message from the LRT to both the lock sender and the lock receiver (ACK to the LCU_2 in response to the transfer from t_1 to t_2), and prevent any lock transfer

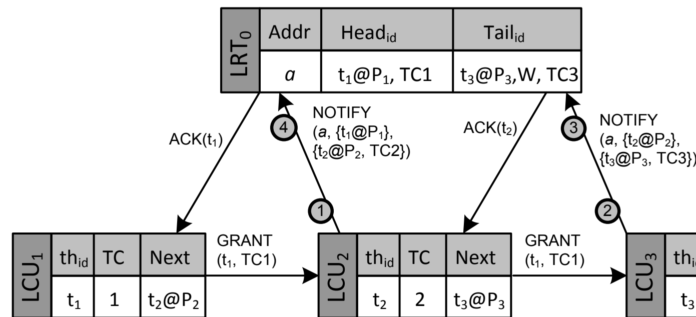


Figure 4-9: Possible data race in the notification mechanism

In the first case, a racy example like the one in Figure 4-9 would lead to an inconsistent **Head$_{id}$** value, pointing to a LCU entry that has been already deallocated. In the second case, the long latency of the remote notification to the LRT will impose a lower bound in the minimum time between consecutive lock transfers, what eventually limits the performance. Instead, the idea of the Transfer Counter **TC** is to use a count of the number of lock transfers, what allows the LRT to detect out-of-order messages. The LRT will acknowledge all transfer notifications to the lock releaser (with an ACK message), but only the messages with a transfer counter value higher than the **TC** in the **Head$_{id}$** field will update this Head pointer, leading to a consistent final value.

Finally, the LCU mechanism relies on RETRY messages being sent when other data races are detected. This occurs in other more infrequent cases. For example, consider a lock taken in uncontended mode. A release might be invoked in the local processor at the same time that a remote enqueue request is being forwarded from the LRT. In such case, the LRT receives the RELEASE message, but detects that the releaser is not the only node in the queue (the new requestor has been already recorded as the queue tail, before forwarding the enqueue message). In this case, the LRT replies with a RETRY to this RELEASE request. However, in such case the release will not be retried: When the LCU (in state REL, after being re-allocated for the release operation) receives the forwarded enqueue request, it will directly forward the lock grant to the requestor. If there was no re-allocation for the lock release, the LCU would erroneously consider that the lock is taken in uncontended mode, and enqueue the requestor. This example shows why a LCU re-allocation is required for lock releases in the case of uncontended locks, in which the LCU entry has been removed.

4.2.2.4. Reader locking

Read-locking employs the same queuing mechanism, with multiple consecutive LCUs allowed to concurrently hold the lock in read mode. This mode makes a distinction between a lock “*grant*”, which is the permission to acquire the lock, and the “*Head*” token, which identifies the single head of the queue. Receiving a GRANT message implies receiving the lock grant, while the Head token is received if the corresponding field of the GRANT message is set. While a single node is the queue head, multiple readers can receive a lock grant. When a waiting

reader LCU entry receives a lock grant, it switches its state to RCV and forwards a GRANT message to any subsequent reader in the queue. Similarly, when a read REQUEST is forwarded to the tail, a GRANT reply is sent if the tail has the lock taken in read mode. For write-locking, by contrast, the Head flag and the lock grant are equivalent.

In this shared mode, all the readers should be allowed to release the lock in any order. To prevent breaking the queue or granting the lock to a waiting writer when there are active readers, we employ the following mechanism, which relies on the **Head** flag.

When the first node in the queue releases the read-lock, the Head token is transferred to the next node and the LRT is notified, as presented before in Figure 4-8. By contrast, when an intermediate node releases its read-lock, it switches to the special state RD_REL ("read-released") without sending any message, waiting for the Head token. This prevents an entry deallocation that would break the queue. The entry is finally released when the LCU receives the Head token, which is bypassed to the next node with the corresponding notification messages to the LRT. Therefore, only Head transfers have to notify the LRT.

An example with 4 concurrent readers and a waiting writer is presented in Figure 4-10. Only the first node in the queue contains the Head flag set. Thread t_2 at LCU_2 has released the lock, but being an intermediate node, the node is preserved in state RD_REL. When the Head token is finally received in the LCU_2, the LRT is notified and the Head token is directly transferred to the next requestor, t_3. When t_3 finally releases the Head, it transfers the lock to t_4, which acquires it in write mode.

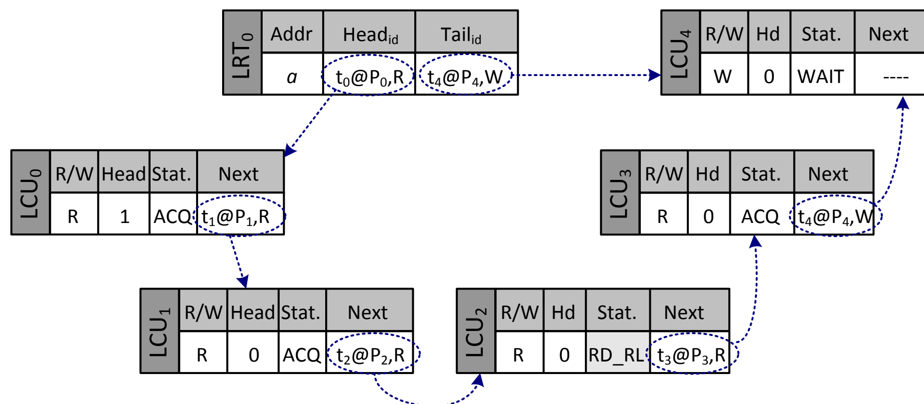


Figure 4-10: Example of concurrent read locking

While a node is in state RD_REL, the local thread can re-acquire the lock in read mode. Although this breaks the original FIFO order (and, therefore, its implementation might be optional), it does not generate starvation: the advance of the Head token along the queue of readers ensures that, eventually, any enqueued writer receives the lock. This behavior prevents contention on a single memory location (note that the notification to the LRT is out of the critical path) or complex management found on software locks [82, 107].

4.2.2.5. The Free Lock Table

As discussed in section 1.3.1.6.1, it is common to find locks that are taken by a single processor. Biasing mechanisms are designed to accelerate this case, providing a fastpath without atomics for the common owner, and penalizing other threads with an *unbias* operation prior to acquiring the lock.

From an architectural point of view, a form of biasing naturally occurs when a thread acquires the same software lock consecutively. Effectively, if no other thread requests the lock, the lock words will remain cached in the processors L1, preventing subsequent cold misses and accelerating execution. Threads running in other processors will require coherence forwarding through the directory controller, what increases their latency. For similarity, this can be considered an *implicit biasing*. Implicit biasing is inherent to software locks, provided by caching of the corresponding memory locations, and it can be also provided by hardware mechanisms that rely on the coherence protocol.

In this work, the proposed mechanism relies on sending each released lock back to the LRT. This implies that subsequent accesses from the same thread/processor will require a remote access to the same LRT. This can make the LRT mechanism actually *slower* than simple software-only locks. To prevent such performance loss, the Free Lock Table (FLT) is an optional mechanism that saves those locks that are likely to be re-acquired in a near future.

The table is made up with a set of entries as presented in Figure 4-4. Each entry contains the identification of the free lock (an address **a**, the $\mathbf{thread_{id}}$ and the **R/W** mode) and two flags used to prevent races, as explained below. When the local thread releases a lock at address **a** without pending requestors, it can allocate a FLT entry instead of sending the release message to the LRT. Subsequent lock accesses to **a** check the local FLT, find the lock free and directly allocate a LCU entry in state RCV without any remote request.

When a lock **a** is moved to the FLT_i, the LRT still considers that **a** is locked at LCU_i and will forward any request for **a** to the LCU_i. Therefore, after receiving remote requests the LCU must check its FLT, and if the lock is found, it will be directly transferred to the requestor.

It is not always safe to allocate a FLT entry when a lock is released. If the lock owner migrates, the LCU must not allocate the lock in the FLT of the new processor. If that happened, the LRT would still record the lock as taken in the initial processor, forwarding new requests to the wrong processor and leading to a possible deadlock. To prevent this, a simple strategy can be used. When the LRT grants a lock, if there is an available entry in the FLT, it is allocated with the **Prediction** flag set to 0. This prediction entry is removed when any remote enqueue request arrives, as in this case the lock is clearly not private to a single thread. At release time, only if the prediction entry remains valid the lock is moved to the FLT. In this case, moving the lock to the FLT actually consists of setting the **Prediction** flag and removing the lock from the LRT.

The LRT assigns locks to threads, recording their access mode (R/W) and the $thread_{id}$. These fields are recorded in the FLT. If a lock is re-acquired from the FLT, but one of these parameters is changed (different read/write mode, or a different thread in the same core), the FLT must notify the LRT before granting the lock. This makes use of the UPGRADE and UPGR_ACK messages shown in Table 4-3. The **Updating** flag indicates that the FLT has sent an UPGRADE message to the LRT, and is still waiting to receive the corresponding answer.

Saving locks in the FLT is only beneficial when their addresses are commonly accessed by a single thread. To determine such case, three options are proposed:

- Letting the programmer or the user enable or disable the FLT for all locks.
- Using ISA hints introduced by the programmer or compiler to determine which locks should be biased.
- Using prediction hardware, such as a two-bit saturating counter, along with the Prediction flag of the FLT.

These three options, however, have not been studied in detail, what is left for future work.

4.2.3. Detailed overview

The previous section 4.2.2 has introduced informal general view of the locking mechanism, without formally explaining the correctness of the mechanism or considering corner cases. This section will detail each of the components of the system and explain the support for corner execution cases, such as lack of resources or logical thread migration from one processor to another. Finally, sections 4.2.3.9 to 4.2.3.11 will introduce two optional mechanisms to improve the performance in different cases (read-only pathological locking and a hierarchical implementation) and a mechanism to support the case of virtual page migrating to disk. These three ideas, however, have not been implemented and are not thoroughly tested; they are left for future work.

4.2.3.1. Design invariants

A formal proof of the correctness of our mechanism is out of the scope of this document, since it would imply a detailed study of the specific implemented model. However, it is interesting to state some invariants which are always obeyed by each different lock in the proposed system:

1. At any point in time, there is at most a single LCU entry (queue node) with the **Head** flag set.
2. A node is considered to be “added to the queue” once it can be reached from the Head node, using the **$Next_{id}$** pointers.

3. Once a node is added to the queue, it never leaves it, until eventually receiving the **Head** token and transferring it.

4. The queue contains a variable amount of readers and writers in any order.

5. A single writer (the Head) or multiple readers hold the lock grant in the corresponding read mode at a given time. The Head always has the grant. If multiple enqueued readers have the lock grant, they are always consecutive, starting from the Head.

6. No readers hold the lock grant in overflow mode if the queue contains any writer.

7. No writer holds the lock grant if there is any reader in overflow mode.

8. A software thread can acquire the lock only when its local LCU entry holds the lock grant in the corresponding mode, represented by the appropriate state field.

4.2.3.2. LRT detailed overview

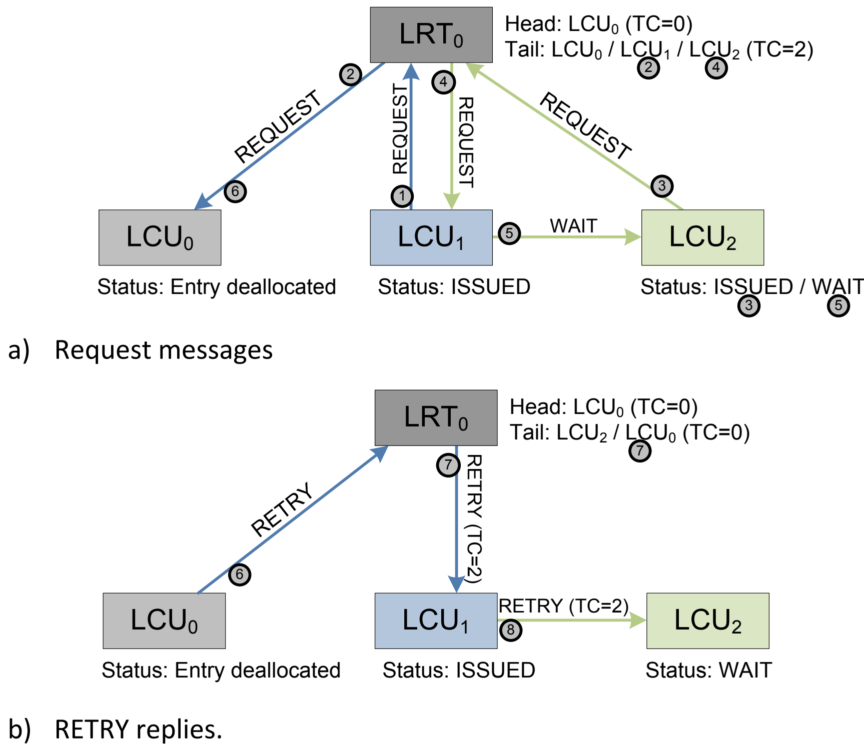
The Lock Reservation Table handles the locks in the system. The basic mechanism has been depicted in sections 4.2.1.2 and 4.2.2.3. This section will detail the enqueueing mechanism, with the races that might occur and how they are handled in a manner that minimizes the communication with the LRT. The mechanisms that handle overflow will be detailed in sections 4.2.3.7 and 4.2.3.8.

The LRT entries hold pointers for the Head and Tail of the queues. Requests for inexistent locks are handled by allocating a new entry and granting the lock. Subsequent requests for existent locks are enqueued by substituting the current $\mathbf{Tail_{id}}$ field with the requestor's information, and forwarding the request to the previous tail, in order to build the pointer chain.

The LRT detects and solves races by means of RETRY messages. Section 4.2.2.3 discussed the case of an enqueue request that overlaps an uncontended lock release. The enqueueing operation described in the previous paragraph is also prone to errors if the lock is owned in uncontended mode and the LCU entry has been deallocated. Consider the example depicted in Figure 4-11 a), which omits most fields for simplicity. The LCU_0 has acquired a lock in uncontended mode, and deallocated the corresponding entry. The numbers indicate the order in the sequence of actions. The messages have been coloured for simplicity, with messages that regard to a given LCU using the same colour as the LCU. LCU_1 and LCU_2 request the same lock (steps 1 and 3), and the LRT subsequently records them as the $\mathbf{Tail_{id}}$ (in request order, steps 2 and 4). Each of these requests is forwarded to the previously recorded tail. Note that the request from the LCU_2 is served before the forwarded REQUEST message from LCU_1 arrives at the LCU_0.

However, if LCU_0 does not have any available ordinary entry to allocate the queue head, it will have to reply with a RETRY message, as depicted in Figure 4-11 b). This message is sent to the

LRT, instead of the previous requestor. When this RETRY is received (step 6), the LRT overwrites the $\textbf{Tail}_{\textbf{id}}$ field with the sender's id, which becomes the queue tail again. The RETRY message is forwarded to the next requestor in the queue, LCU_1 (step 7), which is known because it was included in the RETRY message from step 6. The LRT notifies LCU_1 the last Transfer Count value that was wrongly added to the queue, and the LCU nodes are responsible to forward the RETRY message until the node with this **TC**, LCU_2 in the example.



a) Request messages

b) RETRY replies.

Figure 4-11: Races in the enqueue process

Note that, according to invariant 2, LCU_1 and LCU_2 are never considered part of the queue (despite LCU_2 might be in state WAIT). Then, there is no requirement for them to receive the lock grant and the Head token, as required by invariant 3.

A more complex case might occur if the forwarded REQUEST from LCU_2 (step 4) was delayed in the network, arriving at the LCU_1 later than the RETRY reply forwarded from LCU_0 (step 7). This might happen if the communication from the LRT to the LCU is unordered, or if the RETRY and REQUEST messages are sent on different virtual channels, ordered or not. In any case, the LCU_1 receives a RETRY message that should cause an entry deallocation. However, this RETRY message has to be forwarded to the next waiting node in the queue (until the Transfer Count reaches 2), but this node is not yet enqueued and its identity is not known. If the entry in LCU_1 were deallocated, the reception of the REQUEST message would make the LCU logic believe that the entry is taken in uncontended mode, and it would be incorrectly reallocated. To cope with this situation, the special LCU state "removed" (REM in Table 4-2) is used. This state means that the entry has to be deallocated, but a forwarding of a RETRY message to the next

node is still pending. Once this request arrives and the message is sent, the LCU entry is deallocated. The implementation used in the simulations in section 4.3 used different virtual channels for replies and forwarded requests, so it implemented this special state, which is also considered in the state machine in Figure 4-13.

The presented mechanism might be simplified if the LRT required an acknowledgement of the enqueue operation from the requesting LCU, before attempting a consecutive enqueue on the same lock. However, the main idea in our design was to minimize the amount of LRT accesses in the critical path. The presented case that has to deallocate nodes in state WAIT is highly infrequent, and prevents such notification.

4.2.3.3. LCU state machine

The Lock Control Unit contains the necessary logic to control the state of each handled lock. The possible states for each entry have been presented in Table 4-2. However, many of these states are transient. Transient states are temporarily used to move from one persistent state to another, for example, when a pair of send/reply messages is required. They are required to prevent possible races between the messages sent to or from the LRT or other LCUs, but they do not help to understand the basic mechanism. Similarly, cache coherency mechanisms typically use the 5 MOESI states [149] or a subset of them, but actual implementations typically require more than 15 or 20 states including transient ones.

In order to understand the behavior of the system, Figure 4-12 shows a simplified state machine of the LCU, ignoring transient states. Local requests (**Acq** and **Rel**) are bolded. When a new request is issued, the entry will acquire (ACQ) the lock if it is free, or queue in state WAIT if it is taken. Taken locks can be directly released to state Free when the owner is the queue head, or to the special state RD_REL when read locks in the middle of the queue are released, as discussed in section 4.2.2.4. From this state, they can be either re-acquired, or released, depending on the requests from the local thread and the Head token transfer. All of these states are permanent: the LCU entry can remain in them for an unlimited amount of time. Their change only responds to program advance, such as a different thread granting a just-released lock, or the local thread releasing it.

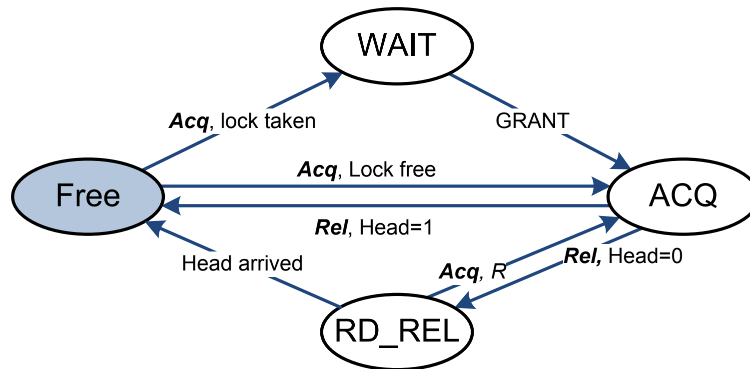


Figure 4-12: Simplified state machine of the LCU entries

By contrast, Figure 4-13 shows the complete state diagram of the LCU. White states are those in Figure 4-12, while transient states are shaded. These states are only used to move from one state to another, and the transition from one of them to another state typically depends on the reception of a reply from a remote entity. Operations that do not modify the state (such as an **Acq** issued on an entry in state WAIT) are not shown.

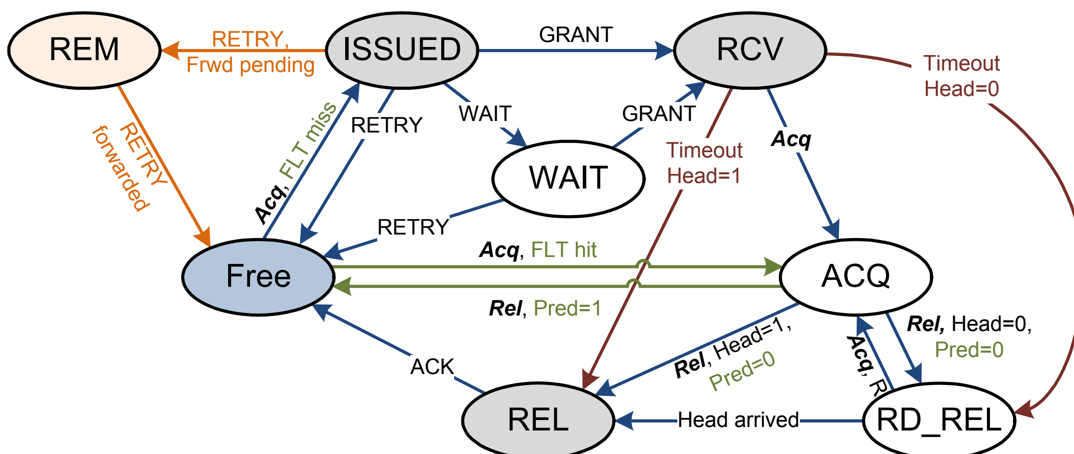


Figure 4-13: Detailed state machine of the LCU entries

Different improvements or optional elements have been highlighted with different colours. The green transitions are valid if the Free Lock table is in use, allowing for direct acquisition/release of the lock from/to the FLT. The orange-shaded state REM is used to solve the race condition discussed in section 4.2.3.2. The timeout mechanism that will be described in section 4.2.3.5 adds the transitions in red to prevent starvation. Finally, the ‘released’ state REL is required despite it unconditionally moves to ‘Free’: It prevents removing the queue head node before the LRT $\textbf{Head}_{\textbf{id}}$ pointer has been updated, as discussed in section 4.2.2.3.

4.2.3.4. Communication primitives

The system makes use of the messages presented in Table 4-3. The table also includes if the message is sent between LCU’s, from one LCU to the LRT or from the LRT to one LCU. Communication between different LRT banks is not needed, since each lock is controlled by a

single LRT. The upgrade primitives are optional, used for FLT-LRT communication, as discussed in section 4.2.1.3.

Acronym	Meaning	LCU-LCU	LCU-LRT	LRT-LCU
REQUEST	Lock request		X	X
WAIT	Wait for grant, request is enqueued	X		
GRANT	Grants the lock and/or the Head token	X		X
NOTIFY	Notifies the transfer of the Head to the LRT		X	
RELEASE	Releases the lock to the LRT		X	
ACK	Acknowledges the release or transfer of the lock			X
RETRY	Retry last operation; race detected or lack of resources	X	X	X
UPGRADE	Upgrades the saved status in the LRT (with FLT)		X	
UPGR_ACK	Acknowledges an upgrade request (with FLT)			X

Table 4-3: Communication primitives

4.2.3.5. Thread suspension and migration

Thread eviction is an issue in two cases:

i) An enqueued requestor thread is suspended or migrates, so it stops iterating on its local LCU entry (the same happens when a trylock expires). Suspension can delay the lock acquisition, reducing performance. By contrast, a trylock expiration or requestor thread migration can cause a deadlock, since the LCU entry is never updated.

ii) A lock owner migrates and the release happens in a remote LCU. If there are enqueued requestors, these never receive the lock grant, leading to a possible deadlock.

Next paragraphs discuss how the model supports both cases preventing the presented problems. It is achieved by triggering a timestamp to detect evicted threads, and by forwarding requests from migrated threads to the original queue node, which has not been updated on migration or suspension events.

In the first case, once received the lock grant, the LCU sets a timer that triggers a release if the lock is not taken within a threshold: this prevents starvation if the local thread is suspended before acquiring the lock, or prevents deadlock if a *trylock* expires or the lock requestor migrates. If a migration occurs while spinning, execution in the remote node will resume in the same **while** or **for** loop, as presented in Figure 4-5. The thread will issue a new **enq** request in the new processor, becoming the tail of the same lock queue. Then, there could be multiple entries with the same $thread_{id}$ along the lock queue. As only one will actually acquire the lock, the others will simply pass it through after the threshold without causing any issues. Figure 4-14 presents an example where thread t_2 has migrated, while waiting, from processor P_2 (LCU_2) to P_9. When the lock grant reaches the LCU_2, the timeout expires (step 3) and the lock is passed to the next requestor, t_3.

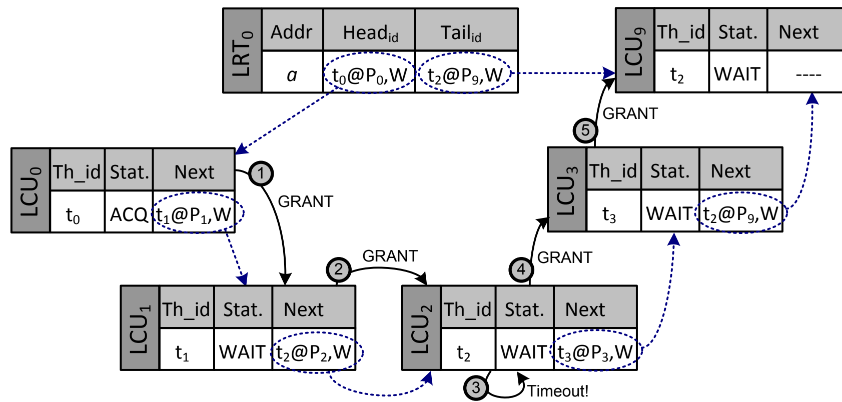


Figure 4-14: Example of migration. Thread t_2 migrates from processor 2 to 9 while waiting

The second case (lock owner migration) is detected when a lock is released from a LCU which differs with the one recorded at the **Head$_{id}$** field of the LRT. Since the remote LCU contains no allocated entry, the RELEASE message will be sent to the LRT as if it was an uncontended entry. The LRT will detect the migration by the difference between the sender and the **Head$_{id}$** field. In the case of a lock in write mode, if a queue exists, it will forward the RELEASE message to the original queue head node. This node will send the lock to the next requestor in the queue. This is the reason to maintain always a valid queue head pointer, as described in section 4.2.2.3.

If the lock is in read mode in this second case, the migrated thread that releases the lock might not be the queue head. In such case, the message is forwarded through the queue until it reaches the proper LCU. Once the appropriate LCU is found, it acknowledges the remote requestor and behaves as if a local Release had been invoked. If no LCU is found, then the lock was taken in overflow mode, and it is forwarded to the LRT. This implies that overflowed locks must always cross the queue when they are released. While this entails a significant cost, it is not very important for a twofold reason: Overflowed locking should be an uncommon case, and the traversal of the queue is out of the critical path of the thread execution, which can proceed ahead.

This mechanism also allows the case of a thread releasing a lock acquired by a different thread by just borrowing its thread id.

4.2.3.6. Types of LCU entries and forward progress

The LCU entries are the interface with the locking mechanism. The thread uses them for the local spin in both lock acquisitions and releases. Therefore, if all entries are in use, the local thread cannot issue any more lock requests and has to wait for the release of one of them. This section presents the deadlock problems that could be originated by a lack of available LCU entries, and the mechanism used to prevent them. Specifically, a new type of “nonblocking” LCU entries is proposed, which can always acquire free locks in uncontended mode. The next section will deal with the mechanisms used to guarantee that a lock is eventually found free.

The forward progress condition of the LCU entries is *blocking*, as defined in section 1.4.8.1. While the call from the thread (**acq** or **rel**) is nonblocking and commits instantaneously (with success or not), the allocated entry remains for an undetermined amount of time, until its request is served. Such time depends on the progress of the remaining threads, for example, the lock owner for an enqueued LCU entry in state WAIT. This makes the mechanism, as a whole, blocking, and introduces deadlock: although there is no dependency between the different threads in the system, the lack of available LCU entries can halt a thread.

The paradigmatic example of such deadlock would be a system with *N* LCU entries per core, with a thread requiring the concurrent acquisition of *N+1* locks. If each of the first *N* locks is also requested by a different thread, the *N* LCU entries will be occupied with the *N* queue head nodes. This prevents the last lock request due to lack of empty entries, halting the local thread and, through the lock dependencies, all of the remaining threads. While the deallocation of uncontended entries presented in section 4.2.1.1 reduces the average usage of LCU entries, it cannot prevent a deadlock case as the one described.

The solution presented in this section involves dividing the LCU entries into two different groups. One of them will be *blocking*, as described above. The other group of entries will be *nonblocking*: These nonblocking entries can be used ordinarily, except that they are not allowed to be enqueued (waiting for the lock or as the queue head). A flag in the request message notifies that requests come from one of these nonblocking entries. If such request had to be enqueued, a RETRY message is sent instead, and the LCU entry is released. The requestor thread will have to retry over and over in a loop as presented in the code in Figure 4-5. The lock will eventually become free, and it will be acquired in uncontended mode with the corresponding entry deallocated. The mechanism used to guarantee that the lock eventually becomes free is discussed in next section. These nonblocking entries are always deallocated in a finite number of steps, after a RETRY message or an uncontended acquisition, so their forward-progress condition is wait-free. Observe that this does not imply that the lock is granted in a finite number of steps, but that the operation finishes (successfully or not) in a finite number of steps, and the entry is deallocated.

A system with these two types of entries can guarantee forward progress. Blocking entries can be used for waiting on a queue and enqueueing other requests, while wait-free entries are used to acquire locks with iterative requests when no more ordinary entries are available. While these two classes of entries are enough to guarantee forward progress, to prevent dependencies between different LCUs without available entries, the mechanism will consider three types of LCU entries with different behavior:

- *Ordinary entries*: Blocking entries as considered above; they can contain lock entries in any state, and they are allowed to be part of a queue.

- *Local-request entries*: Nonblocking entries reserved for requests from the local core. These entries are not allowed to be part of a queue.
- *Remote-request entries*: Nonblocking entries reserved to serve requests that come from a different core. These can be a request for a lock that has been saved in the FLT or a forwarded RELEASE after a migration of the owner thread, as presented in section 4.2.3.5. As with local-request entries, these entries are not allowed to be part of a queue.

A system with, at least, one local-request and one remote-request entries never gets blocked due to lack of resources. Requests from the local thread should find an available ordinary LCU entry most of the times. If all of the entries are in use, a local-request entry, which cannot be enqueued, will eventually complete its operation and get released in a finite amount of time. The same applies to remote-request entries that receive forwarded RELEASE requests from remote LCUs. The mechanisms used with wait-free entries rely on remotely retrying iteratively from the requesting LCU, instead of waiting on a queue. They are described in section 4.2.3.7, along with the necessary "reservation" mechanisms to ensure that they never block.

4.2.3.7. Management of LCU overflow

The previous section has introduced the idea of nonblocking LCU entries to guarantee the acquisition of free locks. However, when the requested lock is not free, there are still two different cases that might suffer the problem of lack of free LCU entries:

1. A lock requestor does not have an ordinary LCU entry available to queue on the lock.
2. The lock owner is uncontended (it does not keep an allocated LCU entry for the lock) and it does not have an ordinary LCU entry available to enqueue remote requests.

This section deals with the mechanisms employed to prevent starvation in both cases. When the requestor has no ordinary LCU entries, it will make requests with nonblocking ones. When it is the lock owner who contains no ordinary entries, it responds with a RETRY message. The mechanism varies in each case and depends on the read or write mode of the request, but in both cases the LRT *reserves* the lock for writers (similar to the Reservation Table in section 3.2) or grants read locks in overflow mode to prevent starvation. The explanation will start with the first case, not having any available ordinary LCU entry.

If there are no ordinary LCU entries, a requestor must use a local-request entry to issue the request, as discussed in the previous section. If the lock is free, it can be granted from the LRT, either in read or write mode. Since the lock will be uncontended, the entry will be released after acquisition, so the operation does not require queuing. If the lock is taken, the behavior will depend on the type of the request.

Write requests from local-request entries cannot be enqueued nor can be granted until the lock becomes free, so a RETRY message is sent back from the LRT, forcing a release of the local-request entry. The requestor will iterate in the request code as depicted in Figure 4-5, allocating a new local-request entry, until the lock is eventually released and it is granted in uncontended mode. However, different requests from other processors might go ahead and add themselves into the existent queue. This behavior might starve the writer with no ordinary entries. To prevent it, a *reservation* mechanism is employed: When the LRT sends this RETRY message, it records the identity of the requestor (**th_{id}** and **LCU_{id}**) in the corresponding reservation fields, **R-th_{id}** and **R-LCU_{id}**, depicted in Figure 4-3. When these fields are set, the reservation mode is active and no request from a different requestor is served; a RETRY message is sent instead. This guarantees that, eventually, the lock will be released, and acquired by the reservation owner. A timestamping mechanism is also employed, to prevent deadlock in the reservation owner stops retrying, due to the requestor being deallocated or the expiration of a trylock. This is similar to the **req_clk** in Figure 3-4 of the Reservation Table, but it has been omitted in Figure 4-3 for simplicity.

Read requests from local-request entries cannot be enqueued if the lock is taken, but they can receive a grant in overflow mode if the lock is taken in read mode. If there are already overflowed readers (**Reader_cnt** > 0) and no writer reservation, the LRT responds to read requests from local-request entries by granting the lock and increasing the current count. If there are no overflowed readers yet, the LRT might need to verify if all of the queue nodes are readers, to obey invariant 6: note that it only records the read/write mode of the head and tail nodes. To perform this check, the acquisition message is forwarded to the queue tail, which will typically respond to the LRT with a RETRY since the request cannot be enqueued. This reply contains a flag indicating if the tail has received the lock grant in read mode; in this case, the overflow mode is activated (setting **Reader_cnt** = 1) and the lock is granted in overflow. Overflowed readers consider their own locks as being uncontended. When they release their lock, the overflow state is detected by the LRT because they are not the queue head, and because they do not follow the proper **TC** index.

Note that overflowed read requests can starve if there are frequent write requests accessing the lock. In such case, it will be unlikely to find the tail of the queue with the lock grant in read mode. The performed simulations have not found this case, since locks being accessed in read/write mode are easily found in read mode without writers in the queue. However, strictly speaking starvation might happen, and it might be necessary for read requests to use the reservation fields, **R-th_{id}** and **R-LCU_{id}**.

Similar mechanisms apply in the second case, when an uncontended lock owner does not have available entries to allocate a queue head node for remote requests. In such case, a RETRY message is sent, as presented in section 4.2.3.2. If the lock is in write mode, it will be

eventually released and remote requests will be served by the LRT. If it is a read lock, other requests can be served in overflow mode.

4.2.3.8. LRT overflow

Given that LRT entries are used to enqueue new requestors to existent queues, their data can never be cleared until lock release. The approach taken in this work is to use the main memory as a backup for LRT overflow.

Each LRT controller is assigned a preallocated hash-table in main memory for overflowed lines. When a request for a new entry arrives at a full LRT, a victim entry is selected and sent to the overflow table. The simulated system uses a pseudo-LRU mechanism for this victim selection. The LRT contains an *overflow* flag indicating if there are overflowed entries in main memory, a counter for these entries and a pointer to the overflow table in main memory. When a request is received with *overflow = true*, the LRT logic must check the memory table if the address is not found in the LRT. A Bloom filter can be used to prevent unnecessary accesses. When an overflowed entry is accessed again, it will be brought back from memory to the LRT, with the corresponding replacement, if required. When all overflowed entries are brought back, the *overflow* flag is cleared. Software exceptions are used in the unlikely event of having to resize the hash table.

Similar offloading mechanisms have been proposed for other hardware structures [23, 159]. The simulations performed showed that a 16-way associative LRT with 512 entries did not suffer from significant overflow problems even without a Bloom filter, and hash table resizing was never required.

4.2.3.9. Read-only locking

Section 1.3.1.6.3 introduced a pathology of read/write locking mechanisms which consists of accessing a given lock in read mode only. This pathology is present in some of the used workloads, for example, the RB-tree or the Skip-list.

This section will introduce an optional mechanism to improve the case of reader/writer locks with frequent accesses in read mode only. The base mechanism presented in previous sections would require each requestor to add itself to the queue, wait for the lock grant, access the lock and then release it, with the same mechanism when the lock is accessed again. While the Read_Released (RD_REL) state allows for faster subsequent accesses if the Head token has not been received yet, it only improves the performance in a few cases. Effectively, the Head node will be always released, and it will deallocate any subsequent nodes of the queue in state RD_REL, forcing all these requestors to wait for the enqueue process on subsequent accesses.

The idea presented here is to use a special “read-only” mode that never deallocates the LCU entries for the shared lock. In this mode, the Head token is not granted to any LCU controller:

Instead, the LRT entry will keep this token, behaving as the queue head, and pointing (via **$Head_{id}$**) to the next element in the queue. Therefore, any lock release will move its LCU entry to RD_REL and will be able to reacquire the lock in read mode without any remote request. This requires at least an additional field in the LRT, not presented in Figure 4-3, but depicted in the example case of Figure 4-15. Observe that, in this example, the first LCU node in the queue, LCU_0, is in state RD_REL, but it does not release its entry or sends any message because it does not hold the Head token, as if it was an intermediate node.

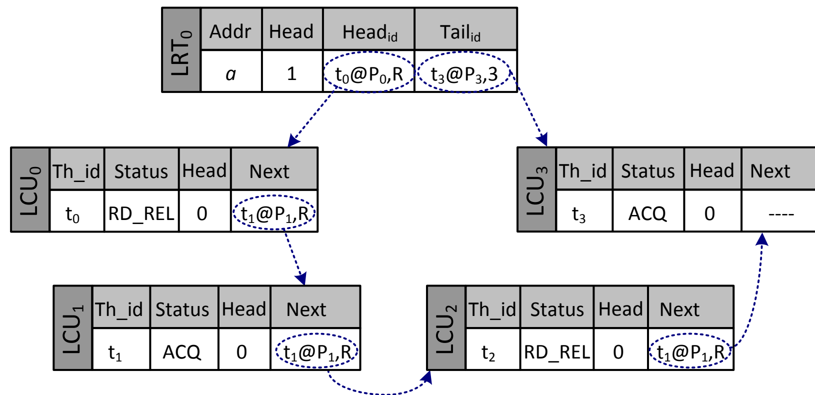


Figure 4-15: Example of read-only locking

This mechanism introduces two sources of complexity into the mechanism. Firstly, it violates invariant 3 from section 4.2.3.1, since LCU entries in this mode never receive the Head token. This restriction implies that the entry is never deallocated, and though it can be reacquired in read mode, it becomes impossible to acquire the lock in write mode. This case might be circumveyed by simply adding a new “WRITE-MODE” request message from the LCU to the LRT, sent by potential writers to the LRT if they find their LCU entry in state RD_REL. Upon receiving such message, the LRT would release the Head token (if present), what would allow the write requestors to eventually deallocate their entry and add themselves to the queue.

The second problem is related with resource utilization. Since this mode permanently allocates LCU entries for a given lock, it has to be clear that the lock is frequently accessed. The problem is twofold. On the one hand, it must be determined which locks should be moved to the “read-only” mode. A simple implementation might include a counter in the LRT, increased with every read request and cleared with write requests. Additionally, a timer might detect long periods without read requests and clear the counter. When the counter exceeds a given threshold, the read-only mode is activated. The transition from the ordinary to the read-only mode might be implemented by granting the read lock in overflow mode until the queue disappears, and then start building a new queue without the Head token passing. On the other hand, once the queue has been build, the LRT is not aware of the lock utilization, so it does not know whether the allocation of resources is still interesting. Solutions to this case might include periodic checks (with new messages) or explicit deallocation messages from the LCUs when they have lack of resources and unused entries in RD_REL.

With this special read-only mode, the proposal can be optimized for all sorts of access patterns. The FLT would optimize the case of a single lock requestor. The read-only mode would benefit the case of multiple requestors, all of them in read mode. For multiple requestors in write mode or mixed modes, the direct transfer using the queue pointers intends to minimize the wasted time in the lock transfer from one thread to another. However, the read-only mode has not been implemented in the simulation tool, what is left for future work.

4.2.3.10. Hierarchical Locking

The proposed LCU mechanism builds a queue with a FIFO policy, according to the arrival order of the requests at the LRT. However, as discussed in section 1.3.1.5, a system with different latencies can highly benefit from a hierarchical implementation. This is the case of multi-CMP systems, where intra-chip latencies are small, while inter-chip latencies can be an order of magnitude higher or more.

The current proposal is appropriate to hierarchical implementations, with some changes on the base design. An example implementation is sketched next, loosely inspired on the software hierarchical lock in [97]. Consider the simplified layout presented in Figure 4-16, with 6 chips containing 12 processing cores each. The LRT_0 should be allocated in the corresponding memory controller, possibly within chip 0 if the memory controllers are implemented inside the chip. All the elements in the Figure refer to the same lock, with the address fields and other fields omitted for simplicity. Only chip 1 is detailed, but the others would have a similar implementation. The global queue will be divided into multiple local queues or “splices”. Each chip contains a Global Enqueue Unit (GEU), with several entries to handle the lock transfer within its chip: **Local Head$_{id}$** (**LH**) and **Local Tail$_{id}$** (**LT**) pointers for its local queue splice and reception of local transfer notifications, functions previously handled by the corresponding LRT. Besides, the GEUs entries form a global queue, using the **Global-Next** (**GN**) pointer in each field. Nodes of the local queues are LCU entries that identify a thread/core pair, while nodes in the global queue are GEU entries that identify a given chip, or, in general, cluster.

The GEU handles the requests from the local cores, allocating new entries for global queues on demand. The GEUs in different clusters forward their requests to the LRT, building a global queue, coloured in red in the Figure. All the requests from a core to the local GEU build a single local queue, depicted in blue in the Figure. The transfer from one local requestor to another is direct, with the corresponding notification to the local GEU instead of the global LRT. However, local LCU entries can be marked as a “splice end” by the GEU on allocation, what has been represented in Figure 4-16 with the letter S in one of the internal cores of Chip 1. When one of these local tail entries releases the lock, it is transferred to the GEU, including the pointer to the next local queue node. In this moment, the GEU transfers the lock to the next node (chip) in the queue, recording the indicated pointer as the local head (LH). Additionally, the GEU re-allocates itself in the end of the global queue to receive the lock for the remaining requests. If there is no such Global-Next node, the lock is transferred to the (just recorded) LH node.

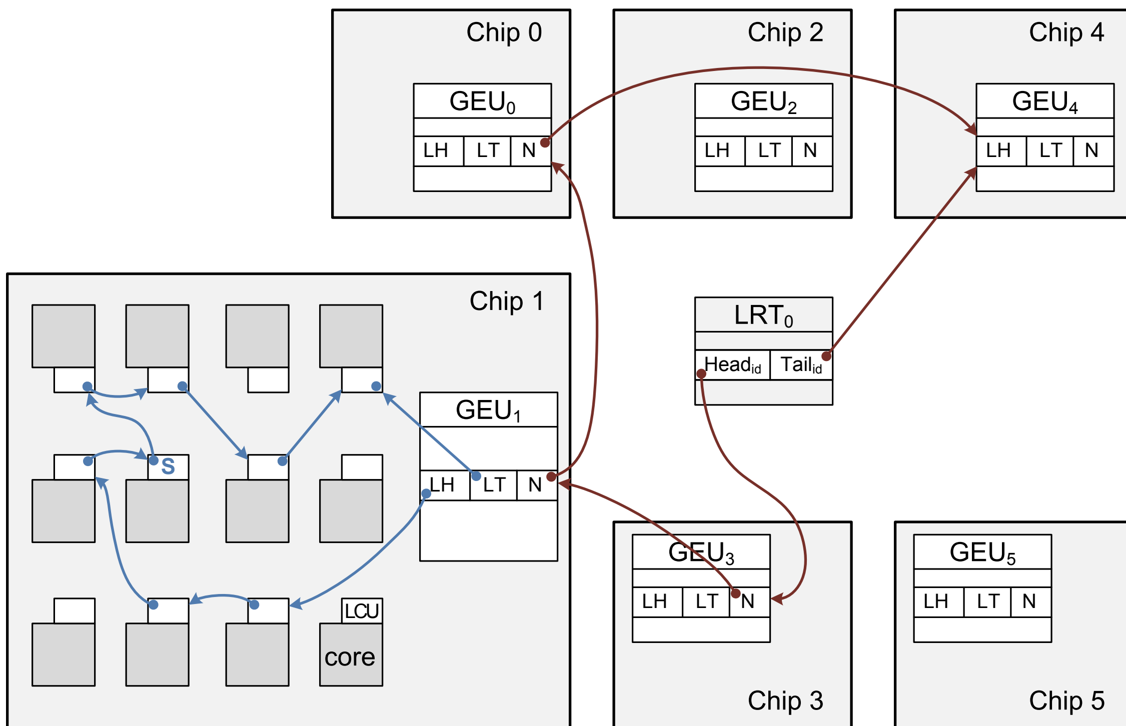


Figure 4-16: Sketch of a hierarchical implementation

A design issue would be then when to assign splice-end marks to local nodes. A simple policy could be the following: Once the lock grant is received in the GEU, the next local enqueue request for this lock is marked as “splice end”. This mechanism allows for enqueuing local requests in the same splice until the lock is received, and permits the construction of the next splice after it. This next splice will be appended to the global queue once the lock is released to the next global node, by sending the corresponding REQUEST message from the GEU to the global LRT. Observe that, with this policy, the situation presented in Figure 4-16 would be impossible: the splice-end mark should not have been assigned since the GEU has not yet become the global queue head, and thus not received the lock grant, unless the transfer notification from GEU_1 to LRT_0 is on-flight.

The ideas presented in this section are mere sketches of a possible implementation. The low-level details, such as overflow issues in the GEU have not been addressed, and the mechanism has not been implemented in the simulation tool detailed in section 4.3. However, they are presented here to show that a hierarchical implementation is feasible with the appropriate design changes.

4.2.3.11. Paging, virtualization and process faulting issues

This section will introduce the problems that can arise from sending virtual pages to disk, using multiple virtual machines or the execution in presence of process failures. The argument of this section is that, while not detailed, OS-supported mechanisms can handle those cases safely, without leaving used LCU/LRT entries behind or mixing different locks. Such support is crucial to make this model feasible as the acceleration mechanism of a real system.

The proposed system makes use of physical addresses, translated via the local TLB. This imposes a challenge when the operating system needs to de-map a virtual page from the memory map and send it to the disk. The management of the acquired locks and associated queues in this situation must consider the possible physical address change once the page is remapped from the disk. Similar issues arise when other virtualization mechanisms are used in a machine, such as the use of one hypervisor to support multiple virtual machines. Other possible conflicting task might be the resource deallocation performed by the operating system once an application crashes. How does the OS know which LCU entries to deallocate when there are multiple applications using the LCU mechanism?

The general idea to cover these cases is that a careful mechanism can deallocate all LCU entries safely, preserving the locking status of the memory addresses and without incurring in incorrect behavior. The key observation is that lock queues can be removed without correctness issues, since subsequent request will allocate new LCU entries and enqueue again. By contrast, “acquired” nodes being part of a queue can be shifted to uncontended mode (in the case of writers or a single reader) or overflowed mode (in the case of multiple readers). Both cases remove the LCU entries and possibly require some notifications to the LRT. Therefore, all of the locking status (owner thread, number of readers) can be safely recorded in the LRT using an appropriate mechanism.

A detailed disquisition follows now, depending on the state of the specific LCU entry to be removed. It will first consider transient or waiting states, and then go into those that actually hold the lock grant:

- Entries in state WAIT or ISSUED can be safely released, as long as the possible on-flight messages are considered. This can involve adding a wait threshold time after the deallocation process starts and the TLBs are flushed, to guarantee that no on-flight GRANT message will be received in LCUs in these states.
- For the same reason, entries in REM can be directly removed since their only mission is to deallocate other entries in WAIT after a data race.
- Released entries (REL) will be removed when the corresponding ACK message is received. Such operation is nonblocking, so this message is received in a fixed amount of time. Similarly, locks found in a FLT can be moved to a LCU entry in state REL, and the corresponding message be sent.
- RD_REL entries can be safely released, since they do not have the lock taken. Their only reason to exist is to preserve the queue, but it is being deallocated.
- ACQ entries can be released with the appropriate mechanism. If the LCU entry is the queue head, the deallocation implies that the owner will shift to uncontended mode and send a deallocation message through the queue. If the entry is an intermediate

node of the queue, it must be a reader. In such case, the entry can be released, with a notification message to the LRT that increases the current **reader_cnt** field.

- Finally, RCV entries have received the lock grant, but have not yet taken it. These entries can be safely released, as if the eviction timeout had expired, following the corresponding action. This moves the entry to REL, and eventually receives an ACK, or to RD_REL, where they can be safely released.

As observed, it is possible to remove all LCU and FLT entries without incurring in incorrect behavior, leaving all the necessary information in the LRT. In order to **support a paging out** operation, the LRT should be able to record the *virtual* address of an entry when its memory page is deallocated, and use some mechanism to identify the owner, such as recording the Address Space Identifier (ASI) of each entry. During a page deallocation process, the OS or the LRT removes all the associated LCU entries and saves the recorded information using the ASI and the virtual address of the lock. When the **page is brought back from the disk**, the OS process should determine if the corresponding LRTs contain any lock associated to that page, and restore the (possibly new) physical address. It is left open how to implement the mechanism to deallocate LCU entries. The simplest mechanism would be to deallocate all LCU entries in the system on paging operations, using some broadcast signal. However, this would be highly inefficient since it breaks any other existent queue, and might have to temporarily halt other lock transfer processes to obey with the required timeouts. Another alternative would be for the LRT to step on all of the concerned locks, and send deallocation messages that traverse their queues. Alternatively, the different LCUs can receive the TLB invalidations and start the deallocation process by themselves.

The most intuitive issue with paging consists of bringing a page P_1 from disk with a saved lock on virtual address $\mathbf{va_1}$, which maps into the same physical address $\mathbf{pa_1}$ as a lock $\mathbf{va_2}$ on a previously moved-to-disk page P_2. However, the deallocation process discussed in the previous paragraph would have removed all LCU entries referencing P_2, and the TLB entries should have been cleared in the paging-out process. Therefore, no LCU can accidentally access the lock in $\mathbf{va_1}$ when pretending to access $\mathbf{va_2}$. Any subsequent access to $\mathbf{va_2}$ would trap and re-allocate the evicted page in a new physical location.

Using **multiple virtual machines** that concurrently access the physical LCUs is not a problem. The virtualized system will eventually resolve all virtual addresses to physical ones, possibly using a two-level TLB or a tagged TLB with virtualization support. Therefore, the same mechanisms used to prevent problems with memory paging should protect different virtual machines from accessing each other’s locks. The concurrent use of the same $thread_{id}$ value by threads on different virtual machines should neither become a problem, since the different virtual machines would never access a lock with the same physical address. Finally, there should be no conflicts with guest operating systems trying to clean-out LCU entries, since the hypervisor is the only one who actually moves pages to/from disk.

Finally, similar issues arise when a **process faults** and terminates prematurely. In this case, the operating system should make sure that the cleaning process does not leave any used LCU or LRT entry behind. In general, the same process that deallocates the assigned memory should clean the allocated LRT and LCU entries.

All the mechanisms in this section have been sketched without a detailed implementation. Besides, they have not been implemented in the simulation infrastructure of section 4.3. These mechanisms are highly dependent on the actual system implementation, rely heavily on operating system support, and their detailed study clearly exceeds the scope of this work. However, it is important to argue that such support mechanisms *can be built*, without conditioning the proposed common-case behavior of the system. Otherwise, the proposed model would not be applicable to real-world systems. The fail to support OS operations such as thread migration, thread eviction or paging-out virtual memory is a common flaw of other hardware-accelerated synchronization systems [23, 55, 123, 159], that restrict their applicability on real commercial products.

4.3. Evaluation

The LCU proposal has been implemented in GEMS [101], as detailed in other evaluations in sections 2.4 and 3.3. In order to guarantee the goodness of the proposal, several conservative decisions and worst-case scenarios were considered. The modeled system was a multi-chip with 32 single core chips which represents an adverse scenario for reader-writer locks since large inter-chip latencies negatively affect performance, as highlighted in [34]. The processor issues up to 4 instructions/cycle thanks to the network multiplier presented in section 2.4.1. The caches use a MESI coherence protocol with 64KB L1 and 1MB L2 private caches. The interconnection model was GEMS's hierarchical switch topology (as presented in [101]) with 6 cycles/hop, leading to a base latency of 48 cycles to communicate any pair of nodes. The DRAM access latency was 80 cycles, leading to a base latency of 176 cycles for a L2 miss. This is a conservative value since larger RAM latency would translate to higher performance gains.

Each LCU contained 8 ordinary entries, plus 1 local-request and 1 remote-request entries to guarantee forward progress, as discussed in section 4.2.3.6. These values lead to a memory overhead of approximately 200 bytes/LCU, with the entry sizes depicted in Figure 4-2, plus the associated control logic. The LCU has an access latency of 3 cycles, with a blocking behavior: Once a LCU access is issued, the processor halts until it receives the answer from the LCU. Note that this latency becomes up to 12 times slower than the ordinary L1D hit time, considering that the processor can make up to 4 L1D accesses/cycle if they all hit.

The FLT, when present, contained 128 entries. There is no elaborate prediction mechanism; instead, the simple strategy presented in section 4.2.2.5 with the **Prediction** flag is employed. By default, all locks are considered private, unless a forwarded request for them is received, what clears this **Prediction** flag. This can cause many false positives, but is employed as a

simple mechanism to test the capabilities of the FLT as a "proof of concept". As argued in section 4.2.2.5, elaborate prediction mechanisms can be developed, what is left for future work.

The model implemented one LRT module per memory controller, with 512 entries, 16-way associative, and an access latency of 6 cycles. This number of entries leads to a memory requirement of less than 16KB per LRT. The overflow to main memory is modeled with the same memory delay, 80 cycles, but it is hardly ever used, since 512 entries serve most of the application's requirements.

The evaluations focused on multi-chip models. Of course, a single-chip CMP would also benefit from the LCU hardware thanks to the reduction in the lock transfer time and the reduced memory usage. However, such architecture seems to be a clear candidate for a hierarchical implementation, as sketched in section 4.2.3.10. A complete hierarchical implementation, and the comparison with a naïve LCU implementation, is specifically left as part of the future work of this thesis.

The system under study has been evaluated with a lock transfer microbenchmark, the STM benchmarks presented in section 2.4.3 and some applications from the Parsec [13] and Splash-2 [157] benchmark suites. However, section 4.3.1 will start with a discussion about the reduction in the number of messages required to transfer the lock. The following sections present the details and results of each benchmark.

4.3.1. Number of messages in the critical path of lock transfer

This section will discuss the number of communication messages (coherence or explicit locking primitives) required to acquire a lock. This number can be typically referred to two different cases: when the lock is free and it is first acquired, or when it is taken by a different thread and a lock transfer occurs. This section will focus in the second case, the lock transfer. The number of messages to transfer a lock is crucial, since it will determine the transfer latency of the lock. This value has been already presented in Table 4-1, considering a typical MESI write-back, write-invalidate protocol for software locks.

Software locks rely on the coherence mechanism to propagate updated memory values to requesting processors. The transfer of software locks typically involves two coherence operations: A memory write from the releaser (which implies a coherence invalidation of the receiver's cache, who is waiting for this update) and a subsequent read/write from the receiver, depending on the lock being queue-based or contended. Each of these coherence operations will require three messages on a typical MESI write-invalidate directory implementation. The update implies a forwarded invalidation: GETX request to the directory controller, INV forwarded to the line sharers (typically, those waiting to acquire the lock), and ACK messages from the sharers to the writer. The subsequent read will also require a forwarding operation that implies three messages: GETS request from the prospective owner

to the directory controller (GETX for contended locks), Fwd_GETS to the exclusive cache that just updated the line, and DATA transfer from the releaser to the owner. Specific protocols (such as Niagara's write-through, no-allocate on stores protocol [81]) can reduce this number to 4 messages.

Contended software locks such as TAS or TATAS rely on all the processors modifying the same memory location. While the number of messages required has been discussed above, the contention generates cache bouncing and poor performance. Queue-based software locks such as [82, 95, 106, 107] assign a queue node in shared memory to each requestor thread. Releasing a lock implies notifying the next thread, by modifying its queue node where it spins. The latency is determined again by the same number of messages discussed before, with no contention issues.

Hardware supported mechanisms can reduce the amount of messages in the critical path. Systems based on remote operations for synchronization ([89, 123, 129, 159]) only require two operations: release and acquire, or transfer and read. Each operation involves two messages, but since these operations can overlap in the best case, Table 4-1 considered a minimum of 2 messages in the critical path. Systems with direct cache-to-cache transfer, such as QOLB [55] or the direct coherence mechanism in Stanford's FLASH [86], reduce this number to 1 message. The evaluated system, with the LRT notification out of the critical path, also requires a single message.

4.3.2. Lock transfer time

A benchmark similar to the one presented in [67] and [77] was employed to evaluate the lock transfer time of the LCU. Multiple threads were set to iteratively access the same critical section, protected by a single lock. The CS is short, corresponding to a few arithmetic operations, so the lock handling time dominates the operation. The number of cycles required to run 50.000 iterations of the CS was measured, including the lock transfer time. Multiple runs were averaged, and the result was divided by the number of CS accessed to calculate the time per CS in cycles.

The LCU proposal is compared with different software and hardware lock implementations. Software locks include **TAS** and **TATAS** locks, the base **MCS** queue-based lock [106] and **MRSW** [107] which is the reader-writer version of **MCS**. The proposed model is also compared with the Synchronization State Buffer [159] hardware (**SSB**). The **SSB** accelerates the lock handling, but does not build a queue of requestors to reduce lock transfer time, and does not provide any fairness guarantees. When possible, different rates of readers and writers accessing the CS were considered, with multiple readers being able to access the CS in parallel.

Figure 4-17 presents the results of the **LCU** system compared with the SSB model. The number of threads varies from 4 to 32, without the possibility of further increasing this value as SSB requires a static assignment of threads to cores. Dashed lines indicate the results of the SSB

model, and lines with different colour represent a different proportion of readers and writers as indicated in the legend. In the mutual exclusion case (100% write locks) our model outperforms SSB on average in 30.6%. This is due to the large latencies involved in the remote SSB bank communication.

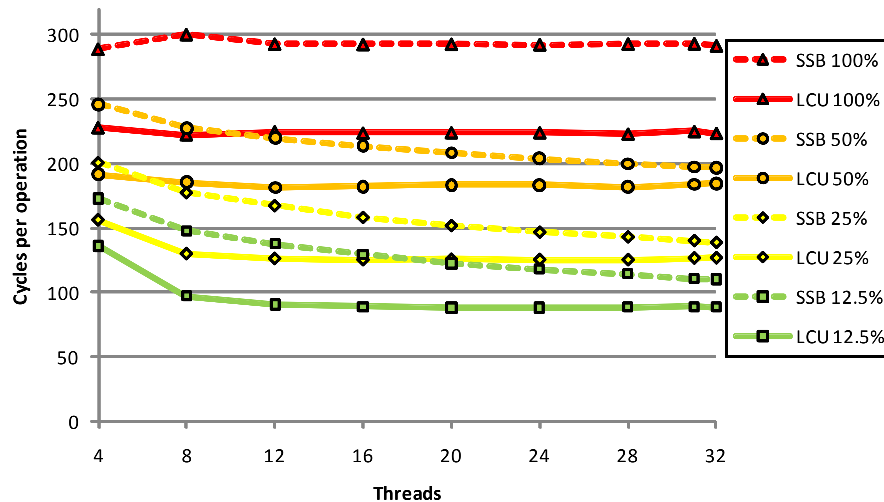


Figure 4-17: CS execution time including lock transfers. SSB vs LCU

The inter-processor latencies are assumed to be very small in the SSB original single-chip model, but in the modeled multi-chip system the direct lock transfer makes a significant improvement over **SSB**. Increasing contention does not affect the lock transfer time in either case, as the performance line remains flat with the thread count. When reader-writer locks are considered, the average critical section access time decreases in both models, as multiple readers access concurrently. In the **SSB** model, we observe that the performance increases with the thread count. This is a side effect of not providing fairness in **SSB**: when the lock is in read mode, readers are allowed to take the lock, independently of the number of pending writers. Therefore, the number of successful read accesses increases, at the cost of starving writers.

Figure 4-18 shows the comparison between software locks and the LCU model. Lines with different colours represent different proportions of readers and writers (**LCU**, **MRSW**), and write-only implementations (**TAS**, **TATAS**, **MCS**) are presented in black.

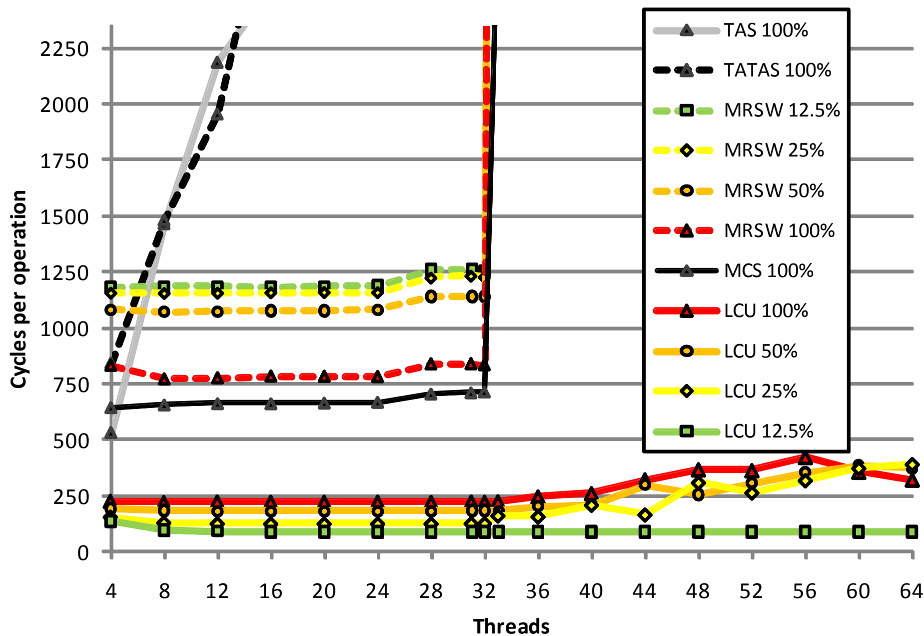


Figure 4-18: CS execution time including lock transfers. LRT vs software locks

The LCU model behaves smoothly with more threads than processors, due to the integrated starvation detection mechanism presented in section 4.2.3.5. Contended locks (**TAS** and **TATAS**) suffer from strong congestion as the number of threads increases. They can behave correctly in the case of having more threads than physical processors; however, in such case the contention is already so high that the lines are out of the limits of the figure. Queue-based locks provide a constant performance up to 32 threads but after that, the described starvation problem dramatically increases the lock transfer time. Compared with **MCS** write-locks, the LCU outperforms the software proposals in more than 2×. This is because, even with the software queue, the coherence operations involved in the lock transfer require a remote coherence invalidation, followed by a subsequent request. The reader-writer queue, **MRSW**, even in the 100% writers case provides worse performance than the write-only **MCS**, due to the increased number of required operations. Moreover, as the readers rate increases, the average time per operation increases too. This is due to coherence congestion in the reader counter contained in the lock, which has to be modified atomically twice per reader (incremented and decremented). Due to this effect, the LCU obtains an average speedup of 9.14× for the case of 75% of the requests in read mode (LCU 25% in the legend). Queue-based locks with timeouts [67] have not been evaluated in the Figure. While they behave nicely when more threads than processors are present, their base latency is higher than **MCS** and reader/writer variants are not known.

Next sections will study how these gains in lock transfer latency translate into whole-program performance. The lock handling typically constitutes a relatively small fraction of the overall application execution time. Therefore, the performance increase will be restricted to those

synchronization overheads. However, we will show how scenarios with fine-grain locking or reader-writer synchronization observe significant performance gains.

4.3.3. Fine-grain locking: STM benchmarks

The STM system presented in section 2.2 makes an intensive use of fine-grain reader-writer locks. It has been ported to use the LCU mechanism in order to study the performance improvement. This section compares the performance of the base system (labeled as **sw-only**) against the same model using LCU RW-locks (labeled **LCU**), with or without the FLT hardware. As a reference, the performance of Fraser's non-blocking OSTM introduced in section 1.4.12.3 (labeled **Fraser**) is also presented. Fraser's OSTM does not require read-locking of the read set during the commit phase. Due to the use of invisible readers, this system fails in the privatization problem, and therefore its performance cannot be directly compared with the other systems based on RW-locks that solve such a problem. Therefore, it is not presented as an 'optimal implementation' reference, but an example of a different system that does not provide the same correctness guarantees. The three benchmarks introduced in section 2.4.3 were used.

The plot in Figure 4-19 studies the scalability of the system. It shows the transaction execution time in the RB-tree benchmark for the different models. The RB-tree has 2^8 maximum nodes and 75% read-only transactions, and the number of threads varies. The single-thread case implements the complete runtime, including lock acquisitions and data validation. With a single thread, the LCU increases the performance of the base sw-only system in a 10.8%. The use of the FLT is beneficial, as it prevents remote accesses when the only thread in the system repeatedly accesses the same location, leading to a speedup of 30.3%. Fraser's nonblocking model outperforms the lock-based one, as its commit phase is much shorter, given the lack of

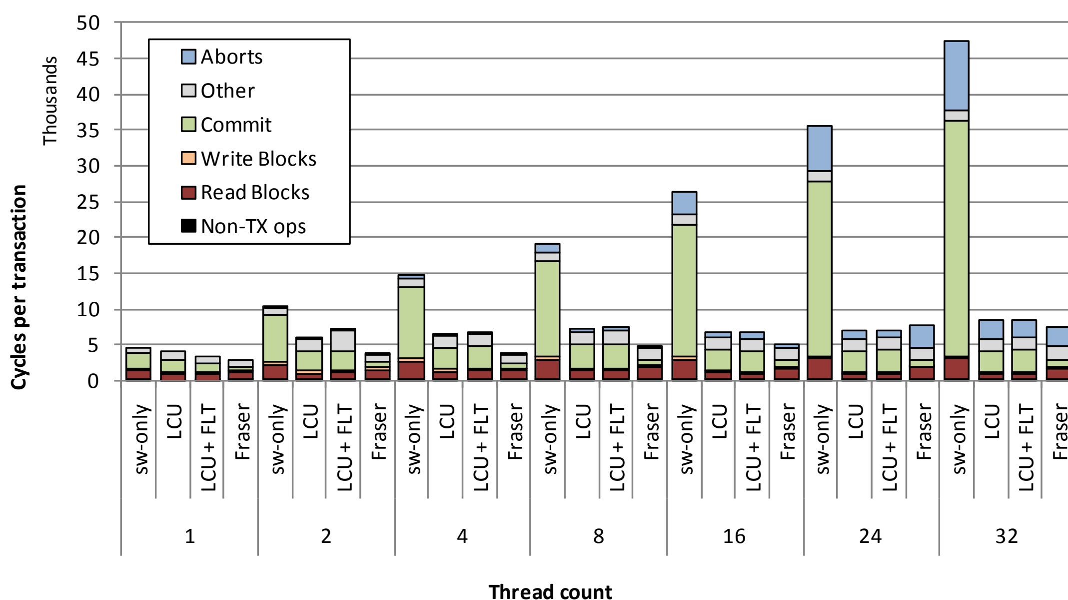


Figure 4-19: Transaction cycle dissection of the RB benchmark with 2^8 max. nodes

reader locking.

As the number of threads increases, the base **sw-only** model gets worse, due to the increase of the commit phase in which locks are acquired. The shared structure presents reader congestion in the root, which affects the overall performance. The LCU model scales almost linearly, maintaining a similar execution time per transaction. The FLT, by contrast, becomes useless with multiple cases, imposing an overhead in some others. Effectively, this benchmark is completely random, and the FLT only introduces overhead in the execution, leading to possibly unnecessary request indirections. Interestingly, for high thread counts, the LCU performance is similar to that of the nonblocking Fraser's system due to the effect of transaction aborts.

Figure 4-20 shows the performance of the previous models with larger problems (2^{15} or 2^{19} maximum number of nodes in the shared structure) considering the three benchmarks and 16 threads. The arrows indicate the speedup obtained from the base sw-only model by using the LCU. The RB benchmark maintains a similar pattern to the one observed in Figure 4-19 . Due to the reader congestion removal, the overall application speedup of the LCU model is 2.97× and 2.41× for sizes 2^{15} and 2^{19} respectively. The skip-list presents the same problem of congestion in the root node, with speedups of 2.37× and 1.53× with the LCU model for the different sizes. The hash-table does not have a single point-of-entry with reader-locking congestion, and its transactions are much shorter. Still there is a 42% speedup thanks to the LCU use.

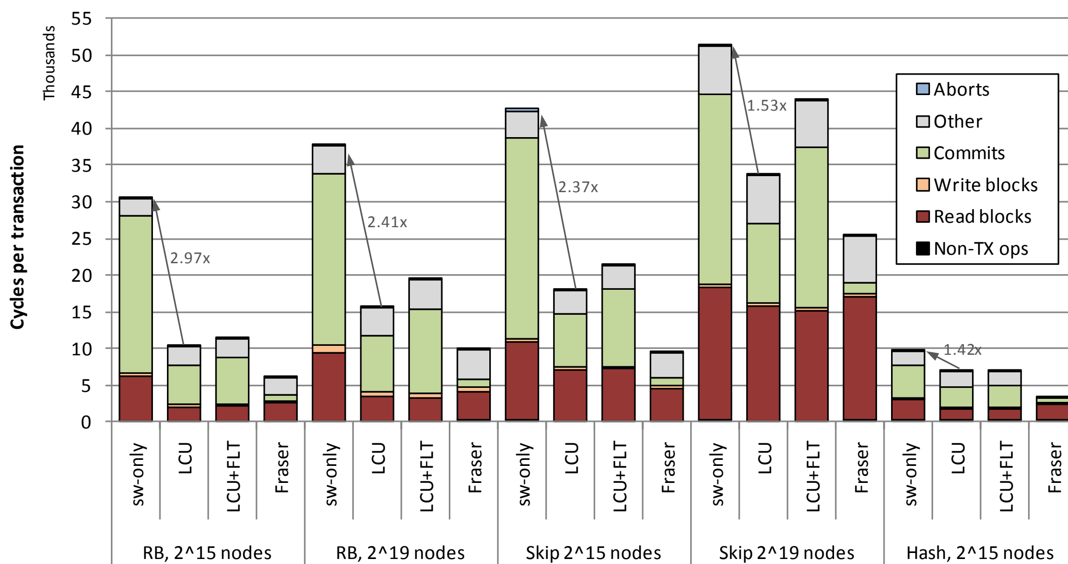


Figure 4-20: Transaction execution time, 16 threads and 75% of read-only transactions

Regarding the remaining columns, it is clear that the FLT negatively affects performance in these benchmarks. The penalty that could be hardly distinguished in Figure 4-19 is clear now with a larger problem size. Recall that, with the lack of a prediction mechanism, all locks are considered 'private' by default, while in fact they are all completely shared. The incorrect

saving of locks in the FLT causes additional request indirections and interconnection traffic, what leads to reduced performance. Additionally, it increases the usage of LRT entries, what can lead to overflow problems. This makes clear the need for such a prediction mechanism, or at least, the capability to disable the FLT. Finally, it is observed that the lack of reader-locking in Fraser's OSTM, especially in long transactions. This result is consistent, since larger workloads require a much higher rate of read accesses, which are prevented by Fraser's model.

4.3.4. Traditional parallel benchmarks

This section evaluates the performance of the LCU with some traditional lock-based programs: The fluid simulation program Fluidanimate from the Parsec benchmark [13] which does an interactive animation; the Cholesky matrix factorization kernel from Splash-2 [157] and the Radiosity ray-tracing application, also from Splash-2. These applications are selected for two reasons: they are lock intensive and present different behavior with respect to the locking pattern. Several runs of each application with 32 (Fluidanimate) or 16 (Cholesky, Radiosity) threads were run and averaged with each of these four systems: the original Posix mutexes in the modeled Solaris system, the **LCU** with or without the **FLT** and the **SSB**. Figure 4-21 shows the execution time of the parallel sections of these applications with different locking mechanisms. The plot shows the resulting values, including the estimated error for a 95% confidence interval, normalized to the Posix result. Note that these plots show the overall application execution time, not only the lock-related code.

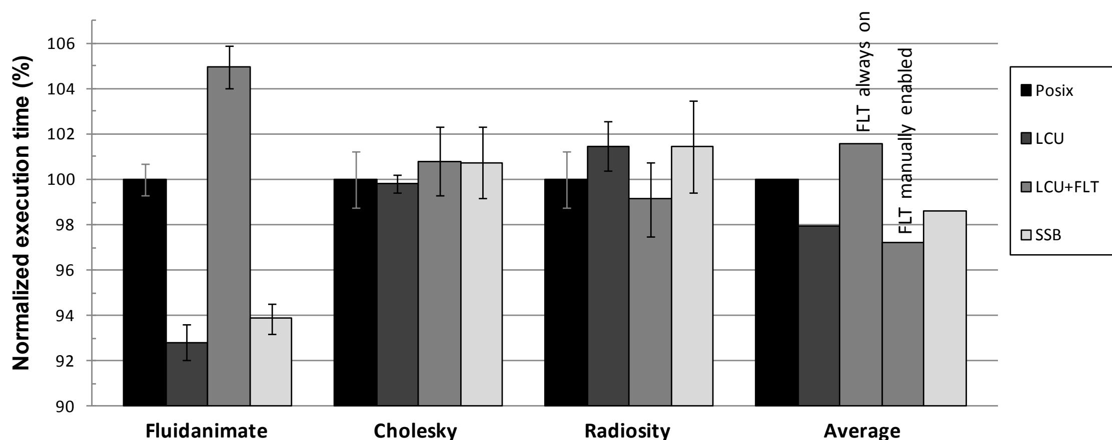


Figure 4-21: Application execution time

Fluidanimate divides the fluid to be simulated into "cells", protecting individual cells with locks. The application performance is limited by the parallelization overhead [13], what prevents finer locking. However, the LCU presents a much lower overhead, what allows for such finer locking. While the original application uses a lock per each modeled cell, the LCU version protects each individual value being updated within a cell with a dynamic lock. This comparison is fair, since a lower overhead is a key contribution of the **LCU** model; comparing

the same code without exploiting the fine-grain capabilities of the **LCU** would underutilize its capabilities. With these concerns, Fluidanimate achieves a speedup of 7.7% over the base model when the **LCU** is used. The direct lock transfer also allows the **LCU** to improve over the SSB performance, which only has a speedup of 6.5% from the base version. By contrast, the inclusion of the **FLT** highly penalizes performance, since all locks in this application are highly shared. Particularly, the version with the **FLT** performs a 4.9% worse than the base, software-only version.

Radiosity, by contrast, shows the opposite pattern. Most lock accesses in the application are to the per-thread private task queues. Only when a thread finishes his work, it accesses remote queues to steal work to be done, in a form of load balancing. In this case the **LCU** without the FLT provides a worse performance than the base model. This expected behavior comes from the absence of FLT that makes all LCU lock accesses to be remote, while the original software version can keep the frequently accessed lock line in the L1 cache. The **SSB** model suffers from the same problem. By contrast, the **FLT** regains the implicit privatization lost with the hardware mechanism, and improves the base model in an average 0.88%.

Finally, Cholesky is a matrix application which does not seem to be almost affected by the lock model, as all the results are within the estimated error range of the base software model. Notwithstanding, it can be observed that the introduction of the **LCU** does not harm performance, despite the conservative values used for the **LCU** simulation, as described at the beginning of section 4.3.

The geometric mean shows that the LCU model can provide an average speedup of 2.1% in these applications that do not make an intensive use of fine-grain locking, despite the bad result in Radiosity. The introduction of the FLT penalizes the average case; however, using the FLT only in those applications with lock locality (Radiosity) leads to an average speedup of 2.9%.

4.4. Summary

This Chapter has proposed the Lock Control Unit (LCU), a distributed hardware mechanism that implements efficient and flexible fair reader/writer locks. The mechanism builds a hardware queue to implement direct core-to-core transfers without additional management messages in the critical path, what minimizes the lock transfer time. LCU entries are dynamically allocated for requested locks and deallocated for uncontended locks, what allows for a low resource usage. Different mechanisms are designed to efficiently support requestor aborts, thread suspension and migration, paging to disk, virtualization, and resource overflow, what makes this proposal viable in general-purpose systems. Additionally, optional mechanisms have been proposed to support lock pathologies, such as single-thread locking, read-only locking or hierarchical accesses to the lock.

The Lock Control Unit has been implemented in a full-system simulator and evaluated with different workloads. Lock transfer microbenchmarks show that our proposal improves the transfer latency in more than a 30% from previous hardware models, and in more than 2× from the best software implementation. This transfer time behaves smoothly when the number of threads exceeds the available execution units. The base lock-based STM presented in Chapter 2 is used as a fine-grain locking benchmark, with performance improvements of almost 3× when using mid-size data structures and 16 threads. Finally, ordinary lock-based applications are evaluated, with an average performance improvement of a 2.9% and maximum improvement of 7.7% thanks to the fine-grain locking capabilities and low overhead of the mechanism.

Chapter 5. Implicit transactional memory

Chapter 5

Implicit transactional memory

Previous Chapters have focused on programmability issues of parallel systems. This Chapter will consider how to exploit a checkpoint-based processor microarchitecture, such as the Kilo-Instruction Processor introduced in section 1.5.3, in a parallel system. Specifically, the contribution of this Chapter is twofold. First, it is explored how the checkpoint-based architecture can be leveraged to support Hardware Transactional Memory and a Sequential Consistency model, both with a native speculation capability that increases the performance. Second, it is studied how this architecture can support locking mechanisms thanks to silent store eviction, and specifically it is discussed how to integrate the Lock Control Unit hardware presented in Chapter 4.

5.1. Atomic sections and processor checkpointing

The performance capabilities of multiprocessors based on processors with large-scale instruction windows has been studied in [49]. Using the SPLASH-2 benchmarks, it was observed that an instruction window of 1024 entries could increase the performance of a parallel application in a factor of 1.13× to 2.04×, from a base system with 64 entries. However, that work did not take into account the internal architecture of the processor, using an upsized model of the base OOO architecture instead.

Given the multi-checkpoint mechanisms in the Kilo-Instruction Processors presented in section 1.5.3.1, this section will focus on how they naturally support a transactional behavior. Specifically, a direct implementation causes the group of instructions between two checkpoints to appear to the rest of the system as a single memory transaction. In particular, memory updates (the store instructions) associated with a checkpoint are managed as a single group. They are globally and atomically validated when the corresponding checkpoint commits. None of them can be made globally visible due to possible misspredictions before the checkpoint commits, and all of them are required to be validated when the commit occurs. Similarly, when coherence invalidations are received, the addresses are checked against the local Load/Store Queue (LSQ) as discussed in section 1.5.3.4. The addresses that are detected as a conflict force a checkpoint rollback, equivalent to a transaction abort. Therefore, a simple

implementation can provide isolation between the groups of instructions in different checkpoints.

These groups of operations belonging to the same architectural checkpoint will be defined as *implicit transactions*. They are called transactions because they preserve the key aspects of atomicity and isolation present in transactional memory, but they are said to be atomic because they are automatically and transparently delimited by the hardware. Every instruction belongs to some implicit transaction, without the needing to know that the system provides such transactional behavior. Therefore, both the programming model and the instruction set are unaffected, and legacy single or multithreaded binaries can be directly executed. It should be made clear that this idea of implicit transactions differs from the concept of transaction defined previously for TM systems, where transactions are programming constructs. In this case, the *implicit transaction* is a micro-architectural concept that is not visible to the programmer. The term checkpoint and the term implicit transaction (or just transaction) will be used interchangeably henceforth, now that the close relationship between a checkpoint and an implicit transaction has been established. Further discussion on implementation issues can be found later on section 5.1.1. Section 5.1.2 discusses the performance aspects of the implicit transaction length.

The potential of implicit transactions is multiple. First, a multiprocessor based on checkpointed processors with implicit transactions can naturally support a bounded HTM mechanism. Such mechanisms have been already considered in previous sections, including the construction of a Hybrid TM system based on a bounded HTM. Second, implicit transactions can be used to simplify the consistency model and provide Sequential Consistency, allowing higher degrees of speculation since a large instruction window is handled speculatively. Finally, implicit transactions can be naturally used as a speculation mechanism on critical sections and past barriers.

5.1.1. Implementation details of implicit transactions

This section will discuss implementation details of a mechanism based on implicit transactions. An example of the operation is presented in Figure 5-1. The instruction flow of four processors P_1 to P_4 is presented, including the different checkpoints that are taken during execution. In-flight instructions are shaded, while committed ones are black. Some processors have checkpoints with all the corresponding instructions committed, but the checkpoint itself cannot commit because it is not the oldest checkpoint in the processor (Chk_{13}, Chk_{22}). The oldest checkpoint in P_3, however, is finished and can commit. The corresponding invalidation coherence messages (or memory updates) are sent to other processors, which detect a conflict with the read values in their LSQ and rollback their checkpoints. In the example of Figure 5-1, P_2 would abort checkpoints Chk_{23} and Chk_{24}, while P4 would abort checkpoints Chk_{42}, Chk_{43} and Chk_{44}.

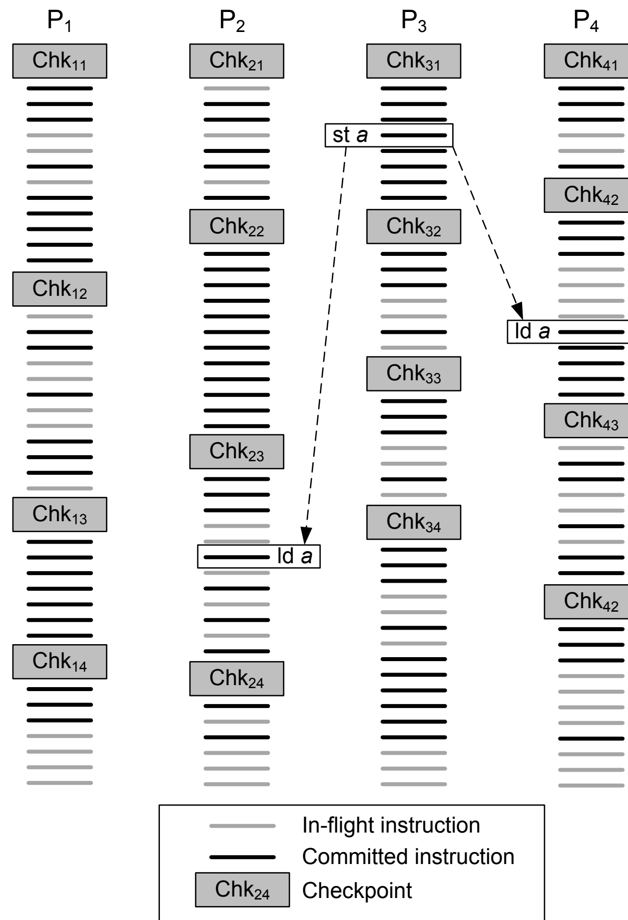


Figure 5-1: Execution flow example with 4 processors

It still remains the issue of how to validate the checkpoint results and detect the conflicts. A small bus-based system might simply broadcast the memory updates on the bus when the implicit transaction commits. This should occur with all the memory lines which are not in an exclusive state in the cache (M or E in the MOESI protocol). The update should be performed in a single step; this is, without releasing the control of the bus to prevent remote updates that might conflict with the committing transaction. This updates the main memory and the remote caches, which use a snooping mechanism to detect the conflicts. The update mechanism is similar to the transactional proposal in TCC [59]; however, in that case all of the code is composed of transactions of a certain size, given by the programmer or the compiler. The advantage in the implicit transactional system is that it is not required to support a certain size of transactions, and when an implicit transaction exceeds the local resources, it can be simply aborted and retried with a smaller size. Alternative designs [19] propose the use of compression in the broadcast phase to accelerate the notification and reduce the bus usage. In this simple system, each transaction can be linearized in any moment during the atomic bus broadcast process.

A distributed system with a directory should make use of specific coherence invalidations, or another mechanism that guarantees the atomicity of the commit process and prevents races

between concurrent committing transactions. Multiple proposals can be considered, similar to those used in HTM systems. In any case, two possible designs can be considered at first:

a) Each store committed during the checkpoint execution makes an access to the local cache to determine if it contains the block with an appropriate exclusive state.

b) Once a checkpoint is ready to commit, all of the required coherence requests and invalidations are sent. When all of the required invalidations are received, the checkpoint is flushed and the LSQ content are sent to the local cache.

The first approach searches for the highest performance by sending the coherence requests and invalidations as early as possible. However, it can increase the rate of unnecessary aborts if the running checkpoint has to abort. The second case only sends update messages when a transaction is about to commit, so it should give the lowest rate of false negatives.

In any of these cases, a mechanism to guarantee the atomicity of the checkpoint validation is required. Multiple approaches can be considered, based on previous works that implement HTM mechanisms. Some alternatives can be:

a) Use a per-directory controller, per processor update mark, so when an implicit transaction is about to commit, it requests the permission to send updates to all of the required directory controllers. The specific directory controllers depend on the memory locations modified during the checkpoint. A similar approach is proposed in [20] to scale the lazy-update TCC model to a directory-based implementation.

b) Send ordinary coherence requests to the directory, which forwards them to the current owners. Such approach might need of **NACK** replies and transaction aborts as in [110] to prevent deadlock issues in the commit phase.

Any of these two alternatives requires the use of a timestamping mechanism to solve deadlocks, either centralized (for example, using a timestamp 'vendor' hardware unit that assigns consecutive numbers to each requestor) or distributed (for example, using the processor local clock and proc_id to build a unique clock value, as in [110]). In general, considering that the implicit transaction length will be shorter than the average explicit transaction length, the commits will occur more often, and the second referred mechanisms would be preferred to minimize the overhead. Even more, in order to minimize the restrictions of the algorithm, a lazy timestamp request would be preferred (to prevent acquiring a timestamp if the implicit transaction does not require it), and a 'range of valid timestamp values' could be considered depending on the received updates, instead of a fixed per-transaction timestamp.

5.1.2. Implicit transaction length and performance

Single-processor checkpoint-based architectures use the checkpoints to simplify the implementation in case of misspredictions, exceptions or interrupts. In these infrequent cases, the processor status is reverted to that of the saved checkpoint. Therefore, checkpoints are a safety mechanism that allows for correct execution. If there were no misspredictions, exceptions or interrupts, the checkpoint mechanism might become almost unnecessary: the rollback operations would not be required except for a few corner cases, such a memory disambiguation issues in the load/store queue. In general, since these cases are uncommon, long checkpoints are preferred to allow for a larger number of in-flight instructions.

However, a parallel architecture introduces concerns about the coherence mechanism. Coherence changes (updates, invalidations) received in a processor must be checked against its local load/store queue to prevent conflicts, as discussed earlier in section 5.1. Conflicting coherence updates would abort the whole checkpoint and the subsequent ones. Therefore, the longer an implicit transaction, the higher the amount of useful work wasted due to the rollback.

The plots in Figure 5-2 estimate the proportion of rollbacks in different Splash-2 benchmarks, depending on the number of processors and a fixed checkpoint size expressed in dynamic instruction. The applications were run on Simics, and on each group of instructions executed by each processor, it was measured how many remote implicit transactions would have to rollback due to a coherence conflict. Although transaction retries were not taken into account, this provides a good estimation of the rate of transactions that would abort due to coherence concerns. As expected, in general the proportion of aborts grows with the number of processors and the length of the transaction. However, with reasonable checkpoint lengths (128 to 512 dynamic instructions, as used in other works on checkpoint-based architectures) the rate of expected aborts is low. Checkpoints with 256 instructions do not exceed a 1% of estimated transaction aborts with any processor count in Barnes, Cholesky or Radiosity, and it is well below the 10% (observe the logarithmic vertical axis) in FFT, Ocean and Radix. Therefore, it is reasonable to consider that the effect of coherence invalidations does not invalidate a checkpoint-based implementation.

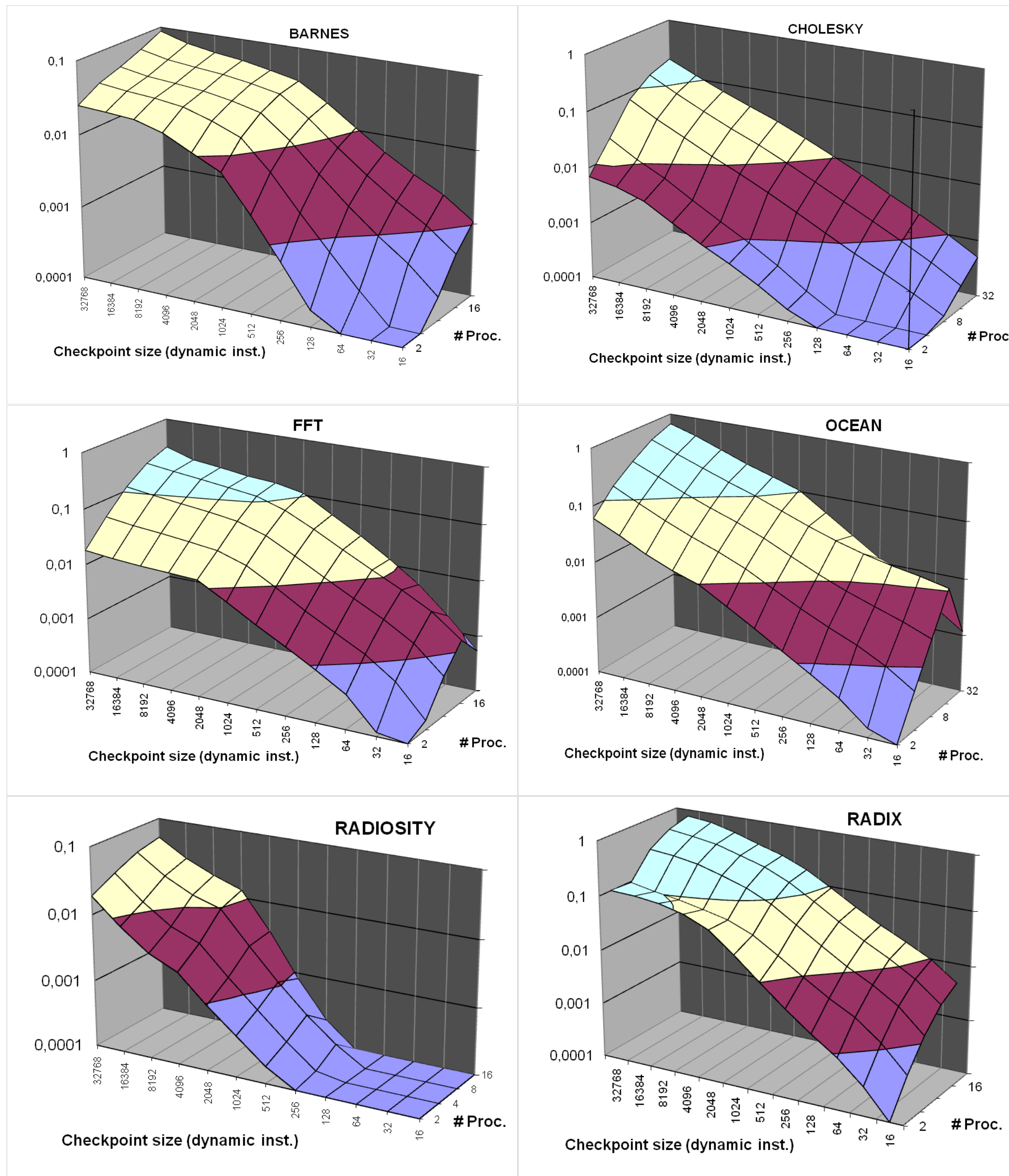


Figure 5-2: Estimation of the proportion of rollbacks in SPLASH

Of course, when an implicit transaction is invalidated, all the subsequent ones are invalidated too, as occurred in the example from Figure 5-1 with checkpoints Chk_{24}, Chk_{43} and Chk_{44}. Therefore, it might be considered that, the larger the instruction window size, the higher the penalty caused by a coherence invalidation. While this is initially true, the invalidated implicit transactions have already issued part of their corresponding memory requests. This means that the invalidated code has effectively performed a prefetch of most of the upcoming memory requests, what accelerates the subsequent reexecution, similar to a runahead mechanism [112].

As a conclusion, the checkpointing mechanism should consider the coherence invalidations for its policy of taking new checkpoints. A variable checkpoint size might be implemented, depending on all the conditioning factors:

- The rate of coherence conflicts that cause an abort, increasing the checkpoint length if the rate is low (program sections with low amount of access to shared data) or decreasing if it is high.
- Long-latency misses, as in the original single-processor approach.
- Other factors, such as exceptions or overflow in the processor resources.

5.2. Sequential consistency with implicit transactions

A basic implementation of Sequential Consistency requires a processor to delay each memory access until the previous one is completed, what is simple but clearly leads to a low performance. However, there are proposals that preserve the SC model without compromising performance much. Specifically, the most related to this work are:

- SC++ [52] makes use of hardware speculation for both load and store operations, and preserves SC by rolling back when a consistency violation is found. In this way, the system can rely on reordering and overlapping memory operations with a performance similar to that achieved with the RC model.
- TCC [59] solves the problem of the consistency by proposing a parallel model based on software delimited transactions, with a sequential ordering between them. Therefore, this model, which takes a software approach to the consistency problem, requires a new programming model to be used.

The implicit transactions idea, based on the previously described behavior, provides SC support in a natural manner: instead of assuring that single memory operations from a processor to be globally performed in order, it requires full transactions to be in order. Fortunately, requiring an order for transactions inside a single processor is straightforward, because the checkpointing mechanism always commits the oldest transaction first.

Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and with the accesses among different processors (arbitrarily) interleaved. In other words, there can be different global orders with different interleaving of memory operations from the different processors, but each such interleaving must maintain all the individual program orderings. Figure 5-3 gives an example of a sequentially consistent global ordering of memory operations from two different processors, labeled [A1, A2, A3] for processor A, and [B1, B2, B3] for processor B. The third column shows a global order that respects the program orders from processors A and B.

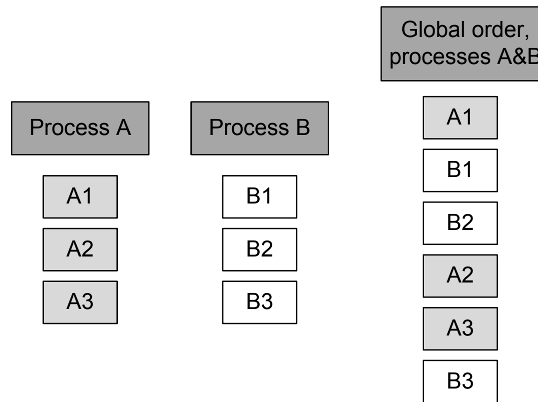


Figure 5-3: Sequentially consistent reordering of memory operations from 2 processors

Note that, for this global order to exist, it is required that each processor observes the same individual order on the memory updates of the remote processors, in the shared locations that are accessed by both processors. Specifically, the problem presented in Figure 1-1 in page 7 only fails when the processor P_2 observes the memory updates from P_1 in the wrong order. This might occur, for example, due to data races in the interconnection hierarchy, and the resulting system would not support a Sequential Consistent memory model.

In the proposed system with implicit transactions, memory accesses from each processor are grouped into implicit transactions by taking checkpoints. It is simple to extend the definition of sequential consistency to such transactions, and require only transactions from each processor to be in order. The resulting global order will be an arbitrarily interleaved succession of transactions that will also meet the basic definition of SC since it corresponds with one of the possible sequentially consistent global orderings. Figure 5-4 presents an example where the instructions are grouped into transactions, labeled [TR_A1, TR_A2] for processor A, and [TR_B1, TR_B2] for processor B. The third column shows that respecting the program order for those transactions, will also respect program order for memory operations.

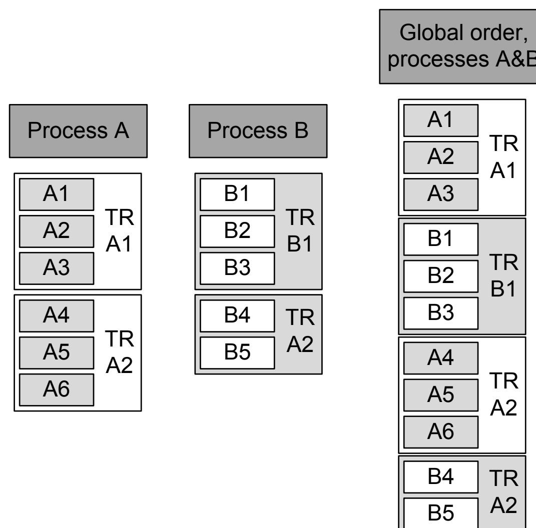


Figure 5-4: Sequentially consistent reordering of checkpoints from 2 processors

Specifically, considering the problem in Figure 1-1 again, when using implicit transactions it would be impossible to obtain the unexpected result of A = 0. The two memory updates from processor P_1 will likely reside in the same implicit transaction. In such case, processor P_2 will receive both updates atomically, with the mechanisms discussed before. In the unlikely event of the two stores being on different checkpoints, the global serialization of different checkpoints in the system prevents any unordered notification to P_2.

Furthermore, the multi-checkpointed model with implicit transactions allows an important improvement that avoids the latency that a simple SC implementation imposes: it allows the execution of memory operations out-of-order. This does not compromise the correctness of the SC model because:

1. The reordering and overlapping of memory operations is allowed only within a single implicit transaction.
2. All the memory operations are executed speculatively and, while the loads bring data into the local cache, the global performing of stores is delayed until the transaction can commit without modifying any cache line.
3. During a transaction, all the speculative loads that match the address of a previous pending store receive the correct value thanks to the usual store forwarding mechanism.
4. The snooping of memory updates from the coherence system ensures that the values speculatively loaded remain valid unless an address match is found, what would produce a rollback of the transaction.
5. Finally, the memory updates from the transaction are atomically broadcast only when the transaction commits, making the pending stores globally performed at this moment.

In this manner, the global result of a transaction is the same, independently of the order of execution of its instructions, making the system behave as in Figure 5-4. Therefore, in the figure an acceptable order for instructions in transaction “TR_A1”, for example, could be “A2, A3 and A1”, instead of the order shown: “A1, A2 and A3”.

5.3. Lock speculation with implicit transactions

Previous sections have discussed the correctness and performance aspects of an Implicit Transactions implementation. This section will deal with locking issues. A processor with implicit transactions can of course run traditional lock-based code with accesses to critical sections, preserving the required correct mutual exclusion. However, this section will discuss how the implicit transactional mechanism can be used to speculate on the access to critical sections, as in Speculative Lock Elision [118]. Two different mechanisms will be discussed: Speculation with centralized software locks based on a Silent Store removal mechanism, and speculation with the hardware queue-based implementation of the LCU introduced in Chapter

4. This section will focus on simple critical sections with a single lock access, not on fine-grain locking mechanisms that can require multiple concurrent lock acquisitions.

5.3.1. Dependencies through locks and critical section speculation

Figure 5-5 shows different possible checkpointing schemes when a critical section is executed. Suppose the lock is initially free. In a) the processor takes a checkpoint in the middle of the critical section. The commit of the transaction T1 makes the lock acquisition visible to the rest of the processors, what forces remote processors to abort if their speculative execution path had already acquired the same lock. When the processor validates T2, the critical section gets unlocked and remote processors can access it.

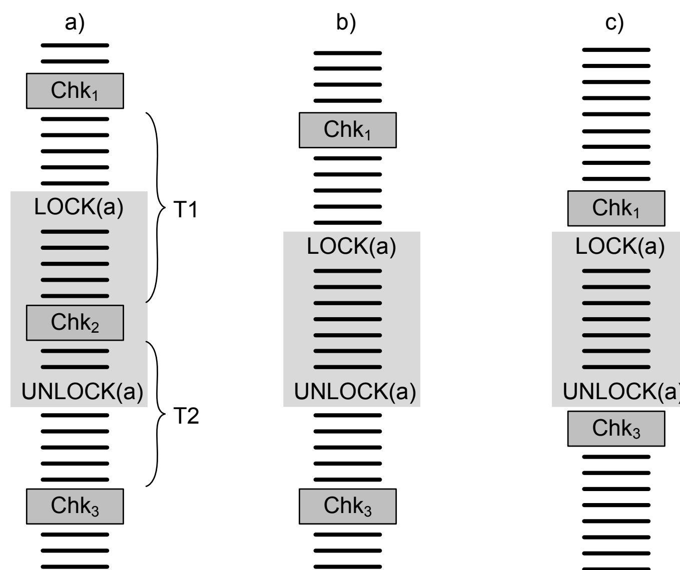


Figure 5-5: Different checkpointing schemes with critical sections

The same invalidation happens even if no checkpoint is taken inside the critical section, as depicted in the case b). In this case, the lock variable is also written, since the lock is acquired and released. Thus, the validation of a transaction that has entirely executed a critical section will cause any other processor that is speculatively executing it to rollback, which means that only a single valid processor stays inside a critical section. This case should be the most frequent one, due to the short nature of critical sections. Case c) shows an optimized situation in which a checkpoint is taken right before the lock acquisition and the next one is taken right after the lock release. This case minimizes the possible sources of conflict, since the transaction will likely abort only if a conflict in the CS is detected, which does not depend on other external accesses. However, it has been shown how the lock variables themselves constitute a form of dependency between different threads accessing a critical section protected by the same lock.

Critical sections admit correct parallel execution in many cases. Rajwar and Goodman [118] discuss some examples, presented in Figure 5-6. In case a), the lock is used to protect a shared

error variable which is hardly ever updated. Multiple threads might access the critical section in most of the cases, except for the infrequent case of two of them having to update the error variable. In case b) two threads use a lock to protect a hash table access. However, their accesses will only conflict in the unlikely case of X = Y and the key not being present in the table. Section 4.3.4 has presented another example in the benchmark Fluidanimate. Coarse-grain locks are used to protect the cell structures; however, all updates are performed on a per-field basis, what would allow for concurrent updates to different fields if the lock granularity was finer.

```
1: LOCK(locks->error_lock);
2: if (local_error > multi->err_multi)
3:       multi->err_multi = local_error;
4: UNLOCK(locks->error_lock);
```

a)

Thread 1

```
1: LOCK(hash_tbl.lock);
2: var = hash_tbl.lookup(X);
3: if (!var)
4:     hash_tbl.add(X);
5: UNLOCK(hash_tbl.lock);
```

Thread 2

```
1: LOCK(hash_tbl.lock);
2: var = hash_tbl.lookup(Y);
3: if (!var)
4:     hash_tbl.add(Y);
5: UNLOCK(hash_tbl.lock);
```

b)

Figure 5-6: Examples of critical sections that often admit parallel execution, from [118]

However, different threads that might execute a critical section in parallel still suffer a dependency through the lock data, as discussed before. Other works handle this problem explicitly. Speculative Lock Elision (SLE [118]) detects the lock acquisition and skips it, starting speculative execution. Speculative Synchronization [103] acquires the lock if it is free, and starts speculation otherwise. In both cases, a conflict detection mechanism is used to provide transactional behavior in the speculation mechanism. Next sections will deal on how to leverage the transactional behavior of the implicit transactions to support speculative execution of parallel sections.

5.3.2. Critical section speculation with software locks

When parallel execution is possible within a critical section, the lock variables are the only interaction between different processors accessing the CS. Let's consider for simplicity a TAS or TATAS lock and the frequent case of the whole critical section being enclosed within a single implicit transaction (cases b and c in Figure 5-5), what is feasible since a CS tends to be as small as possible. The key idea here is that the store executed to release the lock writes a value in a memory position (in the lock header) that already contained that value before the lock acquisition, with the logical meaning of "free lock". This operation constitutes a temporally silent store [93]. Silent stores [92] are those store operations that write a value which was already present in the memory location, typically a 0. Temporally silent stores are pairs of

stores that write a value and then revert it to the original one. This is exactly what happens with the lock acquisition and release pair.

Therefore, the idea to allow for critical section speculation will require the detection and removal of these temporally silent stores within a single implicit transaction. The implementation can rely on the memory disambiguation mechanisms, already required in the load/store queue. It will be composed of two basic mechanisms: A *store merging* mechanism and a *silent store removal* technique.

Store merging consists of squashing multiple stores to the same address in the load/store queue, to reduce the number of committed updates. Such technique is common; for example, it was already present in the Alpha 21164 [39]. The store buffer merging mechanism in such architecture could handle 1 store merging per cycle, enough for the small write buffer of 6 entries used. The Alpha 21164 needs to ensure that between these two stores there is no load to the same address. In the implicit transactions model, the store merging can run before commit, what prevents worries about these race conditions since all loads in the transaction are supposed to be executed. However, such a design would be unfeasible if it had to consider all the possible merges in a large write buffer like the one required for a kilo-instruction architecture.

The proposed design will try to detect the specific addresses that are likely to be part of a lock structure, by noticing that lock acquisition is always performed with an otherwise infrequent atomic instruction. Therefore, the idea will be to record the memory locations successfully modified by atomic instructions during each implicit transaction. A new small buffer (containing 1 or 2 entries, for example) will be used for this operation, named Atomically Updated Address Register (AUAR). At commit time, the store merging mechanism is applied only for those addresses in the AUAR. Additionally, if the LSQ is divided into multiple sections according to the different checkpoints [115], only one of these sections has to be checked, making the operation completely feasible. This merge will likely leave a single store for the lock, which is the last one from the release operation. After the store merging, the remaining store constitutes a silent store, since the lock was initially free at the beginning of the checkpoint.

Conventional *silent store removal* proposals [92] require a new load prior to the commit of each store to determine if the store is silent. Such technique increases the pressure on the L1D cache, especially in the case of the burst commit of a checkpointed architecture, making the idea undesirable. However, a simple idea is to reuse the AUAR buffer and only determine if the contained addresses are silent by accessing the local cache. Alternatively, the LSQ itself could be queried for previous loads to the same address in the same checkpoint, preventing any access to the L1D cache. In any case, the detected silent store would be removed from the LSQ, what allows for the desired behavior of allowing parallel execution of critical sections

which have no real interaction. Finally, notice that the nature of the mechanism prevents any possible ABA problems that would lead to undetected isolation violations.

Two concerns will be discussed next. First, how to ensure that both the lock acquisition and release occur within the same transaction. Second, how to handle other lock implementations, different from centralized locks accessed with atomic instructions and contention.

As discussed in section 5.3.1, the small size of critical sections makes highly probable that a checkpoint will not be taken inside it. However, the desired case would be the one in Figure 5-6 c), in which the implicit transaction exactly encloses the critical section. The AUAR can be leveraged for this task too. When an atomic update is successfully performed, a new checkpoint is taken and the address is recorded in the AUAR. When the same address is modified again (lock release) another checkpoint is taken, what leads to the desired minimum checkpoint length. This mechanism relies in the fact that accesses to the lock inside the critical section are uncommon.

Regarding different lock implementations, it is interesting to note that the proposed implementation might also work with almost any software lock. Consider for example a simple queue-based MCS lock as presented in section 1.3.1.1. In this case, enqueued processors do not acquire the lock with an atomic operation; instead, the previous owner in the queue sets an entry in their queue node, indicating the lock transfer. While this imposes a problem, this case only occurs if the lock is initially taken, what prevents the speculation in the CS by itself. By contrast, if the lock is free, the processor uses an atomic operation to enqueue its own qnode as the head and tail of the queue. Since this operation is speculative, no other processor will enqueue before the checkpoint commit, so the release of the lock will also have to update the same memory location, what will be used for the temporally silent store elimination mechanism.

5.3.3. Critical section speculation with the LCU model

The Lock Control Unit proposed in Chapter 4 accelerates the locking mechanism in a parallel system. This section deals with the issues that a LCU-based implementation can impose to speculative execution in critical sections. The problems discussed in this section will be only focused on the speculation capabilities, since the LCU implementation might work seamlessly with a checkpointed processor without requiring any speculation.

5.3.3.1. Implicit transactions and ordinary LCU access

This section will discuss the interface between the checkpointed processor and the LCU mechanism. The required mechanisms for speculative LCU accesses will be presented and discussed in later sections.

All of the actions executed by a processor based on implicit transactions are speculative until commit. In the case of a software lock acquisition, the write to the lock word is effectively performed at commit time, not when it is recorded in the write buffer. By contrast, the acquisition of locks with the LCU cannot be delayed until the checkpoint commits, since it requires a remote request to the LRT, and it returns a reply status: lock taken or not. Dividing such operation in two steps ("acquire lock" and "check if it is taken") would not be either possible, since the result of the "check" depends on the "acquire" operation being already globally performed.

Therefore, the implementation will require performing LCU acquisition operations in a non-speculative manner. These operations should be recorded and in case of a checkpoint rollback, undone: Locks taken during an implicit transaction must be released if the transaction aborts. By contrast, **release** operations must be delayed until the implicit transaction commits, since the release operation cannot be undone: re-acquiring the lock is different to never having released it. This behavior is analogous to the compensation actions for software locks in a HTM system discussed in [155].

The interface with the LCU will then require of a small buffer, to record the **acquire** and **release** requests issued to the LCU. This structure, which will be called *LCU buffer*, is analogous to the LSQ that records the loads and stores executed during the transaction. At commit time, the buffer is accessed and the **releases** are sent to the LCU. Similarly, on a transaction abort, the buffer is accessed and the acquisitions are undone, by issuing releases to the LCU. The LCU buffer must also allow for merging of operations: two entries for a lock which is released and re-acquired in the same implicit transaction, given the atomicity property, nullify each other. Even more, if this did not happen, it would be impossible to re-acquire the lock before the delayed release occurs. This implies that the LCU buffer must be implemented as a CAM, what is not a problem given its small requirements (LCU accesses will be much more reduced than memory stores), and the fact that merge operations only occur within a single implicit transaction (the LCU buffer will be divided into multiple stages). However, the merge operation must be considered carefully, since it violates the FIFO order in the lock access, and an excess of lock merging can starve other requestors. An alternative idea is to take a new checkpoint when a **release** occurs, effectively releasing the lock. Finally, it should be noted that the **release** operation also returns a value (it can fail if no LCU entries are temporarily available). However, since this case should be very infrequent (definitely, much less frequent than finding a lock being taken) it is safe to speculate on a positive result, and add some logic which iterates until success otherwise.

5.3.3.2. Speculation support in the LCU

There are two important differences between the speculative CS execution with software locks presented in section 5.3.2 and the LCU implementation. First, speculation with software locks relies on the fact that a processor can *check* if the lock is free. Additionally, the isolation

mechanism guarantees that *any remote acquisition of the lock aborts the speculative execution*, since is detected with the coherence updates. By contrast, the presented LCU mechanism does not allow the access to a lock except for the acquisition primitive, with the corresponding enqueueing mechanism. The second problem, which is slightly related, is that the status checks performed in the software locks are completely invisible to other processors (they are simple loads). The same occurs with the speculative lock acquisition which remains private in the processor LSQ until commit. By contrast, the acquisition of a LCU lock requires of an enqueue mechanism controlled by the LRT, what makes the lock to appear as taken from the point of view of another processor, preventing its speculation in the critical section.

These differences suggest that speculation is a priori not possible with the current LCU design. This section will sketch a modification to the base LCU mechanism to allow for such speculative behavior, and discuss some implementation issues. A detailed implementation is not provided, so many of the concerns are not discussed, and left for future work.

The base design of the LCU considers two possible lock acquisition modes: Read/Write (R/W), or shared/exclusive. In both modes the lock is taken, preventing any other acquisition, except for shared reads. The CS speculation mechanism requires that multiple threads can access a write lock concurrently in speculative mode, and any of them is notified if the lock is actually taken in non-speculative mode. Speculative accesses in read mode are unnecessary: if the speculative thread is not going to modify any data in the CS, the read mode itself does the work, without requiring speculative accesses. To support the desired write speculation, a third **Speculative** (**S**) mode will be introduced. Speculative lock requests are sent from the LCUs to the LRT, and they are granted only if no other non-speculative request exists in the queue. Speculative requests build a queue as usual. Multiple speculative requests can acquire a lock concurrently, similarly to read requests. When a lock only has speculative requests, it is recorded with a special *speculative mode* in the LRT. This case is depicted in Figure 5-7, in which the threads running in processors 0 to 2 have accessed a lock in speculative mode.

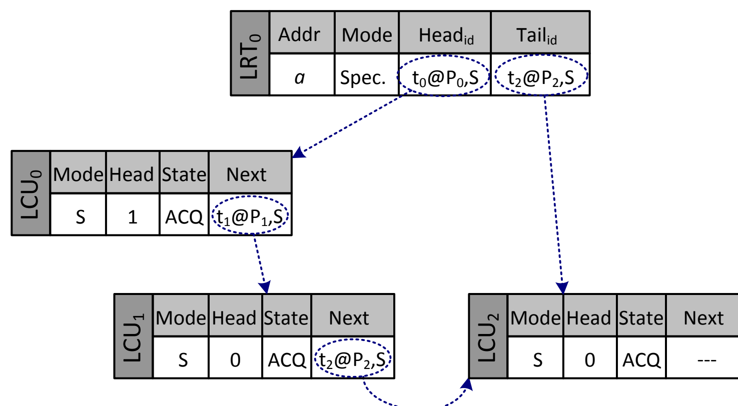


Figure 5-7: Example of speculative access to a lock

When an ordinary request (i.e. write request) is received in the LRT for a lock in speculative mode, the LRT performs two actions. First, it records the requestor as the queue tail as usual, forwarding the request message to the previous tail node in order to enqueue the requestor. Additionally, the LRT sends a RETRY message to the queue head, using the **$Head_{id}$** pointer which is always valid. This RETRY message received on a speculative entry forces a release of the entry, with the corresponding send of the Head token through the queue in further RETRY messages. Eventually, all the speculative entries are released and the new writer receives the Head token, becoming the lock owner. Figure 5-8 shows how this is handled, from the example presented before. The speculative queue deallocation messages are coloured in blue, while the forwarded request message is green. Updated fields are shaded in blue or green, and deallocated entries are marked with state “---”. The forward of the RETRY messages does not need to notify the LRT; they are handled similarly to the abort case presented in section 4.2.3.2. By contrast, the reception of the lock in the last non-speculative node, LCU_9 in the Figure, must send a NOTIFY message to the LRT (not depicted).

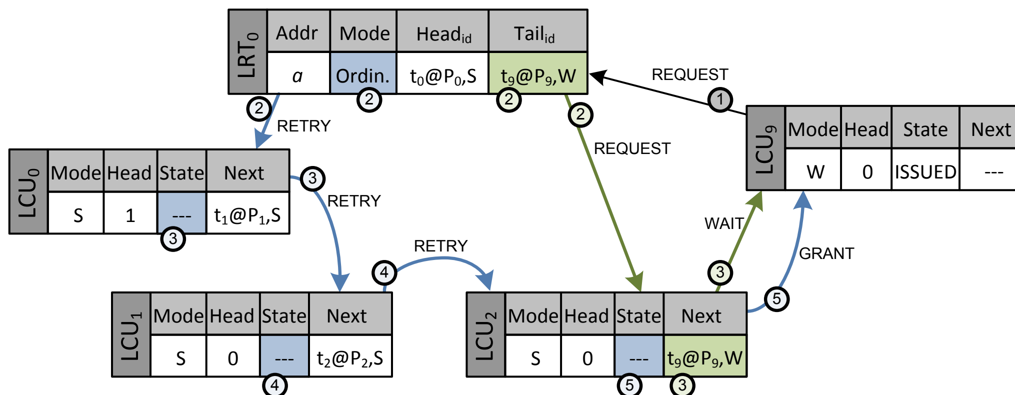


Figure 5-8: Deallocation of speculative entries in the lock queue

If the non-speculative lock requestor had already taken the lock in speculative mode, it will have to remove the queue first, including its own speculative entry. As discussed in section 4.2.3.9 for the proposed read-only locking mechanism, a new type of message should be sent from the requesting LCU to the LRT, which starts the deallocation of the speculative queue. Once the speculative LCU entry is deallocated, a new ordinary entry can be allocated.

As discussed in section 4.2.3.9 for the read-only mechanism, the speculation mode should be used carefully, since it can reserve LCU resources for a long time. Additional mechanisms to ‘clean’ the speculative LCU or LRT resources might be devised.

5.3.3.3. Implicit transactions and speculative LCU access

A processor can decide whether to speculate on a lock or not, probably using hints in the lock instruction, provided by the programmer or the compiler. Speculation is valid if a lock is accessed in speculative mode only within a single implicit transaction, this is, the access to the critical section does not span through multiple transactions. The simplest idea would be to

take a new checkpoint when a lock is acquired in speculative mode, to maximize the probability of the CS fitting in a single transaction. To validate that the speculative lock is released in the same transaction, the LCU buffer must track the speculative accesses within a transaction and check that each acquire operation contains the corresponding release. Otherwise, the transaction is not allowed to commit, it has to abort and switch to a non-speculative LCU mode. Alternatively, the processor might acquire the unreleased locks in non-speculative mode at commit time, with the mechanism discussed in the previous section.

The use of the LCU in speculative mode requires communication in the other way, from the LCU to the processor. If a speculative lock is invalidated due to a non-speculative request, as presented in Figure 5-8, the LCU sends the invalidated address to the processor. If a lock has been speculatively taken on such address in one of the on-flight implicit transactions, the transaction is determined to be invalid, and it aborts. Note the similarity between this mechanism and the LSQ, which detects conflicts with coherence invalidations received in the local cache. By contrast, ordinary non-speculative LCU accesses do not need to be checked, since they never receive external invalidations: A remote processor cannot acquire a taken lock until it is locally released.

5.3.3.4. Speculative LCU accesses and HyTM

If a proper speculative mode in the LCU is implemented, the HyTM system introduced in Chapter 2 might use the accelerated LCU locks. Software transactions would use the ordinary locking mechanism in the LCU, in the corresponding read or write mode. Hardware transactions only require locks to detect conflicts with software transactions. Therefore, they might use the speculative mode for all their accesses. This speculative mode does not generate a conflict when different hardware transactions access the same block in read or write mode; instead, the conflict is detected by the isolation mechanism in the proper memory locations that conflict.

There are plenty of issues that arise when such design is considered. First, the speculative mode does not allow for concurrent hardware and software transactions accessing the same block in read mode. Instead, the software transactions would abort the hardware ones when requesting the lock. Additionally, the LCU buffer in the processor microarchitecture, discussed in section 5.3.3.1, must contain enough entries for all of the explicit transactional accesses. Overflow in the LCU buffer or the LCU entries would be a limit for hardware transactions, similar to overflow in write buffers in many other bounded HTM models. All of these concerns make this idea an open topic for research.

5.4. Speculation beyond flags and barriers

Speculative Synchronization (SS, [103]) studies the possibility of speculative execution past flag and barriers. A checkpoint-based architecture can detect a barrier and execute past it, until the processor resources are full. The study by Martínez and Torrellas [103] determines that

speculative execution past barriers can lead to more than a 10% of overall speedup depending on the application, due to the execution time that threads spend conservatively waiting in barriers.

The proposed LCU design can be adapted to speculate after barriers, in a similar fashion as SS. In such case, there is also a need to detect the barrier code and take a new checkpoint just prior to it. Speculative execution starts after the barrier. However, contrarily to the case of locks, the processor cannot naturally speculate after a barrier. In the case of a lock, it is initially free and the processor can speculatively acquire it. In the case of a barrier, execution proceeds after all of the remaining threads have reached the barrier and set the corresponding flag. Therefore, in the case of barrier speculation it is required that the processor detects the barrier and starts a speculative mode after it.

Barrier detection can be implemented in two different ways. First, the mechanism proposed in [103] can be implemented, what requires a new primitive to notify the processor about the barrier, and add specific changes in the code. Alternatively, the processor might implement a “barrier detection unit” that automatically detects the barrier and starts speculative execution.

The detection unit can be based on the fact that, after the doorway section of the barrier, the processor spins until a flag of the barrier is set (as discussed in [71]). Such iteration consists of a loop in which a single variable is read and tested until it is externally modified, without any store in the loop. The barrier detection unit can determine that the location read in the loop is the flag of a barrier, and speculate on it having a different value that allows the processor to proceed with the execution. As in [103], that location has to be monitorized and, when it is updated with the proper value, the speculative state is committed.

5.5. Summary

This Chapter has introduced the idea of Implicit Transactions as a model of interconnecting checkpoint-based processors. Data validation is performed at the checkpoint level, while other checkpoints continue their speculative execution in the processor, allowing for high degrees of ILP and providing a Sequential Consistent memory model. Simple evaluations with Splash benchmarks have shown that this block validation does not introduce a significant performance penalty, in terms of unnecessary aborted work due to collisions generated by the coarse-grain transaction size. The proposed model can be also used as a bounded HTM if the necessary interface is provided, to support the HyTM system presented in Chapter 2

Additionally, different aspects of the relationship of this checkpoint-based model with reader-writer locks have been discussed. First, a simple mechanism for software-lock speculation has been proposed. This mechanism is based on temporally silent store detection and removal, and relies on the simplified architecture of Kilo-Instruction Processors, specifically a hierarchical LSQ. Second, it has been discussed the interface between the processor and the Lock Control Unit presented in Chapter 4. Specifically, it has been shown how the processor

requires a LCU buffer that records the locking operations of each checkpoint, and how lock releases must be delayed to the commit step, but lock acquisitions must be performed eagerly (before the checkpoint commit) with the corresponding compensation action in case of rollback. Finally, it has been shown how the LCU system, as presented in Chapter 4, does not allow for speculative lock accesses, and a modified *speculative* acquisition mode has been sketched. This speculative mode would allow for speculative access to critical sections, enabling the use of the LCU in the HyTM system introduced in Chapter 2.

Chapter 6. Conclusions

Chapter 6

Conclusions

The current evolution in computer architecture has increased the importance of parallel systems due to constraints in the design complexity of single processors. This makes the mainstream programming task more complex, due to the necessity of parallel programming methods. These methods increase both the complexity of programming and the overhead of the required synchronization and runtime mechanisms. Both aspects impose different limits on the achievable execution speed.

This dissertation has proposed several hardware mechanisms to accelerate the execution of parallel programs, with the main focus on the problems of reader-writer synchronization mechanisms.

The result of this thesis is a set of solutions to accelerate the execution of parallel programs with reader-writer synchronization, from the classical coarse-grain or fine-grain lock-based programs, to the new paradigms of Transactional Memory that simplify the programming model. Additionally, considering the memory wall and other microarchitectural aspects that limit the ILP in a parallel system, it has been proposed a checkpointed architecture for multiprocessors that maintains a simple consistency model while allowing for high levels of speculation.

6.1. Contributions

This thesis has proposed mechanisms for accelerating Transactional Memory, reader-writer locking and building parallel systems based on Kilo-instruction Processors. The main contributions of this thesis are as follows:

1. A Hybrid Transactional Memory system based on reader-writer locking with a generic bounded HTM as the acceleration substrate. The hybrid system allows for slow and safe execution when no HTM substrate is present or, if present, when the HTM resources are exceeded. However, in the general case of short transactions, most transactions can run with the accelerated mechanism, what omits most steps of the software runtime and leads to significant speedups. The main advantages of the

proposed mechanism is that it is highly portable, working in systems with or without HTM support, and that dynamically detects the transactional status of the ongoing transactions.

2. A fairness mechanism for Hybrid Transactional Memory systems. It has been studied how the lack of fairness mechanisms can lead to starvation in the case of reader and writing transactions that collision on a memory location. This problem can be specifically caused by different contention management policies in Hardware or Software transactions in a Hybrid TM system. A new hardware low-cost mechanism, named the “Reservation Table” has been designed to provide fair access to the memory locations to both software and hardware transactions. The deadlock avoidance mechanism of the specific LogTM HTM has been considered, and the design has been consequently adapted to prevent deadlock due to dependencies at different levels. The Reservation Table mechanism minimizes the number of aborts in Hardware mode, what reduces the number of slower software transactions. Overall, the use of the Reservation Table has shown speedups up to 2.5× in simulations of highly congested models that suffered from the starvation problem.

3. A hardware mechanism for fair fine-grain reader-writer locking. Previous reader/writer locking approaches, both software and hardware, are studied to be too costly, have a significant memory or area overhead, or are too inflexible by requiring a limited number of threads tied to the physical processors. A new low-cost distributed mechanism named the Lock Control Unit has been presented. By dynamically allocating each lock and associating it to both the requestor logical thread and physical processor, the system achieves low overhead and high flexibility. The system is designed to support fast lock transfers, removing unnecessary operations from the critical path. Additionally, the corner cases of thread eviction and migration and the overflow of resources are handled with minimum impact in the final performance. The system is evaluated with microbenchmarks, traditional parallel applications and a reader/writer STM which requires fine-grain locking, and it is shown to overcome all the previous software and hardware proposals.

4. An interconnection proposal for checkpointed-based architectures that provides the simplest Sequential Consistency model, while allowing for large degrees of speculation on the processor which reduces overheads. The system, based on so-called *implicit transactions* allows for the validation of blocks of operations. This reduces the coherence and consistency overheads, while maintaining a large instruction window that allows for high levels of speculation to reduce the performance penalty of a slow main memory. Additionally, the relationship between this checkpoint-based model and reader/writer locks has been studied, considering both software locks and the LCU model, and studying both ordinary and speculative accesses.

6.2. Future work

The different techniques presented in this work can be further enhanced or extended. The lock-based Hybrid Transactional Memory system introduced in Chapter 2 has been evaluated by simulation. It is yet pending to observe the behavior with real HTM hardware and determine the effect of the starvation issues studied in Chapter 3 in real implementations. An evaluation of a HyTM model with real hardware was performed [35] by the Scalable Synchronization at Sun Labs with a prototype of the Rock processor [22]. Unfortunately, such processor, which was going to be the first commercial product with HTM support, has been said to be cancelled. Therefore, the evaluation of the HyTM work presented in these sections will have to wait for other HTM products reaching the market.

The Lock Control Unit mechanism proposed for hardware acceleration of reader/writer locks has opened many lines of research. Many issues have been already sketched along the Chapter: Elaborate predictors for the Free Lock Table; Hierarchical implementations in a multi-CMP environment, and its comparison with a simple plain implementation; mechanisms to optimize certain pathologies of the locking mechanism, such as read-only locking; or detailed implementations of the interface between the processor microarchitecture and the LCU hardware. All these lines appear as promising work for future implementations and evaluations.

The idea of implicit transactions to simplify the consistency model presented in Chapter 5 was initially proposed in the ICPS '05 paper listed in the next section. An implementation based on the Bulk model was presented and evaluated in [19] in ISCA’07. Again, the Rock processor from Sun was going to be the first commercial implementation with a checkpoint-based large instruction window to tolerate large memory latencies and provide high single-threaded performance. The ideas presented in Chapter 5 about speculative locking with the LCU hardware are an interesting line of work, especially regarding HyTM implementations.

6.3. Publications

During the training period to achieve the PhD degree, the author of this thesis has been interested in a couple of parallel computing research topics: architectural support to accelerate parallel code execution and interconnection networks. The present thesis document is only devoted to the first research interest. Nevertheless, in addition to the work on reader-writer synchronization mechanisms, Hybrid Transactional Memory and Kilo-instruction Processors, on which this thesis is based, other papers have been published. The following is a comprehensive list of the publications obtained during this period, including works on topological aspects of interconnection networks.

6.3.1. HyTM, kilo-instruction architectures and reader/writer synchronization

1. **Enrique Vallejo**, Marco Galluzzi, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, Jim E. Smith and Mateo Valero. *Implementing Kilo-Instruction Multiprocessors*. In ICPS '05. International Conference on Pervasive Services, 2005, 325-336.

2. **Enrique Vallejo**, Marco Galluzzi, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, Jim E. Smith and Mateo Valero. *KIMP: Multicheckpointing Multiprocessors*. In the Spanish Parallelism Conference, XVI Edition, 2005.

3. **Enrique Vallejo**, Marco Galluzzi, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, James E. Smith and Mateo Valero. *Implementing Kilo-Instruction Multiprocessors*. Universidad Politécnica de Cataluña. TR UPC-DAC-RR-CAP-2005-19., 2005.

4. Marco Galluzzi, **Enrique Vallejo**, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, James E. Smith and Mateo Valero. *Implicit Transactional Memory in Kilo-Instruction Multiprocessors*. In ACSAC '07: Advances in Computer Systems Architecture, 2007, 4697, 339-353.

5. **Enrique Vallejo**, Tim Harris, Adrián Cristal, Osman S. Unsal and Mateo Valero. *Hybrid transactional memory to accelerate safe lock-based transactions*. In TRANSACT '08: 3rd Workshop on Transactional Computing, 2008.

6. **Enrique Vallejo**, Sutirtha Sanyal, Tim Harris, Fernando Vallejo, Ramón Beivide, Osman Unsal, Adrián Cristal and Mateo Valero. *Towards fair, scalable, locking*. In EPHAM'08: 1st Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods, 2008.

7. **Enrique Vallejo**, Sutirtha Sanyal, Tim Harris, Fernando Vallejo, Ramón Beivide, Osman Unsal, Adrián Cristal and Mateo Valero. *Hybrid Transactional Memory with Pessimistic Concurrency Control*. International Journal on Parallel Programming (Accepted for publication), 2010.

8. **Enrique Vallejo**, Ramón Beivide, Adrián Cristal, Tim Harris, Fernando Vallejo, Osman Unsal and Mateo Valero. *Architectural Support for Fair Reader-Writer Locking*. In MICRO-43: The 43rd International Symposium on Microarchitecture (Submitted), 2010.

6.3.2. Interconnection Networks

9. Ramón Beivide, Carmen Martínez and **Enrique Vallejo**. *Gaussian Interconnections for On-chip Networks*. In 1st Microgrid Workshop, 2005.

10. **Enrique Vallejo**, Ramón Beivide and Carmen Martínez. *Practicable Layouts for Optimal Circulant Graphs*. In PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society, 2005, 118-125.

11. Carmen Martinez, Ramón Beivide, **Enrique Vallejo** and Cruz Izu. *Gaussian Interconnection Networks*. In the Spanish Parallelism Conference, XVI Edition, 2005.

12. Carmen Martínez, **Enrique Vallejo**, Miquel Moretó, Ramón Beivide and Mateo Valero. *Hierarchical Topologies for Large-scale Two-level Networks*. Spanish Parallelism Conference, XVI Edition, 2005.

13. Carmen Martínez, **Enrique Vallejo**, Ramón Beivide, Cruz Izu and Miquel Moretó. *Dense Gaussian Networks: Suitable Topologies for On-Chip Multiprocessors*. International Journal on Parallel Programming, 2006, 34, 193-211.

14. Jose M. Cámara, Miquel Moretó, **Enrique Vallejo**, Ramón Beivide, Jose Miguel-Alonso, Carmen Martínez and Javier Navaridas. *Mixed-radix Twisted Torus Interconnection Networks*. In IPDPS '07: IEEE International Parallel and Distributed Processing Symposium, 2007, 1-10.

15. José M. Cámara, Miquel Moretó, **Enrique Vallejo**, Ramón Beivide, José Miguel-Alonso, Carmen Martínez and Javier Navaridas. *Twisted Torus Topologies for Enhanced Interconnection Networks*. IEEE Transactions on Parallel and Distributed Systems (Accepted for publication), 2010.

Chapter 7. References

Chapter 7

References

[1] Standard for information technology - portable operating system interface (POSIX). shell and utilities. *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell*, pages –, 2004.

[2] Sun Fire E25K/E20K systems overview. Technical Report 817-4136-12, Sun Microsystems, 2005.

[3] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.

[4] Nagi Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware support for synchronization in the Scalable Coherent Interface (SCI). In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 141–150, Washington, DC, USA, 1994. IEEE Computer Society.

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[6] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[7] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO '03: Proceedings of the 36th International Symposium on Microarchitecture*, page 423, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *ICS ’90: Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, New York, NY, USA, 1990. ACM.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS ’67 (Spring): Proceedings of the American Federation of Information Processing Societies spring joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[10] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *SCOOL ’05: Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005.

[11] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[12] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Readings in Database Systems*, pages 129–139, 1988.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT ’08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] Jayaram Bobba, Mark Hill, Tim Harris, and Ravi Rajwar (Maintainers). Transactional Memory Online. http://www.cs.wisc.edu/trans-memory/, 2005-2009.

[16] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *IEEE Micro*, 28(1):32–41, 2008.

[17] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA ’07: Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[18] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA ’06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, 2006.

[19] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA ’07: Proceedings of the 34th International Symposium on Computer Architecture*, pages 278–289, New York, NY, USA, 2007. ACM.

[20] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, Chi Cao Minh, Woongki Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA ’07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 97–108, Feb 2007.

[21] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun’s ROCK processor. In *ISCA ’09: Proceedings of the 36th International Symposium on Computer Architecture*, pages 484–495, New York, NY, USA, 2009. ACM.

[22] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[23] Men-Chow Chiang. *Memory System Design for Bus Based Multiprocessors*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 1992.

[24] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, New York, NY, USA, 2006. ACM.

[25] The International Roadmap Comittee. International technology roadmap for semiconductors, 2009.

[26] Computer Laboratory of the University of Cambridge. Practical lock-free data structures website. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/, Aug 2007.

[27] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, 1971.

[28] Travis S Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science, University of Washington., Feb 1993.

[29] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *HPCA ’04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 48–59, Feb 2004.

[30] Adrián Cristal, Oliverio J. Santana, Francisco Cazorla, Marco Galluzzi, Tanausú Ramírez, Miquel Pericas, and Mateo Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, 2005.

[31] Adrián Cristal Kestelman. *Kilo Instruction Processors*. PhD thesis, Departament d'Arquitectura de Computadors . Universidad Politécnica de Cataluña., 2006.

[32] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.

[33] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[34] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *TRANSACT '09: 4th Workshop on Transactional Computing*, 2009.

[35] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS-XIV: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, New York, NY, USA, 2009. ACM.

[36] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[37] Stephan Diestelhorst and Michael Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *EPHAM '08: Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, April 2008.

[38] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, 1965.

[39] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technology Journal*, 7(1):119–135, 1995.

[40] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable NonZero Indicators. In *PODC '07: Proceedings of the 26th Symposium on Principles of Distributed Computing*, pages 13–22, New York, NY, USA, 2007. ACM.

[41] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, Fourth Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[42] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Labs Cambridge, 2006.

[43] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[44] Zhen Fang, Lixin Zhang, John B. Carter, Ali Ibrahim, and Michael A. Parker. Active memory operations. In *ICS '07: Proceedings of the 21st International Conference on Supercomputing*, pages 232–241, New York, NY, USA, 2007. ACM.

[45] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, Aug 2007.

[46] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[47] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[48] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[49] Marco Galluzzi, Valentín Puente, Adrián Cristal, Ramón Beivide, José-Ángel Gregorio, and Mateo Valero. A first glance at kilo-instruction based multiprocessors. In *CF '04: Proceedings of the 1st Conference on Computing Frontiers*, pages 212–221, New York, NY, USA, 2004. ACM.

[50] Amit Gandhi, Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, and Konrad Lai. Scalable load and store processing in latency-tolerant processors. *IEEE Micro*, 26(1):30–39, 2006.

[51] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[52] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA '99: Proceedings of the 36th International Symposium on Computer Architecture*, pages 162–171, May 1999.

[53] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *Computer*, 28(4):23–31, Apr 1995.

[54] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, SCI Committee, March 1989.

[55] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS-III: Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, New York, NY, USA, 1989. ACM.

[56] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the24th Symposium on Principles of Distributed Computing*, pages 258–264, New York, NY, USA, 2005. ACM.

[57] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[58] D.B. Gustavson and Qiang Li. The scalable coherent interface (SCI). *IEEE Communications Magazine*, 34(8):52–63, Aug 1996.

[59] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, 2004.

[60] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[61] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st International Symposium on Computer Architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[63] Tim Harris. Language constructs for transactional memory (or: transactional memory and atomic blocks are not the same thing). In *POPL '08: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Invited talk*. ACM SIGACT-SIGPLA, Jan 2008.

[64] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[65] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[66] Tim Harris and Simon Peyton-Jones. Transactional memory with data invariants. In *TRANSACT '06: 1st Workshop on Transactional Computing*, 2006.

[67] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *HiPC '05: Proceedings of the 11th International Conference on High Performance Computing*, pages 7–18, 2005.

[68] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[69] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[70] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th International Symposium on Computer Architecture*, volume 21, pages 289–300, New York, NY, USA, May 1993. ACM Press.

[71] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Mar 2008.

[72] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, 1998.

[73] IBM. Cell broadband engine architecture, revision 1.02, 2007.

[74] Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual, revision 2.2*. Intel Corporation, Jan 2006.

[75] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Dec 2009.

[76] Harry F. Jordan. Performance measurements on HEP - a pipelined MIMD computer. In *ISCA '83: Proceedings of the 10th International Symposium on Computer Architecture*, pages 207–212, New York, NY, USA, 1983. ACM.

[77] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: let them eat QOLB. In *ISCA '97: Proceedings of the 24th International Symposium on Computer Architecture*, pages 170–180, New York, NY, USA, 1997. ACM.

[78] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[79] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02: Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–141, 2002.

[80] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *ISCA '98: Proceedings of the 25th International Symposium on Computer Architecture*, pages 306–317, Washington, DC, USA, 1998. IEEE Computer Society.

[81] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[82] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204, Washington, DC, USA, 1993. IEEE Computer Society.

[83] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th International Symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[84] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.

[85] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[86] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *ISCA '94: Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[87] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[88] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[89] James Laudon and Daniel Lenoski. The SGI origin: a ccNUMA highly scalable server. *SIGARCH Computer Architecture News*, 25(2):241–251, 1997.

[90] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th International Symposium on Computer Architecture*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[91] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[92] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *ISCA '00: Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, New York, NY, USA, 2000. ACM.

[93] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, New York, NY, USA, 2002. ACM.

[94] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, Aug 2007.

[95] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *SPAA '09: Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures*, Aug 2009.

[96] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGPLAN Notices*, 12(3):128–137, 1977.

[97] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. A hierarchical CLH queue lock. *Lecture Notes in Computer Science (Euro-Par 06)*, 4128:801–810, 2006.

[98] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[99] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.

[100] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006.

[101] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[102] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token coherence: decoupling performance and correctness. In *ISCA '03: Proceedings of the 30th International Symposium on Computer Architecture*, pages 182–193, June 2003.

[103] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, New York, NY, USA, 2002. ACM.

[104] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[105] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[106] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[107] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP '91: Proceedings of the 3rd Symposium on Principles and Practice of Parallel Programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[108] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proceedings of the 20th Symposium on Parallelism in Algorithms and Architectures*, pages 314–325, New York, NY, USA, 2008. ACM.

[109] Mark Moir, Kevin Moore, and Dan Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for rock. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb 2008.

[110] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[111] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, New York, NY, USA, 2006. ACM.

[112] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140. IEEE Computer Society, Feb 2003.

[113] G. Paap and E. Silha. PowerPC: a performance architecture. In *Compcon Spring '93, Digest of Papers.*, pages 104–108, Feb 1993.

[114] M. Pericas, A. Cristal, R. González, D.A. Jimenez, and M. Valero. A decoupled kilo-instruction processor. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 53–64, Feb 2006.

[115] Miquel Pericàs, Adrian Cristal, Francisco J. Cazorla, Ruben González, Alex Veidenbaum, Daniel A. Jiménez, and Mateo Valero. A two-level load/store queue based on execution locality. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 25–36, Washington, DC, USA, 2008. IEEE Computer Society.

[116] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[117] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 241–252, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[118] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[119] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, New York, NY, USA, 2002. ACM.

[120] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, pages 284–298, 2006.

[121] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[122] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. pages 263–272, 2006.

[123] B. Saglam and V. Mooney, III. System-on-a-chip processor synchronization support in hardware. In *DATE '01: Proceedings of the Conference on Design, automation and test in Europe*, pages 633–641, Piscataway, NJ, USA, 2001. IEEE Press.

[124] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[125] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196. IEEE Computer Society, 2006.

[126] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[127] Michael L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *PODC '02: Proceedings of the 21st Symposium on Principles of Distributed Computing*, pages 31–40, New York, NY, USA, 2002. ACM.

[128] Michael L. Scott and William N. Scherer III. Scalable queue-based spin locks with timeout. In *PPoPP '01: Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.

[129] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, New York, NY, USA, 1996. ACM.

[130] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[131] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[132] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[133] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, NY, USA, 2007. ACM.

[134] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 139–150, Washington, DC, USA, 2008. IEEE Computer Society.

[135] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. *SIGARCH Computer Architecture News*, 35(2):104–115, 2007.

[136] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, John, 8 edition, July 2008.

[137] Richard L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2):33–44, 1993.

[138] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *OOPSLA '07: Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 191–210, New York, NY, USA, 2007. ACM.

[139] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processorss. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995.

[140] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[141] SPARC International, Inc., CORPORATE. *The Sparc architecture manual: version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[142] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report TR-915, Computer Science Dept., Univ. of Rochester, Feb 2007.

[143] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing*, Sept 2008.

[144] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Michael Upton. Continual flow pipelines: Achieving resource-efficient latency tolerance. *IEEE Micro*, 24(6):62–73, 2004.

[145] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[146] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[147] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS-XIV: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, New York, NY, USA, 2009. ACM.

[148] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[149] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *SIGARCH Computer Architecture News*, 14(2):414–423, 1986.

[150] Michael Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark Hill, and David Wood. OS support for virtualizing hardware transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb 2008.

[151] The Message Passing Interface Forum. MPI: A message-passing interface standard v2.2, Sept 2009.

[152] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25, 1967.

[153] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[154] D.M. Tullsen, J.L. Lo, S.J. Eggers, and H.M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *HPCA '99: Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 54–58, Jan 1999.

[155] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, 2008.

[156] Gerhard Weikum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[157] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[158] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.

[159] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07: Proceedings of the 34th International Symposium on Computer Architecture*, pages 35–45, New York, NY, USA, 2007. ACM.

[160] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT '06: 1st Workshop on Transactional Computing*, June 2006.

[161] Craig Zilles and Ravi Rajwar. Transactional memory and the birthday paradox (brief announcement). In *SPAA '07: Proceedings of the 19th Symposium on Parallel Algorithms and Architectures*, pages 303–304, June 2007. A longer version is available as Technical Report UIUCDCS-R-2007-2835, March 2007.