In-core best-effort transactional memory with lex order locking

Máster Universitario en Nuevas Tecnologías en Informática

Trabajo Fin de Máster

Author:
Álvaro Rubira García
Advisors:
Alberto Ros Bardisa
Eduardo José Gómez Hernández

2 de Julio de 2025

Facultad Informática Universidad Murcia

Acknowledgements

I would like to thank Alberto Ros for his patience in repeatedly explaining some of the most complex topics in processor architecture, and Eduardo José Gómez for his continued encouragement and guidance during this project. Their combined ideas and advice across many meetings were invaluable and made this work possible. I must also thank Juan Manuel Cebrián for helping me understand the intricacies of simulating computer systems with gem5 and for offering additional ideas and optimizations for the implementation. Lastly, I would like to thank all of our colleagues at the Computer Architecture & Parallel Systems Research Group for their support. They have all been incredibly kind and helpful in spite of their limited time.

Abstract

Parallel algorithms are fundamental for taking advantage of the multiple cores available in current processors. Synchronization is necessary for coordinating work through shared memory. Transactional memory provides a simple alternative for synchronization, such that the programmer defines atomic transactions and the transactional memory system manages the underlying interleaving of threads.

With hardware transactional memory (HTM), processors explicitly incorporate modifications that enable efficient execution of transactions. Processors speculatively execute transactions in parallel while activating a conflict detection system. Upon detecting risks that compromise transaction atomicity, they are resolved by only allowing one of the conflicting transactions to proceed, possibly aborting the other. Aborted transactions need to restore the pre-transaction processor state before either retrying execution or taking a fallback path that ensures forward progress.

Commonly, HTM systems require additional hardware, such as register checkpoints for restoring state in case of abort. We revisit an alternative implementation (initially proposed for speculative lock elision) which buffers transactional execution in the reorder buffer (ROB), taking advantage of existing mechanisms for version management and conflict detection.

We could not find any previous performance evaluations of this alternative, likely because it imposes strict limits on the length and complexity of transactions. However, an HTM that only supports small transactions can still ease the task of programming many parallel algorithms.

In this work, we implement an HTM system that requires minimal modifications to the processor, based on buffering instructions in the ROB. On top of this baseline, we test a mechanism for locking cache lines in a non-deadlocking lexicographical order (lex order). Transactions can activate locks in cache lines, which will delay external conflicting requests, to improve their chances of committing.

We test both versions in the gem5 simulator and execute several benchmarks, mainly focused on concurrent data structures. Our HTM introduces low overhead, with locking in lex order significantly reducing the number of aborts (by 30% for 32 cores on average) and always resulting in equal or better performance than the baseline HTM.

Resumen extendido

Estado del arte El rendimiento de los procesadores mononúcleo se ha ido estancando desde principios de la década de los 2000. Consecuentemente, para seguir mejorando las nuevas generaciones, los fabricantes se han decantado por diseñar chips con más núcleos de propósito general o con arquitecturas especializadas. Los procesadores multinúcleo pueden ejecutar a la vez varios hilos que se comunican entre sí utilizando memoria compartida. Esta comunicación es necesaria en muchos algoritmos que no son trivialmente paralelizables. Para evitar carreras de datos al comunicar el progreso entre hilos, se necesitan mecanismos de sincronización.

Una forma común y sencilla de sincronización es el uso de cerrojos de grano grueso, que consiste en utilizar un único cerrojo para crear una exclusión mutua al acceder a los datos compartidos. En muchas ocasiones, esta forma de sincronización genera una serialización innecesaria de los accesos. Como alternativa, se pueden crear varios cerrojos asociados a partes de las estructuras de datos que se pueden acceder en paralelo. Los algoritmos que utilizan estos cerrojos de grano fino tienen que adquirirlos en un orden que evite *deadlocks*.

Los algoritmos que utilizan cerrojos son vulnerables a las latencias generadas por el planificador de procesos del sistema operativo. Los hilos pueden dejar de ejecutar instrucciones (pueden ser bloqueados) mientras se encuentran ejecutando la sección crítica de un cerrojo. Como consecuencia, previenen el progreso del resto de hilos que necesiten acceder a la sección crítica, lo que puede llegar a bloquear el progreso del programa al completo.

Internamente, los cerrojos suelen ser programados con instrucciones atómicas, que permiten realizar varias acciones (*read-modify-write*) atómicamente. Estas instrucciones atómicas se pueden utilizar individualmente, fuera de los cerrojos, para crear estructuras *lock-free*, que garantizan el progreso del programa mientras al menos un hilo (cualquiera) consiga ejecutar instrucciones. El diseño de algoritmos *lock-free* (y, generalmente, el uso meticuloso de operaciones atómicas en vez de cerrojos) suele ser mucho más complejo que las alternativas bloqueantes, pero promete un mejor rendimiento. Este beneficio se maximiza cuando un número elevado de hilos compite por acceder a recursos compartidos (contención).

Así, se dispone de varias técnicas de sincronización con diferentes niveles de rendimiento esperado y diferentes niveles de complejidad. La memoria transaccional obtuvo reconocimiento como un nuevo mecanismo de sincronización, más sencillo para el programador que el resto de alternativas. Como concepto, la memoria transaccional permite programar con bloques atómicos de varias instrucciones, llamados transacciones. La sincronización entre hilos usando memoria transaccional únicamente requiere delimitar las transacciones, y el mecanismo subyacente se encarga de conseguir ejecutar las transacciones de manera que aparenten ser atómicas.

Para implementar este mecanismo de forma eficiente con soporte *hardware*, los procesadores ejecutan transacciones de forma paralela mientras activan un sistema de detección de conflictos. Al detectar riesgos que puedan comprometer la atomicidad de las transacciones (es decir, situaciones donde dos transacciones en vuelo acceden a la misma dirección y al menos una escribe), el procesador abortará la ejecución de una de ellas para solucionar el conflicto (comúnmente se aborta la transacción que accedió primero a la dirección de memoria). Por tanto, es posible encontrar un conjunto de transacciones que se reintenten y generen conflictos constantemente de tal manera que el programa nunca progrese. Para asegurar que las transacciones se acaben completando, muchos sistemas requieren un camino alternativo, normalmente con cerrojos convencionales de grano grueso. Este tipo de memoria transaccional permite intentar conseguir un rendimiento similar al de cerrojos de grano fino con un código más simple.

Propuesta En este trabajo implementamos un mecanismo de memoria transaccional *hardware* (HTM) que requiere pocas modificaciones al *hardware* existente de un procesador de propósito general, centrándonos en asegurar un alto rendimiento para transacciones cortas.

Una de las partes más complejas de las implementaciones de memoria transaccional es conseguir contener el estado especulativo y revertirlo en caso de conflicto. Seguimos uno de los métodos descritos en *speculative lock elision* para este propósito. En concreto, en lugar de añadir *hardware* dedicado a realizar copias de respaldo de los registros, empleamos *hardware* ya existente que los procesadores necesitan para técnicas de ejecución especulativa.

Los procesadores con ejecución fuera de orden mantienen los resultados de las instrucciones ejecutadas en un *reorder buffer* (ROB). Estos resultados especulativos solamente se guardan en los registros reales cuando las instrucciones hacen *commit* en la cabeza del ROB, cuando son la instrucción especulativa más antigua. Se puede conseguir soporte para la ejecución especulativa (por ejemplo, debido a la predicción de saltos o de dependencias de memoria) descartando los resultados de predicciones erróneas antes de que hagan *commit*. Este mismo mecanismo se puede reutilizar para descartar y reintentar transacciones. Es necesario mantener la transacción al completo en el ROB, lo cual se puede conseguir bloqueando la cabeza del ROB hasta que la transacción al completo esté preparada para hacer *commit*.

La transacción solamente estará lista para hacer *commit* cuando todas sus instrucciones se hayan ejecutado y los bloques que se acceden estén en la caché de primer nivel. Durante esta fase de *commit*, las instrucciones ejecutadas aplican sus modificaciones en los registros y las escrituras pueden acceder a memoria. Para hacer el proceso de escritura atómico, se activan cerrojos (cerrojos sobre líneas de caché, distintos de los cerrojos convencionales) en la caché de primer nivel sobre los bloques con escrituras pendientes. El procesamiento de las peticiones externas a direcciones con cerrojos es pospuesto hasta que se libere el cerrojo.

Los bloques leídos durante la transacción se copian a la caché de primer nivel al ejecutar todas las instrucciones de lectura. Los bloques que son escritos se deben obtener ejecutando *prefetches*, que buscan el bloque con antelación.

Asimismo, para la detección de conflictos utilizamos *hardware* ya presente en los procesadores, lo que evita la necesidad de soporte adicional para marcar líneas de caché accedidas durante transacciones. La caché de primer nivel manda notificaciones al procesador cuando sus bloques son invalidados. Para dar soporte a la ejecución especulativa de lecturas, el procesador descarta toda ejecución especulativa que dependa de lecturas que accedan a dicho bloque. A este mecanismo ya existente, se debe añadir soporte para detectar la pérdida de permisos de escritura en bloques que no son invalidados, mandando notificaciones al procesador cuando esto ocurra. Al comparar todas las notificaciones con las direcciones de los bloques escritos en la transacción, que se encuentran en la *store queue*, se termina de obtener un método para detectar todas las peticiones externas que entran en conflicto con la transacción actual.

Si se detecta un conflicto con un bloque que solamente se ha escrito, solamente es necesario volver a ejecutar los *prefetches* transaccionales que lo necesiten. Para las lecturas, sin embargo, se debe descartar todo el estado especulativo de la transacción. Antes de cada reintento, incluimos un periodo de espera (*backoff*) aleatorio para intentar mitigar el *livelock* entre hilos.

Este sistema presenta varias limitaciones de capacidad que se aplican a las transacciones (a cambio, obtenemos una implementación más simple y eficiente). Todas las lecturas y escrituras de las transacciones deben caber en la *load queue* y *store queue* para poder ser comparadas rápidamente con las notificaciones entrantes. En general, todas las instrucciones de la transacción deben poder ser guardadas a la vez en el ROB sin agotar la capacidad de las estructuras del procesador.

Sobre este sistema base de HTM, añadimos un mecanismo que permite activar cerrojos en líneas de caché antes del inicio de la etapa de *commit* de la transacción para evitar reintentos. Con el objetivo de evitar *deadlocks* entre procesadores, los cerrojos de líneas de caché solamente se pueden activar siguiendo un *lexicographical order* (*lex order*) conocido por todos los procesadores. Este *lex order* depende de la capacidad y asociatividad de las estructuras de la jerarquía de memoria, y permite evitar *deadlocks* tanto en estructuras privadas (cachés privadas) como compartidas (cachés compartidas y directorios *sparse*).

La creación de cerrojos de memoria en *lex order* se presentó anteriormente en el artículo "Efficient, distributed, and non-speculative multi-address atomic operations". A diferencia de los *multi-address atomics*, nuestro mecanismo empieza a adquirir cerrojos antes de que todas las direcciones de memoria de la transacción sean conocidas. Cuando se conocen nuevas direcciones de memoria, puede que algunos cerrojos tengan que ser readquiridos en el orden correcto para mantener una secuencia en *lex order*.

Evaluación Para medir la eficacia de nuestra propuesta, la implementamos en el simulador gem5. Empleamos el modelo *full-sytem* para la arquitectura x86-64. Utilizamos Ruby y SLICC para modelar detalladamente los controladores de memoria. De esta manera, conseguimos simular un procesador moderno con un número variable de núcleos (desde 1 a 32).

En cuanto a *benchmarks*, comenzamos simulando la ejecución de varias cargas de trabajo con estructuras de datos concurrentes. Simulamos operaciones sobre un array (intercambio atómico de elementos), una lista ordenada, un *hash map*, una *deque*, una *stack*, una *queue* y un árbol de búsqueda binario. En *arrayswap*, *deque*, *stack* y *queue*, cada una de las operaciones se puede integrar de forma sencilla dentro de una única transacción. Para la lista, el *hash map* y el árbol de búsqueda binario no podemos utilizar una única transacción, pues sus operaciones contienen bucles que podrían agotar los recursos de nuestra implementación. En su lugar, empleamos transacciones que ejecutan un multi-address compare-and-swap, de manera que cada operación puede necesitar varias transacciones cortas.

En estas pruebas con estructuras de datos concurrentes, comparamos algoritmos basados en cerrojos de grano grueso, basados en la memoria transaccional base y basados en la memoria transaccional con *lex order* locking. Para las dos versiones basadas en HTM, también probamos con variaciones que incrementan el *backoff* cuando se reintenta un prefetch exclusivo.

Al comparar el tiempo de ejecución de cada alternativa, vemos que las dos versiones de HTM son siempre superiores al uso de cerrojos. Además, la versión con *lex order* locking es capaz de reducir el tiempo de ejecución notablemente en escenarios con alta contención (donde hay una alta probabilidad de conflictos entre transacciones).

Otra estadística que recogemos es el número de *aborts* (veces que se ha tenido que reintentar una transacción) dividido entre el número de transacciones completadas. La disminución de esta métrica, a la vez que se mantiene o reduce el tiempo de ejecución, puede ser indicativa de un ahorro de energía, pues supone una reducción en el número de instrucciones que se vuelven a ejecutar. Creemos que la adición de *lex order locking* puede mejorar la eficiencia energética del sistema, ya que reduce el número de *aborts* hasta en un 30% para 32 cores.

Finalmente, probamos a incorporar nuestra memoria transaccional para simplificar secciones de código de los *benchmarks* en Splash-4. En Splash-4 se empleaban operaciones atómicas *compare-and-swap* (CAS) en bucle, dada la ausencia de operaciones atómicas especializadas para coma flotante en x86. Estas operaciones se pueden simplificar con HTM.

Teniendo en cuenta las métricas globales de los *benchmarks* de Splash-4, la sustitución de los bucles CAS por transacciones con nuestra implementación no genera un impacto perceptible en el tiempo de ejecución. Por ello, creamos *microbenchmarks* en los que los procesadores ejecutan continuamente estas operaciones, para comparar la eficiencia de nuestra propuesta ante bucles CAS. Para la operación *fetch-and-add double*, HTM muestra una mejor escalabilidad que los bucles CAS. Aunque en esta prueba solamente se accede a una línea de caché en cada transacción, los cerrojos en líneas de caché consiguen evitar *aborts* (reduciendo así el tiempo de ejecución) al empezar a adquirir cerrojos antes de que la fase de *commit* comience. Para la operación *atomic max*, se consigue un rendimiento similar al del uso de bucles CAS.

Conclusiones y vías futuras En conclusión, hemos desarrollado un sistema de HTM eficiente que requiere pocas modificaciones al procesador. A cambio, las transacciones soportadas adquieren limitaciones, con restricciones principalmente en su longitud y número de líneas de caché accedidas. Sobre esta base, hemos incorporado un sistema que consigue retrasar peticiones para reducir el número de *aborts*.

Como trabajo pendiente, destacamos la falta de un camino alternativo en el mecanismo de HTM. En los *benchmarks*, el *backoff* y los cerrojos en *lex order* han sido los únicos encargados de ayudar a mitigar *livelocks*. Hay varias soluciones posibles que no requieren la definición explícita de un camino alternativo. Principalmente, se puede modificar el método de resolución de conflictos para que garantice el progreso, o incorporar mecanismos *hardware* que se activen tras un número finito de reintentos para asegurar las transacciones se acaban completando.

Contents

List of Figures

List of Tables

Listings

1 Introduction and motivation

Prior to the time when single-core processor designs were unable to keep pace with Moore's law, better performance was mainly achieved by increasing clock speed and using very long pipelines. As challenges with power dissipation appeared, processor designers have switched to single-chip multiprocessors that achieve better energy efficiency.

To extract all available performance from multicores, many threads of execution have to work together on the same problem. If the problem cannot be easily divided into independent subtasks, communication and synchronization between threads (usually through shared memory) is necessary.

Locks are a common way of achieving this synchronization. However, coarse-grained locking can serialize large program sections, reducing the potential speedup of using multiple cores. Avoiding this usually involves fine-grained locking, which can complicate the program. Complicated locking algorithms are especially susceptible to deadlock and livelock problems. In addition, locks imply scheduling problems (such as convoying or priority inversion) that cannot be easily avoided [1].

There is also the possibility of directly using the read-modify-write atomic synchronization mechanisms exposed by the processor's instruction set architecture (ISA). These atomic operations are the synchronization primitives used internally by most locks to provide a higher-level abstraction for critical sections, but they can also be leveraged in an extremely fine-grained fashion to write programs focused on achieving progress guarantees under contention.

This often comes at the cost of added complexity. In lock-based algorithms, all code inside the region protected by a lock can be made atomic to other users of the same lock. With common read-modify-write atomic instructions, however, only specific simple operations exposed by the ISA (e.g. fetch-and-add or compare-and-swap) can be done atomically. Consequently, algorithms programmed with these atomic instructions are hard for even experts to get right [2, 3].

The non-deterministic nature of parallel programs makes bugs notoriously difficult to find and remedy. Moreover, at the advent of multicore processors many programmers were not used to writing correct concurrent code [4].

Transactional memory (TM) appeared as a convenient alternative. TM is simpler even than using locks from the programmer's perspective: it only involves marking sections of code (transactions) that should be executed atomically. The underlying TM system guarantees that the execution of any transaction will appear to be atomic to other transactions [1]. This frees the programmer from the task of managing the underlying synchronization of threads.

With hardware transactional memory (HTM), processors explicitly incorporate modifications that enable efficient execution of transactions. HTM offers the ease of use of TM combined with performance comparable to fine-grained locking. As a tradeoff, the limited hardware resources impose limits on transactions, mainly in the number of different cache lines accessed.

Complexity limitations are not a problem for small transactions that briefly access a few locations in memory. These are prevalent in concurrent data structures [5] and many other parallel workloads. For this reason, it makes sense for processors to provide HTM even if it only supports constrained transactions.

An advantage of reducing the scope of supported transactions is that the hardware implementation can be simplified. Indeed, in an effort to impose fewer restrictions in transactions, recent HTMs have complex implementations, and can involve important changes to the cache coherence protocol (which are to be avoided due to the extensive verification effort they require).

In this thesis, we implement an HTM optimized towards simplicity and performance in small transactions, that requires few modifications to existing hardware. Additionally, we propose a mechanism for reducing transaction aborts that does not require modifications to the cache coherence protocol.

The rest of the document is organized as follows. Chapter 2 provides a necessary background on modern CPUs and HTM systems. Chapter 3 outlines the primary objectives of our HTM proposal, as well as our simulation and evaluation setup. Chapter 4 describes the implementation details of our system. Chapter 5 presents the results of the evaluation of our proposal using several benchmarks. Finally, in Chapter 6 we present our conclusions and suggest directions for future research.

2 Background and related work

The topic of HTM has been deeply researched over recent years. As a result, there exist many alternatives that have different ways of interacting with the processor's microarchitecture to simulate atomicity. These systems have to take into account the mechanisms that optimize the performance of modern processors, which we will review in section 2.1. Later, in section 2.2 we address the difficulty of using these shared-memory multiprocessors to create concurrent algorithms. Lastly, in section 2.3 we summarize previous efforts towards HTM systems that informed our implementation.

2.1 Out-of-order CPUs

Modern processors that optimize for performance do so at the cost of higher power dissipation and more complexity. They can afford transistors for advanced features that aggressively extract every bit of instruction-level parallelism from programs. Specifically, dynamically-scheduled superscalar processors try to avoid stalls by executing instructions speculatively, in an order that can be different from the one assumed by the programmer.

Some aspects of these processors are very relevant for our implementation, so it is worthwhile to dedicate the following sections to reviewing them in detail. It should be noted that these features are specific to "mainstream" out-of-order processors, and although they are certainly widespread, many alternatives exist.

2.1.1 Reorder Buffer

In out-of-order processors, only the front-end (the part that fetches instructions from memory, decodes and "sends" them to where they wait until they are executed) and the commit stage process instructions in-order. Execution can be done out-of-order, whenever the operands and functional units required become available.

In the commit stage, results of instructions that might have executed out-of-order are written to the register file and memory. This stage is done in-order to ensure that the sequence of updates to architectural state is consistent with in-order execution, therefore maintaining correctness even if some instructions were allowed to start executing early to improve performance.

A reorder buffer (ROB), implemented as a circular queue [6], can be used for holding speculative results until they are committed, forwarding them to younger instructions if necessary.

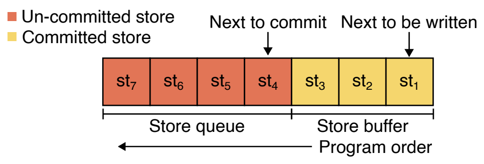


Figure 2.1: Store buffer as part of the store queue.

When an instruction is dispatched, it is assigned an entry at the tail of the queue. Instructions only commit and are removed from the ROB when they are the oldest uncommitted instruction, that is, when they are the head of the queue.

Additionally, processors perform speculation by making a prediction on unavailable information, creating speculative results in the ROB based on that prediction and invalidating and re-executing the speculative actions once the information becomes available if the prediction is resolved as wrong. For example, by predicting the outcome of a conditional branch, younger instructions are fetched and can start executing speculatively before the conditional branch is known to be taken or not.

To discard speculative state, entries in the ROB are marked as invalid, and will be treated as no-ops by the rest of the pipeline, including the commit stage. This is commonly referred to as squashing. Continuing with the conditional branch example, all ROB entries younger than the branch are squashed after an incorrect prediction, and the fetch stage is set to continue at the corrected program counter.

2.1.2 Load queue and store queue

The load queue (LQ) and store queue (SQ) are fundamental for handling correct out-of-order execution of memory accesses. They are complex content-addressable memories that allow quickly searching addresses against all in-flight loads or stores.

In addition to the ROB, load and store instructions are also added to the LQ and SQ, respectively, on dispatch. Similarly to their behavior in the ROB, loads and stores are removed from the LQ and SQ, respectively, on commit.

Although loads access memory when they execute (and can do so speculatively), stores are only written to memory after they commit. This way, the processor emits writes in-order and only for stores that are confirmed to be correct. The processor does not wait for each store to complete when committing, and instead a store buffer (SB) holds pending committed stores. Stores are removed from the SQ and added to the SB on commit. This is often optimized by placing the SB immediately after the SQ in the same structure, as shown in Figure 2.1, and modifying the index that signals the start of the SB instead of removing and inserting.

2.1.3 Cache coherence

The memory consistency model provides a contract between the users of a processor and the team in charge of its implementation. It consists of precise rules that dictate the valid states of the processor after executing loads and stores in shared memory. Knowing these rules, programmers can create code that is correct under all possible valid states of the model. At the same time, the strictness of these rules determines the leeway that chip designers have for introducing optimizations in the processor, because optimizations are not allowed to modify the behavior of the processor in ways that violate the model.

The cache-coherence protocol simplifies the implementation of the memory-consistency model in processors with caches. Its purpose is to make caches functionally invisible from the perspective of the processor, so that cores concurrently executing loads and stores seem to be accessing a single unified memory [7]. The only perceivable effect of having caches between the processor and main memory should be in the timing of accesses.

Most coherence protocols are invalidation-based, meaning that they invalidate old un-updated versions of data before new versions are created by stores. MSI is a simple invalidation-based protocol that implements the "single-writer/multiple readers" principle with three possible states for each line in the first-level cache (we assume write-allocate write-back caches): modified (block can be read and written, and is the only copy), shared (block can be read and multiple copies can exist) and invalid (block is not present). When a processor tries to write to a line that is not in M state, it sends requests to other processors to get the latest value of the block and invalidate other copies before transitioning to M and writing. When a processor tries to read from a line in I state, it needs to transition into S by sending requests that get the latest value for the block and change any private M line to S.

The MESI protocol adds an exclusive state for lines that have no shared copies and have not been modified, eliminating coherence transactions for the common case of writing the only copy of a block right after loading it [7]. More complex variations exist, with different processor manufacturers favoring different approaches (e.g. MOESDIF for AMD64 [8]).

Requests can be sent through a shared bus that processors "snoop" to examine all coherence requests, but for scalability it is more common to use a directory instead. The directory holds state information for all copies of a cache line, and can be distributed, assigning a subset of the cache lines to separate directories (e.g. one directory per bank of L3). Cache misses in first-level caches send a message to the appropriate directory, which will in turn exchange further point-to-point messages to make the block available.

Keeping state information for all main memory blocks can become unfeasible, so instead sparse directories [9] are the most common option. Sparse directories function similarly to caches, and keep state information only for the most recently active entries. All copies of a block are invalidated when its state is evicted from its directory.

2.1.4 Speculative load execution

One of the restrictions set by the memory consistency model is dictating the valid ways that a processor can reorder memory accesses. For example, the x86-TSO model [10] attempts to describe the memory consistency model of x86 processors, and does not allow load instructions to be reordered with respect to other load instructions. Without speculation, this would mean that if a load stalls when accessing memory due to a cache miss, subsequent loads (as well as all instructions that depend on them) would also have to stall, even if their target blocks were present in the cache. In practice, x86 processors speculatively execute loads once their effective address is calculated, without waiting for previous loads to complete.

If a load that is after a slower load in program order is allowed to speculatively execute first, speculative execution is correct as long as, by the time the slower load completes, the value of the block accessed by the faster load has not changed [11]. Instead of issuing additional loads to compare the values of memory blocks each time a reordering is made, most processors opt for a more efficient (although less precise) method to ensure that the block is not modified by other processors: when a block is invalidated in the first level cache, the processor associatively searches the LQ and preemptively squashes all uncommitted executed loads to that block. This covers two possible scenarios:

1. The block was invalidated because another processor requested exclusive access before writing to it. Unless the value to be written is the same as the current value of the block, the block's contents were going to change.

2. The block was invalidated due to the cache replacement policy, and it was not actually being written by another processor. Still, the squash is made as if the block's contents were going to change, because once it leaves the cache the processor no longer receives coherence messages that notify it of updates to the block.

2.2 Synchronization in shared memory

Although it was briefly addressed in the introduction, we will review the issue that TM, and all other synchronization alternatives we cover, try to address: collaboration between threads running on processors that operate in shared memory systems.

2.2.1 Atomic operations

We borrow a toy example of a counter from Herlihy et al. [1]. We can imagine a shared counter, where threads can execute a function that reads the value of the counter and then increments it. One might start by coding the body of the function as "`return counter++;`". The compiler then generates the machine instructions in Listing 2.1. `r1` and `r2` refer to private registers in each processor, and `counter` is the value of the counter in memory.

Listing 2.1: Pseudocode for reading and incrementing a counter

```
Step 1 -> r1 = counter
Step 2 -> r2 = r1 + 1
Step 3 -> counter = r2
# r1 is used as return value
```

Multiple processors can execute the function at the same time, and steps from different processors can interleave. One processor might execute step 1, then other processor executes step 1 as well, and then both processors finish executing the function. The end result is that, although two processors tried to increment the counter, the new value of `counter` stored in memory is `counter + 1`, and both processors return `counter` as the value they found in memory.

Processors provide atomic operations to facilitate collaboration between threads. Atomic operations can perform several actions in memory and guarantee that no other processors modify or read the value while the atomic operation is taking place. The following are a few examples included in many ISAs:

- `test&set rDst, addr`: Sets `rDst` to the value in `addr`, and stores `1` in `addr`.
- `fetch&add rDst, rSrc, addr`: Sets `rDst` to the value in `addr` and then stores `rDst + rSrc` in `addr`.
- `cas rExpected, rNew, addr`: Compare-and-swap (CAS), only stores `rNew` in `addr` if the value in `addr` is `rExpected`, and modifies a register to indicate if the store took place or not (e.g. the x86 implementation sets `rExpected` to the value in `addr` if the values were different and nothing was stored).

As one might imagine from these examples, most atomic operations only perform a few actions. `fetch&add` is enough for a correct implementation of the previous concurrent counter example, but more complicated operations cannot be made thread-safe so easily.

2.2.2 Locks

Locks are a common solution to this problem. Locks can be implemented with a value in memory that indicates if the lock is taken. Threads attempt to acquire the lock in a loop that uses an atomic operation. Only threads that have acquired the lock can execute the critical section, which accesses shared state.

We can now protect more elaborate structures, such as a doubly linked list, by using a single lock. This is scheme is called coarse-grained locking, and introduces unnecessary serialization in many cases. For example, a single lock does not allow for concurrent insertions in distant positions of the list that could be allowed to run in parallel without any unwanted side effects.

As an alternative, multiple locks can be created, for example one lock per node in the list. When inserting a new node, only the two adjacent nodes need to be locked, and disjointed parts of the list can be modified at the same time. This is fine-grained locking.

Needing to acquire more than one lock means that deadlocks can appear when groups of processors mutually need to wait for each other before progressing. The locking order in fine-grained locks of complex structures has to be studied carefully to avoid indefinitely stalling the program due to deadlocks. Locks can also result in other problems such as priority inversion or convoying [1].

Crucially, many algorithms using locks are blocking. If a thread stops executing instructions (e.g. because it gets descheduled by the OS) while holding a lock, it blocks the progress of all other threads that need to acquire the lock. When a single thread stops executing, it can potentially prevent the entire program from making progress.

2.2.3 Lock-free algorithms

Lock-free algorithms guarantee that if at least one thread manages to execute instructions (even if all other threads are descheduled), regardless of which thread it is, it will eventually make progress and complete its desired action.

Lock-freedom is achieved by using atomics to create non-blocking algorithms. The term lock-free can be somewhat misleading: not all algorithms that use locks are blocking (there are even specialized "lock-free locks" [12]), and poorly programmed algorithms that do not use locks can be blocking. The defining property of lock-free programs is that *someone* will make progress independently of however many other threads are blocked.

Non-blocking algorithms are often more complicated than the blocking alternatives. Lock-free structures are programmed using CAS (or load linked/store conditional (LL/SC)). Intuitively, the simplest lock-free algorithm uses a shared pointer that holds a reference to a data structure. Threads then modify private copies of the structure and try to write their progress into the shared state by performing a CAS on the pointer, with the expected value being the value of the pointer when the private copy was made, and the new value being the address of the private copy [13]. If the CAS failed, the whole process (copying, modifying and performing CAS) has to be retried. To improve the efficiency of this method, the copy and CAS can be done for individual nodes or parts of the structure. However, complexity arises from the restricting nature of atomic operations, as only one value in memory can be atomically modified at a time.

2.2.4 MAD atomics and lex order

Atomic operations to more than one address have been proposed. Multi-address atomic operations (MAD atomics) [14] are of particular interest to our work. Processors with support for MAD atomics can expose them through new instructions in the ISA, such as multi-address compare-and-swap (MCAS). MCAS is a version of CAS generalized to more addresses, such that expected and desired values have to be provided for all addresses, and desired values are only stored if all addresses had their expected values.

To modify multiple memory addresses atomically, MAD atomics lock target cache lines. Processing of external requests (e.g. invalidations) to a locked cache line is delayed while the lock is active. Modern processors already have the ability to activate cache line locks (referred to as "cache locking" by Intel [15] and "cacheable locks" by AMD [8]) but only activate them for individual cache lines at a time, likely to avoid the complexity of having to deal with deadlocks.

MAD atomics, however, activates multiple cache line locks in lexicographical order (lex order) [16]. Lex order is a sub-address order that takes the limited capacity of the memory hierarchy into account to avoid all deadlocks. Each cache line address is assigned a lex order that is $line\_addr \bmod (sets \times assoc)$ where *sets* and *assoc* are the number of sets and ways of the structure with the smallest $sets \times assoc$ (both need to be rounded down to the next lower power of two). Thus, in a conventional system, the size of the lex order will be dictated by the L1 cache: 6 bits for the index so it can be VIPT, and a common associativity such as 8-way results in 6 + 3 = 9 bits, so the lex order can be calculated as $line\_addr \bmod 512$.

In the face of MAD atomics where at most one cache line in the group of addresses has a given lex order, locking in lex order is guaranteed to be deadlock-free because conflicting operations will conflict in their minimum common lex order. For MAD atomics that involve a set of cache lines where some have the same lex order, all the ways of the sets in shared structures (directories and shared caches) that hold multiple of the target cache lines have to be locked to guarantee deadlock-freedom [14]. That is, when locking multiple cache lines with the same lex order in an atomic group, the first locking request will have to activate a lock for its entire directory set. A dedicated lock bit can be added to each directory set to allow locking all the entries atomically.

There are limitations on the set of addresses that can be locked at the same time in private caches. Cache lines to be locked might belong to the same set, so the number of addresses cannot be higher than the associativity of the first-level cache.

2.3 Transactional memory

The initial proposal for TM [17] tracks the memory accesses of all ongoing transactions while speculatively executing them. The effects of each transaction are only made visible to other threads (by modifying shared memory) after executing the whole transaction and checking that there were no conflicts with concurrent memory accesses of other transactions. When a conflict is found, the transaction aborts and must be retried.

While many software TM implementations have been developed, their performance overhead makes general adoption difficult [18]. We will focus on HTM, which leverages hardware support for a significant performance improvement.

Before going over some implementations that are the most similar or related to ours, we will start by explaining the categorization of common alternatives for HTM. This will aid in understanding the family of HTM implementations that are the focus of this work: systems that take advantage of existing speculation hardware for providing HTM with eager conflict detection and lazy version management.

2.3.1 Characterization of HTM implementations

Approaches to HTM have mainly been characterized by how they manage concurrency control, conflict detection and version management [4].

Concurrency control

In shared memory systems, conflicts occur when transactions perform operations that break the illusion of atomicity, that is, when they overwrite or read the value written by another ongoing transaction. Concurrency control determines how the system avoids committing conflicting operations.

In optimistic concurrency control, the conflict is detected once the conflicting operation has happened, and requires some form of conflict resolution, usually by aborting one or all of the conflicting transactions and undoing speculative state.

In pessimistic concurrency control, the conflict is detected before the conflicting action is taken, and can be resolved by e.g. delaying the action. A form of pessimistic concurrency control might place a memory lock on all locations accessed by a transaction before executing, and other transactions that access the same addresses will have to wait if they find a memory location locked.

Most recent research and commercial HTM systems employ optimistic concurrency control, which is often assumed by default, and thus more attention is given to choices made in conflict detection and version management.

Conflict detection

Conflict detection usually refers to the timing of conflict detection on HTM with optimistic concurrency control. It can be broadly characterized as eager, if conflicts are detected as soon as they happen, or lazy if conflicts are only detected right before starting the transaction commit stage.

Conflict detection also addresses the granularity of the mechanism that detects conflicts in memory. Ideally, only accesses to the same bits in memory should cause conflicts. Many HTMs, however, detect conflicts with cache line granularity, because their conflict detection relies on the cache coherence protocol. As a consequence, accesses to different locations of the same cache line may still trigger conflicts (*false* conflicts).

It is important to mention that most recent HTM systems display strong atomicity [19] (atomicity not only between transactions, but also between transactions and non-transactional code), because non-transactional accesses can also cause conflicts.

Version management

Version management dictates how the state modified by ongoing transactions is handled. Version management can be eager or lazy. In both cases, two versions of the memory locations written during a transaction are kept: one version with the value before the transactions and another with the last value written in the transaction. This is done so that either the initial values are kept if the transaction aborts, or the transactionally written values are persisted if the transaction commits. The only difference is which version is stored in-place and which one is stored somewhere else [20].

Transactions with eager version management write speculative state in-place, so commit is faster as data to be committed is written in advance to shared memory. When a transactional write is performed, the pre-transaction values can be stored in a separate redo-log.

Lazy version management, on the other hand, stores speculatively written state in a private copy and only overwrites the real locations on commit.

2.3.2 Speculative Lock Elision

Presented by Rajwar and Goodman [21], speculative lock elision (SLE) detects lock acquisitions and releases and tries to elide critical sections using HTM. Transactional execution is attempted until elision is successful or a finite number of aborts is reached, and the lock can be acquired as a fallback that resorts to the usual serializing execution of critical sections. Processors attempting speculative execution start by checking that the lock is free to prevent conflicts with a processor in the fallback path.

In addition to contributing the idea of automatic detection of locks, which allows transactional execution of unmodified lock-based programs, they describe efficient mechanisms for HTM. Essentially, SLE buffers speculative updates to memory by not letting stores exit the SB until lock elision is validated, and two alternatives are proposed for buffering the architectural register state: using the ROB (by not letting instructions commit until the transaction is successful) or using a register checkpoint (instructions are free to commit, but stores are still buffered in the SB).

Each alternative has its own method for detecting conflicts. If the ROB is used, in processors with speculative load execution invalidations already send snoops to the core which can be used for detecting conflicts by checking them against entries in the LQ and SQ. If the register checkpoint is used, a bit is added to each block's cache metadata that marks blocks accessed during the transaction. This bit can then be checked to detect atomicity violations when conflicting external requests are received.

Thus, in addition to not requiring dedicated hardware to quickly create a full backup of the register file, the ROB alternative does not need to mark accessed blocks. Furthermore, the register checkpoint alternative creates a backup of all registers at the start of the transaction, many of which might not be overwritten during the transaction, meaning that many unnecessary backups are created (especially in small transactions). The ROB alternative requires the least modifications to existing hardware and avoids creating unnecessary register backups, but has the drawback of not being able to commit instructions until transaction commit starts.

In a later article, Rajwar and Goodman propose adding a timestamp-based conflict resolution mechanism to SLE [22]. It guarantees forward progress of the program in the face of any number of conflicting processors without having to acquire the underlying lock.

2.3.3 Transactional-Execution Facility in the z/Architecture

IBM's Transactional Execution Facility was introduced with the zEC12 processor. We are particularly interested in its constrained transaction execution mode. Constrained transactions impose a series of limitations on transaction length and complexity, such as having a maximum of 32 instructions, limiting the number of accessed cache lines or forbidding jumps to lower PCs (hence forbidding loops and function calls) [23].

In exchange, constrained transactions that meet these requirements do not require a fallback path and are guaranteed to eventually complete on their own. Not needing to provide a fallback path eases the work of developers, and having a reliable way to ensure forward progress makes these transactions more suitable for critical software such as operating systems.

To ensure forward progress, a combination of increasing backoff between retries and a reduction of the size of the window of speculative execution are applied when transactions keep aborting, and a message to temporarily stop all other CPUs from executing conflicting instructions is used as a last resort [23].

2.3.4 clEAR

Cacheline-locked executed Atomic Region (clEAR) [24] monitors the first execution of atomic regions (can be critical sections or transactions, no distinction is made), tracking the memory addresses accessed and the immutability of the set of addresses. If the set is immutable, a retry is guaranteed to access the same addresses. If the first speculative execution is aborted, the next attempts take advantage of the information collected, adapting by choosing one of several possible modes of execution.

As with all other alternatives covered in section 2.3, performance evaluation was only done on a system that uses register checkpoints while speculatively retiring instructions, but a design that buffers speculative state in the ROB (in-core speculation) is also described. In the version with in-core speculation, the first execution checks that the atomic region can finish without exhausting the core's resources (ROB and SQ entries, etc.), and goes straight into fallback mode if the limits were reached. Otherwise, the following execution modes exist for subsequent retries:

- Non-speculative with cache line locking: In the best-case scenario, the set of addresses is immutable and can be locked in L1, so the addresses are locked in the same way as MAD atomics before executing the atomic region without the need for conflict detection.

- Speculative with cache line locking: If the set might change but can be locked in L1, locking is still done but conflict detection is active and the transaction might abort.

- Speculative retry: If the set cannot be locked or a speculative execution with cache line locking aborted, the atomic region is attempted with conflict detection as in conventional HTM, without cache line locking.

After too many retries, as a last resort, the lock of the critical section or a global HTM fallback lock is taken.

2.3.5 Constrained HTM with NACKs for forward progress

Nagabhiru and Byrd [5] advocate for constrained transactions, for the main use-case of lock-free programming with MCAS implemented through HTM. Similarly to constrained transactions in z/Architecture's HTM [23], they guarantee forward progress without the need of a fallback path, but this time through a NACK-based approach.

Transactions can respond to a conflicting transactional request from a core with higher core-ID with a NACK that aborts the requesting transaction. Needless to say, this method requires the addition of NACK support to the cache coherence protocol. After a number of retries, transactions can also NACK non-transactional accesses, which will have to be retried.

Additionally, some adjustments are made to increase performance, such as automatically promoting transactional read requests to write requests after a number of retries (this targets the specific access-pattern of MCAS) or increasing a linear back-off for multiple retries caused by the same transaction.

3 Objectives and methodology

In the previous work referenced in section 2.3, there are alternatives that describe mechanisms for transactional execution by buffering state in the ROB. However, none of them show a performance evaluation of this option, ostensibly because the register checkpointing alternative does not impose such strict limits on transactions. In this work, we are interested in exploring the effectiveness of the ROB buffering approach, targeting small transactions.

We also want to decrease the number of aborts and the possibility of livelocks in this alternative. Previous work focuses on combining the usual requester-wins policy of HTM with a requester-loses policy that ensures forward progress. In directory systems, requester-loses policies require the addition of NACKs, which can complicate the cache coherence protocol. Instead, we want to investigate the combination of HTM with previous efforts for atomic operations to multiple cache lines, specifically cache line locking in lex order.

Therefore, this thesis has the following objectives:

1. Implement an HTM based on the ROB buffering version of SLE. That is, an HTM that buffers speculative state in the ROB and uses existing mechanisms to detect atomicity violations.

2. Create a variant of the HTM system that tries to reduce the number of aborts by activating cache line locks in lex order.

3. Evaluate the performance of the two variants in benchmarks for small transactions.

In the following sections, we first describe the simulation setup chosen to implement the HTM. Later, we present the benchmarks used to perform the evaluation.

3.1 Simulated system

As prototyping our design in real hardware would be impractical, we implement it in a simulator. gem5 [25, 26] is an open-source cycle-level simulator that has become the standard for evaluating and exploring computer architecture designs. It is a modular simulator, allowing simulation of many ISAs with several representative CPU models.

We choose to simulate gem5-24's out-of-order CPU model in full-system mode for the x86-64 ISA. We try to define a CPU similar to an Intel Alder Lake P core [27, 28] with the parameters shown in Table 3.1. The memory subsystem is simulated with Ruby, as we need a detailed model that allows modifying the coherence protocol. The directory design and

Parameter	Value
Pipeline width	8 fetch/6 decode/6 rename/12 dispatch/12 issue/8 commit
Physical registers	332 integer + 332 floating-point
ROB	512 entries
LQ	192 entries
SQ	114 entries
RAS	64 entries
Branch predictor	TAGE-SC-L [30]
Memory dependence predictor	Store sets [31]
Caches	64B line size, write-back, write-allocate, strictly inclusive
L1 instruction	32KiB, 8-way, 1-cycle latency
L1 data	48KiB, 12-way, 1-cycle latency
L2	1MiB, 16-way, 5-cycle round-trip latency
L3	Shared, 4MiB, 16-way, 17-cycle round-trip latency
Coherence	MESI protocol with directory (embedded in L3, not sparse), SimplePt2Pt interconnection

Table 3.1: Configuration parameters

cache latencies needed for Ruby in recent gem5 versions can be unrealistic when compared with contemporary chips. Recent advances investigate the cause of this issue and incorporate modifications that allow for a more accurate simulation [29]. Additionally, we could not find an easy way to change the interconnection network in the three-level hierarchy to a more reasonable topology, such as a crossbar, instead of an all-to-all point-to-point network. We reserve the inclusion of these improvements for future work.

Regarding lex order, our system has a 48KiB 12-way L1D. Associativity is not a power of two, so for calculating the range of the lex order we choose the next lower power of two [16]. Lex order range is therefore $64\ sets \times 8\ ways = 512$.

3.2 Benchmarks

In this section, we present the benchmarks for experimental evaluation of our HTM. We use two categories of benchmarks. First, we evaluate the performance of our mechanism as a synchronization primitive in concurrent data structures. Second, we employ targeted microbenchmarks to measure its performance as a substitute of small constructs where atomic operations are commonly used.

When running each benchmark, we take a checkpoint right before starting the operations and extract the simulation statistics (simulation time, number of aborts, ...) when all threads have completed. The statistics are averaged between 10 runs of the same benchmark. gem5 is a deterministic simulator, so we introduce variability by adding different delays before creating the checkpoint in each run.

3.2.1 Data structures

We evaluate our proposal in the benchmarks for concurrent data structures developed by Kankava [32]. We incorporate the corrections suggested in [33][1]. The benchmarks measure the time taken to perform a set number of operations in the following structures:

- Arrayswap: Threads atomically swap two random elements of an array.
- Binary search tree: Threads perform a mix of insertions, deletions and lookups. The tree is not automatically balanced.
- Sorted list: A mix of insertions, deletions and lookups to a doubly-linked sorted list.
- Hash map: A mix of insertions, deletions and lookups. All implementations define the hash map as an array of lists.
- Deque, queue and stack: Insertions and deletions.

We omit *mwobject*, which was included in the original benchmarks [32], because we could not obtain a small enough difference in statistics between runs, even when performing a high number of operations.

Operations for arrayswap, deque, queue and stack only require a single transaction. However, algorithms for operations in the binary search tree, sorted list and hash map include looping constructs. We cannot afford to include these loops inside transactions because they could exhaust our HTM's capacity limits. Instead, we employ fine-grained HTM that follows the structure of MCAS-based versions of these data structures.

3.2.2 Microbenchmarks

We introduce two microbenchmarks for small critical sections outside of data structures, and provide a real use case for the Splash-4 [34] benchmark suite. The Splash suite evaluates the performance and scalability of multicore processors. Splash-4 is an updated version that introduces atomic operations to replace small critical sections, and reduces the overhead of barrier synchronization. Some atomic operations in Splash-4 had to be performed as a CAS-loop because dedicated atomics were not available [34]. Specifically, many processors do not have dedicated `fetch-and-add` and `atomic-max` atomics for floating point values. In Splash-4, they were implemented as shown in Listing 3.1 and Listing 3.2. `atomic_compare_exchange_weak` is a wrapper for the underlying CAS operation of the processor, it sets `addr` to `newValue` on success and `oldValue` to the current value in `addr` in case of failure.

[1]Source code is available at `https://github.com/alvaro-r-g/data-structure-benchmarks`.

Listing 3.1: `fetch-and-add` with CAS loop

```
double oldValue = *addr;
double newValue;
do {
  newValue = oldValue + increment;
} while (!atomic_compare_exchange_weak(addr, &oldValue,
   newValue));
return oldValue;
```

Listing 3.2: `atomic-max` with CAS loop

```
double oldValue = *addr;
do {
  if (newValue <= oldValue) break;
} while (!atomic_compare_exchange_weak(addr, &oldValue,
   newValue));
```

Using HTM, we rewrite them as shown in Listing 3.3 and Listing 3.4. In Listing 3.4 we introduce a check to avoid the overhead of creating a transaction if possible.

Listing 3.3: `fetch-and-add` with HTM

```
TM_BEGIN();
double oldValue = *addr;
*addr = oldValue + increment;
TM_END();
return oldValue;
```

Listing 3.4: `atomic-max` with HTM

```
if (newValue > *addr) {
    TM_BEGIN();
    if (newValue > *addr) *addr = newValue;
    TM_END();
}
```

The new versions are simpler and their correctness can be verified at a glance. Their effect in the overall execution time of Splash-4, however, is negligible. We measured the Splash-4 benchmarks updated with the HTM constructs[2] in gem5 and verified that, as these operations are surrounded by long blocks of computation, they do not generate a perceivable effect on execution time.

As an alternative, we create microbenchmarks to test their isolated performance. For `fetch-and-add`, threads constantly try to atomically increase a shared counter. In `atomic-max`,

[2]The affected Splash-4 benchmarks are Ocean contiguous, Ocean non-contiguous, Water-NS and Water-SP.

we initialize arrays with random elements for each thread, and threads perform `atomic-max` between each element of their local array and a shared global maximum. Elements of the arrays are chosen from a random function with an upper bound that increases with the index of the array, in order to increase the frequency of updates to the global maximum.

Both microbenchmarks depict unrealistic scenarios. Atomics are expensive (in terms of performance), and most well-written programs avoid performing them with this much frequency. Still, we include them to show that the overhead introduced by our HTM is comparable to atomics, and to test if the lex order locking mechanism is beneficial even in these simple scenarios.

4 Design and implementation

We now describe the details of our HTM. Most of the tradeoffs and intricacies of the design space were already studied in great detail by Rajwar [35], who characterized similar implementations. Our implementation can be thought of as equivalent to the SLE variant that buffers speculative state in the ROB, without automatic detection of locks, and with an added cache line locking mechanism for avoiding conflicts. It employs eager conflict detection and lazy version management.

Our implementation's behavior can be summarized in the following steps:

1. Transactions can start executing freely, but their micro-ops are not committed to architectural state (and cannot modify shared memory) until transaction commit starts. Transaction commit can only start when (1) all micro-ops are executed and ready to commit and (2) all blocks that need to be accessed are present in L1 in their required states. Meanwhile, speculative state is buffered in the ROB.

2. Blocks are loaded into L1 when loads and stores in the transaction are executed. Blocks accessed by stores are fetched by exclusive prefetches generated on the execute stage.

3. Transactional blocks previously acquired by loads and store prefetches might be invalidated or lose write permissions due to requests from other processors. Some of these requests can be delayed until the transaction is completed if the conditions explained in section 4.6 are met. Otherwise, loads and exclusive prefetches need to be retried to bring blocks back to their required state.

4. During transaction commit, the SQ and SB are drained of transactional stores. To make the draining process atomic, blocks that have pending transactional stores are locked.

A more detailed breakdown of the system is provided in the following sections.

4.1 `begin` and `end` instructions

Our interface for defining transactional regions consists of two new instructions: `begin` and `end`. An unused combination of opcode, prefixes and attributes is chosen for encoding the instructions in machine code. As an alternative, the opcodes for `XBEGIN` and `XEND` from Intel's Transactional Synchronization Extensions (which are disabled in most processors[36]) can be reused.

The front-end of x86 processors transforms complex CISC instructions into an internal RISC representation, so each instruction (macro-op) is converted into one or more micro-ops. For `begin` and `end`, each macro-op is decoded into only one micro-op of the same name. The two micro-ops are also new and are added to the internal micro-op ISA.

Micro-ops between `begin` and `end` are marked as transactional in the ROB, LQ and SQ. Other than being specially handled by the pipeline as transaction delimiters as explained in the following sections, both `begin` and `end` behave like full memory barriers (like *mfence*s), as is the case for atomics in x86 [15]. Full memory barriers prevent younger memory micro-ops from executing until the barrier reaches ROB head and the SB is empty. This simplifies the design of our HTM, preventing unwanted interactions with memory operations from outside the transaction.

4.2 Buffering speculative state

Micro-ops that are part of a transaction are buffered in the ROB. This is achieved by not committing the `begin` micro-op once it reaches ROB head until it is squashed or the transaction commit stage starts. As micro-ops can only commit and retire in-order through the ROB head, this effectively delays the whole transaction (and any other ROB instructions), and prevents it from being committed to architectural state.

Consequently, transactional modifications to the cache are also delayed because stores only access memory once they commit. Loads, on the other hand, access memory when they execute, and can forward the speculatively loaded values to other micro-ops in the transaction before the transaction commit stage starts.

4.3 Misspeculation and squashes

Conditional and unconditional branches can be executed in our transactions, which might cause some instructions in the transaction to be squashed. Modern processors also perform memory dependence prediction, so speculatively executed loads that are incorrectly predicted to access a different memory address than previous stores can also cause squashes.

Depending on how the processor handles misspeculation, squashed instructions can contribute towards reaching the capacity limits of the transaction. Specifically, removal of squashed instructions can be done lazily or eagerly. If the processor implements lazy squashing, instructions in the ROB are marked as squashed and can only be retired when they reach the ROB head. Once a transaction starts executing, the ROB head is blocked by `begin`, so squashed instructions can only retire if `begin` itself is squashed (during a transaction abort) or if it commits (during transaction commit). Until squashed instructions are retired, they keep occupying resources in the processor's structures (ROB, LQ, SQ and register allocation table (RAT)). Therefore, our implementation is more suited for processors with eager squashing, which immediately free ROB entries upon misspeculation. Still, our implementation maintains the lazy squashing implemented in gem5, as we did not find any capacity issues with the short transactions that we use.

4.4 Loading required blocks into L1

To make transactional execution atomic, all blocks that will be accessed during the transaction need to be present in L1 before transaction commit starts. Blocks only accessed by loads need to have any state other than I before the transaction commit begins. Blocks accessed by one or more stores need to be in E or M state.

Loads already bring their required block into L1 when they execute. When stores execute, on the other hand, they do not need to access the block they write to. However, many prefetching strategies have been researched and implemented for normal execution (not necessarily transactional) that preemptively ask for the block to be loaded with write permissions in order to prevent a cache miss once the store accesses memory after committing. Many modern processors perform prefetches when stores commit [15], before the stores reach the head of the SB. We generate an exclusive prefetch (it loads the block[1] in E state) when transactional stores execute.

Usually, prefetches can only affect performance, and not correctness of the program, and as a result they can be discarded depending on the state of the block they find once they reach the cache controller. For example, the exclusive prefetches that are available in gem5's MESI three-level protocol only attempt to acquire exclusive permission for a block if it was in I state. They also do not need to notify the core once they succeed in loading the block.

In this case, however, exclusive prefetches for transactional stores need to complete for the transaction to progress, so they should not be discarded and should eventually send a response to the processor, notifying that the data block is ready. We have modified the cache coherence protocol and cache controllers to support these special prefetches. Firstly, we included a new type of coherence message, named `Tx_PF_Store`, that the core can send to the L1 cache when requesting a transactional exclusive prefetch. Secondly, we created two new intermediate states for cache blocks named `Tx_PF_IE` (invalid block waiting for exclusive permission) and `Tx_PF_SE` (block in shared state waiting for exclusive permission), that are used when the `Tx_PF_Store` reaches a block without exclusive permission in L1. Lastly, we modified the cache controller to send a notification back to the core once the data requested by a `Tx_PF_Store` is ready in L1. When the core receives this response, it marks the exclusive prefetch as completed.

[1]We keep saying "block", but a load or store might have to access more than one block if it is an unaligned access to a cache line boundary. Our implementation takes this into account. For example, exclusive prefetch requests can generate multiple packets to load multiple cache lines.

4.5 Conflict detection and resolution

Conflicts appear in the process of loading blocks accessed during the transaction into L1. Conflicts are detected by checking if L1 notifications of invalidations or losses of write permissions change transactionally accessed blocks from their required state. Any stores or loads that required the block will request the block again. External requests that cause load or store retries can come from other processors which may or may not be executing a transaction, so our implementation provides strong atomicity [19].

In the case of stores, conflicts with external accesses are detected by searching the SQ whenever notifications of invalidations or losses of write permissions are received by the processor. Processors usually do not have any reason to be notified when they lose write permissions for a cache line if the block is still valid, so this is a specific additional requirement of our implementation that we include in the cache controllers. If there is a coincidence with an executed transactional store, it needs to recover by re-sending its exclusive prefetch.

Loads are only affected by invalidation notifications. However, their recovery mechanism is not so simple: in addition to retrying the load, any execution depending on the loaded value (which was just invalidated) needs to be discarded to maintain atomicity. As explained in subsection 2.1.4, CPUs already do this by default, triggering a full flush of the processor pipeline in the case of Intel [37]. As an aside, a valid implementation of the mechanism that searches the LQ to find loads affected by invalidations might skip the first entry of the LQ. This entry contains the oldest uncommitted load, so there is no older load that it could have speculatively overtaken. However, during transactions, this first entry also needs to be taken into account.

gem5 models the retry of speculative loads by marking invalidated loads with a fault. When this fault reaches the ROB head, it triggers an exception that flushes the pipeline and starts re-executing from the PC of the invalidated load. In our implementation, this exception would be triggered and handled during transaction commit, which would break the atomicity of the process. To prevent this, we trigger an early processor flush whenever a transactional load is marked with a fault. As the entire transaction is being buffered in the ROB, processor flushes result in a complete transaction retry, which is similar to a transaction abort in conventional HTM implementations.

Figure 4.1 shows an example of this situation. Initially, the entire transaction is buffered in the ROB. After an invalidation to block B, the LQ and SQ are searched. The store `st r2, B` had performed its exclusive prefetch, so it will need to retry and complete the prefetch again before the transaction can commit (this does not require any squashes). The load `ld B, r2` was also executed and is marked with a re-execution fault. When the re-execution fault is detected in a transaction, all in-flight instructions (including those after the transaction) are squashed. After this, fetching starts at the `begin` micro-op and the transaction is retried once the squashed instructions are drained.

To prevent livelocks between conflicting transactions, we introduce a random exponential backoff period between retries depending on the number of aborts. We also test a variant where the backoff is increased when exclusive prefetches are retried.

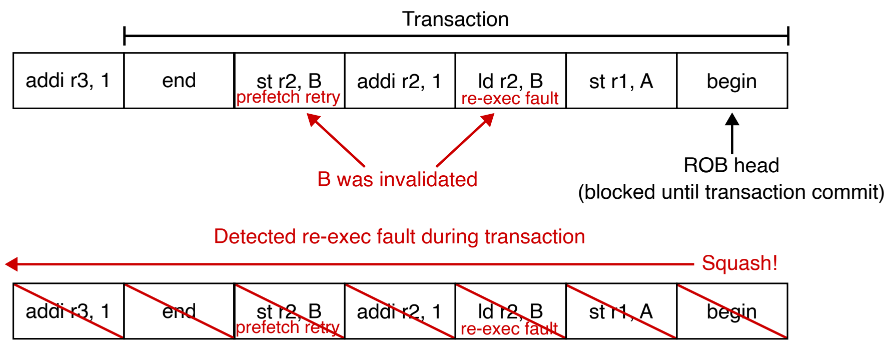


Figure 4.1: Example showing a transaction being squashed after receiving a conflicting invalidation.

4.6 Locking

To increase the chances that a processor will, at some point, have all the blocks it needs in L1, we activate cache line locks that prevent invalidations and downgrades. In contrast to cache line locking in CLEAR (subsection 2.3.4), instead of accessing the cache only after blocks have been locked, we access memory freely and lock as many cache lines as possible once blocks are brought to L1. On one hand, we avoid the bottleneck of having to wait for lock acquisition before sending subsequent requests, and our procedure easily accommodates read and write sets that vary between retries. On the other hand, this procedure is speculative and vulnerable to invalidations (which cause aborts) at many points in time.

We keep track of the read and write set of the transaction (the number of transactional loads and stores that access each cache block) to know which blocks should try to be locked. The read and write set of the transaction can be derived from the target addresses of all executed (and not squashed) transactional entries in the LQ and SQ.

In conventional processor designs, the L1 cache controller can be easily connected to the LQ and SQ. In gem5, however, there is currently no direct way of generating communication between them. We simulate a realistic interaction by sending instantaneous messages to the cache controller whenever blocks are added or removed from the read or write sets, so the cache controller can keep track of the blocks accessed during the transaction.

Locks can only be acquired for blocks in L1 that are in the state required for the transaction (a state such that they are ready to be accessed in the transaction without sending or waiting for additional coherence requests). A locked block delays any coherence requests that would change its state to one that is not ready for the transaction. The delay lasts until the block is unlocked.

The read and write set of a transaction changes as it executes. Thus, we are presented with the challenge of acquiring locks in a deadlock-free manner for a changing set of addresses. Our approach is to try and lock as many addresses as possible in a locking process that follows the lex order explained in subsection 2.2.4. This means that a block that is ready can only be locked if all previous blocks (according to the lex order) have been locked. When a new block is added to the transaction, it can only be locked if the complete locking sequence follows lex

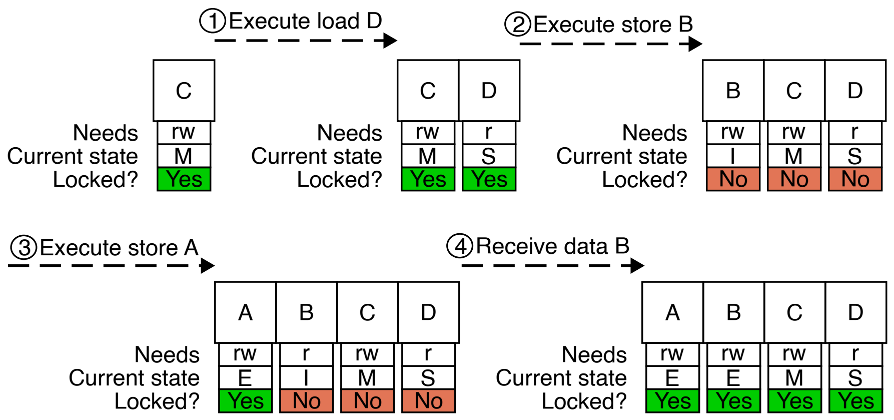


Figure 4.2: Example showing locking in lex order as blocks are added to the read and write sets of a transaction.

order. If it cannot be locked, some locks need to be released and reacquired in a different order than they were originally locked in, taking into account the added step of locking the new address.

Figure 4.2 shows an example for a transaction that is in the middle of executing. Block addresses are represented with letters, and the lex order follows the lexicographical ordering of the letters. At the initial time, the only block accessed is C, and C needs to be written. C is present in M state, so any requests from other processors that attempt to read or write to C can be delayed.

Next, in ① the transaction executes a load to D. The cache happens to already have read access to D, and all previous blocks (C) are also in their required state, so any write request from other processors to D can be delayed. External read requests to D are allowed because they do not change the state of D. C can still be locked and delays read and write requests. In ②, the transaction executes a write to address B. The processor does not yet have access to B, and requests it. While B is being fetched with write permissions, B cannot be locked (because the block is not present). As a consequence, all following blocks in lex order (C and D) cannot be locked. In ③, the transaction executes a write to address A. Block A happens to be available with write permissions, and can be locked. In ④, block B is received with write permissions, so all transaction blocks can be locked.

A consequence of this locking method is that once all transaction blocks are ready in L1, they can all be locked, so the same locking mechanism is used to retain blocks through the commit stage.

The locking in lex order that we have discussed so far avoids all possible deadlocks in private structures. For shared structures (shared caches and sparse directories), it remains to activate locks for entire sets when activating locks for blocks with the same lex order, as indicated in subsection 2.2.4. The lex order conflict can appear when a new access is added to the transaction, and a conflicting block might already be locked in the directory. In this case, to avoid deadlock the individual lock has to be released before activating the special lock for all the ways in the directory. We do not implement this mechanism because the deadlock scenario is highly unlikely.

4.7 Transaction commit

There are two conditions before transaction commit can start:

1. All micro-ops of the transaction must have executed and must be ready to commit. This ensures that all memory accesses of the transaction have executed and all modifications to register state are buffered in the ROB. This also allows checking that no exceptions will be found in the process of committing the transaction.

2. All blocks accessed during the transaction must be present in L1 in a state that allows them to be accessed without generating or waiting for additional coherence messages (as described in section 4.4).

Both conditions will be met if all transactional loads and stores are executed (and not squashed) and all the exclusive prefetches have completed, which is what our implementation checks before starting transaction commit. Once the conditions are met, `begin` is allowed to commit, and so are all instructions of the transaction until `end`. Stores, after committing, are allowed to access memory. The locking mechanism explained in section 4.6 ensures that transactional modifications to the cache are perceived as atomic by other processors. As a result, we unlock blocks once their last store completes (when stores complete and are removed from the SB, they are also removed from the write set of the transaction).

Blocks that are only read need to be in L1 before the transaction commit stage starts, but they do not need to remain locked through the commit stage, because all loads have already executed and loaded values are stored in the ROB. Thus, we clear the transaction read set on transaction commit stage start, so blocks that are only read are no longer candidates for locking. We disable squashing due to invalidation notifications for transactional loads while the transaction is in the process of committing.

When the `end` instruction reaches ROB head, as it behaves like a full memory barrier, it can only commit once all transactional stores have completed and the SB is empty.

4.8 Exceptions and interrupts

We follow the naming convention that treats exceptions as internal events (e.g. divide-by-zero error) and interrupts as external events (e.g. timer or I/O device). We need to discuss interrupts and exceptions because a requisite for atomicity is that interrupts and exceptions cannot be handled during transaction commit [35].

Once the CPU notices that an interrupt is pending, it stops fetching and waits until all in-flight instructions have finished. This requirement implies that interrupt handling waits until the ROB is empty, therefore we know that interrupts will be automatically delayed until the current transaction ends. However, the CPU might stop fetching halfway through a transaction. We detect these doomed transactions when fetch is stopped, `begin` is blocking the ROB head and no `end` micro-ops are in-flight. We squash doomed transactions to drain the ROB, and handle the interrupt before they are retried. In our benchmarks, interrupts are infrequent and result in very few (if any) aborts.

Exceptions, on the other hand, cannot be delayed. Exceptions in gem5 are caused by "faulty" instructions. Before each instruction is committed, the processor checks if it has an associated fault. If a fault is found, the processor is forced to instantly squash all in-flight instructions and handle the exception corresponding to the fault. To avoid doing this midway through transaction commit, before starting transaction commit we first wait for all instructions to be completely executed, so they produce all the faults they can, and then we scan the ROB to see if any transactional instruction is faulty. Some exceptions (such as the one mentioned in section 4.5) can be handled by just squashing the entire transaction and re-executing. This is not the common case, and most faults will keep appearing after re-execution until the handling routine is carried out. We have not yet developed a solution for the rest of exceptions (although page faults can be mitigated by prefaulting the memory accessed by transactions). Many HTMs systems handle exceptions by taking the fallback path.

4.9 Limitations

We now summarize the main limitations of our system:

- Interrupts and exceptions cannot be handled during transactions (this includes system calls, which invoke exceptions).
- To make sure that all blocks accessed in the transaction fit into L1, the number of cache lines cannot be greater than L1 associativity. In our simulated CPU, this means that transactions can access a maximum of 8 different cache lines.
- Transactions have a capacity limit set by the size of the ROB, LQ, SQ and RAT. The programmer is held responsible for creating simple enough transactions so that this does not happen.
- Misprediction squashes count towards capacity limits in processors with lazy squashing. It is recommended not to use complex control flow inside transactions, such as difficult-to-predict conditionals inside loops.
- The system does not have a fallback path. Instead, we rely on the effectiveness of the backoff mechanism.

We also highlight the overhead of executing transactions. This might be a performance attribute, more than a limitation, but it could be improved and needs to be taken into account.

The innate overhead of transactions (without taking conflicting transactions into account) is due to (1) the fences at the start and end of transactions (which have been shown to harm performance in atomic instructions [38]) and (2) temporarily refraining from committing instructions to buffer transactional state. Similarly to atomics, it is convenient for performance to only execute transactions if necessary.

5 Evaluation

We simulate variants of our implementation in multiple benchmarks. A detailed description of the benchmark workloads is included in the following sections, as well as a discussion of the gathered results.

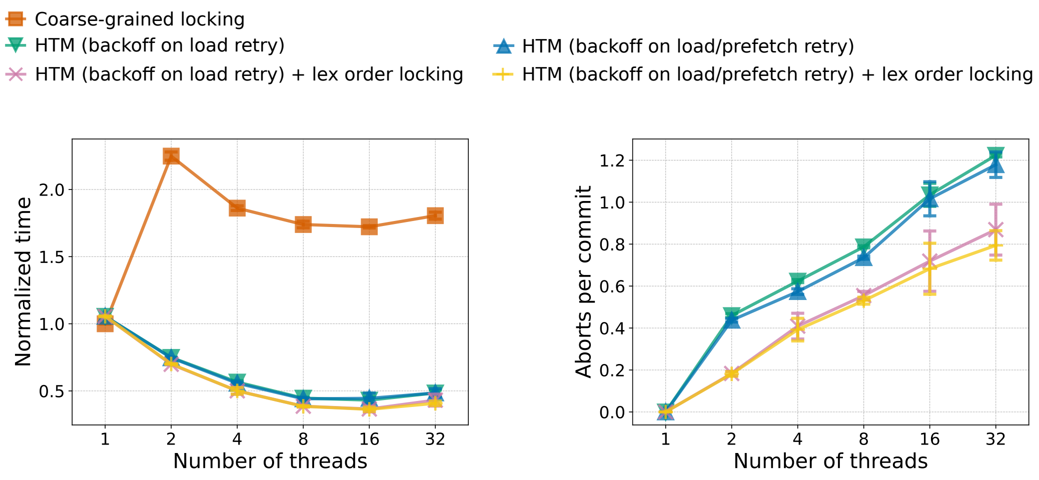
5.1 Data structure benchmarks

We do not use pinned threads in any of the tests, and instead thread placement is handled by the OS, resulting in a more realistic benchmarking environment. In each benchmark, the same total number of operations has to be performed regardless of the number of threads. We try to choose a high enough number of operations for each benchmark so that the standard deviation of the results between runs is low, but we still include error bars in the results.

For data structures that allow different operations, we try to generate a representative mix. For the deque, queue and stack we generate 50% insertions and 50% deletions. For the sorted list, hash map and binary search tree we generate a first stage only with lookups, a second mixed stage (10% insertions, 10% deletions and 80% lookups) and a third stage of only updates (50% insertions and 50% deletions).

We compare the execution time of coarse-grained locking (except for arrayswap, where there is one lock per element of the array, and therefore uses fine-grained locking) against two major HTM variants. The first variant consists of our in-core HTM, without lex order locking, that performs conflict resolution like a conventional HTM with exponential backoff. The second variant includes our lex order locking mechanism for reducing aborts. For both, we include two sub-variants that increase the exponential backoff when an exclusive prefetch has to be retried (i.e., in these sub-variants, both loads and store retries increase the counter that determines the upper limit of the random backoff delay).

In Figure 5.1 we summarize the results of the benchmarks over all data structures. Specifically, Figure 5.1a plots the geometric mean of the normalized execution time. Time is normalized over the execution time of the lock-based algorithm for 1 thread. We also plot the average number of aborts (transactions that are completely squashed due to interrupts or invalidations of loaded blocks) per commit in Figure 5.1b. It is difficult to generate accurate energy metrics from computer architecture simulations, but a reduction in the number of aborts can be a good proxy that correlates with a decrease in energy consumption. Each abort avoided with our locking method is achieved by delaying a requesting transaction instead of aborting and re-executing the transaction that receives the request. Hence, by reducing aborts we prevent discarding useful speculative state.



(a) Geometric mean of normalized execution times.

(b) Average number of aborts per commit.

Figure 5.1: Summary of results for data structure benchmarks.

In what follows, we explore the specific results for each of the data structures in more detail. The results are grouped separately for execution time and number of aborts per commit in Figure 5.2 and Figure 5.3, respectively.

5.1.1 Arrayswap

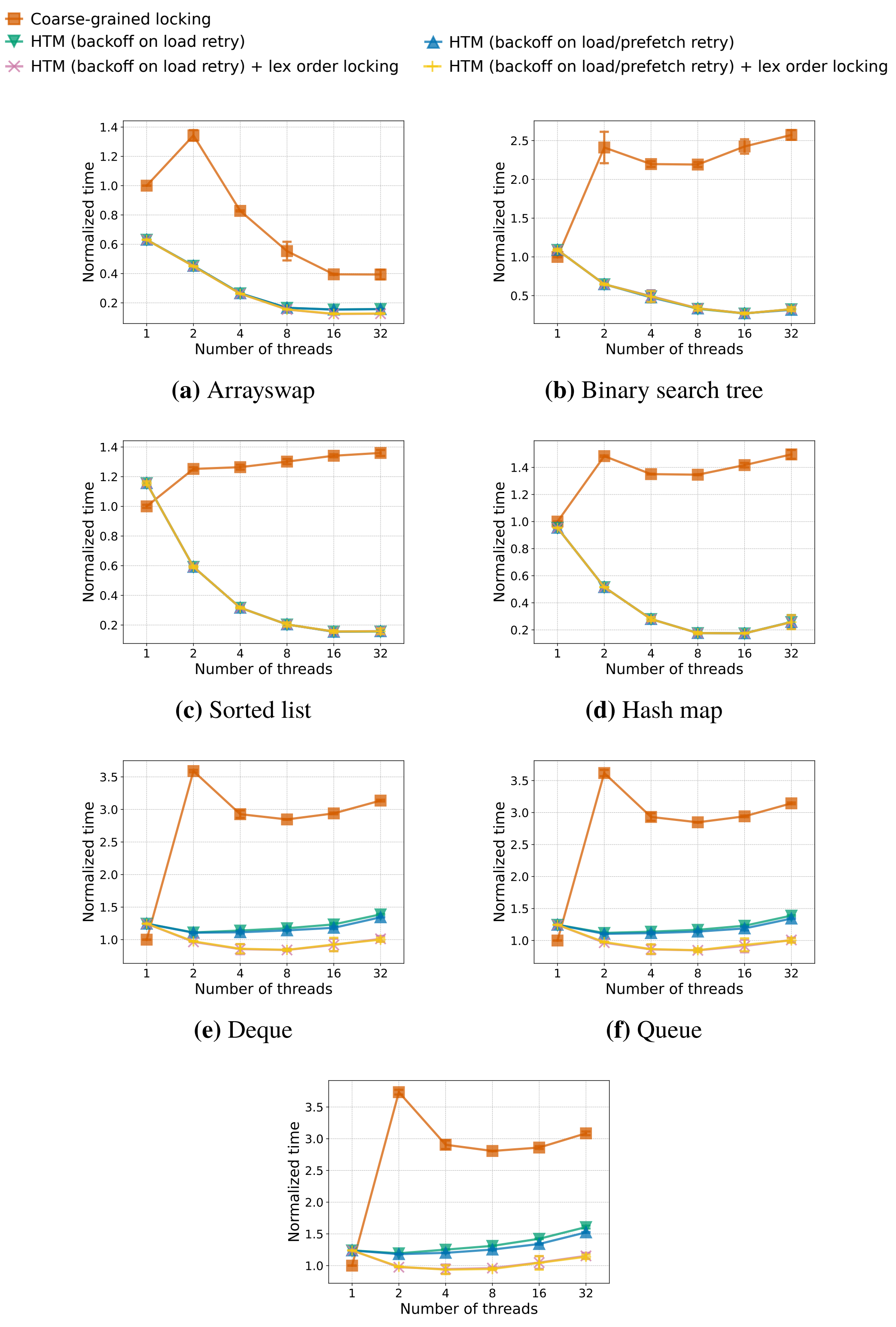
In arrayswap, all HTM versions outperform the locking alternative. The size of the array is constant (16 cache lines, specifically) so, understandably, the speedup decreases when having a higher number of cores. The alternative with lex order locking significantly reduces the number of aborts and, for high core counts (and therefore high contention), it is slightly faster.

5.1.2 Binary search tree, sorted list and hash map

The HTM variants perform similarly due to the low number of aborts, but they all outperform the locking alternative. The low ratio of aborts is due to using small MCAS transactions in structures with many different possible conflict points, and due to taking many lookup operations into account. Still, lex order locking manages to halve the number of aborts on average.

5.1.3 Deque, queue and stack

These are the benchmarks where lex order locking is most beneficial for execution time. This is due to the high contention (as we can see by the rate of aborts in Figure 5.3e, Figure 5.3f and



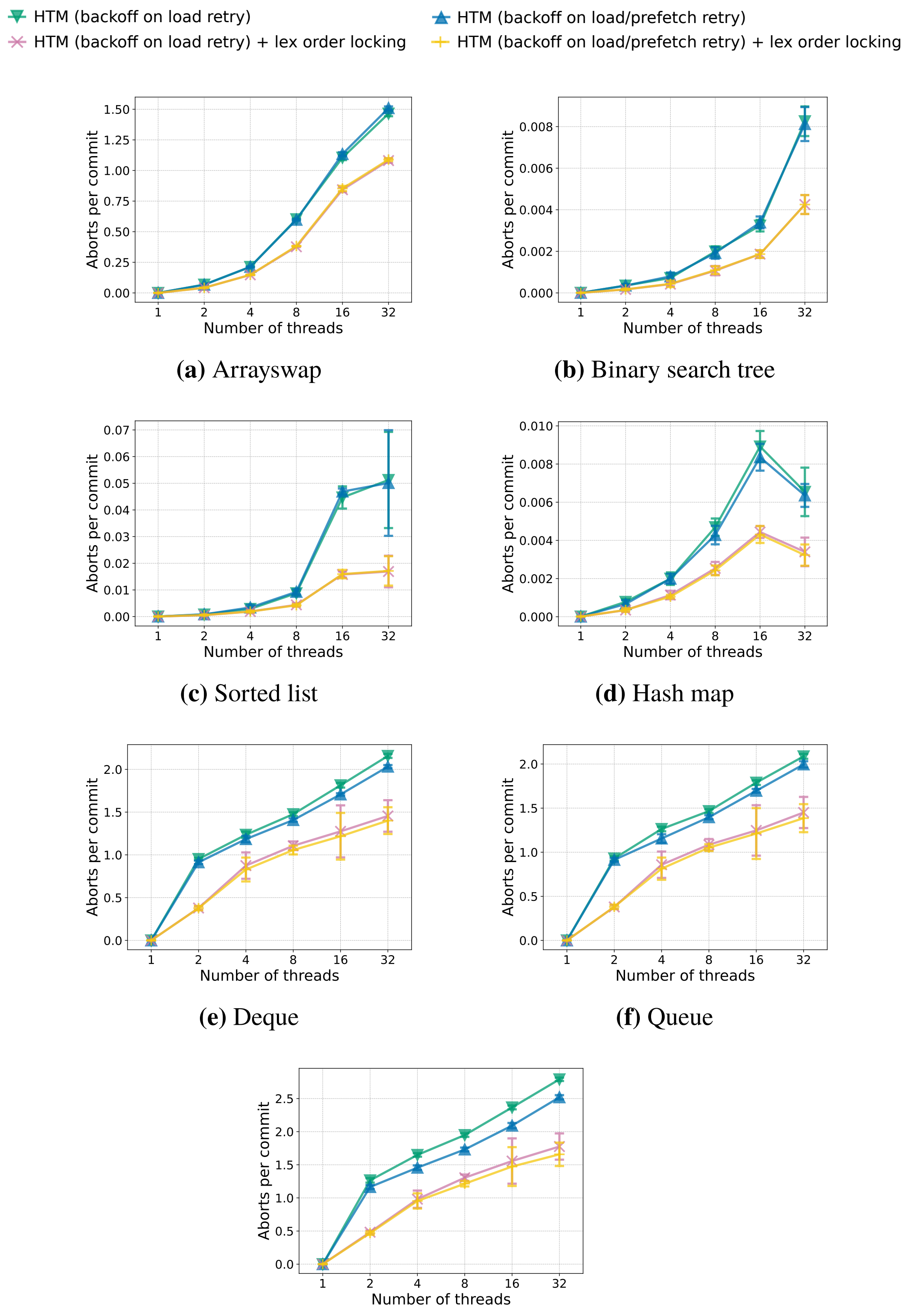
(a) Arrayswap **(b)** Binary search tree

(c) Sorted list **(d)** Hash map

(e) Deque **(f)** Queue

(g) Stack

Figure 5.2: Normalized execution time in data structure benchmarks



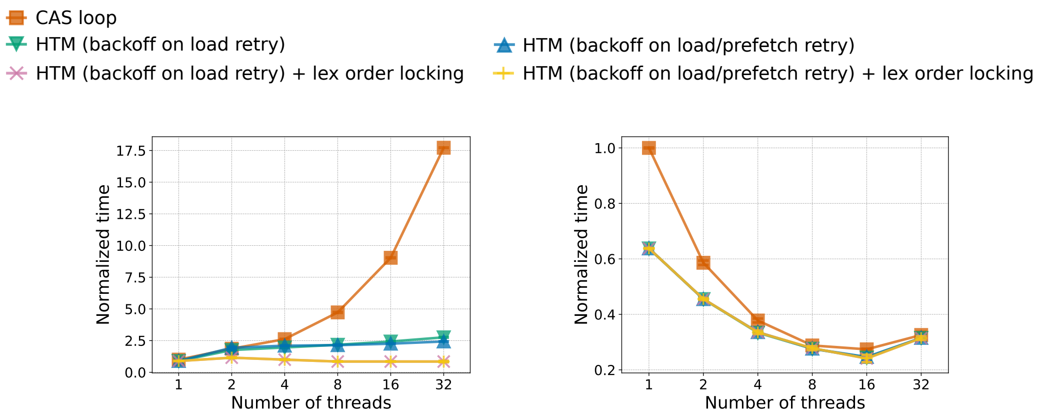
(a) Arrayswap **(b)** Binary search tree **(c)** Sorted list **(d)** Hash map **(e)** Deque **(f)** Queue **(g)** Stack

Figure 5.3: Aborts per commit in data structure benchmarks

Figure 5.3g), as threads are performing constant updates in, at most, two different insertion and deletion points. Once again, lex order locking significantly reduces the number of aborts.

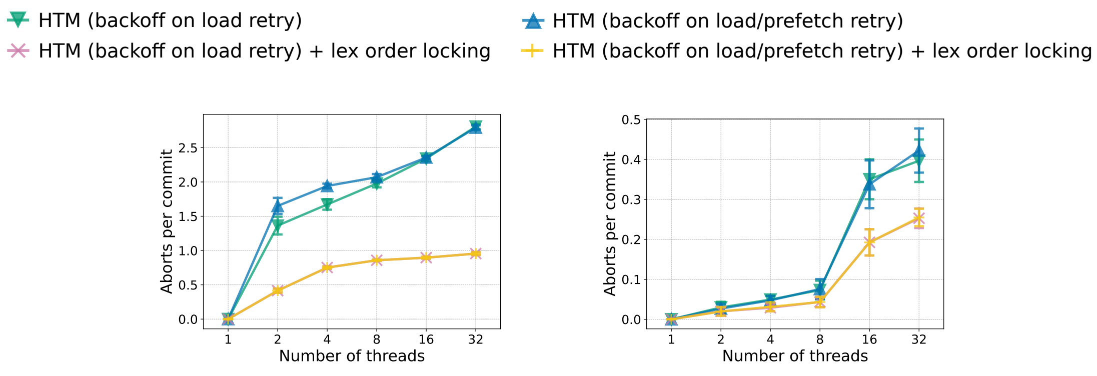
5.2 Microbenchmarks

Figure 5.4 and Figure 5.5 plot the normalized execution time and number of aborts per commit in the microbenchmarks, respectively.



(a) Fetch-and-add double **(b)** Atomic max

Figure 5.4: Normalized execution time in microbenchmarks



(a) Fetch-and-add double **(b)** Atomic max

Figure 5.5: Aborts per commit in microbenchmarks

The scalability of HTM outperforms the CAS loop in fetch-and-add. In this scenario, even when a single cache line is being modified in each transaction, the variant with cache line locking reduces execution time and the number of aborts. The order in which locks are acquired is irrelevant in this case because there is a single cache line. What generates the reduction in the number of aborts is being able to instantly lock the cache line when it is received with write permissions, even before transaction commit starts. In the baseline, a transaction might have all the required blocks in L1 but will have to wait in a vulnerable state for the core to start transaction commit (the core needs to check that all instructions have executed and all loads and exclusive prefetches have completed).

In atomic max, updates to the global maximum are still not too frequent (even after initializing the arrays from a random function with an increasing upper bound), and execution time becomes similar to the CAS loop version when the number of cores increases.

6 Conclusion and future work

In this work, we implemented an HTM that buffers speculative state in the ROB. The HTM mechanism required minimal changes to existing hardware. In our benchmarks, it performs better than coarse-grained locking and similarly or better than CAS loop constructs, which highlights the low overhead of the implementation. Although HTMs that buffer transactions in the ROB have not been thoroughly explored in the past, our results suggest that they are viable for efficiently supporting small transactions.

We later incorporated locking of transactional blocks in a non-deadlocking lex order. We measure an important reduction in the number of aborts (by 30% on average for 32 cores) that, in addition to reducing execution time in contended scenarios, is likely to result in energy savings. Variants of the HTM which increase backoff when exclusive prefetches are retried were tested, but they did not provide any benefits when already using lex order locking.

The main pending work in our HTM is to tackle the limitations listed in section 4.9. For instance, the limit of 8 cache lines can be prohibitive for some transactions. Other main limitations include proper handling of exceptions and ensuring forward progress. Forward progress can be guaranteed by software (by requiring a fallback path) or by hardware (through the conflict resolution mechanism [5, 22] or by stopping all conflicting processors [23]). Ensuring forward progress without the need of a fallback path seems to be the most attractive option, as this would finish establishing our HTM as a versatile replacement for many short atomic operations.

Bibliography

[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] Dmitry Vyukov, Sanjay Ghemawat, Mike Burrows, Jeffrey Yasskin, Kostya Serebryany, Hans Boehm, and Ashley Hedberg. The danger of atomic operations, Jun 2021. URL https://abseil.io/docs/cpp/atomic_danger. Accessed: 2024-11-10.

[3] Herb Sutter. The trouble with locks. *C/C++ Users Journal*, 23(3), 2005.

[4] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. ISBN 9781608452361.

[5] Mahita Nagabhiru and Gregory T Byrd. Achieving forward progress guarantee in small hardware transactions. *IEEE Computer Architecture Letters*, 2024.

[6] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on computers*, 37(5):562–573, 1988.

[7] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. *A primer on memory consistency and cache coherence*. Springer Nature, 2 edition, 2020.

[8] *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices, 2024. URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. Accessed: 2025-05-07.

[9] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. *Reducing memory and traffic requirements for scalable directory-based cache coherence schemes*. Springer, 1992.

[10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[11] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. *Two techniques to enhance the performance of memory consistency models*. Computer Systems Laboratory, Stanford University, 1991.

[12] 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.

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

[14] 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.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2024. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[16] Alberto Ros and Stefanos Kaxiras. Non-speculative store coalescing in total store order. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 221–234. IEEE, 2018.

[17] 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.

[18] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, 2008.

[19] Colin Blundell, E Christopher Lewis, and Milo MK Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17–17, 2006.

[20] Rubén Titos-Gil. Sistemas de memoria transaccional hardware: Políticas, conflictos y su influencia en el rendimiento. Master's thesis, Universidad de Murcia, 2007.

[21] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 294–305. IEEE, 2001.

[22] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs. *ACM SIGOPS Operating Systems Review*, 36(5):5–17, 2002.

[23] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM system z. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36. IEEE, 2012.

[24] Eduardo José Gómez-Hernández, Juan M Cebrian, Stefanos Kaxiras, and Alberto Ros. Bounding speculative execution of atomic regions to a single retry. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 17–30, 2024.

[25] 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.

[26] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[27] Agner Fog. The microarchitecture of Intel, AMD and VIA cpus: An optimization guide for assembly programmers and compiler makers, 2024. URL `https://www.agner.org/optimize/microarchitecture.pdf`. Accessed: 2025-05-07.

[28] Agner Fog. Instruction tables: Instruction latencies, throughputs and micro-operation breakdowns. `http://www.agner.org/optimize/instruction_tables.pdf`, 2023. Accessed: 2025-05-07.

[29] Joaquín Ferrer, Juan M Cebrian, Ricardo Fernández-Pascual, and Manuel E Acacio. Precise characterization of coherence activity in multicores using gem5. *The Journal of Supercomputing*, 81(8):1–30, 2025.

[30] André Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[31] George Z Chrysos and Joel S Emer. Memory dependence prediction using store sets. *ACM SIGARCH Computer Architecture News*, 26(3):142–153, 1998.

[32] Nodari Kankava. Exploring the efficiency of multi-word compare-and-swap. Master's thesis, Uppsala University, 2020.

[33] Álvaro Rubira García. Análisis de técnicas de sincronización en estructuras de datos concurrentes. Bachelor's thesis, Universidad de Murcia, 2024.

[34] Eduardo José Gómez-Hernández, Juan M Cebrian, Stefanos Kaxiras, and Alberto Ros. Splash-4: A modern benchmark suite with lock-free constructs. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–64. IEEE, 2022.

[35] Ravi Rajwar. *Speculation-based techniques for transactional lock-free execution of lock-based programs*. The University of Wisconsin-Madison, 2002.

[36] 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: 2025-06-17.

[37] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468, 2021.

[38] Ashkan Asgharzadeh, Juan M Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. Free atomics: hardware atomic operations without fences. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 14–26, 2022.

Acronyms and abbreviations

CAS	Compare-and-swap.
HTM	Hardware transactional memory.
ISA	Instruction set architecture.
LL/SC	Load linked/store conditional.
LQ	Load queue.
MAD atomics	Multi-address atomic operations.
MCAS	Multi-address compare-and-swap.
RAT	Register allocation table.
ROB	Reorder buffer.
SB	Store buffer.
SLE	Speculative lock elision.
SQ	Store queue.
TM	Transactional memory.