Análisis de técnicas de sincronización en estructuras de datos concurrentes

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Álvaro Rubira García
Tutores:
Alberto Ros Bardisa
Eduardo José Gómez Hernández

Facultad
Informática
Universidad
Murcia

3 de Junio de 2024

Resumen

Cuando el fin del escalado de Dennard interrumpió las ganancias exponenciales de rendimiento predichas por la ley de Moore, gran parte del interés de la industria, que previamente se centraba en desarrollar procesadores mononúcleo, se destinó a crear arquitecturas multinúcleo eficientes.

Mientras que un subconjunto de algoritmos "vergonzosamente paralelos" pueden modificarse fácilmente para obtener un aumento del rendimiento considerable cuando se ejecutan de forma concurrente, muchos otros programas necesitan comunicarse a través de memoria compartida de maneras no triviales para lograr una sincronización eficiente.

La programación de este último subconjunto de algoritmos debe tener en cuenta la naturaleza asíncrona de los sistemas informáticos modernos, en los que la ejecución de los distintos procesos se entrelaza de manera impredecible. En este entorno, las carreras de datos son comunes y difíciles de evitar.

Como resultado, al crear algoritmos concurrentes para estructuras de datos, la complejidad del código original se incrementa. Además, se debe tener en cuenta la dificultad de depurar estructuras concurrentes con muchos hilos en ejecución. Por lo tanto, se buscan métodos sencillos que permitan diseñar estas estructuras.

Asimismo, el auge del *software-as-a-service*, en el que servidores remotos generan un número elevado de hilos para procesar solicitudes entrantes, exige el desarrollo de algoritmos de estructuras de datos que puedan funcionar con eficacia cuando son utilizados por muchos hilos al mismo tiempo.

Una forma común y sencilla de sincronización es el uso de cerrojos de grano grueso. La simplicidad de este método tiene como desventaja problemas como la inversión de prioridades o el *convoying*. Además, se serializa la ejecución de la sección protegida, reduciendo en mayor medida el rendimiento del programa cuanto más larga sea la sección. Otro obstáculo para la eficiencia al utilizar cerrojos es que si un hilo es bloqueado tras adquirir un cerrojo, muchos otros tendrán que esperar sin poder avanzar en sus propias tareas, estancando el progreso de gran parte del programa. Este problema empeora conforme se aumenta el número de hilos y, en consecuencia, el número de conflictos entre ellos.

Para abordar estos problemas se han investigado algoritmos que cumplen determinadas condiciones de progreso. Por ejemplo, los algoritmos *wait-free* garantizan que, si un hilo ejecuta instrucciones continuamente, progresará (esto es, completará la operación) en un límite finito de pasos. Estos algoritmos son especialmente útiles en aplicaciones de tiempo real, en las que es importante garantizar que tareas concretas no se retrasarán

indefinidamente.

En particular, los algoritmos *lock-free* mantienen condiciones de progreso más débiles en comparación con los *wait-free* para obtener un rendimiento más alto. Garantizan que si un hilo sigue dando pasos, algún hilo progresará. En un algoritmo *lock-free*, los hilos bloqueados no impiden que todo el sistema progrese. Es decir, se garantiza que, si un proceso queda bloqueado en una sección crítica, no forzará a todos los demás a esperar sin conseguir progreso alguno. Por otro lado, los algoritmos *lock-free* no consiguen evitar el problema de la inanición a diferencia de los algoritmos *wait-free*, pero siguen siendo no bloqueantes, y normalmente más rápidos.

Las primeras estructuras *lock-free* se basaban en el cuidadoso uso de operaciones atómicas simples como Compare-And-Swap (CAS) para la sincronización entre hilos. Esto da lugar a estructuras que superan con creces la complejidad de sus equivalentes con cerrojos. Como consecuencia, este enfoque ha demostrado ser propenso a errores y tener complicadas demostraciones de corrección. Además, es un enfoque que debe adaptarse a cada caso y no se puede generalizar fácilmente a estructuras nuevas o diferentes.

Si bien existen multitud de formas de intentar mejorar este primer enfoque, nos centraremos en los *lock-free locks*, la memoria transaccional y el Multi-address Compare-And-Swap (MCAS) (aunque la implementación *hardware* concreta que utilizamos para estos dos últimos mecanismos de sincronización no es *lock-free*).

El término *lock-free locks* se utilizó en varios artículos que describen métodos para crear estructuras de datos *lock-free* partiendo de estructuras basadas en cerrojos, modificando un código que originalmente utiliza la semántica simple e intuitiva (en comparación con los algoritmos *lock-free* con CAS) propia del uso de cerrojos. Estos algoritmos consiguen ser *lock-free* permitiendo que los hilos en ejecución ayuden a otros hilos que han quedado bloqueados en código protegido por cerrojos.

Los intentos anteriores de crear *lock-free locks* fueron considerados demasiado ineficientes, ya que requerían guardar continuamente el contexto del proceso para permitir la ayuda de otros. En un artículo más reciente, se ha revisado este enfoque y, en su lugar, se propone crear secciones críticas idempotentes. De esta forma, los hilos que ayudan solo necesitan ejecutar la sección crítica desde el inicio para colaborar, sin preocuparse por el contexto del hilo bloqueado.

En este nuevo enfoque se utiliza un *log* o registro para hacer que la sección crítica sea idempotente. El registro se encuentra en un descriptor, visible para todos los hilos que quieran acceder a la sección crítica asociada, junto al código que el hilo que adquirió el cerrojo necesita ejecutar. El registro contiene suficiente información sobre el progreso de la ejecución de esta sección crítica (resultados de accesos a memoria, reserva y liberación de memoria) para hacer posible que varios hilos con la intención de ayudar ejecuten el mismo código teniendo el mismo efecto que si solo fuese ejecutado por un único hilo.

Como puede que las secciones críticas se ejecuten varias veces desde el principio, esta estrategia es especialmente eficaz para adaptar estructuras de datos basadas en cerrojos

de grano fino. Las estructuras con cerrojos de grano fino utilizan un mayor número de cerrojos que las estructuras con cerrojos de grano grueso. Esto da lugar a secciones críticas más pequeñas, lo que a su vez permite una mayor concurrencia, a costa de tener que adquirir más cerrojos. Es un enfoque considerablemente más complejo que su equivalente de grano grueso.

La memoria transaccional, por otra parte, parece ser la técnica más sencilla a la hora de programar las estructuras. Solo conlleva marcar secciones de código que no deben interferir entre sí, de modo que su ejecución parezca atómica para las demás. Esto se implementa de manera eficiente, con suerte con soporte del *hardware*, realizando solamente cambios especulativos hasta que se termine de ejecutar toda la transacción, momento en el los cambios se escriben en memoria compartida y se hacen visibles a otros hilos si no se han detectado conflictos. La implementación *hardware* debe funcionar en conjunción con los protocolos de coherencia de caché para detectar conflictos entre transacciones.

Como pueden aparecer conflictos, será necesario reintentar las transacciones. Para evitar reintentar una transacción demasiadas veces, algunas implementaciones en *hardware* recurren al uso de cerrojos cuando se alcanza un número determinado de reintentos. Por lo tanto, no pueden considerarse estrictamente *lock-free*, y los problemas asociados a los algoritmos bloqueantes pueden aparecer de nuevo si hay conflictos entre hilos.

Por último, el MCAS intenta resolver los problemas del CAS simple permitiendo operaciones atómicas con múltiples direcciones de memoria. Normalmente, en las estructuras de datos *lock-free*, CAS se utiliza después de leer de memoria compartida, para intentar cambiar el dato leído por uno nuevo solo si el original no ha sido modificado ya por otro hilo. La restricción de poder detectar cambios únicamente en una ubicación hace que el diseño de estas estructuras sea notoriamente difícil.

Un CAS para múltiples direcciones de memoria solamente escribe los nuevos valores proporcionados si todas las direcciones todavía tienen sus antiguos valores antes de escribir atómicamente (o se escriben todos los nuevos valores o ninguno). Sin embargo, se ha demostrado que tener acceso incluso a dos direcciones sigue provocando problemas similares a los descritos anteriormente para el CAS simple.

Las operaciones atómicas para varias direcciones de memoria intentan ampliar este límite, y se pueden utilizar estructuras de datos especiales para ejecutar MCAS a un número arbitrario de direcciones de memoria, aunque en este trabajo consideramos cuatro como máximo. Así, el uso de MCAS permite estructuras de datos más simples.

Dado que aún no se dispone de una implementación extendida de MCAS en *hardware*, para obtener una medida fiable del rendimiento de MCAS es necesario ejecutar *benchmarks* en simuladores de arquitecturas de computadores como gem5. Uno de estos *benchmarks* se puede encontrar en el repositorio de GitHub *mcas-benchmarks*, que utilizamos como punto de partida para añadir estructuras de datos con nuevos mecanismos de sincronización.

Al partir de *mcas-benchmarks*, se nos proporcionan implementaciones de varias es-

tructuras de datos *lock-free*. En particular, comenzamos con implementaciones *lock-free* basadas en CAS y en MCAS, además de basadas en cerrojos de grano grueso para comparación. Las estructuras evaluadas son la lista ordenada doblemente enlazada, el *hash map*, el árbol de búsqueda binario, la *deque*, la cola y la pila. Las pruebas también evalúan el rendimiento de los algoritmos *arrayswap* (que intercambia dos índices aleatorios de un *array*) y *multi-word object update* (varios hilos que intentan actualizar atómicamente la misma línea de caché).

Originalmente, el objetivo de este Trabajo de Fin de Grado era incluir en *mcas-benchmarks* estructuras *lock-free* con el método idempotente descrito anteriormente (pero acabamos añadiendo también otros mecanismos de sincronización). El artículo "*Lock-Free Locks Revisited*", que describe este método, proporciona su implementación en la librería `flock`. Empezamos intentando reproducir los resultados presentados en el artículo, para asegurarnos de que la funcionalidad de la biblioteca se ejecuta eficientemente en nuestro entorno de pruebas.

Hemos obtenido resultados muy similares a los presentados en el artículo, demostrando que con `flock` se pueden crear estructuras *lock-free* a partir de cerrojos de grano fino que pueden mantener su rendimiento en condiciones de alta contención. Las estructuras creadas con `flock` también superaron a otras alternativas *lock-free* del estado del arte.

Incorporamos implementaciones *lock-free* con `flock` para los algoritmos utilizados en *mcas-benchmarks*. Algunas de estas estructuras (lista ordenada doblemente enlazada y *hash map*) ya habían sido creadas por los autores de `flock`, por lo que solo las modificamos ligeramente para añadirlas a las pruebas. También proporcionaron implementaciones de árboles de búsqueda binarios, pero con variaciones en la estructura interna que los hacían demasiado diferentes para compararlos con los árboles ya presentes, por lo que no se incorporaron.

El resto de las estructuras de datos se derivaron sustituyendo por cerrojos `flock` los cerrojos de las estructuras *lock-based* ya presentes en *mcas-benchmarks*. Para el árbol de búsqueda binario, esto significa que la implementación con `flock` utiliza cerrojos de grano grueso, algo que no es deseable. También intentamos crear un árbol de búsqueda binario con `flock` partiendo de implementaciones con cerrojos de grano fino, pero descubrimos que la forma en que se debe estructurar el código con `flock` plantea problemas para realizar el recorrido *hand-over-hand locking* utilizado en la implementación.

Después, se añadieron a las pruebas estructuras basadas en memoria transaccional. Tenemos acceso a Hardware Transactional Memory (HTM) a través de las Transactional Synchronization Extensions (TSX) de Intel. Aunque estas extensiones se desactivaron mediante actualizaciones de firmware en muchos procesadores debido a vulnerabilidades de la implementación, tenemos acceso a procesadores Intel Xeon E5-2695 en los que esta función se puede activar.

Cuando se alcanza el límite de intentos fallidos al realizar una transacción, nuestras implementaciones utilizan un *spinlock* global. Probamos versiones de todos los mecanismos de sincronización que utilizan memoria transaccional con y sin *backoff* exponencial

entre estos intentos.

Teniendo acceso a HTM, la utilizamos para derivar estructuras con HTM a partir de las estructuras basadas en cerrojos ya presentes en *mcas-benchmarks* y a partir de las estructuras basadas en MCAS, con el objetivo de ver qué tipo de algoritmo se beneficia más del uso de HTM.

Por último, añadimos nuevas variaciones de la sincronización basada en MCAS. Necesitamos realizar benchmarks con un elevado número de hilos y comparar varios mecanismos de sincronización, por lo que en lugar de utilizar gem5 para simular el rendimiento de MCAS en *hardware*, utilizamos un cerrojo global para conseguir modificaciones de múltiples direcciones de memoria que se perciben como atómicas.

Además, creamos una variación de MCAS que internamente utiliza HTM para la sincronización en lugar de un cerrojo. Nótese que la estructura resultante es diferente de la anterior que adapta las estructuras basadas en MCAS con HTM pero no utiliza MCAS.

También probamos variantes de estas versiones de MCAS en las que la sentencia `if` propia del MCAS para comparar los valores leídos con los esperados se sustituye por código que evita el uso de una operación de salto condicional.

Al crear las nuevas estructuras, encontramos varios errores en las estructuras ya presentes en *mcas-benchmarks*. Proporcionamos soluciones para los problemas que pudimos resolver. Además, mejoramos la precisión de la medición del tiempo realizada en los *benchmarks* añadiendo barreras antes y después de que los hilos realicen sus operaciones.

En *mcas-benchmarks*, medimos el tiempo tardado en completar un número determinado de operaciones que se asignan uniformemente entre todos los hilos utilizados. Esto se hace para cada uno de los mecanismos de sincronización y estructuras de datos.

Obtenemos resultados que indican que `flock` es superior a los demás mecanismos de sincronización cuando se utiliza en estructuras con cerrojos de grano fino. HTM también ofrece un rendimiento competitivo y puede programarse de forma más sencilla, con un estilo de grano grueso, ya que encontramos que las estructuras que utilizan HTM basándose en la adaptación de estructuras lock-based de grano grueso son más rápidas que las que adaptan estructuras basadas en MCAS, en la mayoría de los casos. Además, las variantes que eliminan saltos condicionales en la implementación de MCAS o utilizan *backoff* tienen efectos mínimos en el rendimiento para estos *benchmarks*.

También observamos que el anclar hilos a núcleos específicos reduce drásticamente la eficacia de los enfoques lock-free en escenarios con alta contención.

Por último, la sincronización basada en MCAS, ya sea implementada con cerrojos o HTM, es únicamente mejor que sus equivalentes con cerrojos o HTM no basados en MCAS para el árbol de búsqueda binario. Es decir, MCAS implementado con HTM funciona peor que el simple uso de HTM de grano grueso, y MCAS implementado con un cerrojo es peor que la sincronización lock-based de grano grueso en la mayoría de los casos, pero no pudimos hacer pruebas con MCAS para la lista ordenada y el *hash map* debido a errores.

Extended Abstract

As the end of Dennard scaling abruptly slowed the raging performance gains predicted by Moore's law, a great part of the industry's interest which was previously focused on developing single-core processors has now been displaced to creating performant many-cores architectures.

While a subset of "embarrassingly parallel" algorithms can be easily modified so that a huge speedup will be gained when executed concurrently, many other programs will need to communicate through shared memory in non-trivial ways to achieve safe synchronization.

The programming of this latter set of algorithms has to account for the asynchronous nature of modern computer systems, where the execution of each thread is scheduled in unpredictable ways. In this environment, data races are common and difficult to avoid.

As a result, when creating concurrent algorithms for data-structures, the complexity of the original code is further incremented. In addition, the difficulty of debugging concurrent structures must be considered. Therefore, simple methods for creating concurrent data-structures are needed.

Moreover, the rise of software-as-a-service, where remote servers spawn many threads in order to handle all incoming requests, calls for the development of data-structure algorithms that can perform efficiently when used by many threads at the same time.

A common and simple way of synchronization is the usage of locks. The simplicity of this method comes at the cost of scheduling problems such as priority inversion or convoying. In addition, execution of the protected section is serialized, further reducing the program's efficiency as the section grows bigger. Another glaring obstacle for efficiency when using locks is that if a thread is delayed indefinitely while holding a lock, many others will have to wait without progressing in their own tasks, stalling the progress of most of the program. This issue becomes more common as the number of threads, and contention as a result, increases.

To address these problems, special algorithms that adhere to specific progress conditions have been investigated. One such example are wait-free algorithms, which guarantee that every thread that keeps taking steps makes progress in a finite number of steps. This kind of algorithms is well-suited for time-critical applications, where it is important to guarantee that a particular task will not be delayed indefinitely.

In particular, lock-free algorithms grant weaker progress conditions when compared to wait-free ones in exchange for efficiency. They guarantee that if a thread keeps taking steps, some thread will make progress. In a lock-free algorithm, delayed threads do not prevent the whole system from making progress. That is, they ensure that threads

blocked inside protected sections will not force all other threads to wait. As a result, lock-free algorithms are not starvation-free unlike wait-free algorithms, but they are still non-blocking, and usually faster.

Early lock-free structures were largely based on careful usage of simple atomics such as CAS for synchronization between threads. This results in structures that greatly exceed the complexity of their blocking equivalents. As a consequence, this approach has been shown to be error-prone and to have complicated correctness proofs. It is also an approach which needs to be tailored to each specific case, and cannot be easily generalized to new or different structures.

Many ways of improving upon this first approach have been investigated. We focus on lock-free locks, transactional memory and MCAS (although the current hardware implementation we use for the latter two is not lock-free).

The term "lock-free locks" appears in papers that describe methods for creating lock-free data-structures by adapting lock-based ones, with modifications to code that can still use the simple and intuitive (when compared to CAS-based algorithms) semantics of lock acquisition. Lock-freedom is achieved by allowing running threads to help other threads that were blocked while holding a lock.

Previous attempts at creating lock-free locks were considered to be too inefficient, as they needed to continuously save the context of the process in order to allow helping from others. In a more recent paper, this approach was revisited. Said article proposed creating idempotent critical sections. This way, helping processes only need to execute the section from the beginning in order to help, without worrying about the context of the blocked process.

In this new approach, a log is used for making the critical section idempotent. This log is found in a lock descriptor, visible to all threads that try to access said lock, along with the code that the thread which currently holds the lock needs to run. The log contains enough information (results of memory accesses, allocation and freeing of memory) as to make it possible for many helping threads to run the same code and have the same effect as if it was only ever run by a single thread.

As critical sections between locks might be executed multiple times from the beginning, this strategy is specially effective for adapting fine-grained lock-based data-structures. Fine-grained locking uses a higher number of locks, as opposed to coarse-grained locking. This leads to smaller critical sections, which in turn enable higher concurrency, at the cost of needing to acquire more locks. It is also significantly more complex.

Transactional memory, on the other hand, seems to be the simplest approach of the three when it comes to programming. It only involves marking sections of code which should not interfere with each other, so that their execution will appear as atomic to the others. To achieve this in an efficient way, this is hopefully done with hardware support, by only making speculative changes until the whole transaction has been executed, at which point it is committed to shared memory and made visible to other threads if no conflicts were detected. As it might be expected, this is highly related to and ideally

engineered to work in conjunction with cache coherence protocols.

As conflicts might appear when using transactional memory, transactions will need to be retried. To avoid retrying a transaction too many times, some hardware implementations resort to using a lock when reaching a given number of retries. Thus, they cannot be strictly considered lock-free, and problems associated with blocking algorithms may appear again with high contention.

Finally, MCAS attempts to solve many of the problems of simple CAS by enabling atomic operations to multiple memory addresses. Typically, in lock-free data-structures CAS is used after reading some shared data, as it tries to change it to a new value only if the original was not already changed by another thread. The constraint of only being able to detect changes in one location makes designing these structures notoriously difficult.

CAS for multiple memory addresses only writes all of the provided new values if all locations still have their old values before writing atomically (either all or none of the new values are written). However, it has been shown that having access to even two locations still leads to problems similar to the ones described above for simple CAS.

Multi-address atomic operations try to extend this limit on the number of locations, and special data-structures can be used for executing a MCAS to arbitrarily many addresses, although we only consider up to four. This allows for simpler data-structures.

As a commercial hardware implementation of MCAS has yet to be available, benchmarks need to run in computer architecture simulators such as gem5 in order to correctly measure the efficiency of hardware MCAS. One such benchmarks can be found in the GitHub repository "mcas-benchmarks", which we use as a starting point for adding data-structures with new synchronization mechanisms.

When starting from mcas-benchmarks, we are provided with implementations of several lock-free data-structures, which can then be tested against concurrent reads and modifications by many threads. In particular, we start with lock-free implementations based on CAS, on MCAS and lock-based implementations for comparison. The tested structures are sorted doubly-linked list, hash-map, binary search tree, deque, queue and stack. These benchmarks also evaluate the performance of two other algorithms known as arrayswap (exchanging the values in two random indexes of an array) and multi-word object update (multiple threads that try to atomically update the same cacheline).

Originally, the aim of this bachelor's thesis was to include lock-free lock synchronization with the idempotent method described above into mcas-benchmarks (but we ended up adding other synchronization mechanisms too). The paper "*Lock-Free Locks Revisited*", which describes said method, also provides its implementation in the `flock` library, for practical application of the technique. We started by attempting to reproduce the results presented in the `flock` article, in order to ensure that the library's functionality would run efficiently on our benchmarking machines.

We achieved very similar results to the ones presented in the paper, supporting the claim that `flock` can create structures with lock-free fine-grained locks that can

maintain their throughput under heavy contention. The structures created with `flock` were also able to outperform other state-of-the-art lock-free alternatives.

We incorporated lock-free implementations with `flock` for the algorithms used in mcas-benchmarks. Some of these structures (sorted doubly-linked list and hash-map) had already been created by `flock`'s authors, so we only modified them slightly in order to add them to the benchmarks. They also provided binary search tree implementations, but with variations on the inner structure that made them too different for comparison with the already present trees, so they were not incorporated.

The rest of the data-structures were derived by replacing normal locks from the lock-based structures with locks from the `flock` library. For the binary search tree, this means that the `flock` implementation uses coarse-grained locking, which is not ideal. We also tried creating a binary search tree with `flock` starting from existing fine-grained blocking implementations, but found that the way `flock` code needs to be structured poses problems for doing the hand-over-hand locking required in this structure.

Next, structures using HTM were added to the benchmarks. We have access to HTM through Intel's Transactional Synchronization Extensions. Although support for these extensions was disabled through firmware updates in many processors due to software vulnerabilities found in the implementation, we have access to Intel Xeon E5-2695 processors where this feature can be activated.

When 6 failed attempts at committing a transaction are reached, our implementations use a global spinlock. We test versions of all synchronization mechanisms that use transactional memory with and without exponential backoff between these attempts.

Having access to HTM, we use it to derive HTM-based structures from the lock-based ones and also from the MCAS-based ones, in order to see which kind of algorithm benefits most from the use of HTM.

Lastly, we add new variations of MCAS-based synchronization. We need to benchmark with a high number of threads and against many other synchronization mechanisms, so instead of using gem5 to simulate hardware MCAS we use a global lock to achieve modifications of multiple memory addresses which are perceived as atomic.

In addition, we create a variation of MCAS which relies internally on HTM for synchronization instead of a lock. Note that this version is different from the previous one that adapts MCAS-based structures with HTM but does not use MCAS.

We also test variants of both lock-based and HTM-based MCAS where the internal if statement used for comparing read values with expected (old) values is replaced with code that avoids using a conditional branch operation.

When creating the new structures, we found several errors in the ones already present in mcas-benchmarks. We provide solutions for the problems we were able to solve. Furthermore, we improve the accuracy of timekeeping done in the benchmarks by adding barriers before and after threads carry out their operations.

In mcas-benchmarks, we measure the time taken to complete a given number of operations which are evenly distributed among threads. This is done for each of the

synchronization mechanisms and data structures.

We found `flock` to be superior than the other synchronization mechanisms when used in structures with a fine-grained locking approach. HTM also offers competitive performance and can be programmed in an easier, coarse-grained manner, as we found that HTM structures based on adapting coarse-grained lock-based structures are faster than the ones which adapt MCAS-based ones, in most cases. Moreover, variants created by removing conditional branches from the MCAS implementation or using backoff with HTM have minimal effects in our results.

Our experiments also showed that pinning threads to specific cores severely reduced the effectiveness of the lock-free approaches with overcontention.

Finally, MCAS-based synchronization, whether implemented with a lock or HTM, is only better than its simple lock-based and HTM equivalents for binary search trees. That is, MCAS implemented with HTM performs worse than simply using coarse-grained HTM, and MCAS implemented with a lock performs worse than simply using coarse-grained lock-based synchronization in most cases, but we were unable to test MCAS implementations for sorted list and *hash map* due to errors.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Introducción

Algunos algoritmos "vergonzosamente paralelos" [1] se pueden adaptar sin problemas al nuevo paradigma multihilo establecido por el fin de la ley de Moore, donde varios núcleos colaboran concurrentemente, obteniendo un gran aumento del rendimiento respecto a la ejecución secuencial con un sobrecoste de sincronización despreciable. Por desgracia, la adaptación de muchas estructuras de datos que se deben compartir entre hilos no se puede realizar con tanta facilidad. La comunicación entre los distintos hilos de ejecución se realiza modificando posiciones de memoria compartida, donde aparecen fácilmente carreras de datos.

Así, la creación de estructuras de datos secuenciales que ya de por sí pueden tener una implementación complicada, debe ser adaptada con sumo cuidado si se quiere obtener una estructura de datos correcta y altamente concurrente.

Un gran peligro aparece al utilizar estructuras de datos concurrentes cuya implementación es compleja o no tan intuitiva. La depuración de estructuras secuenciales puede llegar a parecer relativamente fácil si se compara con la de estructuras concurrentes, donde es difícil razonar sobre el comportamiento del programa teniendo en cuenta el efecto de varios hilos cuya ejecución se planifica de manera impredecible. Por lo tanto, se buscan técnicas simples, capaces de ahorrar tiempo de desarrollo y que faciliten la tarea de comprobar la corrección de estos algoritmos.

Una manera sencilla y ampliamente utilizada en la actualidad de generar estructuras concurrentes es la utilización de cerrojos o mútexes para restringir el acceso a secciones críticas delimitadas en el código. El uso de cerrojos acarrea una serie de problemas de rendimiento que se agravan en situaciones de alta contención, donde aumenta el número de conflictos entre hilos que intentan interactuar a la vez con estructuras compartidas.

Teniendo en cuenta el reciente auge del *software-as-a-service*, donde gran parte del procesamiento realizado antes localmente se traslada a servidores remotos [2] que deben atender a la vez múltiples peticiones manejando para ello una cantidad elevada de hilos, estos escenarios de alta contención son cada vez más comunes.

En estos escenarios es posible encontrar hilos bloqueados dentro de secciones críticas, que impiden como consecuencia el progreso de todos los demás hilos que necesitan acceder a las mismas, provocando una severa degradación del rendimiento.

Se han desarrollado algoritmos especializados con condiciones de progreso que intentan garantizar el avance del programa en estos casos. En concreto, en este trabajo nos centramos en mecanismos de sincronización originalmente concebidos para crear algoritmos *lock-free* (aunque en la práctica las implementaciones *hardware* que empleamos para varios de ellos no son *lock-free*, como se explicará posteriormente). Estas técnicas

se basan en el uso de secciones críticas idempotentes, memoria transaccional y CAS para varias direcciones de memoria.

Hemos desarrollado implementaciones de varias estructuras de datos con estos procedimientos y comparamos su eficiencia con la de otros algoritmos *lock-free*. En el proceso de desarrollar estas implementaciones, hemos descubierto errores en estructuras de datos previas que evidencian la necesidad de técnicas simples para crear estructuras de datos concurrentes.

2 Estado del arte

A continuación tratamos los avances realizados en el ámbito de varias técnicas para la creación de algoritmos concurrentes, con un recorrido que empieza por los primeros intentos de un método universal para crear algoritmos *lock-free*, para seguir con el desarrollo de *lock-free locks*, atómicos de varias direcciones de memoria, memoria transaccional y finalizar con *lock-free locks revisited*.

2.1 Algoritmos lock-free

Los algoritmos *lock-free* son aquellos que proporcionan la garantía de que, en todo momento, *alguien* está progresando (esto es, algún hilo consigue ejecutar código perteneciente a una sección crítica), independientemente de los retrasos o fallos que puedan sufrir los demás. Esto permite eliminar por completo la posibilidad de problemas como el *convoying* o la inversión de prioridades, que pueden aparecer al utilizar secciones críticas convencionales, aunque el problema de la inanición, que ocurre cuando un proceso nunca recibe tiempo de procesamiento porque otros lo acaparan continuamente, sigue presente [1]. Se garantiza que, si un proceso queda bloqueado en una sección crítica, no forzará a todos los demás a esperar sin conseguir progreso alguno. En el resto del trabajo se utiliza el término "lock-free" para referirse a este tipo de algoritmos no bloqueantes.

Una gran desventaja de las estructuras lock-free es su complejidad. Las implementaciones lock-free de estructuras comunes se suelen basar en la cuidadosa utilización de operaciones atómicas como CAS [3, 4], por lo que emplean enfoques muy especializados difíciles de generalizar a otras estructuras de datos y cuya corrección es difícil de demostrar. Su implementación dista de ser trivial y varios investigadores han propuesto otros algoritmos que intentan mejorar su generalidad.

Crear copias completas o parciales de una estructura de datos antes de realizar cualquier modificación sobre la misma es un método sencillo para generar una versión concurrente de cualquier estructura de datos. Tras realizar la operación sobre la copia local, se utiliza alguno de los mecanismos de sincronización lock-free (disponibles en la arquitectura, p. ej. CAS o Load Linked/Store Conditional) para unificar los datos de las copias locales. En caso de existir un conflicto entre las estructuras, la copia se descarta y se vuelve a realizar una copia limpia para intentar la operación de nuevo. Propuesta en 1993 como un ejemplo ilustrativo, "esta transformación es lo suficientemente sencilla como para ser realizada por un compilador o preprocesador" [5].

La generalidad de este método tiene un precio: aparte del sobrecoste de generar la copia, solamente uno de los hilos que quieran modificar la estructura a la vez realiza trabajo útil. Aún así, muchos métodos lock-free se basan en técnicas similares, realizando un trabajo en local (aunque no necesariamente partiendo de una copia) e intentando ponerlo en común con CAS.

En la actualidad disponemos de múltiples soluciones alternativas para crear, no solo estructuras de datos, sino programas lock-free. Entre ellas destacamos: *lock-free locks*, atómicos de varias direcciones de memoria, memoria transaccional y *lock-free locks revisited*.

2.2 Lock-free locks

Turek et al. [6] y Barnes [7] intentan mejorar la concurrencia de los métodos lock-free generales introduciendo el concepto de cooperación entre hilos. Nos referimos a estos métodos como *lock-free locks* porque permiten convertir cualquier estructura de datos basada en el uso de cerrojos o mútexes convencionales en una estructura lock-free.

Lo consiguen utilizando un método basado en la cooperación: se incluyen descriptores en los distintos elementos de la estructura de datos, con suficiente información sobre el estado de la operación que el hilo que los intenta modificar quiere realizar como para que otros hilos que quieran acceder a la vez puedan ayudar si el primer hilo se bloquea.

La cooperación entre hilos es más común en los algoritmos *wait-free*. Los algoritmos *wait-free* conllevan garantías de progreso más robustas que los lock-free, ya que imponen un límite finito de pasos que las operaciones individuales pueden tardar en completarse, eliminando como consecuencia el problema de la inanición. Esto se suele conseguir haciendo que los hilos comprueben continuamente si hay otros hilos que necesitan ayuda (porque no consiguen progresar en su operación) para que los hilos más rápidos ayuden a los más lentos antes de que estos últimos alcancen el límite de pasos [8, 9].

En el caso de *lock-free locks*, el hecho de guardar el estado frecuentemente implica guardar el contexto del proceso (*program counter* y registros) cada vez que se lleva a cabo una instrucción de lectura o escritura. Turek et al. [6] lo comparan con la implementación de una máquina virtual, en la que los descriptores apuntan al estado de la máquina.

Como se señala en "*Lock-Free Locks Revisited*" [10], esta transformación del código protegido por cerrojos para que guarde el contexto del proceso es difícil de realizar sin un compilador especial y hace que el algoritmo sea muy ineficiente cuando no se necesita ninguna ayuda. Al ser considerado poco práctico, se optaba por otros enfoques como los descritos en los apartados 2.3 o 2.4.

2.3 Atómicos de varias direcciones de memoria

Una de las razones por las que las implementaciones tradicionales de algoritmos lock-free son tan complicadas es el hecho de que las primitivas de sincronización disponibles en los sistemas actuales solamente permiten tratar una dirección de memoria a la vez.

En un intento de facilitar la elaboración de estos algoritmos altamente concurrentes se desarrolló el Double Compare-And-Swap (DCAS). Es una operación atómica que lee el contenido de dos direcciones de memoria independientes y solamente escribe el nuevo valor designado para cada dirección si ambos valores coinciden.

Si bien el uso de DCAS permite crear con mayor facilidad algoritmos no bloqueantes para algunas estructuras de datos como la *deque* [11, 12] (lista en la que solamente se permite insertar o eliminar elementos por los extremos [13]) y llegó a implementarse en hardware en el procesador Motorola 68040 [14], generalizar su uso a otras estructuras de datos sigue siendo una tarea complicada y la corrección de estas implementaciones es difícil de demostrar [15].

Se ha propuesto la utilización de instrucciones atómicas MCAS que accedan a aún más direcciones de memoria, argumentando que el DCAS no es suficiente ya que comparte muchas de las desventajas observadas en las implementaciones de estructuras con CAS [15]. Podemos imaginar que el DCAS es generalizable a más direcciones de memoria (TCAS, QCAS…), teniendo en cuenta que si en cualquiera de ellas el valor esperado difiere del leído no se escribirá en ninguna. Si el número de registros del procesador empieza a ser un límite a la hora de especificar las direcciones de memoria que forman parte de la operación, se pueden llegar a utilizar estructuras de datos adicionales para este propósito [16].

En implementaciones *hardware*, al tener que modificar varias direcciones de memoria atómicamente surge la posibilidad de generar *deadlocks* al bloquearlas ante modificaciones externas mientras se realiza la operación atómica. En "*Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations*" [17] se expone en detalle una manera de evitar estos *deadlocks* bloqueando las líneas de caché en un orden establecido, a cambio de un pequeño sobrecoste de memoria.

2.4 Memoria transaccional

El paradigma de la memoria transaccional permite crear regiones de código que obtienen propiedades de serialización y atomicidad análogas a las de las transacciones atómicas utilizadas en sistemas de bases de datos [18].

En la propuesta inicial de la memoria transaccional, cada transacción realiza sus cambios en memoria de manera tentativa, y solamente se intenta llevarlos a cabo al terminar la transacción, en la fase *commit*. En esta etapa, si se detectan conflictos con otras transacciones, la transacción falla y se descartan todos los cambios. De lo contrario, la transacción se hace efectiva para todos los hilos [19]. Esta propuesta inicial se centraba en la memoria transaccional como técnica para facilitar el desarrollo

de algoritmos lock-free, pero en la actualidad es un enfoque más apreciado por permitir al programador diseñar algoritmos concurrentes desde un alto nivel de abstracción, sin especificar en gran detalle la manera en la que la sincronización se deberá llevar a cabo.

Al implementar memoria transaccional en *hardware*, tanto en procesadores con protocolos de coherencia de caché basados en mecanismos de fisgoneo como en los basados en sistemas de directorio, los protocolos utilizados para detectar conflictos entre accesos individuales a memoria se deben extender para detectar conflictos entre transacciones. Además, se suelen incluir *buffers* para almacenar las modificaciones especulativas antes de ser puestas en común con los demás hilos [20].

Si bien es cierto que la memoria transaccional se originó como un modelo de programación lock-free, las implementaciones más comunes, como las basadas en *lock elision*, no buscan esta propiedad. En estas implementaciones cada transacción se intenta un número limitado de veces, tras lo cual se recurre a un cerrojo global como alternativa. Como consecuencia, pueden volver a aparecer problemas asociados con la sincronización a través de cerrojos. Es decir, en este caso HTM se puede utilizar para intentar mejorar la concurrencia de una estructura existente, pero no garantiza ninguna condición de progreso [1].

Existen implementaciones hardware de memoria transaccional en procesadores Intel [21] y ARM [22], aunque fueron deshabilitadas con firmware en procesadores de algunas generaciones de Intel debido a *bugs* [23] y fallos de seguridad [24].

2.5 *Lock-Free Locks Revisited*

En "*Lock-Free Locks Revisited*" se presenta un método práctico para generar estructuras de datos lock-free utilizando *lock-free locks*. Inspirándose en el modelo de los *lock-free locks*, se intenta conseguir un mecanismo más práctico y fácilmente aplicable. Para conseguir la garantía de progreso característica de los algoritmos lock-free, utilizan secciones críticas idempotentes.

2.5.1 Idempotencia

El concepto de idempotencia fue originado en el campo de las matemáticas para referirse a operaciones que producen el mismo resultado independientemente del número de veces que se apliquen tras la primera vez. En el contexto de la informática, la idempotencia se ha utilizado tradicionalmente a la hora de clasificar las características de operaciones en bases de datos o sistemas de archivos.

Más recientemente, podemos encontrar aplicaciones de la idempotencia en protocolos diseñados para obtener fiabilidad en redes con muchos fallos [25], orientados particularmente a la comunicación eficiente entre nodos de sistemas de computación de alto rendimiento.

También han habido avances en la generación automática de código idempotente a través de compiladores especializados para recuperar el estado del programa en caso

de error [26, 27]. A diferencia de "*Lock-Free Locks Revisited*", en vez de crear secciones idempotentes (es decir, en vez de convertir en idempotente código que normalmente no lo es) se intenta preservar las secciones naturalmente idempotentes del código durante la etapa de compilación, renunciando para ello a algunas optimizaciones pero permitiendo recuperar el estado del programa en caso de error simplemente repitiendo la última sección idempotente.

2.5.2 *Logs*

En "*Lock-Free Locks Revisited*", la idempotencia permite que si un hilo falla o queda bloqueado mientras accede a una sección crítica, otros hilos que intenten acceder puedan completar la operación del primero, liberando así el cerrojo asociado. Al ser una operación idempotente, el hecho de que varios hilos intenten ejecutar simultáneamente una sección crítica que puede estar a medias no es problemático.

Cuando un hilo adquiere uno de estos *lock-free locks*, establece un descriptor en el mismo que es visible para todos los hilos que intentan acceder. El descriptor incluye el fragmento de código que el hilo que adquirió el cerrojo quiere ejecutar junto con un *log* o registro. Dicho registro guarda los resultados de lecturas a direcciones de memoria compartida, reservas y liberaciones de memoria, en cierto modo anotando los resultados del progreso de todos los hilos que ayudan en la ejecución de la sección. De esta manera, varios hilos diferentes pueden ayudar en la ejecución del código del descriptor, leyendo resultados de ejecuciones parciales anteriores para conseguir el mismo efecto que si hubiese sido ejecutado por un solo hilo.

2.5.3 Sincronización de grano fino y grano grueso

El uso del registro hace que esta implementación esté especialmente orientada a la sincronización con cerrojos de grano fino: para secciones críticas demasiado largas (al utilizar sincronización de grano grueso) se debe tener en cuenta el sobrecoste de añadir entradas al registro y que todo hilo que intente ayudar volverá a completar la operación desde el principio de la sección crítica.

Mientras que la sincronización de grano grueso proporciona simplicidad, la sincronización de grano fino emplea un número superior de secciones críticas, pero más cortas y eficientes. Por ejemplo, a continuación se muestra el código de la operación de inserción de una lista ordenada con cerrojos de grano grueso:

Código 2.1: Inserción en lista con cerrojos de grano grueso

```
void insert(int data) {
  Node *new_node = new Node();
  new_node->data = data;
  Node *parent = head;

  list_lock.lock();

  Node *curr = head->next;
  while (curr != tail && curr->data < data) {
    parent = curr;
    curr = curr->next;
  }

  new_node->next = curr;
  parent->next = new_node;

  list_lock.unlock();
}
```

Se adquiere un único cerrojo para recorrer la lista e insertar el elemento. Suponemos que la lista tiene nodos `head` y `tail` permanentes en cada extremo y que se permiten elementos repetidos. En cambio, en una lista con cerrojos de grano fino se conseguiría de la siguiente manera:

Código 2.2: Inserción en lista con cerrojos de grano fino

```
void insert(int value) {
  Node *new_node = new Node();
  new_node->data = data;

  Node *parent = head;
  (parent->lock).lock();

  Node *curr = head->next;
  (curr->lock).lock();

  while (curr != tail && curr->data < data) {
    parent.unlock();
    parent = curr;
    curr = curr->next;
    (curr->lock).lock();
  }

  new_node->next = curr;
  parent->next = new_node;

  (parent->lock).unlock();
  (curr->lock).unlock();
}
```

Vemos que ahora se asigna un cerrojo a cada nodo. Se adquieren los cerrojos de los nodos recorridos con *hand-over-hand locking*, adquiriendo el cerrojo del nodo siguiente antes de liberar el del padre [1]. Esto es necesario porque al utilizar sincronización de grano fino aparecen nuevos problemas de sincronización. En este caso, el *hand-over-hand locking* evita que se inserte el nuevo nodo después de un nodo padre que ha sido eliminado. También existen optimizaciones que evitan tener que adquirir todos los cerrojos previos al lugar de inserción con *hand-over-hand locking* en listas de grano

fino [1], pero se busca presentar un ejemplo simple.

Esta última versión con cerrojos de grano fino, aunque significativamente más compleja, permite un mayor grado de concurrencia en escenarios de alta contención, donde un número elevado de hilos compiten por leer y modificar recursos compartidos. Al asociar cada sección crítica con los cerrojos de nodos específicos, se intenta que operaciones en partes diferentes de la lista pueden llevarse a cabo en paralelo.

Por otro lado, un mayor número de cerrojos conlleva un sobrecoste de memoria y tiempo que puede ser contraproducente en situaciones de baja contención, pues es común que las operaciones de grano fino necesiten adquirir más cerrojos que sus equivalentes de grano grueso.

2.6 Librería `flock`

Los autores de "*Lock-free Locks: Revisited*"[10] proporcionan la librería `flock` (el código fuente se encuentra disponible en `https://github.com/cmuparlay/flock`), que implementa los métodos descritos en el artículo. Tal y como describen, se trata de un enfoque práctico que facilita enormemente la creación de algoritmos lock-free a partir de estructuras basadas en cerrojos.

En concreto, proporcionan el cerrojo `flck::lock`. Con su método `try_lock`, al que se llama con un *thunk* (expresión *lambda* sin argumentos en C++), se intentará ejecutar el *thunk* una vez adquirido el cerrojo y se devolverá un booleano: `false` si no se puede acceder al cerrojo (tras ayudar, si es necesario, al hilo que lo esté utilizando) o el booleano devuelto por el *thunk* si se ha conseguido completar la operación.

Además de utilizando `try_lock`, se puede ejecutar código protegido por un cerrojo con `try_lock_result` y `with_lock`. `try_lock_result` devuelve un `std::optional` que tendrá el valor de retorno del *thunk* solamente si se ha conseguido obtener el cerrojo, permitiendo devolver valores que no sean booleanos. `with_lock` es similar a `try_lock` pero garantiza que siempre se adquirirá el cerrojo (en este sentido es equivalente a intentar `try_lock` en un bucle hasta conseguir acceder) y por ello siempre devuelve el valor de retorno del *thunk*.

En el artículo "*Lock-free Locks: Revisited*" se demuestra que la operación `try_lock` es lock-free siempre que los cerrojos estén simplemente anidados. Para ello se deben cumplir las siguientes condiciones:

1. Entre todos los cerrojos debe haber un orden parcial, de tal manera que siempre se adquieran en el mismo orden. Esto implica evitar *deadlocks* y ciclos entre cerrojos.

2. En cada `try_lock` solo puede haber un único `try_lock` directamente anidado. Este segundo `try_lock` también puede tener otro `try_lock` directamente anidado dentro, permitiendo recursividad.

Esta propiedad no llegó a demostrarse para la implementación de `with_lock`, que asegura que se obtendrá el cerrojo al intentar adquirirlo continuamente, pero indican que la demostración también debería ser adaptable a este método.

La clase `flck:lock` se debe utilizar junto al *wrapper* `flck::atomic<T>`. Los valores compartidos (exceptuando variables de solo lectura) a los que se acceda dentro de un *thunk* deben ser envueltos en `flck::atomic<T>`. Así, los resultados de operaciones de lectura o escritura a estas variables compartidas quedarán guardados en el registro asociado a la ejecución de la sección crítica. Para guardar también el progreso de operaciones de reserva o liberación de memoria dentro de la sección crítica, se deben utilizar los métodos `new_obj` y `retire` de una `flck::memory_pool<T>`.

3 Análisis de objetivos y metodología

Marcamos como objetivo comparar diferentes técnicas que facilitan el desarrollo de algoritmos concurrentes, creando estructuras de datos con ellas para verificar su facilidad de uso y posteriormente comparar su rendimiento. Antes de esto, reproducimos pruebas para confirmar que el enfoque de la librería `flock`, que es el primer algoritmo de sincronización que nos proponemos incorporar, es realmente viable para nuestro entorno de pruebas.

A la hora de llevar a cabo *benchmarks* que miden la eficiencia de estructuras de datos concurrentes, es conveniente utilizar un equipo con un número elevado de núcleos para observar de manera realista cómo escala el rendimiento ante escenarios de alta contención. Es por eso que se nos concedió acceso a nodos `tsx` del cluster de computación ECHO de la Universidad de Murcia.

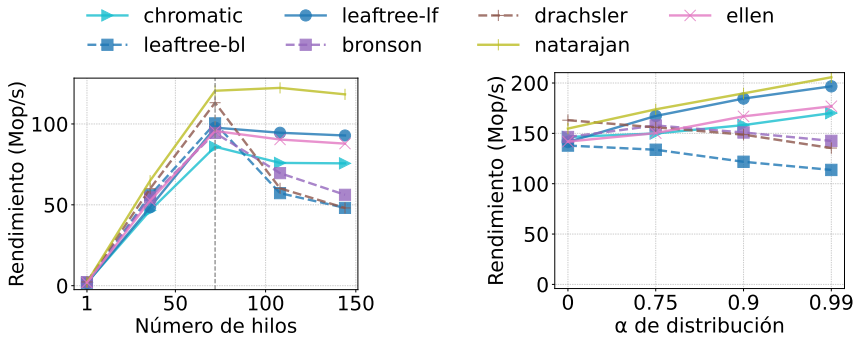
Se ha utilizado uno de los nodos `tsx` para todas las pruebas de este trabajo, contando con 2 CPU Intel Xeon E5-2695 v4, cada una con 18 núcleos a 2,10 GHz y 45 MB de caché L3. En total, utilizando *hyperthreading*, se dispone de 72 hilos. Las pruebas se realizaron en Ubuntu 18.04.6 LTS y se compilaron con g++ 9.2.0, con optimización `-O3`.

3.1 Reproducción de los resultados de *Lock-Free Locks Revisited*

Dado que el *artifact* que permite reproducir los resultados del artículo "*Lock-Free Locks Revisited*" se encuentra públicamente disponible en GitHub (`https://github.com/cmuparlay/flock/blob/ae/README.md`), decidimos reproducir los resultados obtenidos para confirmar que es factible, en términos de rendimiento, utilizar la librería `flock` en nuestro equipo.

Tras instalar las dependencias necesarias, utilizamos el *script* proporcionado en el repositorio mencionado anteriormente para realizar los mismos experimentos que generan las gráficas expuestas en el artículo. El *script* se modificó para adaptarlo al número de hilos de nuestros procesadores, realizando pruebas con 1, 36, 72, 108 y 144 hilos.

Aunque los autores recomiendan utilizar una máquina con al menos 64 núcleos para ejecutar los *benchmarks* de `flock`, en ECHO (de 36 núcleos) se obtuvieron resultados muy similares a los originales y que respaldan la promesa de concurrencia en condiciones de alta contención presentada en el artículo.



(a) 100.000 elementos, 50% modificaciones, $\alpha = 0{,}75$

(b) 100.000 elementos, 108 hilos, 5% modificaciones

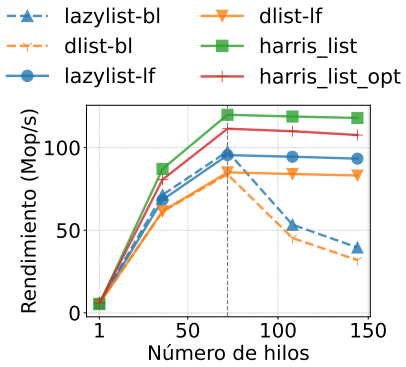
Figura 3.1: Rendimiento de árboles binarios de búsqueda con *benchmarks* de "*Lock-Free Locks Revisited*"

Como ejemplo se incluyen algunas de las gráficas más importantes. Las figuras 3.1a y 3.1b muestran el rendimiento de varias estructuras de datos del estado del arte en árboles binarios de búsqueda, donde las líneas discontinuas son estructuras bloqueantes y las continuas estructuras lock-free. El árbol de búsqueda binario que utiliza la librería `flock` es *leaftree-lf* y se incluye una versión de la misma estructura con cerrojos bloqueantes llamada *leaftree-bl*.

En el pie de foto de cada gráfica se indica el número de elementos de las estructuras, el porcentaje de operaciones que son modificaciones (las cuales pueden ser con igual probabilidad inserciones o eliminaciones, por lo que el tamaño de las estructuras se mantiene estable), y α, un parámetro que varía la distribución de los valores accedidos o modificados, donde $\alpha = 0$ es una distribución uniforme y valores de α más cercanos a 1 aumentan la desigualdad de las probabilidades y hacen que determinados valores se accedan con más frecuencia, provocando un mayor número de colisiones entre los hilos.

En las figuras 3.1a y 3.1b podemos ver que las estructuras lock-free mantienen un mejor rendimiento que el de las bloqueantes ante la contención, ya sea provocada por utilizar un número excesivo de hilos o un alto valor de α. En ambas la estructura *leaftree-lf* es mejor que las estructuras bloqueantes y varias de las otras estructuras lock-free al aumentar la contención.

En la figura 3.2 se encuentra el resultado de un *benchmark* similar, esta vez para listas simple y doblemente enlazadas. Las listas lock-free implementadas con `flock` son *lazylist-lf* (simplemente enlazada) y *dlist-lf* (doblemente enlazada). De manera similar a los resultados anteriores, las estructuras lock-free no experimentan una degradación del rendimiento tan importante como la de las bloqueantes. Esta vez las estructuras implementadas con `flock` son ligeramente peores que las otras implementaciones lock-free del estado del arte, pero este es un compromiso asumible debido a su mayor facilidad de implementación.



(a) 100 elementos, 5% modificaciones, $\alpha = 0{,}75$

Figura 3.2: Rendimiento de listas con *benchmarks* de "*Lock-Free Locks Revisited*"

3.2 Pruebas con *mcas-benchmarks*

Para poder comparar las estructuras desarrolladas, partimos del repositorio *mcas-benchmarks* (`https://github.com/kankava/mcas-benchmarks`), creado para el Trabajo de Fin de Máster "*Exploring the Efficiency of Multi-Word Compare-and-Swap*" [28]. Incluye varios *benchmarks* que permiten comparar la efectividad de estructuras de datos basadas en cerrojos, lock-free basadas en MCAS o lock-free basadas en CAS.

Los *benchmarks* miden el tiempo tardado en realizar un determinado número de operaciones sobre una de las estructuras de datos. El conjunto de operaciones se divide equitativamente entre los hilos lanzados. Se tienen en cuenta las siguientes estructuras de datos:

- *Multi-word object update* (mwobject): Una estructura con cuatro contadores, cada uno representado con una variable de 8 bytes, y que todos los hilos intentan incrementar de manera atómica. La intención del *benchmark* de esta estructura es que todos los hilos realicen lecturas y escrituras dirigidas a la misma línea de caché, para simular un escenario con la máxima contención posible.

- *Arrayswap*: Un *array* en el que se intercambia el dato contenido en dos índices elegidos aleatoriamente.

- Lista ordenada: Todas las implementaciones de esta estructura emplean listas doblemente enlazadas y permiten almacenar valores repetidos.

- *Hash map*: También conocida como *hash table*. La implementación definida para todos los mecanismos de sincronización utiliza un *array* de listas. Este *array* se accede según el *hash* del valor a buscar, insertar o eliminar, y posteriormente se recorre la lista encontrada en dicho índice. Al igual que la lista ordenada, admite elementos repetidos.

- *Deque*, pila y cola: la *deque* es una lista que admite inserciones y eliminaciones por los dos extremos, la pila solamente por un extremo y la cola admite inserciones por un extremo y eliminaciones por el otro.
- Árbol de búsqueda binario: Todas las implementaciones presentan nodos con dos hijos como máximo, admiten elementos repetidos y no se balancean automáticamente.

En el repositorio *mcas-benchmarks* ya se proporciona una implementación para cada una de estas estructuras con 3 algoritmos de sincronización diferentes [1]: un algoritmo de sincronización basado en cerrojos (nos referimos a estas implementaciones como "lock-based"), lock-free utilizando CAS y lock-free utilizando MCAS.

Como no existe una implementación *hardware* de MCAS, en *mcas-benchmarks* se definen versiones de las operaciones CAS, DCAS, TCAS y QCAS para su ejecución en el simulador gem5 [29]. También se incluyen alternativas que pueden ser ejecutadas en *hardware* real, pero donde todas las operaciones utilizan un cerrojo global para lograr la sincronización, por lo que no son lock-free. Utilizamos esta última versión en los *benchmarks*, como alternativa a simularlos en gem5, para poder realizar pruebas con un número elevado de hilos cuya ejecución no se alargue demasiado en el tiempo.

Finalmente, queremos destacar una pequeña modificación que llevamos a cabo en la medición del tiempo de los *mcas-benchmarks* originales. Inicialmente se utilizaba un temporizador activado antes de empezar a lanzar los hilos y que se pausaba cuando todos los hilos terminaban su ejecución. Esto puede ser problemático ya que mientras unos hilos todavía no se han lanzado, otros pueden avanzar y realizar operaciones. El temporizador tendrá en cuenta el tiempo total del hilo más lento, que incluye tiempo transcurrido desde que se inició el temporizador hasta que lanzó dicho hilo, además del tiempo gastado en realizar operaciones.

Para acercar el tiempo medido al tiempo real que los hilos emplean en realizar operaciones, probamos a incorporar una *sense-reversing centralized barrier* [30]. La barrera nos permite detener el progreso de los hilos hasta que todos hayan llegado a la barrera. Utilizamos una barrera antes de empezar a realizar operaciones, para iniciar el temporizador solamente cuando todos los hilos estén preparados, y otra a la que cada hilo llega cuando ha terminado de realizar sus operaciones asignadas. Tras comprobar que la *sense-reversing centralized barrier* introducía una gran sobrecarga de tiempo al ser utilizada con contención (algo que se advertía en el artículo original, pero que comprobamos experimentalmente), optamos por cambiar el tipo de barrera a `pthread_barrier`.

En el repositorio `https://github.com/AlvaroRGum/data-structure-benchmarks` publicamos el código fuente del programa final tras realizar las modificaciones descritas en este trabajo. De esta manera, el código queda a libre disposición para su utilización en investigaciones futuras.

[1] Excepto para las estructuras mwobject y arrayswap, para las que solamente se incluyen implementaciones basadas en cerrojos o en MCAS.

4 Diseño y resolución del trabajo realizado

Para los algoritmos evaluados en *mcas-benchmarks* desarrollamos estructuras basadas en mecanismos de sincronización no presentes todavía en los *benchmarks*. En concreto, a continuación se expone el desarrollo de estructuras basadas en *Lock-Free Locks* (con la librería `flock`), memoria transaccional y variantes de MCAS. Después, tratamos los errores encontrados en los algoritmos originales presentes en *mcas-benchmarks*.

4.1 Incorporación de `flock`

Inicialmente, priorizamos la incorporación de la librería `flock` en los *benchmarks*. El código de `flock` disponible en el repositorio indicado en "*Lock-free Locks: Revisited*" solamente se puede utilizar sin modificar si los hilos lanzados en el *benchmark* son manejados con ParlayLib [31]. ParlayLib es una librería que facilita la programación de algoritmos paralelos con funciones que permiten, entre otras cosas, ejecutar secciones de código entre varios hilos (de manera similar a OpenMP), modificar el *scheduling* de los hilos o generar números aleatorios para algoritmos paralelos. `flock` hace uso del algoritmo de reserva de memoria de ParlayLib, integrado con `jemalloc` para gestionar la memoria reservada dentro de los cerrojos.

En la versión original, la reserva y liberación de memoria depende del uso de un `worker_id` proporcionado por ParlayLib para funcionar correctamente. Es por eso que utilizamos una versión modificada, disponible en `https://github.com/cmuparlay/verlib/tree/main/include/flock`. Esta versión permite utilizar `flock` con hilos que no se gestionan con ParlayLib (en nuestro caso los gestionamos con `std::thread`).

4.1.1 Lista ordenada y *hash map*

Los autores de `flock` crearon varias estructuras de datos con la librería. En *mcas-benchmarks* ya se realizaban pruebas con algunas de estas estructuras de datos (lista ordenada doblemente enlazada, *hash table*) y pudimos incluirlas, tras unas pequeñas modificaciones, en nuestros *benchmarks*. En concreto, partiendo de las estructuras disponibles en la versión más reciente del repositorio, realizamos modificaciones para permitir la inserción de elementos repetidos (dado que las estructuras ya presentes en *mcas-benchmarks* también lo permitían), adaptar la interfaz de las clases a la utilizada en nuestros *benchmarks* y guardar valores individuales en vez de pares clave-valor.

La implementación del resto de estructuras parte de las estructuras *lock-based* ya presentes en *mcas-benchmarks*, como se detalla a continuación.

4.1.2 mwobject y arrayswap

La implementación del *benchmark* mwobject con `flock` se basa en la implementación *lock-based*, con un solo cerrojo para los cuatro contadores. Aparte de sustituir el cerrojo de la librería estándar por un `flck::lock`, incrementamos punteros (en particular, del tipo `char *`) en vez de variables de tipo `long`, a diferencia de todas las demás implementaciones. Esto se debe a que el tipo `flck::atomic` solamente admite tipos con un tamaño máximo de 4 bytes o punteros, y el tipo `long` tiene un tamaño de 8 bytes en sistemas Linux de 64 bits.

Para `arrayswap` realizamos una implementación prácticamente idéntica a la lock-based, en la que se sustituyen los cerrojos utilizados por cerrojos de la librería `flock`. Al igual que la implementación lock-based original, se utiliza un cerrojo por elemento del *array*.

4.1.3 *Deque*, pila y cola

Creamos la *deque* y a partir de ella derivamos implementaciones para la pila y la cola. Esto se hacía también en las estructuras originales basadas en MCAS y cerrojos, basta con restringir el acceso a métodos de la *deque* para convertirla en una pila o cola.

La *deque* lock-based original utiliza un único cerrojo, que sustituimos por un `flck::lock`. A modo de ejemplo, a continuación se muestra el código con `flock` que define los nodos de la *deque*:

Código 4.1: Deque con flock

```
class Deque {
private:
  struct Node {
    // No se utiliza flck::atomic<> para data porque es de sólo lectura
    int data;
    flck::atomic<Node *> prev;
    flck::atomic<Node *> next;
    Node() = default;
    Node(int data) : data(data), prev(nullptr), next(nullptr){};
  };

  Node *head;
  Node *tail;

  flck::lock deque_lock;
  flck::memory_pool<Node> node_pool;
```

Como se mencionaba en la sección 2.6, empleamos el *wrapper* `flck::atomic<T>` para variables que se deben modificar dentro de las secciones críticas. Una vez definidos los nodos, `push_front` se podría implementar de la siguiente manera:

Código 4.2: push_front de deque con flock

```
1  // push_left
2  void push_front(int data) {
3    flck::with_epoch([=] {
4      deque_lock.with_lock([=] {
5        Node *new_node = node_pool.new_obj(data);
6        Node *hn = (head->next).load();
7
8        new_node->next = hn;
9        new_node->prev = head;
10
11       hn->prev = new_node;
12       head->next = new_node;
13       return true;
14     });
15   });
16 }
```

Se llama a `with_lock`, que siempre adquiere el cerrojo y no puede fallar, para ejecutar el código de la sección crítica tras adquirir el cerrojo de la *deque*. Para leer el valor de una variable de tipo `flck::atomic` (línea 6) se emplea el método `load`, mientras que el operador de asignación utilizado en las líneas 8 a 12 está *overloaded* al método `store` de `flck::atomic` y se sustituye automáticamente por una llamada a `.store(valor)`. El nuevo nodo se debe crear llamando al método `new_obj` de una `node_pool`, como sucede en la línea 5.

Además, toda la operación está envuelta en una llamada a `flck::with_epoch`. Este *wrapper* se añadió en las últimas versiones de `flock` y es necesario para el correcto funcionamiento del algoritmo de reclamación de memoria de `flock`.

4.1.4 Árbol de búsqueda binario

Entre las estructuras creadas para el artículo "*Lock-free Locks: Revisited*" se encuentran varios tipos de Binary Search Tree (BST), pero todos ellos incluyen modificaciones que los alejan demasiado de la estructura ya presente en nuestros *benchmarks* para otros algoritmos de sincronización. Por ejemplo, presentan varias versiones de *leaf-oriented* BST, también llamado *external* BST o *leaf tree* [32]. Estos árboles *leaf-oriented* solamente guardan elementos en los nodos hoja.

Desarrollamos una implementación basada en cerrojos de grano fino, con un cerrojo por nodo, siguiendo el esquema del BST implementado con MCAS ya presente en *mcas-benchmarks*. Aunque completamos una implementación funcional a primera vista, posteriormente se decidió que era demasiado complicada. Uno de los objetivos de `flock` es permitir la creación de estructuras *lock-free* con un proceso simple, normalmente partiendo de estructuras basadas en cerrojos, por lo que decidimos intentar llevar a cabo otra implementación que demuestre la utilidad de `flock` para este propósito.

Exploramos alternativas simples de BST basados en cerrojos de grano fino [33], pero encontramos problemas a la hora de implementar recorridos con *hand-over-hand locking* con `flock`.

En un recorrido con *hand-over-hand locking*, siempre se adquiere el cerrojo del si-

guiente nodo antes de liberar el del nodo actual. Este tipo de recorridos se vuelve incómodo al utilizar cerrojos con la clase `flck::lock`, ya que el código a ejecutar en la sección crítica se tiene que proporcionar dentro de una función sin argumentos. Por ejemplo, el código en C++ para implementar el método `insert` de un BST con `flock` y *hand-over-hand locking* podría ser el siguiente:

Código 4.3: insert de BST con flock y hand-over-hand locking

```
1  void insert(const int value) {
2    flck::with_epoch([=] {
3      root_lock.with_lock([=] {
4        Node *root_load = root.load();
5        if (root_load == nullptr) {
6          root = node_pool.new_obj(value);
7          return true;
8        }
9        root_load->with_lock([=] {
10         root_lock.unlock();
11         recursive_insert(root_load, value);
12         return true;
13       });
14       return true;
15     });
16   });
17 }
18
19 void recursive_insert(Node *node, int new_value) {
20   Node *parent = node;
21   Node *curr;
22   if (new_value < parent->value.load()) {
23     curr = parent->left.load();
24     if (curr == nullptr) {
25       parent->left = node_pool.new_obj(new_value);
26       parent->unlock();
27       return;
28     }
29   } else {
30     curr = parent->right.load();
31     if (curr == nullptr) {
32       parent->right = node_pool.new_obj(new_value);
33       parent->unlock();
34       return;
35     }
36   }
37   curr->with_lock([=] {
38     parent->unlock();
39     recursive_insert(curr, new_value);
40     return true;
41   });
42 }
```

Como se puede apreciar, parece que es necesario utilizar funciones recursivas. Para otras operaciones más complejas, como `remove`, el código puede complicarse dado que se debe utilizar recursividad mientras se adquieren los cerrojos de hasta 5 nodos a la vez. Cabe mencionar que ninguno de los BST implementados para el artículo "*Lock-free Locks: Revisited*" utiliza el método `unlock` ni realiza recorridos con *hand-over-hand locking*. En los *leaf-oriented* BST, por ejemplo, no es necesario utilizar *hand-over-hand locking*.

Se decidió optar por una alternativa más simple, adaptando el BST con cerrojos de grano grueso ya presente en los *benchmarks*. Esta estructura tiene solamente un cerrojo que se utiliza para todo el árbol. Se entiende que no dará lugar a la estructura más óptima, pero nos permitirá probar la efectividad de `flck` para estructuras de grano grueso.

4.2 Incorporación de memoria transaccional

Utilizamos las extensiones TSX de Intel para acceder a una implementación de HTM, ya que disponemos de procesadores Intel Xeon E5-2695 en ECHO donde esta característica se puede activar a pesar de haber sido deshabilitada.

Estas extensiones ofrecen la interfaz Hardware Lock Elision (HLE) y la más reciente Restricted Transactional Memory (RTM). Hemos utilizado RTM porque permite un mayor grado de flexibilidad a la hora de especificar el código ejecutado cuando se falla demasiadas veces al intentar ejecutar una sección de memoria transaccional [34].

Para delimitar las secciones que utilizan sincronización con HTM definimos las funciones `TM_BEGIN` y `TM_END`. Internamente, estas funciones utilizan los *intrinsics* de Intel `_xbegin` y `_xend` para marcar el inicio y el final de una sección RTM, respectivamente.

En `TM_BEGIN` especificamos el comportamiento a seguir ante fallos al intentar llevar a cabo una transacción. Al llegar a 6 intentos fallidos se recurre a un *spinlock* global para la sincronización. Se ha incluido opcionalmente (se puede activar o desactivar al compilar el código) un tiempo de espera o *backoff* exponencial entre intentos.

Además de `TM_BEGIN`, utilizamos `TM_BEGIN_NO_CONFLICT` para iniciar transacciones que no se reintentan si encuentran un fallo por conflicto. Esta función se explica con más detalle en la sección 4.3.

Hemos incorporado dos versiones de estructuras de datos con HTM: la versión "HTM lock", que parte de las estructuras lock-based, y la versión "HTM MCAS", que adapta las estructuras basadas en MCAS. De esta manera, tenemos varias versiones de estructuras con HTM que nos permiten comprobar cuál es el tipo de recorrido o algoritmo que se beneficia más de utilizar HTM.

4.2.1 HTM lock

Para conseguir estructuras con sincronización basada en HTM a partir de estructuras lock-based hemos sustituido las instrucciones que delimitan cada sección crítica protegida con un cerrojo por `TM_BEGIN` y `TM_END`. Por ejemplo, la operación `push_front` de la *deque* se implementó de la siguiente manera para HTM lock:

Código 4.4: push_front de deque con HTM lock

```
1 void push_front(int const& data) {
2   Node *new_node = new Node();
3   new_node->data = data;
4
5   TM_BEGIN(0);
6
7   new_node->next = head->next;
8   new_node->prev = head;
9
10  head->next->prev = new_node;
11  head->next = new_node;
12
13  TM_END(0);
14 }
```

El código desde la línea 7 a la 11 estaba originalmente protegido por un cerrojo global. Ahora la memoria transaccional es la encargada de garantizar que esta sección es percibida como atómica por los demás hilos.

Las funciones `TM_BEGIN` y `TM_END` tienen un entero como argumento, pero se utiliza únicamente por propósitos de *debugging* y no tiene ningún efecto en la ejecución de la transacción.

4.2.2 HTM MCAS

Las operaciones de inserción o eliminación de las estructuras originales basadas en MCAS contienen un bucle que engloba la mayor parte del código del método. En cada iteración de este bucle se leen valores de memoria y, si es necesario, se recorre la estructura hasta llegar al lugar donde se debe insertar o eliminar un nodo. Es entonces cuando se realiza un MCAS, que puede fallar si los datos leídos al principio del bucle han sido modificados o, más generalmente, se han realizado cambios que hacen que la dirección de inserción o eliminación a la que se ha llegado pierda su validez. El bucle exterior tiene el propósito de reintentar la operación hasta que el MCAS se lleve a cabo con éxito.

Así, para una operación como `pop_front` de la *deque* tendríamos originalmente el código del listado 4.5. En la llamada a TCAS se pasan como argumento tres veces un puntero a una dirección de memoria, el valor esperado y el valor a escribir. Este TCAS comprueba si el valor de `LeftHat` sigue siendo `lh`, el de `lh->R` sigue siendo `lhR` y el de `lh->L` sigue siendo `lhL`, para después cambiar el valor de `LeftHat` a `lhR`, el de `lh->R` a `lh` y el de `lh->L` a `lh` solamente si la comprobación anterior es correcta.

El equivalente con HTM se consigue sustituyendo la llamada a MCAS por las modificaciones que se realizarán si el MCAS no falla y cambiando el bucle exterior por los delimitadores de una sección HTM.

Código 4.5: pop_front de deque con MCAS

```
1  int pop_front() {
2    while (true) {
3      Node *lh = LeftHat;
4      Node *lhL = lh->L;
5      Node *lhR = lh->R;
6
7      if (lhL == lh) {
8        if (LeftHat == lh)
9          return -1;
10     } else {
11       if (tcas(
12               reinterpret_cast<uint64_t *>(&LeftHat),
13               reinterpret_cast<uint64_t>(lh),
14               reinterpret_cast<uint64_t>(lhR),
15
16               reinterpret_cast<uint64_t *>(&lh->R),
17               reinterpret_cast<uint64_t>(lhR),
18               reinterpret_cast<uint64_t>(lh),
19
20               reinterpret_cast<uint64_t *>(&lh->L),
21               reinterpret_cast<uint64_t>(lhL),
22               reinterpret_cast<uint64_t>(lh))) {
23         int result = lh->data;
24         return result;
25       }
26     }
27   }
28 }
```

Utilizar HTM nos asegura que la fase *compare* de un CAS es innecesaria, pues las escrituras de otros hilos que puedan invalidar los datos leídos (en este caso los valores leídos en las líneas 3, 4 y 5 del listado anterior) harían que se descartase y volviese a intentar la transacción.

De esta manera, el código del listado anterior se podría adaptar como se indica a continuación:

Código 4.6: pop_front de deque con HTM MCAS

```
1  int pop_front() {
2    int result;
3
4    TM_BEGIN(2);
5    Node* lh = LeftHat;
6    Node* lhL = lh->L;
7    Node* lhR = lh->R;
8    if (lhL == lh) {
9      TM_END(2);
10     return -1;
11   } else {
12     LeftHat = lhR;
13     lh->R = lh;
14     lh->L = lh;
15     result = lh->data;
16   }
17   TM_END(2);
18   return result;
19 }
```

4.3 Incorporación de nuevas versiones de MCAS

Al no disponer de una implementación *hardware* de MCAS, en el repositorio original se proporciona la opción de emular el comportamiento de MCAS con un cerrojo global si no se utiliza gem5. Partiendo de las estructuras de datos *lockfree* que utilizan MCAS creamos otra versión en la que la atomicidad del MCAS está garantizada gracias a la HTM. Es decir, el código que define las operaciones de estas estructuras sigue siendo el mismo que el de la versión original con MCAS, pero para esta nueva variante el MCAS está implementado internamente utilizando HTM. La estructura resultante es diferente de la HTM MCAS descrita en la sección 4.2.2 que adapta las estructuras con MCAS utilizando HTM pero no utiliza MCAS en la versión final.

Con esto obtenemos dos familias de estructuras de datos distintas, a las que llamamos "mcas-lock" (la original) y "mcas-htm" (la basada en HTM). Por ejemplo, el código que define el DCAS de la versión original es el siguiente:

Código 4.7: DCAS original

```
1 uint64_t dcas(uint64_t* addr0, uint64_t old0, uint64_t new0,
2                uint64_t* addr1, uint64_t old1, uint64_t new1) {
3   std::lock_guard<std::mutex> lock(global_cas_lock);
4   {
5     if ((*addr0 == old0) && (*addr1 == old1)) {
6       *addr0 = new0;
7       *addr1 = new1;
8       return true;
9     }
10    return false;
11  }
12 }
```

Para implementar esta operación con "mcas-htm" las instrucciones protegidas por el cerrojo se delimitan como una transacción con `TM_BEGIN` y `TM_END`:

Código 4.8: DCAS con HTM

```
1 uint64_t dcas(uint64_t* addr0, uint64_t old0, uint64_t new0,
2                uint64_t* addr1, uint64_t old1, uint64_t new1) {
3   if (!TM_BEGIN_NO_CONFLICT(1)) {
4     return false;
5   }
6
7   if ((*addr0 == old0) && (*addr1 == old1)) {
8     *addr0 = new0;
9     *addr1 = new1;
10    TM_END(1);
11    return true;
12  }
13  TM_END(1);
14  return false;
15 }
```

Se puede apreciar que en vez de `TM_BEGIN` se utiliza `TM_BEGIN_NO_CONFLICT`. En nuestra primera versión, en lugar del código de las líneas 3 a 5, se llamaba simplemente a `TM_BEGIN(1);`, pero nos dimos cuenta de que podíamos realizar una pequeña optimización de esta manera.

Se trata de una versión de `TM_BEGIN` que devuelve `false` si la transacción falla por conflicto con otros hilos y `true` en cualquier otro caso. Si la transacción falla por conflicto con otro hilo sabemos que esto ocurre porque se ha modificado alguna de las posiciones de memoria que se van a comparar con sus valores "viejos", por lo que es muy posible que el MCAS falle.

Dicho de otra manera, es una heurística que evita desperdiciar tiempo en reintentar una transacción cuando se sabe que, al haber fallado anteriormente por conflicto, el MCAS exterior también va a fallar aunque se complete la transacción (va a encontrar valores diferentes al comparar con los "viejos").

Además, incluimos una variante de cada una de estas dos versiones en la que se intenta eliminar el salto condicional creado por el `if` que comprueba si los valores leídos son iguales a los esperados. Por ejemplo, el DCAS de esta variante para la versión mcas-lock es el siguiente:

Código 4.9: DCAS sin salto condicional

```
uint64_t dcas(uint64_t* addr0, uint64_t old0, uint64_t new0,
              uint64_t* addr1, uint64_t old1, uint64_t new1) {
  bool match = false;
  uint64_t trash = 0;

  const uint64_t i_trash = (uint64_t)&trash;
  const uint64_t i_addr0 = (uint64_t)addr0;
  const uint64_t i_addr1 = (uint64_t)addr1;
  {
    std::lock_guard<std::mutex> lock(global_cas_lock);
    match = (*addr0 == old0) & (*addr1 == old1);
    *(uint64_t*)((i_addr0 * match) + (i_trash * (1 - match))) = new0;
    *(uint64_t*)((i_addr1 * match) + (i_trash * (1 - match))) = new1;
  }
  return match;
}
```

Esta variante hace que siempre se escriban los nuevos valores, pero la dirección de memoria destino se cambia a la de una variable local `thrash` si alguno de los valores esperados difiere del leído. Se obtiene el mismo resultado que con la versión original, donde los cambios solamente son visibles para otros hilos si los valores `new` y `old` coinciden. Nos referimos a estas variantes como "mcas-lock-no-if" y "mcas-htm-no-if".

En resumen, se dispone de cuatro variaciones de MCAS: mcas-lock, mcas-htm, mcas-lock-no-if y mcas-htm-no-if.

4.4 Errores de las estructuras originales

Durante la creación de las estructuras descritas anteriormente, al partir de estructuras ya presentes en *mcas-benchmarks* basadas en MCAS, CAS o lock-based, encontramos algunos errores de implementación. Los detallamos a continuación para facilitar el trabajo de quienes tengan intención de utilizarlas en el futuro.

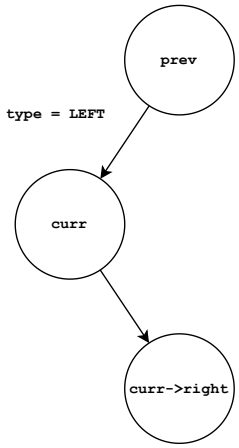


Figura 4.1: Fallo de BST MCAS al eliminar `curr`

4.4.1 Árbol de búsqueda binario basado en MCAS

El código de esta implementación procede del apéndice de "*A hardware implementation of the MCAS synchronization primitive*" [16]. Encontramos un error que provoca fallos a la hora de eliminar nodos con un solo nodo hijo. Para la situación de la figura 4.1, donde se llega hasta un nodo `curr` que contiene el valor a eliminar, se ejecuta el siguiente código:

Código 4.10: Código erróneo de MCAS BST

```
1          } else { // subtree with one child
2            if (type == LEFT) {
3              if (curr->left) {
4
5              ...
6
7              } else {
8                Node *present = curr;
9                Node *temp = nullptr;
10               {
11                 if (dcas(reinterpret_cast<uint64_t *>(&prev->right),
12                          reinterpret_cast<uint64_t>(present),
13                          reinterpret_cast<uint64_t>(present->right),
14
15                          reinterpret_cast<uint64_t *>(&curr),
16                          reinterpret_cast<uint64_t>(present),
17                          reinterpret_cast<uint64_t>(temp))) return;
18               }
```

La variable `type` indica cuál es el puntero del nodo padre que apunta al nodo a eliminar. Se realiza un DCAS en el que se comprueba si el puntero derecho del nodo padre sigue apuntando al hijo (líneas 11 y 12), pero al nodo hijo se accede por el puntero izquierdo del nodo padre, por lo que este DCAS siempre falla. Solucionamos el error cambiando la dirección proporcionada en la línea 11 por `&prev->left`.

Se comete un error similar para la situación inversa, donde se llega al nodo a eliminar por el puntero derecho del padre y el nodo a eliminar tiene un hijo a su izquierda. De

manera análoga al error anterior, lo solucionamos cambiando el primer argumento del DCAS de `&prev->left` a `&prev->right`.

Finalmente, existe otro error en el método `insert` que no proviene del código original de "*A hardware implementation of the MCAS synchronization primitive*", sino de la adaptación realizada para incluir este código en *mcas-benchmarks*. En particular, al inicio de `insert` se comprueba con un `if` si el árbol está vacío, y si lo está se intenta añadir un nuevo nodo como raíz con un CAS. Si el CAS tiene éxito, se regresa de la llamada a `insert`, pero si no lo tiene se continúa e intenta recorrer el árbol para insertar el nuevo nodo, aunque puede que el árbol esté vacío, en cuyo caso se produce el error. Se puede solucionar cambiando el `if` que comprueba si existe el nodo raíz por un `while`.

4.4.2 Estructuras lock-based

Estos errores provienen del código creado originalmente para *mcas-benchmarks*. Por una parte, el BST lock-based contiene un error en el código que inserta un nuevo nodo, que se muestra en el listado 4.11.

Código 4.11: Código erróneo de BST lock-based

```
1 void insert(const int value) {
2
3 ...
4
5     while (curr) {
6       prev = curr;
7       if (value < curr->value) {
8         curr = curr->left;
9         type = LEFT;
10      } else {
11        curr = curr->right;
12        type = RIGHT;
13      }
14
15      if (type == LEFT) {
16        prev->left = new_node;
17      } else {
18        prev->right = new_node;
19      }
20    }
21  }
22 }
```

El `if` final que inserta el nuevo nodo en `prev->left` o `prev->right` solamente debería realizarse al terminar el bucle `while`. Como consecuencia de este error, la implementación lock-based solamente podría tener tres nodos como máximo. Para arreglarlo basta con mover este último bloque `if` fuera del bucle `while`.

También se encontró un error en la `deque`, en el que no se establecían correctamente los punteros de todos los nodos tras una eliminación. En concreto, el código erróneo en la operación `pop_front` se muestra a continuación:

Código 4.12: Código erróneo de deque lock-based

```
1  int pop_front() {
2    int data = -1;
3    bool found = false;
4    Node *tmp;
5    {
6      std::lock_guard<std::mutex> lock(deque_lock);
7      if (head->next != tail) {
8        data = head->next->data;
9        tmp = head->next;
10       head->next = head->next->next;
11       found = true;
12     }
13   }
14   if (found)
15     delete tmp;
16   return data;
17 }
```

No se establece correctamente el valor del puntero `prev` del nodo siguiente al eliminado. Tras ejecutar este código, el puntero `prev` de este nodo seguiría apuntando al nodo eliminado, cuando debería apuntar al nodo anterior al eliminado. Podemos comprobar que se obtiene un error con el siguiente programa, por ejemplo:

Código 4.13: Programa que provoca error en deque lock-based

```
1 deque.push_front(1);
2 std::cout << deque.pop_front() << std::endl;
3
4 deque.push_back(2);
5 deque.push_back(3);
6 std::cout << deque.pop_back() << std::endl;
7 std::cout << deque.pop_front() << std::endl;
```

La salida resultante de ejecutar este programa indica que se han obtenido los valores 1, 3 y -1. Este error se puede solucionar añadiendo `head->next->prev = head;` después de la línea 10 del listado 4.12. El error también está presente en el método `pop_back` y se arregla de manera análoga, esta vez con la línea `tail->prev->next = tail;`.

4.4.3 `get_min` de árboles de búsqueda binarios

El código original de la operación `get_min` de los árboles de búsqueda binarios recorre los punteros desde la raíz hacia la izquierda, hacia valores más pequeños, hasta encontrar un nodo sin hijo izquierdo, donde se continúa el recorrido por su hijo derecho si es posible. Durante todo el recorrido se van comparando los valores de los nodos para encontrar el valor mínimo.

No encontramos una explicación para este tipo de recorrido cuando para encontrar el valor mínimo de un árbol de búsqueda binario basta con seguir todos los punteros a la izquierda desde la raíz y tomar el valor del primer nodo que no tenga un hijo izquierdo. Hemos modificado todos los árboles binarios de búsqueda para que utilicen este último recorrido.

4.4.4 Árbol de búsqueda lock-free basado en CAS

Aunque no utilizamos el árbol de búsqueda lock-free basado en CAS como base para ninguna implementación, hemos observado que no permite insertar elementos repetidos. Esto le otorga una ventaja injusta en los *benchmarks* respecto a los demás árboles de búsqueda, ya que siempre contendrá un número menor o igual de elementos para la misma secuencia de inserciones. Este error también estaba presente en "*A hardware implementation of the MCAS synchronization primitive*".

No fuimos capaces de modificar el árbol para que permita elementos repetidos mientras se conserva el resto de su funcionalidad, por lo que mantenemos la estructura pero tenemos en cuenta este dato a la hora de interpretar los resultados de los *benchmarks*.

5 Pruebas y resultados

Intentamos medir el tiempo tardado en realizar 10^6 operaciones con todos los algoritmos de sincronización y estructuras de datos, ya que hemos comprobado que es un número suficiente para mantener un error despreciable con la mayoría de algoritmos, pero aparecieron fallos adicionales en algunas de las estructuras originales que no fuimos capaces de resolver y que nos obligaron a modificar estas condiciones.

Por una parte, encontramos errores al utilizar un número elevado de hilos con la lista ordenada y la *hash table* basadas en MCAS. No fuimos capaces de conseguir ejecuciones sin errores para un número suficientemente elevado de operaciones, por lo que la lista ordenada y la *hash table* que se basan en el algoritmo que utiliza MCAS (mcas-lock), incluyendo sus variantes (mcas-lock-no-if, mcas-htm y mcas-htm-no-if), no se incluyen en las pruebas.

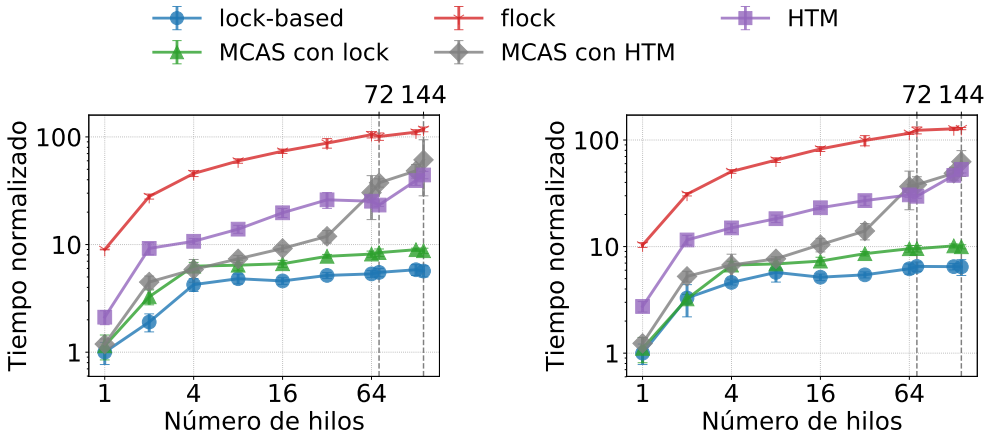
Además, aparecen errores en la implementación lock-free basada en CAS de la cola al intentar realizar 10^6 operaciones con un número elevado de hilos, pero este error desaparece al reducir el número de operaciones, por lo que solamente ejecutamos 10^5 operaciones en la prueba dedicada a la cola.

Variamos el número de hilos entre los que se reparten las operaciones, desde 1 hasta el doble del número de hilos lógicos del procesador. Realizamos más pruebas al inicio, con pocos hilos, para visualizar mejor la escalabilidad de los algoritmos mientras no se alcanza el máximo de hilos lógicos. También indicamos el punto en el que se alcanza el número de hilos lógicos del procesador (72) y el doble de hilos lógicos (144) en cada gráfica.

Las pruebas se realizaron con la versión de HTM sin *backoff* y la versión con *backoff* exponencial. Para cada estructura y algoritmo se realizan 5 ejecuciones de calentamiento y se calcula la media de las 20 posteriores. El tiempo resultante se normaliza respecto al de la versión lock-based con 1 hilo y se incluyen barras de error con la desviación típica obtenida en las ejecuciones. Utilizamos escala logarítmica en ambos ejes para visualizar con mayor facilidad las diferencias de tiempo entre los algoritmos en la parte inferior de las gráficas.

En mwobject y arrayswap, al ser algoritmos muy simples, no tiene sentido introducir varias versiones de algunos mecanismos de sincronización. Por ejemplo, las versiones de HTM que adaptan las estructuras basadas en cerrojos y MCAS tendrían la misma implementación, por lo que solamente se incluye una versión con HTM. Tampoco se incluye una versión lock-free con CAS simple para estos algoritmos.

Para la lista ordenada, el *hash map* y el árbol de búsqueda binario se pueden realizar lecturas además de modificaciones, por lo que realizamos una prueba con solo lecturas,



(a) mwobject sin *backoff* en HTM **(b)** mwobject con *backoff* en HTM

Figura 5.1: Tiempo normalizado de mwobject

otra con solo modificaciones (50% inserciones y 50% eliminaciones) y una última con un 80% de lecturas, 5% de inserciones y 5% de eliminaciones, a la que nos referimos como *mixed*.

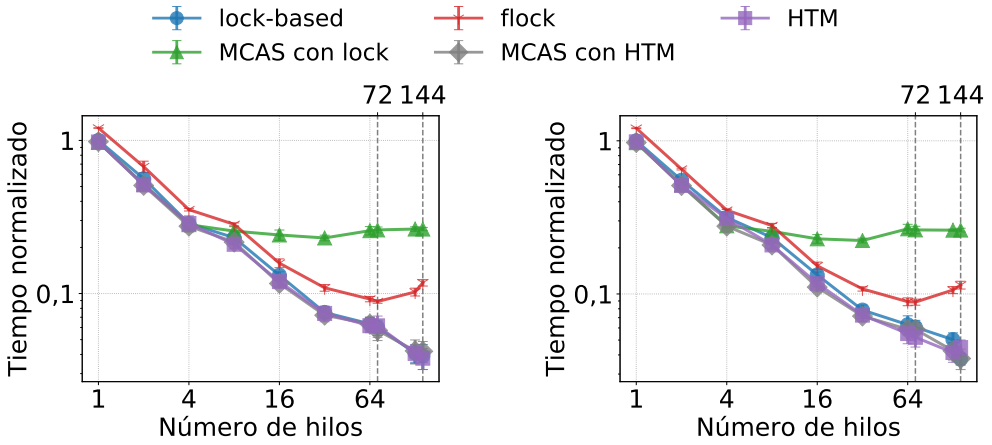
Tras obtener los resultados de estas pruebas, se observó que el uso de variantes sin `if` de los mecanismos basados en MCAS no afecta al tiempo de ejecución de forma notable. La muy pequeña diferencia en tiempos de ejecución que encontramos no los mejora ni empeora consistentemente en ninguna de las pruebas. En consecuencia, solamente se incluyen resultados de los algoritmos originales con `if` para mejorar la claridad de las gráficas.

5.1 mwobject

En la figura 5.1 vemos que la implementación más rápida, incluso con contención, es la lock-based, obteniendo un rendimiento similar al de la basada en MCAS con cerrojos. Esta última versión utiliza un cerrojo global, por lo que tiene sentido que presente un rendimiento parecido al de la lock-based, si bien es menos eficiente porque el MCAS puede fallar.

Se obtiene el peor tiempo con `flock` para todos los números de hilos. Esto nos da una idea de la sobrecarga de rendimiento que introduce `flock` en casos simples donde es difícil (imposible en este caso extremo) conseguir modificaciones concurrentes.

Los tiempos de la estructura basada en MCAS HTM tienen un mayor valor de error al aumentar el número de hilos y son ligeramente peores para alta contención que los de la versión con HTM simple.



(a) arrayswap sin *backoff* en HTM **(b)** arrayswap con *backoff* en HTM

Figura 5.2: Tiempo normalizado de arrayswap

5.2 arrayswap

En la figura 5.2 mostramos los resultados del *benchmark* para arrayswap. Podemos ver que se produce un aumento logarítmico del rendimiento (aumenta mucho al principio y menos al acercarse al límite de hilos, produciendo una evolución aparentemente lineal en gráficas con escala logarítmica) incluso tras superar el máximo de 72 hilos para las estructuras basadas en HTM, lock-based y basadas en MCAS con HTM.

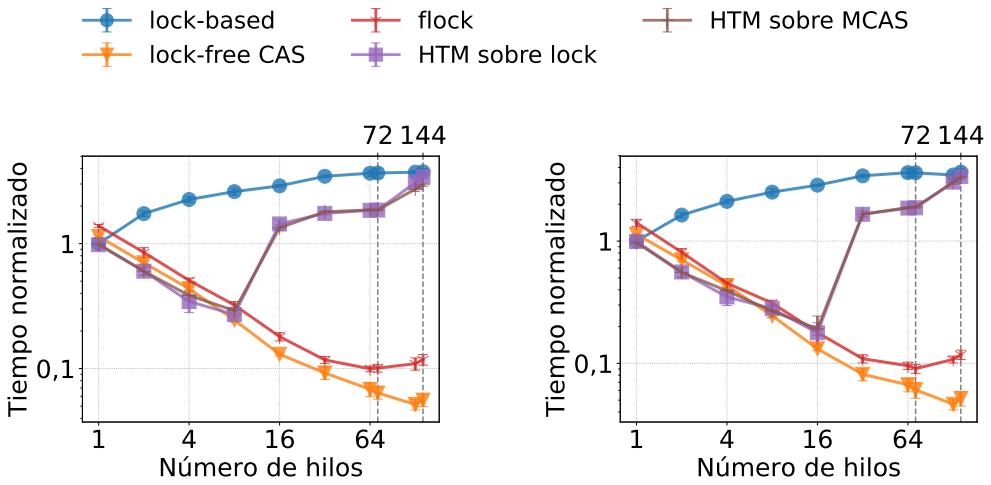
Las estructuras que reportan el peor tiempo son las basadas en MCAS simple, dado que se emula la implementación de MCAS en *hardware* utilizando un cerrojo global. Las demás implementaciones basadas en cerrojos, incluyendo `flock`, tienen un mejor rendimiento ya que utilizan un cerrojo por índice del *array*.

Al igual que en mwobject, se obtiene un rendimiento muy similar para las versiones con y sin *backoff* de los algoritmos que utilizan HTM.

5.3 Lista ordenada

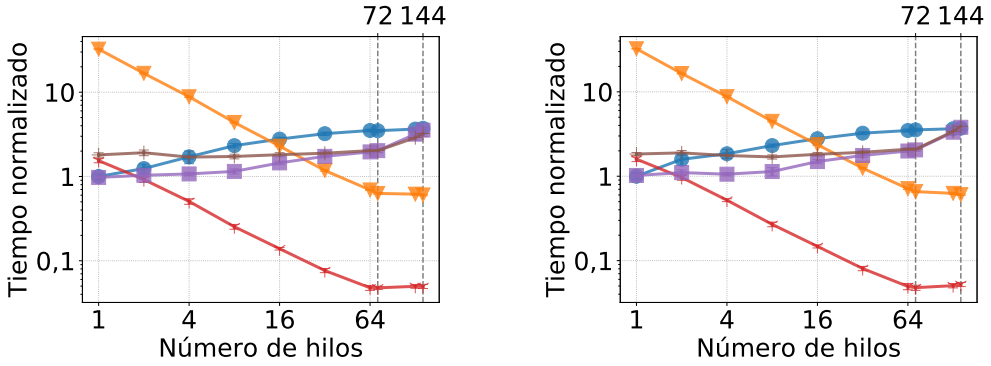
En la figura 5.3 mostramos los tiempos obtenidos para la lista ordenada. Se consigue un resultado muy similar con las versiones de HTM que parten de estructuras lock-based y basadas en MCAS, aunque HTM sobre lock-based es ligeramente mejor para modificaciones con pocos hilos. Estas dos versiones con HTM simple son ligeramente mejores que la implementación lock-based hasta llegar al máximo de hilos disponibles.

Pasado este punto, el rendimiento de las versiones con HTM empeora hasta ser el mismo que el de la estructura lock-based ya que las transacciones fallidas por la alta contención obligan a recurrir al *spinlock* con mucha frecuencia.



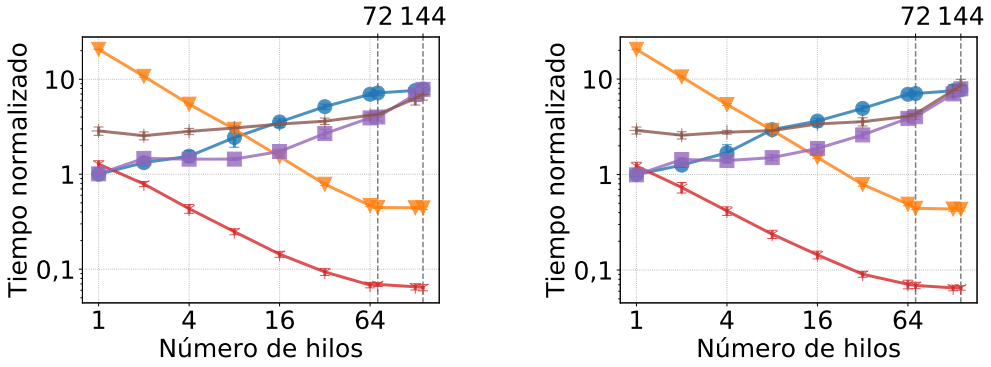
(a) Lista ordenada lecturas sin *backoff* en HTM

(b) Lista ordenada lecturas con *backoff* en HTM



(c) Lista ordenada modificaciones sin *backoff* en HTM

(d) Lista ordenada modificaciones con *backoff* en HTM



(e) Lista ordenada *mixed* sin *backoff* en HTM

(f) Lista ordenada *mixed* con *backoff* en HTM

Figura 5.3: Tiempo normalizado de lista ordenada

	Sin *backoff*	Con *backoff*
Tiempo (ms)	2048,62	261,02
tx-abort	5364238	357289
tx-commit	158951	965432
tx-conflict	5211824	186140
tx-start	5523190	1322739
tx-capacity	365	19302

Tabla 5.1: Tiempo y contadores RTM para 10^6 lecturas con 16 hilos en lista HTM lock

El algoritmo lock-free basado en CAS utilizado para la lista permite lecturas muy rápidas y modificaciones que consiguen un tiempo mejor que la versión lock-based al aumentar la contención. La implementación con `flock` presenta una evolución similar pero con tiempos hasta 10 veces más bajos para modificaciones. Esta estructura con `flock` utiliza cerrojos (si bien son cerrojos lock-free) de grano fino, a diferencia de la lock-based.

Curiosamente, al utilizar *backoff* ambas implementaciones de HTM siguen mejorando su rendimiento en lecturas hasta los 16 hilos (figura 5.3b), algo que sin *backoff* solamente ocurre hasta los 8 (figura 5.3a). El *backoff* tiene efecto aunque solamente se realizan lecturas, y en modificaciones no tiene efecto.

Intentamos explicarlo con `perf`, contando las veces que ocurren eventos relacionados con la extensión RTM en lecturas de la estructura HTM sobre lock-based. Realizamos una media de las estadísticas para 30 ejecuciones con y sin *backoff*.

Medimos un total de 5 tipos de eventos *hardware* con `perf`, para lo que son necesarios 5 contadores *hardware*. Como nuestros procesadores solamente disponen de 4, para realizar la media de 30 valores de cada contador en realidad debemos llevar a cabo 60 ejecuciones. En la primera tanda de 30, medimos los 4 primeros contadores, y en el resto únicamente el último contador.

También existe la opción de realizar solamente 30 ejecuciones multiplexando los contadores, esto es, concediéndoles acceso a monitorizar el *hardware* por turnos (en grupos de 4), de tal manera que siempre hay periodos de tiempo en los que un contador pierde acceso a anotar eventos que se deberían tener en cuenta. Al terminar la ejecución, se añade una estimación de los eventos perdidos al conteo final [35]. Descartamos esta última opción por la pérdida de precisión asociada.

Ejecutamos las pruebas para 16 hilos, contando solamente los eventos generados para las 10^6 lecturas, produciendo como resultado la tabla 5.1. Aunque solamente hay lecturas, se detecta un número considerable de fallos por conflicto, y el número de fallos por conflicto con *backoff* es casi 30 veces menor que sin *backoff*. Aunque reduce el tiempo total y el número de fallos por conflicto, en este caso el *backoff* aumenta el número de fallos por capacidad.

5.4 *Hash map*

En la figura 5.4 podemos ver que el algoritmo lock-free basado en CAS solamente tiene un rendimiento mejor que el lock-based para las lecturas con alta contención. Además, `flock` vuelve a conseguir los tiempos más bajos para modificaciones con alta contención al utilizar cerrojos de grano fino.

Las implementaciones con HTM mantienen tiempos muy bajos, especialmente para lecturas y el uso de *backoff* sí parece tener efecto. En concreto, empeora el rendimiento de las versiones con HTM para modificaciones, algo que podemos ver al comparar las figuras 5.4c y 5.4d (y como consecuencia también afecta al desempeño en la prueba *mixed*, en las figuras 5.4e y 5.4f).

La versión HTM MCAS obtiene tiempos ligeramente mejores que HTM lock para modificaciones con alta contención pero peores para lecturas.

5.5 *Deque*, pila y cola

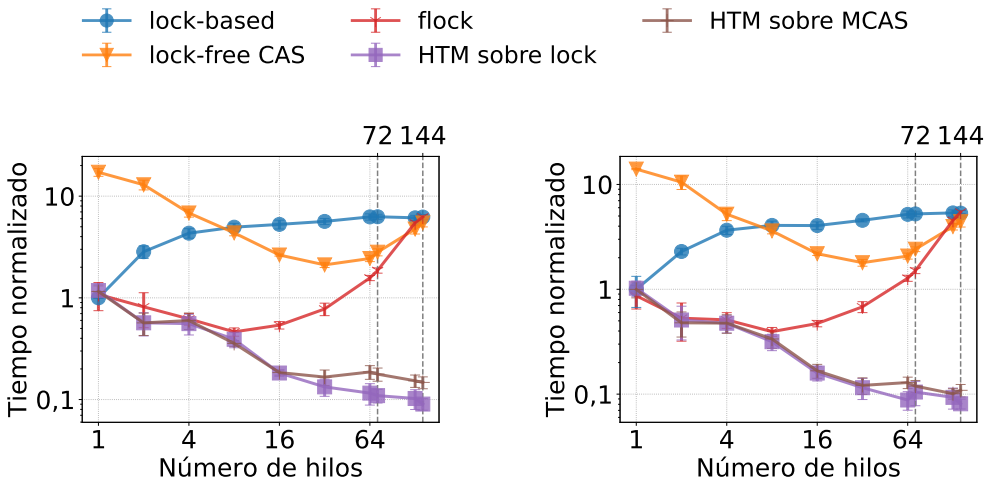
En el caso de la *deque* (figuras 5.5a y 5.5b), el algoritmo lock-free con CAS elegido tiene una sobrecarga muy alta con pocos hilos, pero termina siendo el mejor con alta contención.

La implementación con `flock` es la peor después de MCAS con HTM para estas tres estructuras, pero para la pila es ligeramente mejor que la implementación *lockfree* basada en CAS con alta contención.

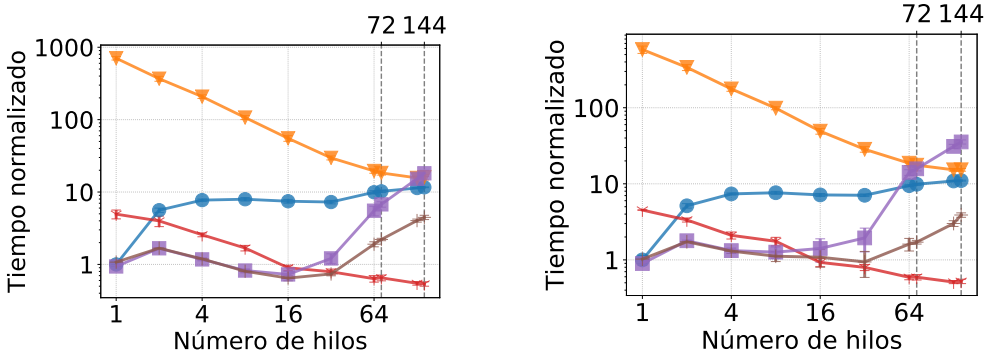
El algoritmo lock-based es mejor que los algoritmos lock-free en varias ocasiones, incluso con alta contención, para la pila y la cola. De forma similar a lo ocurrido en arrayswap, consigue el mismo rendimiento que MCAS con cerrojos.

Los tiempos de *deque* y pila con y sin *backoff* son idénticos. En la cola los tiempos de MCAS con HTM cambian bastante al utilizar *backoff*, pero hemos comprobado que se debe al alto valor de error encontrado en MCAS sobre HTM para esta estructura. Si bien para la cola se realiza un menor número de operaciones que para las demás estructuras debido a los errores de la estructura lock-free con CAS, encontramos diferencias demasiado importantes entre ejecuciones consecutivas.

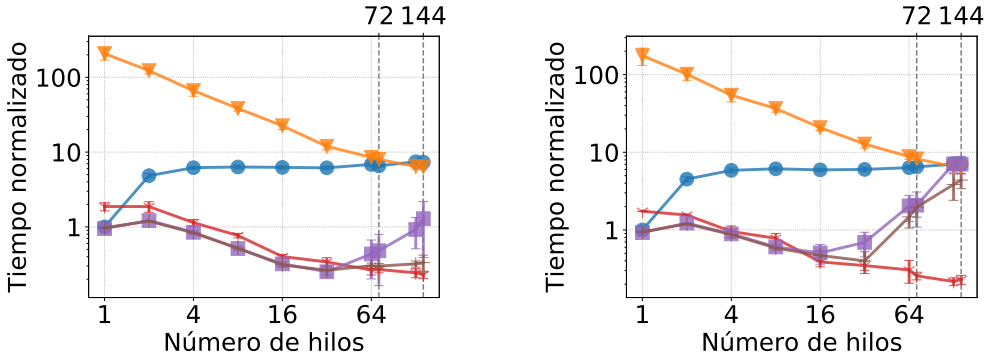
Volvemos a utilizar `perf` para investigar el alto valor de error en MCAS HTM. Encontramos que para un tercio de las ejecuciones se produce un drástico incremento del número de fallos por conflicto y *aborts* de memoria transaccional.



(a) *Hash map* lecturas sin *backoff* en HTM **(b)** *Hash map* lecturas con *backoff* en HTM

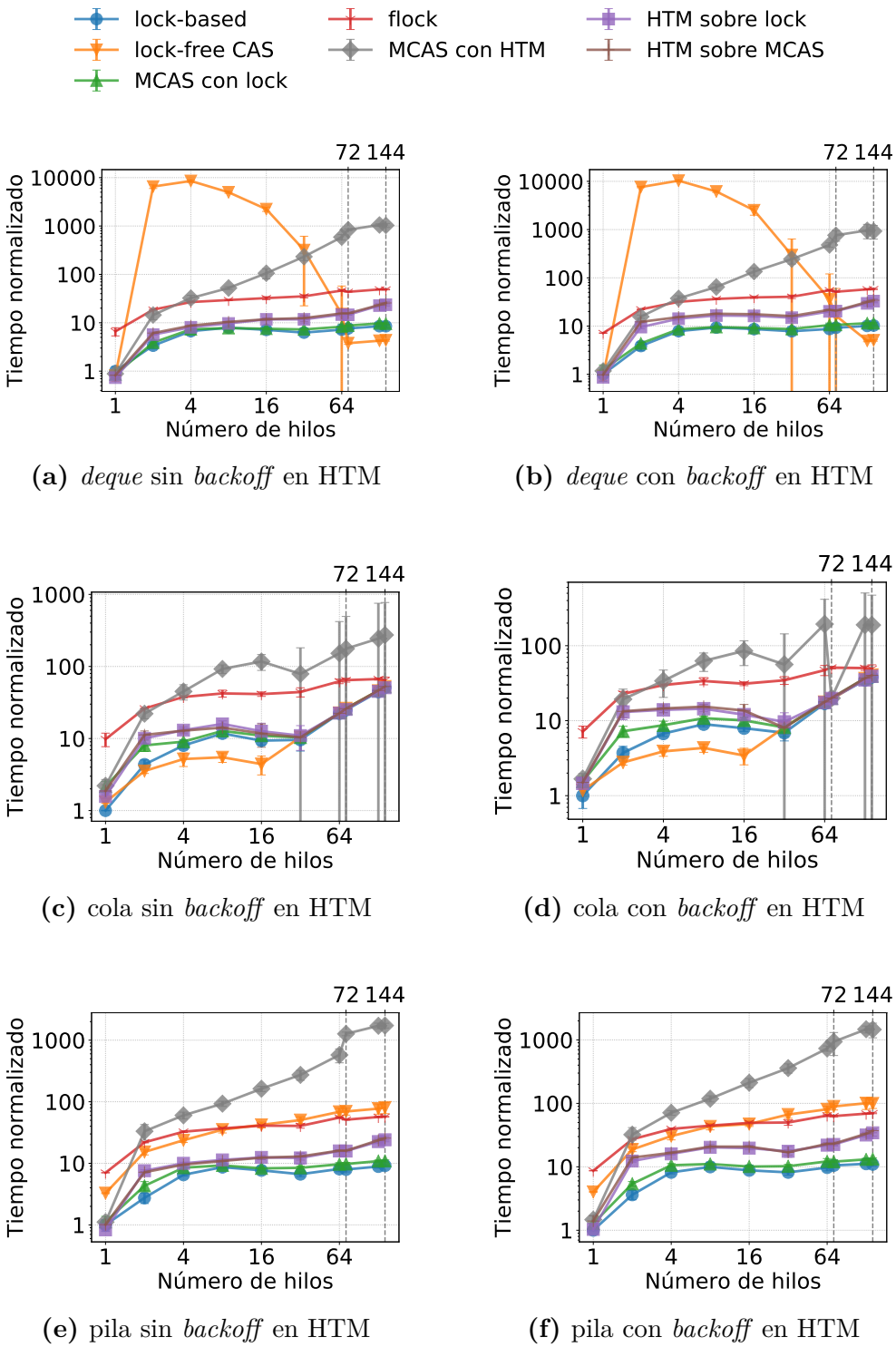


(c) *Hash map* modificaciones sin *backoff* en HTM **(d)** *Hash map* modificaciones con *backoff* en HTM



(e) *Hash map mixed* sin *backoff* en HTM **(f)** *Hash map mixed* con *backoff* en HTM

Figura 5.4: Tiempo normalizado de *hash map*



(a) *deque* sin *backoff* en HTM

(b) *deque* con *backoff* en HTM

(c) cola sin *backoff* en HTM

(d) cola con *backoff* en HTM

(e) pila sin *backoff* en HTM

(f) pila con *backoff* en HTM

Figura 5.5: Tiempo normalizado de *deque*, cola y pila

	Low conflict	*High conflict*
Tiempo (ms)	36,39	967,38
tx-abort	2144	28939566
tx-commit	102064	127090
tx-conflict	3275	39476532
tx-start	104208	29066790
tx-capacity	2	11655

Tabla 5.2: Tiempo y contadores RTM para 10^5 modificaciones con 36 hilos en cola MCAS HTM

Separamos estos casos especiales con un tiempo de ejecución muy superior al normal, refiriéndonos a ellos como *high conflict*, para diferenciarlos así de los más comunes, *low conflict*, que tienen un menor tiempo de ejecución. Realizamos una media de las estadísticas para 30 ejecuciones de cada tipo (de manera similar a lo ocurrido con la tabla 5.1, en realidad ejecutamos 60 pruebas). Ejecutamos las pruebas para 36 hilos sin *backoff*, pues el uso de *backoff* no parece tener efecto al intentar reducir el alto número de fallos en los casos *high conflict*. Obtenemos la tabla 5.2.

Más allá de confirmar que la lentitud de los casos *high conflict* se debe a un aumento del número de transacciones fallidas, no conseguimos determinar la causa de dichos fallos por conflicto adicionales. No se observa este mismo efecto en la pila y la *deque*, aunque su implementación con MCAS HTM es muy similar a la de la cola. Además, hemos comprobado que no se debe al uso de `TM_BEGIN_NO_CONFLICT`.

En general, los tiempos de MCAS implementado con HTM son los peores con alta contención en estas estructuras derivadas de la *deque*.

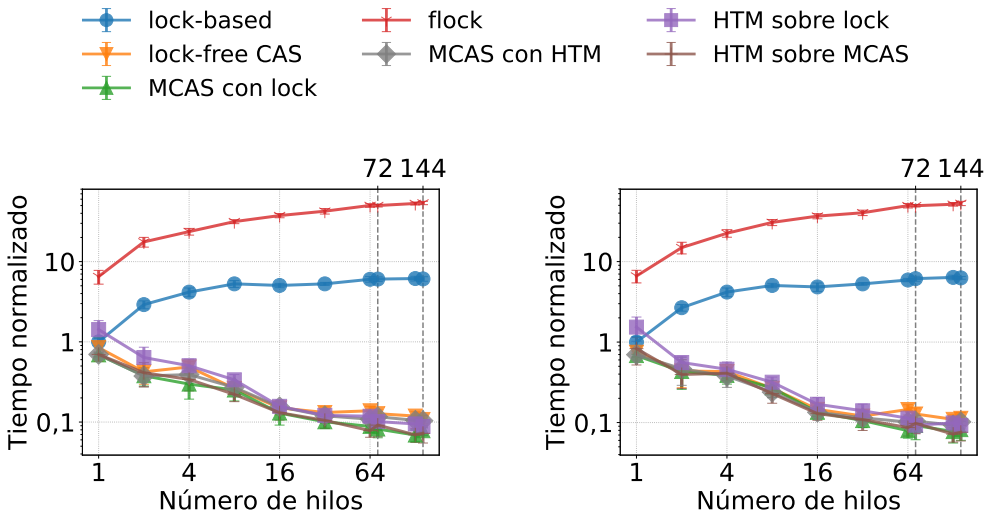
Finalmente, las estructuras basadas en HTM (HTM sobre lock y sobre MCAS) obtienen un rendimiento similar, ligeramente peor que el de la versión lock-based.

5.6 Árbol de búsqueda binario

Incluimos gráficas de los tiempos para las versiones con y sin *backoff* en la figura 5.6. Destaca el mal desempeño de `flock` para esta estructura, confirmando que la librería `flock` está diseñada para adaptar estructuras de grano fino, con secciones críticas pequeñas (la implementación con `flock` del BST se llevó a cabo partiendo de una estructura de grano grueso).

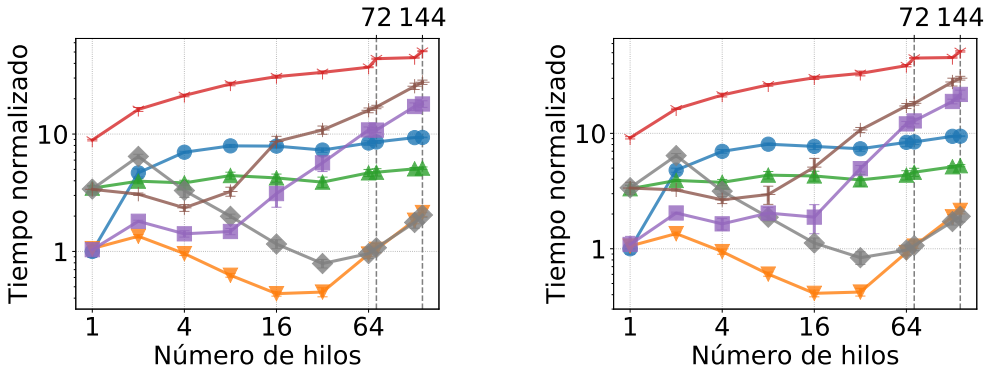
En este caso la estructura lock-free con CAS es la mejor en modificaciones, incluso sin contención. Como mencionamos en la sección 4.4.4, esto se puede deber a que no admite elementos repetidos, a diferencia de las demás estructuras.

Las estructuras basada en MCAS con cerrojo tienen un rendimiento peor que las basadas en MCAS implementado con HTM. Todas las versiones con MCAS son mejores que la lock-based para modificaciones.



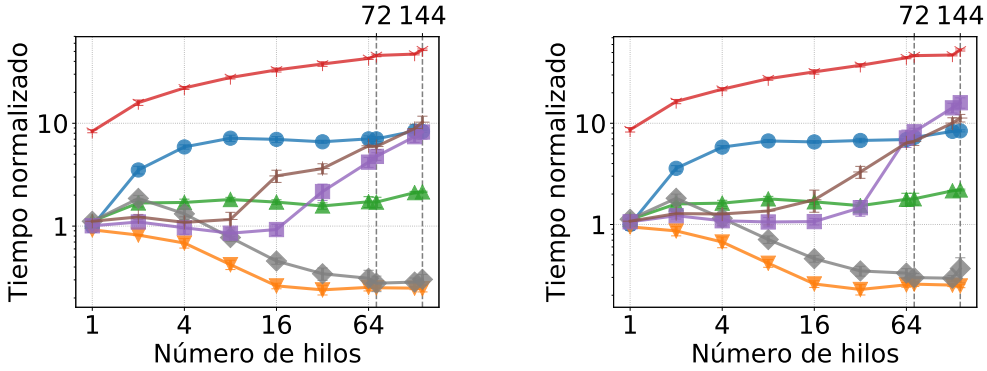
(a) Árbol de búsqueda binario lecturas sin *backoff* en HTM

(b) Árbol de búsqueda binario lecturas con *backoff* en HTM



(c) Árbol de búsqueda binario modificaciones sin *backoff* en HTM

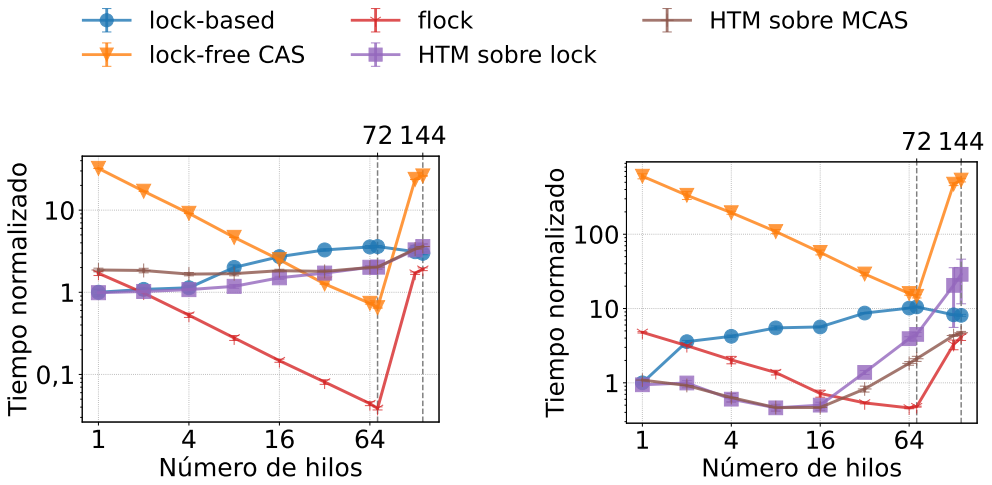
(d) Árbol de búsqueda binario modificaciones con *backoff* en HTM



(e) Árbol de búsqueda binario *mixed* sin *backoff* en HTM

(f) Árbol de búsqueda binario *mixed* con *backoff* en HTM

Figura 5.6: Tiempo normalizado de árbol de búsqueda binario



(a) Lista ordenada modificaciones con hilos *pinned* y sin *backoff*

(b) *Hash map* modificaciones con hilos *pinned* y sin *backoff*

Figura 5.7: Tiempo de pruebas con hilos *pinned*

El uso de *backoff* mejora el rendimiento de HTM sobre la estructura lock-based para modificaciones (figuras 5.6c y 5.6d) pero lo empeora ligeramente en *mixed* (figuras 5.6e y 5.6f) y no tiene un efecto importante para las demás estructuras con HTM. Se obtienen mejores tiempos con la versión de HTM sobre la estructura lock-based que sobre la basada en MCAS, excepto puntualmente al utilizar *backoff* con mucha contención en la prueba *mixed*.

5.7 Pruebas adicionales

Originalmente, en *mcas-benchmarks* se asignaba un núcleo de preferencia a cada hilo lanzado, pero los resultados anteriores proceden de pruebas donde desactivamos esta característica. Inicialmente, al establecer la "máscara de afinidad" de los hilos creados, anclando su ejecución a un solo núcleo, se conseguía una ligera mejora en las estructuras lock-based a cambio de provocar un deterioro importante del rendimiento en las estructuras lock-free (lock-free con CAS y `flock`) al superar el máximo de hilos lógicos del procesador.

Incluimos gráficas donde este efecto, resultado de asignar núcleos de preferencia a los hilos, es más visible. En las figuras 5.7a y 5.7b se puede comprobar que la mayor diferencia respecto a las pruebas sin hilos anclados se encuentra al superar los 72 hilos. El rendimiento de las alternativas lock-free empeora hasta ser igual o peor que la estructura lock-based. En definitiva, el uso de hilos anclados parece eliminar los beneficios característicos de los algoritmos lock-free (buen rendimiento bajo *overcontention*) para estas dos implementaciones.

6 Conclusiones y vías futuras

6.1 Conclusiones

La sincronización en aplicaciones paralelas es un apartado complicado de diseñar e implementar correctamente, aún más si se desea maximizar el rendimiento de la aplicación. En este trabajo hemos realizado la implementación y comparativa de varios métodos de sincronización (HTM, variantes de MCAS y lock-free con `flock`) sobre distintas estructuras de datos. Además de tener en cuenta el rendimiento, nos fijamos en la facilidad de uso de cada uno de estos métodos.

HTM, teniendo en cuenta la implementación concreta basada en *lock elision* que empleamos, permite intentar extraer concurrencia cuando hay relativamente poca contención, pero su rendimiento es similar al de estructuras lock-based si se realizan modificaciones con muchos hilos porque se termina recurriendo al cerrojo de respaldo. Aún así, permite lecturas (cuando hay pocas escrituras) muy rápidas. Todo esto se puede conseguir adaptando estructuras simples de grano grueso a diferencia de `flock`, que solamente consigue una mejora del rendimiento al adaptar estructuras de grano fino.

En cuanto a las versiones con HTM simple, la creada al adaptar estructuras basadas en cerrojos ha sido mejor en la mayoría de ocasiones que la obtenida al adaptar estructuras basadas en MCAS. Es decir, se ha obtenido una estructura más eficiente al adaptar código basado en cerrojos, que además es más simple e intuitivo.

Por otra parte, vemos que la librería `flock` sobresale al adaptar implementaciones de grano fino (lista ordenada, *hash map*) pero confirmamos que no es aplicable a estructuras de grano grueso. Con respecto a las dificultades que presenta al desarrollar estructuras con *hand-over-hand locking*, creemos que no es un problema demasiado grave ya que muchos algoritmos actuales utilizan optimizaciones que permiten evitar este tipo de recorridos (el *leaf-tree*, por ejemplo).

En cuanto a la comparación entre MCAS lock-based y con HTM, al no poder hacer pruebas con la lista ordenada y el *hash map* por errores en la implementación original, es difícil llegar a conclusiones. No hay una alternativa claramente mejor: la versión lock-based, por ejemplo, solamente es preferible en mwobject y derivados de *deque*.

Ocurre algo similar para la versión MCAS con HTM al compararla con HTM lock, que utiliza HTM convencional. Vemos que la versión basada en MCAS con HTM es únicamente mejor que la que utiliza HTM convencional para modificaciones con alta contención en el árbol de búsqueda binario (y para mwobject, pero este tipo de escenario de máxima contención no es frecuente en la realidad). Asimismo, la versión de MCAS que utiliza un cerrojo es solamente mejor que la lock-based para el árbol de búsqueda

binario. Las variantes que intentan eliminar un salto condicional en MCAS no tienen un efecto notable.

Similarmente, en nuestros *benchmarks* el uso de *backoff* en HTM no parece tener efecto excepto en casos puntuales y de manera impredecible.

Resumiendo, `flock` y HTM son alternativas relativamente simples de programar y que permiten mejorar el rendimiento en múltiples estructuras de datos concurrentes (`flock` especialmente en escenarios de alta contención) respecto a versiones lock-based. El uso de MCAS tiene una mayor complejidad asociada y solamente parece aconsejable para árboles binarios de búsqueda, pero no pudimos hacer pruebas con MCAS para la lista enlazada y el *hash map* debido a errores.

6.2 Vías futuras

Sería interesante, para permitir comparaciones más justas entre algoritmos lock-based y lock-free, añadir estructuras lock-based de grano fino a *mcas-benchmarks*. Además, se podría hacer algo similar con el MCAS lock-based, utilizando un *array* de cerrojos en vez de un único cerrojo global. Se elegirían los cerrojos a adquirir según las direcciones de memoria seleccionadas para el MCAS, constituyendo así un MCAS más parecido a su eventual versión *hardware*, aunque todavía considerablemente diferente. Un ejemplo de esta implementación se presenta en el apéndice de "*A hardware implementation of the MCAS synchronization primitive*" [16].

De cara a futuros trabajos, muchos algoritmos wait-free se basan en algoritmos lock-free. Por ejemplo, los autores de "*Lock-Free Locks Revisited*" desarrollaron una investigación que parte de ideas similares a las utilizadas en `flock` para crear algoritmos wait-free, también basados en el uso de idempotencia y que mantienen la semántica de cerrojos para su utilización [36].

También se han investigado maneras de crear estructuras wait-free "automáticamente" a partir de estructuras que utilizan algoritmos lock-free normalizados [37], por lo que se podría intentar aplicar esta transformación a las estructuras lock-free estudiadas en el trabajo.

Por último, se podría explorar una manera de comparar los mecanismos de sincronización anteriores frente a pruebas de rendimiento para MCAS realizadas en gem5, con el objetivo de contrastar los resultados con a una estimación realista de las características de MCAS *hardware*.

Bibliografía

[1] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Morgan Kaufmann Publishers (imprint of Elsevier), 2 edition, 2021.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, 6 edition, 2019.

[3] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.

[4] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[5] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5): 745–770, 1993.

[6] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, 1992.

[7] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, 1993.

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46(8):223–234, 2011.

[9] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[10] Naama Ben-David, Guy E Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 278–293, 2022.

[11] Ole Agesen, David L Detlefs, Christine H Flood, Alexander T Garthwaite, Paul A Martin, Nir N Shavit, and Guy L Steele Jr. Dcas-based concurrent deques. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 137–146, 2000.

[12] David L Detlefs, Christine H Flood, Alexander T Garthwaite, Paul A Martin, Nir N Shavit, and Guy L Steele Jr. Even better dcas-based concurrent deques. In *International Symposium on Distributed Computing*, pages 59–73. Springer, 2000.

[13] Donald Ervin Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 3rd edition, 1997.

[14] *M68000 Family Programmer's reference manual*. Motorola Inc., 1989.

[15] Simon Doherty, David L Detlefs, Lindsay Groves, Christine H Flood, Victor Luchangco, Paul A Martin, Mark Moir, Nir Shavit, and Guy L Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224, 2004.

[16] Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, and Smruti R Sarangi. A hardware implementation of the mcas synchronization primitive. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 918–921. IEEE, 2017.

[17] Eduardo José Gómez-Hernández, Juan M Cebrian, Rubén Titos-Gil, Stefanos Kaxiras, and Alberto Ros. Efficient, distributed, and non-speculative multi-address atomic operations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 337–349, 2021.

[18] James R Larus and Ravi Rajwar. *Transactional memory*. Springer Nature, 2022.

[19] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[20] Yan Solihin. *Fundamentals of parallel multicore architecture*. Chapman & Hall, 2016.

[21] Intel. Intel® Transactional Synchronization Extensions. URL `https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html`. Accessed: 2024-03-27.

[22] ARM. Overview of Arm Transactional Memory Extension, 2022. URL `https://developer.arm.com/documentation/102873/0100`. Accessed: 2024-03-27.

[23] Intel. Intel® Xeon ® E3-1200 v3 Processor Product Family 61 Specification Update August 2020, 2020. URL `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf`. page 61. Accessed: 2024-03-29.

[24] MITRE. CVE-2019-11135. URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11135`. Accessed: 2024-03-27.

[25] Jeremy Brown, Jeff P Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 248–257, 2002.

[26] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 475–486, 2012.

[27] Marc De Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.

[28] Nodari Kankava. Exploring the efficiency of multi-word compare-and-swap. 2020. URL `https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-467147`.

[29] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[30] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[31] A Toolkit for Programming Parallel Algorithms on Shared-Memory Multicore Machines. URL `https://cmuparlay.github.io/parlaylib/`. Accessed: 2024-04-20.

[32] Anne Kaldewaij and Victor J Dielissen. Leaf trees. *Science of Computer Programming*, 26(1-3):149–165, 1996.

[33] Ye Lu and Sean Zhou. Implementation of binary search tree with fine-grained locking and lock-free methods, Dec 2021. URL `https://louisluscu.github.io/15418_Project/`. Accessed: 2024-05-05.

[34] James R. Transactional synchronization with Intel® CoreTM 4th generation. URL https://www.intel.com/content/www/us/en/developer/articles/community/transactional-synchronization-in-haswell.html. Accessed: 2024-04-20.

[35] Perf wiki. 2024. URL https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events. Accessed: 2024-05-27.

[36] Naama Ben-David and Guy E Blelloch. Fast and fair randomized wait-free locks. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 187–197, 2022.

[37] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM SIGPLAN Notices*, 49(8):357–368, 2014.

Lista de acrónimos y abreviaturas

BST	Binary Search Tree.
CAS	Compare-And-Swap.
DCAS	Double Compare-And-Swap.
HLE	Hardware Lock Elision.
HTM	Hardware Transactional Memory.
MCAS	Multi-address Compare-And-Swap.
RTM	Restricted Transactional Memory.
TSX	Transactional Synchronization Extensions.