In-Core, Low-Cost Hardware Transactional Memory

Álvaro Rubira-García, Eduardo José Gómez-Hernández, Rubén Titos-Gil and Alberto Ros
Computer Architecture and Parallel Systems Group
University of Murcia
Murcia, Spain
{alvaro.r.g, eduardojose.gomez, rtitos}@um.es, aros@ditec.um.es

***Abstract*—Transactional memory provides a simple alternative for synchronization, such that the programmer defines transactions and the underlying system guarantees their atomicity. With hardware transactional memory (HTM), processors incorporate modifications that enable efficient execution of transactions, mainly through speculative execution. These modifications commonly require fast register checkpoints and non-trivial changes in latency-critical first-level caches.**

Motivated by the growing number of in-flight instructions in current processors, this work revisits and extends an alternative approach that leverages existing out-of-order execution support to implement HTM with minimal microarchitecture changes. In this way, the design and verification complexity is reduced, while still easing the task of programming concurrent data structures.

To our knowledge, this is the first performance evaluation of an in-core HTM that buffers instructions in the reorder buffer. Using the gem5 simulator, we compare our in-core, low-cost hardware transactional memory (LHTM) implementation against a lock-based baseline and a full-blown HTM model of Intel TSX with register checkpointing and versioning in cache. In spite of its simplicity, for 32 cores LHTM achieves performance on par with TSX on a benchmark suite of concurrent data structures.

***Index Terms*—hardware transactional memory, concurrency, multithreading**

I. Introduction

The unpredictable nature of parallel applications makes programming for large-scale, multi-core systems a significant challenge. An unexpected race condition can introduce a deadlock, a livelock, crash the application, or even produce wrong results. Hardware transactional memory (HTM) addresses this issue by letting programmers delimit groups of instructions to be executed atomically as transactions, while the underlying system leverages hardware mechanisms to efficiently enforce atomicity. This is usually done by speculatively executing transactions while monitoring concurrent accesses to shared memory; detected conflicts cause the transaction to abort and typically restart [1].

At first glance, HTM appears to be the perfect solution, but this illusion dissipates when considering the hardware costs associated with it: a checkpoint is required to recover the state of the CPU before starting the transaction in the event of an abort [2]–[4]; for handling large transactions, hardware might require managing speculative data in the cache by versioning the cachelines [4]; and caches are also modified to track the read and write sets of the transaction [2], [3].

However, researchers have noted that modifications to primary caches and coherence protocols should be minimized [5], [6]. As a consequence, HTM adoption has been limited by hardware complexity and verification challenges. Most HTM designs aim to simplify implementation by reusing existing hardware mechanisms (e.g., leveraging the cache coherence substrate to detect conflicts), although this has not stopped bugs and vulnerabilities from occurring [7]–[9]. We follow the same principle and go further, reusing existing microarchitectural components to build a minimalist HTM with reduced complexity and overhead.

Speculative lock elision (SLE) [10], an early proposal for hardware-enabled critical sections, suggests two alternatives: 1) using the reorder buffer (ROB) to store all speculative information and 2) using a register checkpoint. At the time, the speculative structures available in processors were very limited, establishing the checkpointing solution as the only one that made sense to implement and evaluate. However, over 20 years have passed: computations and, more importantly, resources inside the CPUs have increased drastically. In the Pentium 4 era, processors aimed for around 128 ROB entries, with reserve stations in the order of 60 entries, and load/store queues in the range of 30-40 entries each [11]. Today, processors' speculative structures are over 3 to 5 times larger than the Pentium 4 figures. With these new resources, using the speculative structures for checkpointing in a critical section is no longer that limiting and provides an appealing picture for revisiting the ROB-based implementation (although we do not explore the lock acquire and release detection that SLE incorporates [10]).

This alternative still imposes limits on large transactions, but we focus on the usage of HTM for simple, short atomic operations that are ubiquitous in concurrent data structures and scientific workloads. Commercial HTM has been shown to be most efficient in these short transactions [12], and provides a versatile alternative that allows manufacturers to avoid constantly modifying the ISA with new atomic operations.

This work proposes in-core, low-cost hardware transactional memory (LHTM). We extend SLE's proposal in order to design and evaluate a complete low-cost implementation for the x86-64 ISA. By reusing the core's own speculative execution facilities, LHTM is able to stall in-flight transactional modifications and detect conflicts. Then, it commits transactional

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO, grant agreement No. 819134) and from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00).

updates through a coarse-grained version of the mechanism used by read-modify-write (RMW) atomic operations that target data in first-level caches.

Using the gem5 simulator [13] and a benchmark suite of concurrent data structures [14], we demonstrate that the discarded idea of using the speculative structures of the CPU for checkpointing has become viable in recent years. LHTM is able to maintain performance similar to a more complex Intel TSX, without significant modifications to first-level caches.

II. Background

Our HTM takes advantage of many mechanisms that are already present to optimize performance in modern processors. For this reason, we start by providing necessary background on modern CPUs. Later, we introduce the implementation proposed in SLE [10], which this paper extends.

A. Out-of-order cores

1) Reorder buffer: Dynamically scheduled superscalar processors reduce stalls by executing instructions speculatively and out-of-order (OoO), as soon as operands and functional units are available. Only the front-end and commit stages operate in-order, ensuring a consistent architectural state. Processors may also speculate on unavailable information, storing predicted results in the reorder buffer (ROB) [15]; if a prediction is wrong, the affected entries are squashed to discard incorrect state. Likewise, the ROB could be used to try and execute a group of instructions atomically [10], squashing them if the predicted input values (the transaction's read set) have potentially been modified by a concurrent thread.

2) Load queue and store queue: Load and store instructions are tracked in the load queue (LQ) and store queue (SQ), respectively, from dispatch until commit. Loads may access memory speculatively, while stores defer their updates until commit to preserve in-order visibility. Once a store commits, it is removed from the SQ and placed in the store buffer (SB), allowing the core to continue without waiting for the memory write to complete. The SQ naturally lends itself to version management [10], as it buffers stores until it is safe to make them visible according to the memory consistency model.

3) 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 [16] 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 [17]. 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 yet conservative 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. As a side effect, cores need to be notified with a snoop whenever other cores invalidate any of their valid cachelines (usually before writing), offering a convenient way to detect conflicts in transactional execution [10].

B. Hardware cache locking

Modern systems include hardware cache locking mechanisms to facilitate the execution of atomic RMW operations. Special requests can lock cachelines with exclusive permissions, delaying all invalidations and downgrades until a corresponding store releases the lock [18]. Only external requests are stalled, and local operations can access locked cachelines. Although current processors do not fully realize the potential of cacheline locks for efficient multi-address atomics (e.g., multi-address compare-and-swap (MCAS)), they could be used to atomically commit transactional updates [19].

C. Speculative lock elision

Proposed by Rajwar and Goodman [10], 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 conventional 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, SLE describes efficient mechanisms for HTM. Essentially, SLE buffers speculative updates to memory by not letting the stores exit the SB until the lock elision is validated, and two alternatives are mentioned to buffer the architectural register state: using the ROB or using a register checkpoint. We focus on the ROB-based alternative, which avoids modifying shared state until the transaction is successful by preventing instructions from committing. In processors with speculative load execution, it takes advantage of invalidation snoops, which can be used to detect conflicts by checking them against LQ and SQ entries.

III. In-Core, Low-Cost Hardware Transactional Memory

The main goal of our in-core, low-cost hardware transactional memory (LHTM) implementation is to achieve an efficient HTM that involves minimal verification effort. The base mechanisms employed by LHTM for buffering speculative state, loading cachelines and detecting conflicts are derived

from Rajwar's work [20]. We fully develop the implementation and include a simple method for committing speculative updates, in order to evaluate its performance. However, our work focuses on simplicity and thus does not detect lock acquire and releases; instead, we assume that the ISA has extensions for delimiting transactions.

To support LHTM, the ROB and SQ need to hold, as speculative state, all transactional modifications to the state of the system. Before transaction commit, cachelines accessed by loads and stores will be requested with the necessary permissions. During this process, the LQ and the SQ are searched by invalidation or downgrade snoops to detect conflicts. Once all needed cachelines are present with adequate permissions in the private cache and every instruction in the ROB is ready to commit, the first-level data cache (L1D) is locked, and speculative modifications are allowed to commit, thus guaranteeing the transaction's atomicity.

A. ISA interface

Our proposal targets the x86-64 ISA. In 2012, Intel introduced the Transactional Synchronization Extensions (TSX), and reserved several opcodes for the `XBEGIN`, `XEND` and `XABORT` instructions, among others. Despite being disabled in most recent processors [7], the opcodes are still reserved for this purpose.

To reduce the amount of changes required, LHTM reuses these opcodes, only tweaking their behavior as needed. Specifically, in our implementation both micro-ops need to behave like full memory barriers. This simplifies the design of our HTM, preventing unwanted interactions with memory operations from outside the transaction.

As with TSX, `XBEGIN` updates the `rax` register with a status code signaling that the transaction successfully started, or with an abort cause otherwise. To ensure forward progress, we follow the approach of conventional best-effort HTM: depending on the returned abort code and the number of failed attempts to execute the transaction speculatively, the abort handler may opt for executing the transaction through an alternative path with regular lock-based synchronization. This is commonly known as the fallback path.

`XEND` is used to delimit the end of transactions, and `XABORT` generates an explicit abort. The abort handler may choose not to count explicit aborts, as `XABORT` is mainly used to re-start the transaction in the software handler when a different thread takes the fallback lock.

B. Proposed design

1) Buffering speculative state: As micro-ops can only commit and retire in-order through the ROB head, LHTM simply stalls `XBEGIN` at commit. 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 can 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.

As a result, we avoid the need for dedicated hardware to create a backup of the register file. Furthermore, a checkpointing alternative would create a full backup of all registers at the start of the transaction, many of which might not be overwritten, meaning that many unnecessary copies would have been created. Checkpointing the register file is particularly inefficient if transactions are small and frequent, and it may contribute to imposing an additional fixed overhead to transaction execution. While leveraging the ROB to buffer transactional instructions minimizes hardware modifications, it has the drawback of not allowing instructions to commit until transaction commit starts, which can specially affect performance when committing large transactions.

2) Loading required cachelines: Transaction atomicity is guaranteed by locking L1D to delay all external requests from sources other than the local core. This ensures that all modifications are made globally visible at once after the transaction commits. To avoid deadlocks once the lock is activated, all cachelines accessed during the transaction must be in a state that does not require generating or waiting for additional coherence messages when completing the transaction (i.e., for a MESI protocol, cachelines only accessed by loads need to be in M, E or S state; and cachelines accessed by stores need to be in M or E state).

Loads already bring their target block into L1D when they execute. When stores execute, on the other hand, they do not need to access the block they write to. However, many store prefetching strategies have been researched and implemented for regular, non-transactional execution that preemptively ask for blocks to be loaded with write permissions [17] in order to prevent a cache miss once stores access memory. Many modern processors perform prefetch requests (i.e., read for ownership) when stores commit [21], before they reach the head of the SB. We opt for the alternative of generating a request for exclusive ownership when transactional stores execute. Unlike traditional exclusive prefetches, which are best-effort in nature, the ownership requests sent when transactional stores execute in LHTM must eventually be fulfilled by the cache, which will respond to the processor with an ACK.

As cachelines are requested during the execute stage for both loads and stores, read and write set tracking is imprecise. Requests generated from speculatively executed loads and stores (e.g., after a branch or value prediction which might be incorrect) can abort conflicting transactions.

3) Conflict detection and resolution: Conflicts can appear as the transaction loads cachelines into L1D. Specifically, the core can detect a conflict if an L1D snoop signals a change to any transactionally accessed block from its required state. This can be done by comparing snoops against all in-flight loads and stores if `XBEGIN` is at the head of the ROB. `XBEGIN` and `XEND` are full memory barriers, so all in-flight memory operations belong to the transaction.

Loads are only affected by invalidation snoops. When a completed load is found upon an external snoop, in addi-

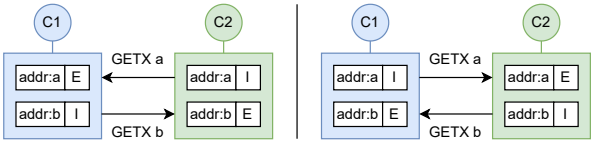

Fig. 1. Simplified livelock example between two cores (`C1` and `C2`) executing transactions that write to the same two cachelines. Only first-level caches are shown, and communication with the directory is omitted. If ownership requests are retried, it is not guaranteed that any of the cores will acquire both cachelines at the same time in order to commit atomically and achieve forward progress.

tion to retrying the load to recover the lost cacheline, any execution depending on the loaded value would need to be discarded to maintain atomicity. We follow the behavior of modern CPUs [22], and conservatively trigger a full flush of the processor pipeline. As the entire transaction is being buffered in the ROB, the flush is akin to a transaction abort in conventional HTM implementations.

Stores are affected by both invalidation and downgrade snoops. In principle, it is not necessary to squash the transaction in this case since stores remain in the SQ and have not yet modified memory, so it is only necessary to reissue ownership requests. This approach would avoid unnecessary squashes on downgrade snoops and could, in some cases, prevent aborts on invalidations if the affected location had not been previously read by the transaction. Nevertheless, such an alternative increases design complexity and introduces potential livelock scenarios, as continuously retrying ownership requests without aborting may prevent progress when multiple cores write to the same cachelines (see Fig. 1). Ensuring forward progress in these situations would require additional hardware mechanisms. Given our design goal of simplicity, we adopt a more conservative approach and choose to squash the entire transaction in these cases.

In summary, any conflicting external request will abort the receiving transaction. External requests can come from other processors that may or may not be executing a transaction, so our implementation provides strong atomicity [23].

Loaded cachelines could also be invalidated by the cache replacement policy, but it is highly unlikely given that our goal is to provide efficient HTM support for transactions composed by a relatively low number of instructions, more so considering that the replacement policy would generally prevent the eviction of recently used lines.

4) Transaction commit: The following conditions must be met before transaction commit can begin:

- All instructions between `XBEGIN` and `XEND` must have executed and be ready to commit.
- All required cachelines must be present in L1D. For cachelines accessed by loads, this is already verified when checking that all instructions are ready to commit. To account for cachelines accessed by stores, all ownership requests must have been completed.

If exceptions were disregarded, checking that all loads and ownership requests have been completed would be enough to start the transaction commit. However, to ensure that the entire transaction is committed atomically, it is necessary to check that none of its instructions are marked as faulty after executing (instructions marked with a fault will generate an exception once they try to commit at the head of the ROB).

Once transaction commit starts, the entire L1D is locked and `XBEGIN` is allowed to commit, as well as all other instructions in the transaction. Stores are naturally allowed to complete after committing. Strictly speaking, cachelines that were only read during transactions do not need to be locked through the commit stage, because all loads already executed and their loaded values are kept in the ROB. Cachelines that were not accessed during the transaction do not need to be locked either. Thus, only cachelines with pending transactional stores need to be locked to guarantee an atomic transaction commit. However, this would imply storing the write set of the transaction in the cache with additional metadata or sending messages before transaction commit to lock only the required cachelines. In addition to avoiding more complex modifications, we favor the coarse-grained locking option because the time during which the cache is locked should be small, as all the instructions have been executed and the latency for loading missing cachelines into L1 has already been paid.

When `XEND` reaches the head of the ROB, it behaves like a full memory barrier, so it can only be committed once all transactional stores have been completed and the SB is empty. Therefore, it is safe to unlock the cache after this point.

5) Aborts: In LHTM, aborts are performed by squashing from `XBEGIN`, which is at the head of the ROB. As previously mentioned, conflict aborts are generated when completed loads or ownership requests are hit by a snoop. Explicit and exception aborts need to be triggered when any instruction of the transaction is marked with a fault. Specifically, explicit aborts are generated if the fault was created by `XABORT`, and exception aborts follow any of the remaining faults.

If the processor's fetch stage is blocked due to lack of resources, a capacity abort is performed if there is not a valid `XEND` instruction (i.e., an `XEND` that is not squashed) in the ROB. Capacity aborts are also generated if all accessed cachelines cannot be loaded at the same time into L1D. If the blocking was instead activated because of a pending interrupt, the transaction needs to be aborted in order to handle it.

C. Microarchitectural changes

This section lists our proposed modifications for adding LHTM support to an OoO processor, which are also summarized in Fig. 2.

Firstly, when `XBEGIN` reaches the head of the ROB, it is stalled by preventing it from committing until the conditions for transaction commit are met. A 2-bit register can indicate the state of the transaction: `ts_none` when executing non-transactional code, `ts_exec` when `XBEGIN` reaches the head of the ROB (and while waiting for transaction commit) or `ts_commit`.

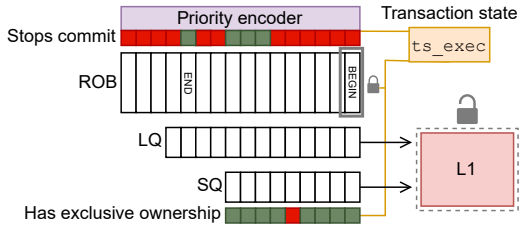

Fig. 2. Modifications needed to support LHTM in an OoO processor

When in `ts_exec` mode, the ROB needs to be monitored to check if the current transaction can be committed "in one go". For this, we include a bitmap that indicates whether the corresponding ROB entry will stop the transaction commit process. Each bit is set by `OR`ing three conditions (ignoring squashed instructions): the entry is an `XEND`, the entry is not executed, or the entry is marked with a fault. A priority encoder can then be used to get the index of the entry closest to the head of the ROB that would stop the transaction commit. If this entry is an `XEND`, LHTM can enter `ts_commit` (pending completion of ownership requests). If the entry is an instruction that is not ready or none of the bits are activated in the entire array, LHTM remains in `ts_exec`. If the entry is marked with a fault, an abort is triggered to avoid generating an exception in the transaction commit. Capacity or interrupt aborts also need to be generated if the fetch stage is blocked and the entire bitmap is unset.

Meanwhile, the number of accessed cachelines can be calculated by looking at the LQ and SQ, in order to trigger a capacity abort if at any point the L1 associativity limit is exceeded.

The remaining condition to enter `ts_commit` requires checking that all ownership requests have been completed. Ownership requests must be generated for stores that execute in `ts_exec`. A bit per SQ entry can indicate whether the ownership request corresponding to each store has been completed. By always forwarding downgrade snoops from the L1D in addition to the already-present invalidation snoops, the core can search the SQ to find stores that are hit and that completed their ownership request.

Lastly, all L1D cachelines must be locked during transaction commit. This locked state can be signaled for the entire cache with a single "locked" flag. When locked, the cache will delay all external requests. The flag would then be activated and deactivated after the core signals `ts_commit` start and end, respectively.

D. Known limitations

LHTM requires holding the transaction in the speculative state of the processor, introducing severe restrictions that we summarize now. If any of these restrictions cannot be met, an abort will squash the transaction.

- There cannot be system calls, exceptions or interrupts that prevent transaction commit.
- All instructions from `XBEGIN` to `XEND` must fit in the ROB. In this respect, misprediction squashes count towards capacity limits in processors with lazy squashing (i.e., in processors where resources for squashed instructions are not freed immediately, and instead are only released once the squashed instructions reach the head of the ROB). As a consequence, it is recommended to incorporate eager squashing or to avoid complex control flow within transactions.
- To ensure that all blocks accessed in the transaction fit into L1D, the number of accessed cachelines should not be greater than L1D associativity (e.g., a maximum of 8 different cachelines for our simulated system).

Therefore, capacity aborts in our proposal can be caused either because the transaction overflows the core's structures —ROB, SQ, LQ and register allocation table— or because accessed cachelines do not fit at the same time in L1D.

Pipeline width	5 fetch/5 decode/5 rename/10 dispatch/ 10 issue/10 commit
Physical registers	180 integer + 180 floating-point
ROB	352 µop entries, eager squash
LQ	128 entries
SQ	72 entries
RAS	64 entries
Branch predictor	LTAGE
Memory dependence	Dependence prediction with store sets
Caches	64B line size, write-back, write-allocate, strictly inclusive, LRU replacement policy
L1 instruction	32KiB, 8-way, 1-cycle latency
L1 data	48KiB, 12-way, 1-cycle access latency
L2	512KiB, 8-way, 10-cycle access latency
L3	Shared, 4MiB per core, 16-way, 45-cycle access latency
Memory	80-cycle access latency
Coherence	Three-level MESI protocol interconnected with a crossbar, directory embedded in LLC

TABLE I
CONFIGURATION PARAMETERS

IV. EXPERIMENTAL METHODOLOGY

We develop an implementation of LHTM in gem5 [13], an open-source cycle-level simulator that has become the standard for evaluating and exploring computer architecture designs. Specifically, we simulate gem5's OoO CPU model (O3CPU) in full-system mode for the x86-64 ISA.

We define a processor with 32 cores, each configured after an Intel Ice Lake core [24] with the parameters shown in Table I. The memory hierarchy is simulated with Ruby and the interconnect is modeled with GARNET [25]. The underlying OS is Ubuntu 24.04 with the Linux kernel 6.8.0.

To evaluate the advantages of LHTM, we run several data structure benchmarks [14], including arrayswap (atomically exchanging elements on two random array indexes) and operations on a binary search tree (BST), sorted-list, hashmap, deque, queue and stack. Operations for arrayswap, deque, queue and stack require a single simple transaction. The remaining data structures use MCAS-like transactions: before inserting or deleting elements, the addresses of adjacent

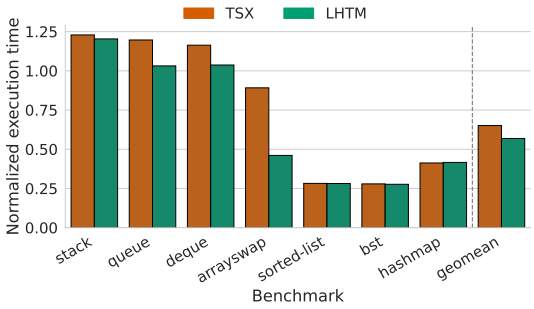

Fig. 3. Execution time normalized to locking baseline

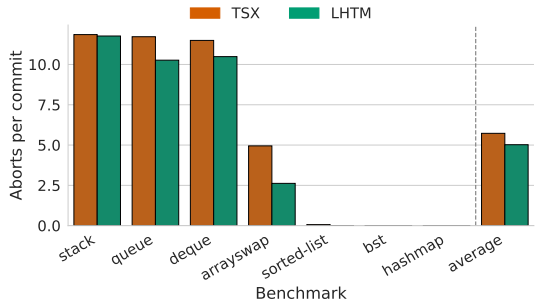

Fig. 4. Aborts per committed transaction

elements are read, and the update is performed (with a single atomic operation) only if adjacent elements were not changed by an update from another thread. If the MCAS operation fails, it needs to read adjacent elements again and be retried [14]. We avoid using STAMP [26], a widely adopted benchmark suite for HTM proposals, due to its coarse-grained transaction style, as most of its workloads (except for kmeans and SSCA2) diverge significantly from our motivating applications with small, constrained transactions.

We report the results of performing a mixed selection of operations (10% insertions, 10% deletions and 80% lookups) for the sorted list, hash map and binary search tree. For stack, queue, deque and arrayswap only updates are performed.

For each benchmark, we compare a coarse-grained locking alternative[1] and an Intel TSX-like [2] HTM against our proposal. The software wrapper for starting and finishing transactions in TSX and LHTM is identical. This wrapper includes an abort handler that introduces a random exponential backoff period between retries and resorts to a single global lock in the fallback path. Additionally, to avoid the lemming effect [27] it waits until the lock is free before retrying and does not count conflict-induced aborts that occur while the lock is held towards the retry limit. Since it is unrealistic to expect programmers to manually tune the retry policy for each application, we set the maximum number of retries to the configuration that reported the best overall performance between 1 and 10 retries (specifically, 6 retries for TSX and 5 for LHTM). Results are averaged between 10 runs of each benchmark. gem5 is a deterministic simulator, so we introduce variability between runs by adding different delays before entering the region of interest.

V. Experimental Results

This section compares the locking baseline, the TSX-like HTM and LHTM. In all figures shown, benchmarks are placed from left to right following an order of high to low contention.

A. Execution time

Fig. 3 shows the normalized execution time for all benchmarks. Deque, queue and stack provide high contention tests for the HTMs, as it is difficult to extract any speedup through parallelism. Both HTMs generate overhead when compared to the simple locking alternative in these scenarios. In arrayswap, with more room for parallel accesses, the HTMs manage to outperform fine-grained locks. In all the mentioned workloads (deque, queue, stack and arrayswap), LHTM achieves a lower execution time than TSX. The main difference between TSX and LHTM that results in this performance disparity lies in LHTM's ownership requests. LHTM's ownership requests are generated as soon as the target address of transactional stores is determined. In contrast to TSX, cachelines accessed by stores are added to the write-set before the store instruction is committed. In these benchmarks with extremely short transactions (specially arrayswap's, with just 4 `mov` instructions), we found that triggering conflict-related aborts earlier in the transaction reduced the final number of total aborts, which is reflected in execution time. The average performance of LHTM and TSX for the remaining workloads, which have less contention, is similar.

B. Aborts per commit

Fig. 4 shows the average number of aborts per commit in each workload. Given that we perform many lookup operations in the low-contention workloads and that a backoff period is included between retries, the final aborts per commit ratio in sorted-list, BST and hashmap is small (0.065, 0.008 and 0.008 respectively on average for TSX). In the high-contention workloads, in line with TSX's increased execution time, TSX performed 13.87% more aborts per commit on average.

C. Abort cause

In Fig. 5, we group aborts per abort cause and normalize each group against TSX's total aborts. The vast majority of aborts are caused by conflicting threads, as the short transactions we evaluate are unlikely to suffer interrupt, capacity or exception aborts. As the ratio of aborts per commit in the low-contention workloads is small, differences between TSX and LHTM in Fig. 5 for sorted-list, BST and hashmap do not heavily impact execution time.

D. Commit breakdown

In Fig. 6 we compare the number of transactions that committed speculatively, grouped by the number of retries before

[1]Except for arrayswap, for which we instead evaluate the simple fine-grained alternative of protecting each element with a separate lock.

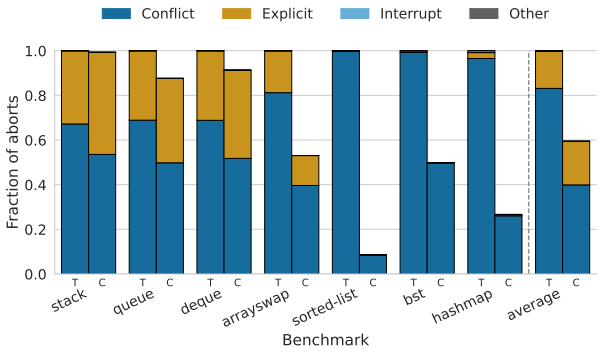

Fig. 5. Abort cause breakdown of TSX (T) and in-core LTHM (C), normalized to TSX's total aborts

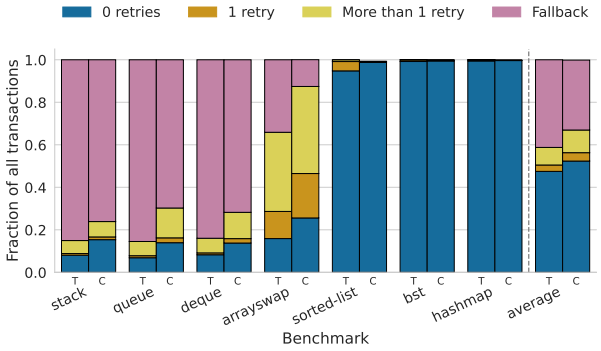

Fig. 6. Speculative commits per number of retries and fallback commits for TSX (T) and in-core LHTM (C), normalized to TSX's total commits

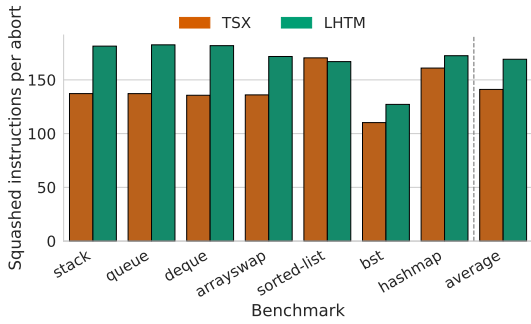

Fig. 7. Squashed instructions retired from the ROB (due to a transaction abort) per abort

committing, and the number of transactions that committed through the fallback path for each HTM. We normalize against TSX's total commits to show that, although some of our benchmarks use MCAS-like transactions, which might have to be repeated even if they commit correctly, the total number of commits is similar between HTMs.

E. Squashed instructions

Fig. 7 shows the number of instructions that were squashed and retired from the ROB for each transaction abort. Although TSX has access to a checkpoint to rollback on abort, it still has to flush the entire pipeline before doing so. As a result, it needs to squash a considerable number of instructions, similarly to our proposal. However, TSX does not buffer transactional instructions in the ROB and can start retiring them before transaction commit. This results in a smaller average number of remaining instructions in the ROB that need to be squashed when transactions are aborted in TSX.

VI. Related Work

The Rock processor's HTM design [28] took advantage of the checkpointing system that it used as an alternative to the ROB. Additionally, it committed atomic updates by acquiring locks in L2 and draining the store queue, which buffered transactional stores. It was described as an HTM implementation that introduced minimal additional changes to the existing processor design (just the ability to set locks in L2) and targeted "efficient execution of moderately sized transactions that fit within the hardware resources" [28].

Similarly to Rock, checkpointed multiprocessors [29] incorporated checkpointing into the base speculation support of the processor, although usually in conjunction with a ROB. Thus, implementations such as Bulk [5] can naturally introduce HTM support in-core without modifications to cache coherence protocol or first-level caches.

An implementation based on taking advantage of already present speculation mechanisms was considered for AMD's Advanced Synchronization Facility (ASF) proposal, but was ultimately discarded [30]. ASF targeted guaranteeing forward progress in the absence of contention and exceptions for transactions that access few cachelines, and therefore could not allow the spurious aborts caused by the limitations of the speculation hardware (e.g., store queue overflows) [30].

Nagabhiru and Byrd [31] advocate for constrained transactions, for the main use-case of lock-free programming with MCAS implemented through HTM. As with constrained transactions in z/Architecture's HTM [32], they guarantee forward progress without the need of a fallback path, but introduce important changes to the cache-coherence protocol.

Lastly, Kafousis [33] also makes the case for simplified HTM as a universal multi-word atomic primitive. Like ours, this implementation generates ownership requests in preparation for committing modifications atomically in L1, but defers them until the end of the transaction to reduce the number of aborts. Contrary to our proposal, the focus is on modifications to the first-level cache.

VII. Conclusion

When early HTM proposals started to show promising benefits, several companies incorporated HTM support in their processors. Over the years, however, interest diminished and many of these implementations have been disabled [7], [34]. The unfavorable tradeoff of a complex hardware implementation that benefits a limited number of use cases likely factored heavily in this decision.

With LHTM, we show that with simple changes, many of the benefits of HTM, particularly for small transactions,

can be obtained. Furthermore, by reusing in-core support for speculation, we reduce the attack surface for potential vulnerabilities that might have taken advantage of a checkpointing mechanism.

REFERENCES

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

[2] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[3] ARM holdings plc, "Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture," 2022, Accessed: 2025-10-26. [Online]. Available: https://documentation-service.arm.com/static/62ff4dcbc3b04f2bd53e21d5

[4] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 127–136.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk disambiguation of speculative threads in multiprocessors," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 227–238.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*. Paris France: ACM, Apr. 2010, pp. 27–40.

[7] Intel, "Intel® Xeon ® E3-1200 v3 Processor Product Family 61 Specification Update August 2020," 2020, page 61. Accessed: 2025-10-20. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf

[8] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Addendum 1 to RIDL: Rogue in-flight data load." 2019, Accessed: 2025-10-26. [Online]. Available: https://mdsattacks.com/files/ridl-addendum.pdf

[9] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 380–392.

[10] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *34th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2001, pp. 294–305.

[11] C. Lam, "Intel's Netburst: Failure is a Foundation for Success," Aug. 2025, Accessed: 2025-10-26. [Online]. Available: https://chipsandcheese.com/p/intels-netburst-failure-is-a-foundation-for-success

[12] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. Edmonton AB Canada: ACM, Aug. 2014, pp. 3–14.

[13] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[14] N. Kankava, "Exploring the efficiency of multi-word compare-and-swap," Master's Thesis, Uppsala University, 2020.

[15] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on computers*, vol. 37, no. 5, pp. 562–573, 1988.

[16] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[18] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: hardware atomic operations without fences," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 14–26.

[19] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. 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*, 2024, pp. 17–30.

[20] R. Rajwar, "Speculation-based techniques for transactional lock-free execution of lock-based programs," Ph.D. dissertation, University of Wisconsin-Madison, 2002.

[21] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel64-and-ia32-architectures-optimization.html

[22] H. Ragab, E. Barberis, H. Bos, and C. 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)*, 2021, pp. 1451–1468.

[23] C. Blundell, E. C. Lewis, and M. M. Martin, "Subtleties of transactional memory atomicity semantics," *IEEE Computer Architecture Letters*, vol. 5, no. 2, pp. 17–17, 2006.

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

[25] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[26] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Int'l Symp. on Workload Characterization (IISWC)*, Sep. 2008, pp. 35–46.

[27] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum, "Applications of the adaptive transactional memory test platform," in *3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008, pp. 1–10.

[28] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, Mar. 2009.

[29] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "Cava: Using checkpoint-assisted value prediction to hide l2 misses," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 3, no. 2, pp. 182–208, 2006.

[30] S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, J.-W. Chung, and L. Yen, "Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core," 2010.

[31] M. Nagabhiru and G. T. Byrd, "Achieving forward progress guarantee in small hardware transactions," *IEEE Computer Architecture Letters*, 2024.

[32] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM system z," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 25–36.

[33] K. Kafousis, "Limited Read-Write/Set Hardware Transactional Memory without modifying the ISA or the Coherence Protocol," University of Crete, Bachelor's Thesis, Sep. 2025. [Online]. Available: http://arxiv.org/abs/2510.15888

[34] IBM Corporation, *Power ISA Version 3.1*, IBM, 2020, Accessed: 2025-10-29. [Online]. Available: https://wiki.raptorcs.com/w/images/f/f5/PowerISA_public.v3.1.pdf