2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00081

LockillerTM: Enhancing Performance Lower Bounds in Best-Effort Hardware Transactional Memory

Li Wan, Fu Chao, Qiang Li, Jun Han*
State Key Laboratory of Integrated Chips and Systems, Fudan University
Shanghai, China

***Abstract*—Concurrent access to shared data has always been a challenge for developing multi-threaded programs and a bottleneck in the performance of Chip-Multiprocessor (CMP) systems. The challenge has been exacerbated by the need to augment processor cores and network bandwidth to fulfill the low-latency demands of ever-expanding data processing. Existing commercial best-effort Hardware Transactional Memory (HTM) is a common and effective solution. However, its architectural constraints prevent transactions from surviving in exceptions, cache overflow, and coexisting with a non-speculation fallback path, leading to unstable performance and diminishing favor. In this paper, we propose three lightweight mechanisms designed to mitigate the limitations of the best-effort HTM architecture to enhance performance stability. One is the recovery mechanism that supports the dynamic revocation of toxic conflicting requests, dramatically reducing the potential of livelocks. The second is the HTMLock mechanism with hardware and software co-design, which allows transactions using HTM and locks to run concurrently except when encountering actual conflict. Lastly, the switchingMode mechanism enables a running transaction to proactively attempt to switch to HTMLock mode in the event of a non-conflict-induced abort. Gem5 infrastructure is extended to validate and evaluate our mechanisms in a 32-core tiled CMP system. Experimental studies show that LockillerTM outperforms the coarse-grained locking scheme under STAMP benchmarks except for the yada workload, irrespective of thread number and cache size. Furthermore, our approach achieves an average of 1.86x and 1.57x speedup in all benchmarks and different threads under a typical cache size and a maximum of 7.79x and 6.73x speedup in high-contention benchmarks under extreme scenarios with only 8KB L1 cache and 32 threads, compared to best-effort HTM and state-of-the-art HTM respectively.**

***Index Terms*—Concurrent control, hardware transactional memory, coherence protocol**

I. Introduction and Motivations

Transactional memory is a general and efficient concurrency synchronization architecture that works against any data structure and is lock-free [1]. It can be implemented by a software called STM. However, it will introduce relatively high performance overhead [2], or it can be implemented by hardware called HTM, which can obtain relatively high performance at the cost of increasing hardware complexity. A wide range of researchers have favored HTM for a long time due to its potential to achieve performance comparable to the lock-free scheme while providing the advantages of coarse-grained locking in terms of generality and ease of use [3].

The authors are with the State Key Laboratory of Integrated Chips and Systems, Fudan University, Shanghai 201203, China. E-mail: wanl21@m.fudan.edu.cn, {cfu19, qiangli19, junhan}@fudan.edu.cn

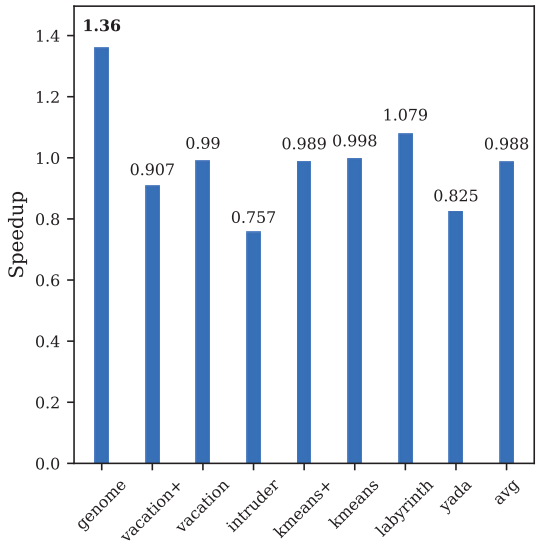

Fig. 1. The speedup of requester-win best-effort HTM with respect to coarse-grained locking scheme under STAMP benchmarks using two threads.

Despite endless academic exploration of HTM performance, hardware vendors eventually adopted the implementation of eager HTM with the most straightforward requester-win conflict management strategy to compromise on the complexity of implementation and the difficulty of validation. Moreover, hardware vendors have introduced specific architectural constraints to the HTM implementation, resulting in unconditional transaction aborts when exceptions and cache overflow occur. A non-speculative fallback path is required to ensure that the transaction proceeds as it is also known as best-effort HTM [5]–[7]. As a result, best-effort HTM often exhibits unstable performance and even performs worse than using coarse-grained locking when running transactional workloads with high contention, relatively large read/write sets, and is prone

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00081

to exceptions, as shown in Fig. 1. This results in gradually diminishing the favor for HTM.

We would like to know if we could make relatively lightweight incremental modifications that guarantee that best-effort HTM outperforms coarse-grained locking in most common scenarios. By observation, we find that the main problem where best-effort HTM performs worse than locking is friendly-fire, a known problem where a transaction is defeated by a transaction it has defeated. The two transactions do not advance over time, and as a result, they both end up starting to use locks. Instead of pessimistically predicting which requests will cause conflicts again, we propose a recovery mechanism that optimistically assumes that all requests will not cause conflicts and withdraw requests only when they cause conflicts and the originators of the request have a lower user-defined priority.

To ensure a minimum level of system performance more than relying solely on the recovery mechanism is required because, besides memory conflicts, various events like acquiring the fallback lock, exceptions, and cache overflow can also trigger transaction aborts. Such aborts may unexpectedly lead to a "black swan event," where a transaction that has been relied upon ends up aborting for unforeseen or unforeseeable reasons. Firstly, to address the problem of the fallback path unconditionally terminating all running transactions, even if there is no conflict between them, we propose the HTMLock mechanism to allow the fallback path and transactions to securely run together by modifying the HTM programming interface and enhancing cache coherence protocol and bookkeeping mechanisms. Secondly, to tackle the issue of the remaining unconditional transaction aborts caused by exceptions and cache overflow, we introduce the switchingMode mechanism built upon the HTMLock mechanism. This mechanism enables a transaction to proactively attempt a transition to HTMLock mode before triggering an abort and can avoid the rollback of transactions if successful.

To summarize, we have made the following contributions:

1) We introduce a lightweight recovery mechanism that upholds the Single Writer Multiple Readers (SWMR) property of cache coherence, empowering users to employ more effective conflict management strategies. When combined with the committed instructions-based (insts-based) policy outlined in the paper, this approach significantly reduces the occurrence of friendly fire by selectively revoking harmful conflict requests, enhancing system stability and performance in typical scenarios.
2) We propose an innovative HTMLock mechanism meticulously engineered in hardware and software layers, effectively dissolving the conservative exclusivity between fallback paths reliant on a fallback lock and HTM-based transactions. This approach facilitates concurrent execution of HTM-based and lock-based transactions unless there is a conflict, greatly enhancing system parallelism, particularly in scenarios with many threads.
3) To alleviate the architectural constraints found in best-effort HTM, which typically does not support transactions enduring exceptions and cache overflow events, we adopted a non-invasive approach named switching-Mode. This method involves a subtle enhancement to the HTMLock mechanism, allowing transactions to switch to HTMLock mode proactively when faced with such situations and avoid transaction wasted work in case of no other thread in HTMLock mode.
4) LockillerTM performs better than the coarse-grained locking scheme regardless of thread count and cache size under the STAMP benchmarks suite, except for the yada workload on a 32-core tiled CMP system with gem5. Furthermore, on average, it obtains a 1.86x and 1.57x speedup over requester-win best-effort HTM and state-of-the-art HTM under a typical cache size.

II. Related Work

It has been 30 years since Herlihy first proposed an HTM blueprint in 1993 [1]. Due to the potential significance of HTM in concurrency control, researchers have optimized HTM from various dimensions during this period.

Early HTM researchers utilized memory-resident data to save and restore transaction-related information to achieve unbounded HTM [8], [9], which significantly broadened its range of uses. Since then, more research has focused on improving the performance of HTM [4], [32], broadly categorized into two main types: reducing the occurrence of conflicts and reducing the cost of transaction rollback. Lazy HTM, such as TCC [11], can avoid RAW conflicts by delaying conflict detection until the commit phase of a transaction. However, the drawbacks of lazy HTM are also apparent, leading to more wasted transactional workload and the need for serialization of transaction commits. EazyHTM [12] and FlexTM [13] provide distributed commits by recording conflicts as they happen. DynTM [14] supports the simultaneous execution of eager and lazy transactions through an extended cache coherence protocol. ForgiveTM [15] and DeTras [16] achieve a similar effect to lazy HTM in the eager system by delaying the write request, which is not observed externally. From the point of view of weakening the semantics of transactional memory, DATM [17], SONTM [18], and Wait-n-GoTM [19] eliminate some unnecessary conflicts by logging the dependencies between transactions so that conflicts only need to be resolved if there is a cyclic dependency. Furthermore, SI-TM [20] and OverlayTM [21] eliminate read-write conflicts by providing a consistent view of transaction data through a multi-version memory system. From the perspective of conflict arbitration, PleaseTM [22] and LosaTM [23] consider the characteristics of the transaction and current execution information when encountering a conflict, improving system performance by making more reasonable arbitration decisions. Through dynamic scheduling of transactions, ATS [24], PTS [25], BFGTS [26], and SEER [27] monitor conflict events in the system, predict the confidence level of conflict, and then prevent recurrence of conflict. Furthermore, some works aim to reduce the cost of transaction rollback by adding checkpoints before potentially conflicting requests [28], repairing conflict-

ing data by RETCON [29], and prefetching data invalidated by a transaction after it aborts [30].

Although academics have proposed a variety of schemes to improve HTM performance as described above, hardware vendors, for the sake of strong transactional semantics and minimal changes to the architecture at the expense of some performance, have unanimously adopted eager HTM, known as best-effort HTM where there is no guarantee that a transaction will move forward. There needs to be a non-speculative fallback mechanism [5]–[7].

The most relevant works for our recovery mechanism are LogTM [10] and LosaTM [23]. LogTM uses NACK solely to indicate to the requester that a conflict has occurred and adopts the conflict resolution policy of retrying after a period. Although unnecessary rollbacks can be reduced, retry timing is difficult to determine and introduces the possibility of deadlocks. LosaTM solves the problem of difficulty in determining the retry time through the wake-up mechanism. However, its arbitration logic is so complex that the cache controller needs an extra cycle of delay in exceptional cases, and introducing a retry arbitration option will also introduce deadlock risk. Unlike the above two works, this paper only draws on the NACK method to help establish a global transaction priority from the perspective of stable performance. It only requires guarantees that at least one core of the system can survive so that the design can be relatively simple. Moreover, the papers need to discuss the details of the implementation of NACK and its impact on legacy systems, which will affect whether vendors consider supporting them in the future.

Concurrent Irrevocable Transaction (CIT) [31] is the closest work to our HTMLock mechanism. It implements the simultaneous execution of transactions with HTM and irrevocable transactions with lock by not acquiring the fallback lock when it enters a non-speculation fallback path. The translated lock address must, however, be stored in the MMU. Cache controllers block conflict requests, write to the translated lock address, and abort ongoing transactions using HTM when they encounter conflict requests during CIT mode. The most significant difference between our work and CIT is that, to the best of our knowledge, we implement the first complete parallel execution of transactions using HTM and transactions based on a fallback lock because transactions using HTM do not need to subscribe to the lock address before proceeding. It is important to note that the execution of a transaction with a lock will only abort the transactions using HTM that conflict with it and will not abort all transactions using HTM, thus increasing the parallelism of the system. Additionally, to stabilize the performance of best-effort HTM, we propose a switchingMode mechanism. Since there is no need to record the lock's address in the MMU, a transaction attempts to proactively switch to HTMLock mode when encountering abort events due to system limitations; thereby, there is a certain probability of avoiding the currently wasted work, which the CIT does not support.

III. LockillerTM

In this section, we discuss the three proposed mechanisms to enhance its stability by easing architectural constraints step by step.

A. Recovery Mechanism

The recovery mechanism proposed in this paper is an incremental modification based on best-effort HTM to achieve more stable performance with as few changes as possible to the original cache coherence protocol. The high-level idea of the recovery mechanism is that, for requests issued by a user-defined low-priority transaction conflicting with a user-defined high-priority transaction, the recovery mechanism is responsible for withdrawing them. Theoretically, at least one of the highest-priority transactions can be successfully committed to achieve performance at least as good as coarse-grained locking.

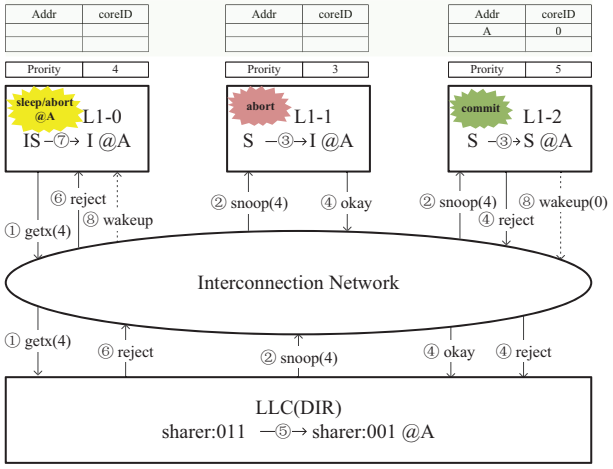

Fig. 2. The overall architecture of the recovery mechanism.

The architecture of the recovery mechanism, shown in Fig. 2, provides a framework for undoing messages sent from the cache to the interconnect, with no stipulation on the priority between transactions, nor on the action after a request has been withdrawn, which means that it can be implemented relatively easily. It is worth noting that the recovery mechanism will still function even if it chooses to abort the transaction that triggers a conflict request.

Consider a scenario where a transaction initiates a read request to the interconnect, and the interconnect forwards the request to the cache controller that owns the block. The cache controller is in the HTM transaction state, and the metadata indicates that the current transaction has written the block. Following the requester-win conflict resolution policy, the transaction should be aborted, but the transaction's written data cannot be accessible outside the transaction. Therefore, a mechanism must be established to notify the next level cache of the responsibility of bringing the original data to the requester before the transaction. A direct way to implement it

in best-effort HTM is to make the bus support a response type like NACK. As shown in Fig. 3, we take the MESI protocol as an example for illustration. According to the traditional protocol, the owner provides dirty data to the requester while writing it back to the lower-level cache. The requester jumps to the S state and sends a response of unblock to the lower-level cache after receiving the data. The directory transitions from the transient state to the stable state SS only after receiving the data written back and the unblock message, as shown in the black arrow in Fig. 3. In order to support HTM, a response message type of NACK in Fig. 3 is used to tell the directory that the owner from the upper-level cache has invalidated itself and that the directory needs to change the owner of the current cache block to the requester, and at the same time, send exclusive data to the requester. The requester is entirely unknown to the whole process and only needs to turn into a stable state E and send the unblock message to the directory when the data is received, as shown in the red arrow in Fig. 3.

Our recovery mechanism is divided into three parts: 1) support for carrying user-defined data used to resolve conflicts, 2) a mechanism to reject toxic requests selectively and recover the state, 3) wake up the requests that are rejected ever (it is not necessary, as we will see later, but we keep the option for illustration)

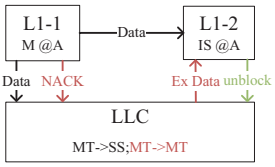

Fig. 3. NACK-like message in the best-effort HTM.

User-defined priority of transactions: As mentioned earlier, the requester-win policy is prone to livelocks due to priority inversion. Suppose there is a global consistent priority between transactions that ensures that the same transaction always wins when two transactions request each other's data. In that case, livelocks can be avoided, and at least one thread is guaranteed to move forward even in highly contention scenarios. The priority can be determined before the transaction and remain unchanged during execution or change dynamically. If the priority is determined before execution, there is no problem with priority inversion, but selecting a reasonable priority is difficult. In this paper, we adopt a dynamic priority policy based on the number of instructions committed during a transaction, which broadly weakens the effect of livelocks caused by friendly-fire because the defeated core re-executes with the lowest priority and can also help quite a bit to avoid the unfair situation. In the ACE bus, priority information can conveniently be encoded in the ARUSER field of the AR channel [33].

Reject toxic requests and recover the state: The cache controller can identify toxic requests more quickly and efficiently with customized priorities. Fig. 4 illustrates the process flow when the cache controller receives a request, with the green part representing the revised additional logic. Transactions receiving external requests initiate parallel tag match checks and overall priority comparisons (when carrying the same priority, the processor ID is compared, with smaller IDs having greater priority). In the absence of conflict, it adheres to the original process; if a conflict arises and its priority is higher, the request is denied, and its status remains unchanged; if its priority is lower, it follows the original process of aborting the transaction. As with a NACK message, the reject message is sent as a data-less message that can easily be encoded on the CRRESP signal of the CR channel of the ACE bus [33]. Note that the topology of the interconnection network does not limit this framework, provided the topology ensures that any two nodes are reachable. Assuming L1 nodes can communicate directly, the response containing reject information can be sent directly to the requester. Upon receiving the response, the requester must record the current state of the cache block of the responder based on whether it is okay or rejected. In addition, as shown in Fig. 3, the state information about the cache block in each L1 cache is piggybacked onto the unblock message to inform the directory how to update the state of the cache block and jump to the corresponding stable state accordingly. In the topology where L1 caches can only communicate through subordinates, as shown in Fig. 2, the directory keeps track of the state of the cache block of the above caches according to the response received in ④; when the last response is received, it updates the sharer list to L1Cache-2 only, as in ⑤, and ⑥ sends a response with reject information back to the original requester.

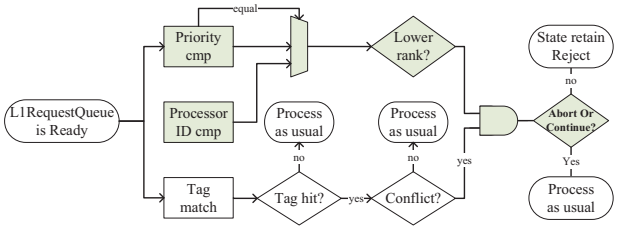

Fig. 4. The enhanced flow of handling external requests in the L1 cache controller.

Wake up rejected requests: When the requester receives the response with reject information, the response cannot be processed according to the original logic. It is necessary to hold the request in the MSHR, mark it as incomplete, and restore it to the state before sending the request, as shown ⑦ in Fig. 2. The requester has three options: abort directly, pause for a fixed period before retrying, or wait for a wake-up before retrying. The recovery mechanisms do not specify these options. In the case of waiting for wake-up, the cache controller must record which cores need to be woken up each time it rejects a request, as shown in the green shaded table in Fig. 2, and the table entry is checked at the time of transaction

abort or commit. If it is not empty, a wake-up message needs to be sent to the corresponding core, as shown ⑧ in Fig. 2. As with the stash transaction in ACE [33], the core retries the request after receiving the wake-up message, but it needs to extend the AWSNOOP signal to identify it.

In summary, the recovery mechanism does not compromise the SWMR properties of the cache coherency protocol nor impose significant requirements on the interconnect topology. It can be implemented by simply extending specific bus signals to include user-defined data and adding a small amount of additional parallel logic to the cache controller to handle harmful requests. When there is no wake-up support, less hardware overhead is required.

B. HTMLock Mechanism

Every time the fallback lock is acquired, it will cause other transactions in the process to abort, even if they are not in conflict with it, which leads to a large amount of wasted transaction work and, at the same time, consumes an attempt count, which negatively leads the transaction one step closer to falling back and executing with a lock. Ultimately, the system degrades to the point where all the transactions want to resort to the fallback path, which exhibits poorer performance than a lock-based scheme. To solve this problem, we propose the HTMLock mechanism, which does not require locks and transactions to be executed mutually exclusively, further improving the system's potential parallelism and performance stability. For description purposes, we will refer to transactions executed with HTM as HTM transactions and transactions executed with the fallback path as lock transactions. If not specified, transactions will be assumed to be HTM transactions by default.

While supporting simultaneous execution of HTM and lock transactions is not easy, there are two challenges to overcome: 1) lock transactions cannot be rolled back. Hence, the thread in a lock transaction must be guaranteed to see consistent data through execution. 2) The semantics of transactions stipulate that transactions can only read values before the start of other transactions or values after committing other transactions.

The typical HTM programming interface, utilizing the Intel RTM instructions, is depicted in Listing 1. Upon execution of the xbegin instruction, signaling the initiation of a transaction, the fallback lock is accessed to include its address in the transaction read set. In case any subsequent transaction acquires the fallback lock and reverts to a lock transaction, all HTM transactions will be aborted. If no thread is currently involved in a lock transaction, the current thread can proceed uninterrupted. However, if a thread is engaged in a lock transaction, the current transaction needs to be aborted, as indicated in the eighth and ninth lines.

We propose the HTMLock hardware-software co-design mechanism to address the above two challenges, shown in Fig. 5. We define an HTMLock mode to mark that the current lock transaction is running with our HTMLock mechanism, similar to the fallback path, but without aborting other HTM transactions in progress.

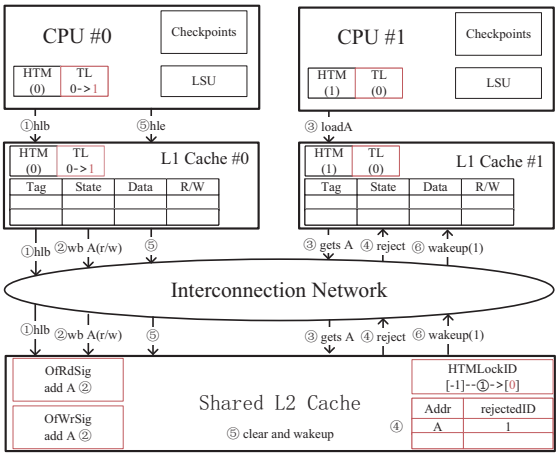

Fig. 5. The overall architecture of the HTMLock mechanism.

Listing 1 Recommended Software Implementation of Programming Interfaces on Best-Effort HTM and Modifications under HTMLock Mechanism.

```
1: void lock_acquire_elided(lock_t *lock)
2: {
3:   int num_retries = TME_MAX_RETRIES;
4:   uint64_t xstatus;
5:   do {
6:     xstatus = _xbegin();
7:     if(xstatus == SUCCESS) {
8:       if(!lock_is_free(lock)) {
9:         _xabort(TME_LOCK_IS_ACQUIRED);
10:      } else {
11:        return; // Execute using HTM
12:      }
13:    }
14:    --num_retries;
15:  } while (retry_strategy(xstatus,
       &num_retries, lock) == 1)
16:  lock_acquire(lock);
17:  _hlbegin(); // Execute using Lock
18:  return;
19: }
20:
21:
22: void lock_release_elided(lock_t* lock)
23: {
24:   if (lock_is_free(lock)) {
25:     _xend();
26:   } else{
27:     _hlend();
28:     lock_release(lock);
29:   }
30:   return;
31: }
```

Software: The modifications required to support the HTMLock mechanism are highlighted with a grey background frame, as depicted in Listing 1. Initially, we adapted the lock_acquire_elided programming interface of best-effort HTM by removing the code responsible for adding the address of the fallback lock to the read set of the transaction. This modification eliminates the undesirable limitation where a thread unconditionally terminates all HTM transactions upon acquiring a lock, even without conflicts. Simultaneously, we retain the original fallback lock, which ensures that only one thread can enter HTMLock mode, as it is not feasible for two threads to access the critical region non-speculatively simultaneously. Additionally, a line of code, hlbegin, must be incorporated into the source code at line 17 of the lock_acquire_elided programming interface to indicate to the CPU and memory subsystems that HTMLock mode has been entered. Furthermore, augmenting the lock_release_elided programming interface is essential by adding an extra line, hlend, to line 27 of the original source code to facilitate the exit from HTMLock mode by the CPU and memory subsystems.

Hardware: Two new instructions, hlbegin and hlend, have been incorporated into the ISA, occupying the same coding space as the original xbegin and xend instructions. The TL flag in Fig. 5 indicates that the CPU and memory subsystem have entered HTMLock mode upon executing hlbegin, a load instruction. In contrast to xbegin, hlbegin does not require a return value, since it is guaranteed to succeed. Additionally, hlend, another load instruction, is used to direct the CPU and memory subsystem to exit HTMLock mode and clear the read and write sets.

When the L1 cache controller receives the request from the hlbegin instruction, it needs to set TL to one to indicate that it enters HTMLock mode, as shown ① in Fig. 5. In this mode, the lock transaction also needs to record each subsequent memory access in the read/write set, the same as the transaction in the best-effort HTM, which ensures that the memory conflict between the HTM transaction and the lock transaction can be detected. However, detecting conflicts is not enough; it is also essential to resolve them correctly. The recovery mechanism mentioned above can be used to overcome challenge 1. Setting the priority of the transaction currently in HTMLock mode to the highest global priority will ensure that any other HTM transactions will not modify data read by the lock transaction. Similarly, this method can also overcome challenge 2, killing two birds with one stone.

However, transactions in HTMLock mode must survive events such as exceptions and cache overflow to serve as an irrevocable fallback path. Inspired by LogTM-SE [35], we place two signatures in LLC to record the read set and write set overflowed from the L1 cache in HTMLock mode, as illustrated in Fig. 5. In HTMLock mode, core0 replaces transaction data on address A in the L1 cache, and as shown in ②, the LLC is notified to add the address to the corresponding signature. As shown in ③, the HTM transaction in CPU1 wants to load the data on address A and then sends the request to the LLC due to cache miss. When receiving external requests, the LLC needs to check whether the address hits OfWrSig or OfRdSig while checking the Tag; if the request address hits OfWrSig, the LLC rejects the current request, and if it is contained in the OfRdSig and there is no other copy in the upper level caches the LLC must reject the request as well. The reason for this is that if there is no other copy, the requester who receives the exclusive data has the privilege to store and commit, which could result in subsequent HTMLock mode transactions loading A with inconsistent data but unable to rollback, causing the program in the risk of a crash.

C. SwitchingMode Mechanism

To cope with unstable system performance caused by exceptions and cache overflow, this paper introduces a mechanism called switchingMode, which proactively switches from speculative HTM transaction mode to non-speculative HTMLock mode. If the attempt to switch the mode fails, the transaction will be rolled back in the same manner as in the best-effort HTM; While if it succeeds, the work already performed by the transaction will not be lost. It should also be noted that even if the HTM transaction switches to HTMLock mode successfully, it will not affect other HTM transactions since we have modified the programming interface to allow both HTM transactions and lock transactions to run simultaneously unless there is an actual conflict.

As previously mentioned, the typical entry into HTMLock mode is indicated by the hlbegin instruction. In order to support both switching to HTMLock mode proactively and typical entry into HTMLock mode without adding too much hardware complexity, we stipulate that: 1) switchingMode mechanism is only considered when the HTM transaction encounters a non-conflict induced abort event such as exceptions and cache overflow. 2) Only one transaction permitted to be in HTMLock mode at any time.

Firstly, the CPU and cache controllers introduce an extra one-bit flag called Switched Transactional Lock (STL). This flag distinguishes between proactively switching to HTMLock mode and typically entering HTMLock mode, known as Transactional Lock (TL). Both STL and TL transactions are lock transactions. In systems that do not support the switching-mode mechanism, there are no STL transactions, and only TL transactions go into HTMLock mode, so only one transaction in TL mode is guaranteed by grabbing the lock indicated by line 16 in Listing 1. However, under the switchingMode mechanism, TL transactions must now acquire the LLC's authorization in addition to obtaining the lock due to contention with STL transactions. Therefore, the serialization of LLC ensures the atomic and exclusive switch of STL transactions into HTMLock mode from HTM transaction mode without obtaining the lock. Additionally, our approach of LLC's authorization can seamlessly extends to distributed LLCs by adding a lightweight centralized arbiter module. Second, the function of the original instruction ttest is extended. According to ARM TME Spec [33], the return value of the ttest instruction is the nesting depth of the transaction. In general, programs are unlikely to nest many layers of transactions inside a single

transaction. Therefore, we can agree on two relatively large numbers. If the CPU is in STL mode, the instruction return value can be set to 0x0FFFFFFF. While In TL mode, the return value can be set to 0x1FFFFFFFFF. As a result, the current transaction state can be determined through the ttest instruction. With the enhanced ttest instruction, we can rewrite the lock_release_elided programming interface, as shown in Listing 2. Regardless of whether the transaction is in STL or TL mode, hlend instructions must be executed by the end of the critical section. As the transaction is in STL mode and is switched from HTM transaction mode, there is no need to release the lock, while TL mode requires it.

Listing 2 Enhanced Software Implementation of Programming Interfaces with HTMLock and SwitchingMode Mechanism.

```
1: void lock_release_elided(lock_t* lock)
2: {
3:   uint64_t tstatus = _ttest();
4:   if (tstatus == STL) {
5:     _hlend();
6:   } else if (tstatus == TL){
7:     _hlend();
8:     lock_release(lock);
9:   } else {
10:    _xend();
11:  }
12:  return;
13: }
```

The processing flow of proactive switching to HTMLock mode is shown in Fig. 6. Here is an example scenario illustrating how a transaction's read/write set overflowing the L1 cache can trigger the switchingMode mechanism. If the cache is in HTM transaction mode and receives a CPU request that needs to replace the cache block that was accessed by the transaction, and if the transaction has not attempted to switch to HTMLock mode before. Then, the request is revoked, the current cache state is changed to applyingHLA, and all external requests are blocked. At the same time, the request for applying to enter the STL mode is initiated to the LLC. Upon receiving the response from the LLC, set the mode in L1 Cache to STL and signal the CPU to switch to STL mode if the LLC grants the request. Regardless of whether the application is approved or rejected, the applyingHLA state is pulled down, unblocking requests and, simultaneously, waking up the request that was previously withdrawn due to cache replacement to enter the pipeline again. If the application fails, the transaction will be aborted in the same manner as before. Whether it is an exception or cache overflow, STL or TL transactions will not be aborted and then rolled back, as they behave similarly to the fallback path in the best-effort HTM.

The proactive switching mode should undergo a similar operation if it is triggered by exceptions, which will not be discussed in detail here. It should be noted that most of the exceptions in the transaction are the program's problems and can be avoided mostly at the software level [34]. In addition, supporting switchingMode triggered by exceptions requires

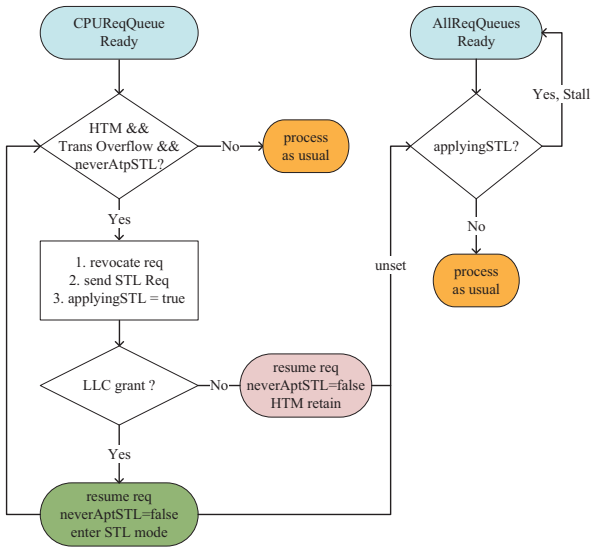

Fig. 6. The processing flow under the switchingMode mechanism.

additional processing logic for the carefully designed and validated CPU, and context switching during the transaction may introduce unknown security risks. Therefore, we choose not to support switchingMode under exceptions now but to abort the transaction like before.

IV. Experimental Results

This section presents our experimental methodology and the analysis of the experimental results.

A. evaluation methodology

We use gem5-22 as our experimental platform to validate and evaluate the performance of the proposed LockillerTM. It is equipped with best-effort HTM piggybacking on the MESI-Three-Level-HTM cache coherence protocol implemented by the ARM team [36]. However, this protocol is only a preliminary implementation, which adds a private intermediate-level cache to simplify transactional data maintenance in the L1 cache. It introduces some odd designs, such as invalidating data from the L1 cache by flushing it to the middle cache even when the other cores try to load data. Therefore, we modified the MESI-Three-Level-HTM protocol into a more streamlined, efficient, and generalized MESI-Two-Level-HTM, assuming that the next level of cache is shared, as the protocol of our baseline best-effort HTM.

The parameters of the critical modules of our modeled system are shown in Table I. We simulate a 32-core tiled CMP system where the CPU model is an in-order core that supports ARM TME extensions. The memory subsystem is configured with a two-level cache hierarchy; the L1 cache is private, and all cores share the L2 with an inclusive inclusion policy, although this is not mandatory. Tiles are interconnected using a 4x8 mesh network with an X-Y routing algorithm.

To evaluate the performance of our HTM system, we use the unmodified STAMP benchmarks and the recommended program inputs as our workloads [37]. Bayes application is excluded due to its known unpredictable behavior and highly variable execution time [38], and for both kmeans and vacation applications, both low-contention and high-contention configurations were considered.

TABLE I. System Model Parameters

Component Parameter	Value
Number of Cores	32
Frequency	2 GHz
Core Detail	In-Order, Single-issue, ARM ISA
Cache Line Size	64 bytes
L1 I&D caches	Private, 32KB, 4-way, 2-cycle hit latency
L2 cache	Shared, unified, 8MB, 16-way, 12-cycle hit latency
Memory	8GB, 100-cycle latency
Coherence protocol	MESI, directory-based
Topology and Routing	2-D mesh(4 x 8), X-Y
Flit size/message size	16 bytes / 5 flits (data), 1 flit (control)
Link latency/bandwidth	1 cycle / 1 flit per cycle

To provide a more comprehensive assessment of the stability of the HTM system, we conducted simulations with thread counts ranging from 2 to 32 on a 32-core system. Each thread is bound to a single core and does not involve OS scheduling. Table II summarizes the HTM systems we will evaluate. In order to make a fair comparison with coarse-grained locking, the same source code was used, compiled with the same compilation options, and run with the same number of threads, except for the functions of entering and exiting the critical sections, which are overloaded.

Although there are some simplifications in the currently modeled system, it is enough to demonstrate the benefits of LockillerTM using STAMP benchmarks, which are widely recognized and considered authoritative in the field. We leave a more comprehensive evaluation of LockillerTM in a more complex and closer to the actual system, with larger working sets and different transaction sizes for future work.

TABLE II. Evaluated Systems

CGL	Coarse-grained locking with the same granularity of transactions
Baseline	Best-Effort HTM with requester-win
LosaTM-SAFU	LosaTM without False Sharing and Capacity Overflow OPT
LockillerTM-RAI	Baseline + Recovery + SelfAbort + InstsBasedPriority
LockillerTM-RRI	Baseline + Recovery + SelfRetryLater + InstsBasedPriority
LockillerTM-RWI	Baseline + Recovery + WaitWakeup + InstsBasedPriority
LockillerTM-RWL	Baseline + Recovery + WaitWakeup + HTMLock
LockillerTM-RWIL	LockillerTM-RWI + HTMLock
LockillerTM	**LockillerTM-RWI + HTMLock + SwitchingMode**

B. performance evaluation

Fig. 7 illustrates the speedup of each application relative to a coarse-grained locking scheme using the same thread number on the evaluated systems. The experimental results demonstrate that LockillerTM outperforms the locking scheme across all workloads, ranging from a minimum of 2 threads to a maximum of 32 threads, except for the yada workload due to many exceptions, which the best-effort HTM and LockillerTM do not support.

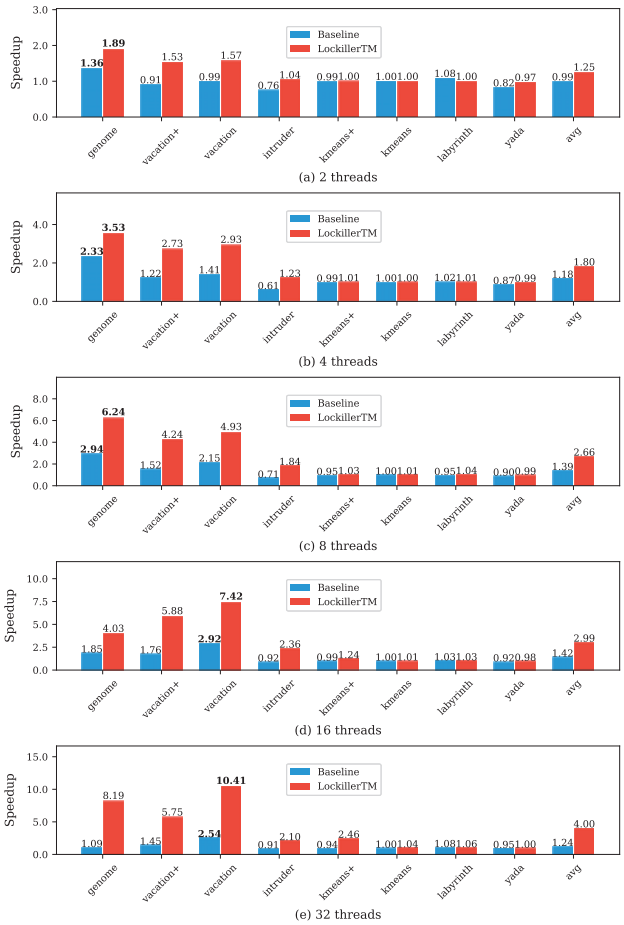

Fig. 7. The speedup of the evaluated systems compared to coarse-grained locking in the typical cache size under five different number of threads.

a) Recovery Mechanism and Insts-based Priority: As can be seen from Fig. 7, after the addition of recovery and insts-based priority optimization, the performance of the best-effort HTM has improved substantially. Even in the most straightforward LockillerTM-RAI configuration, they enable the HTM to perform better in most workloads. The priority mechanism ensures that a transaction that restarts execution after abort will have a lower priority if it conflicts with a transaction that previously defeated it. The recovery mechanism is used in conjunction with the priority mechanism to revoke requests issued by lower-priority transactions, dramatically reducing the occurrence of friendly-fire and improving the commit rate of transactions. As shown in Fig. 8, after the recovery mechanism and Inst-based priority are applied, the transaction commit rate is significantly increased by 1.4, 1.69, and 1.63 times, respectively, compared with base-effort HTM.

b) HTMLock Mechanism: We divide the execution time into the following six categories: speculative transactions (htm and aborted), lock transactions (lock), non-transactional and

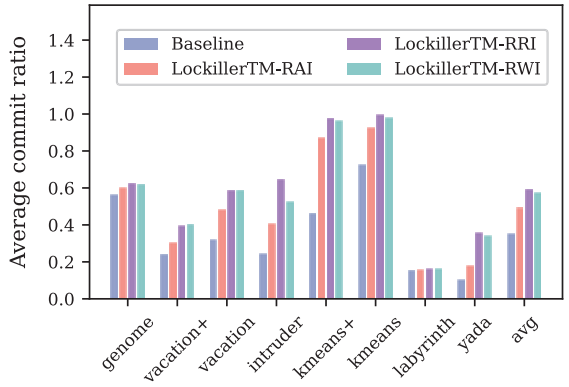

Fig. 8. The average transaction commit rate of HTM systems equipped with recovery mechanisms under five different number of threads.

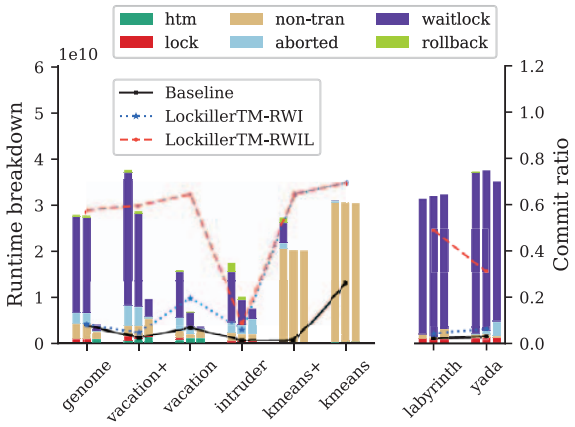

Fig. 9. The breakdown of execution time and the transaction commit rate of the evaluated systems under 32 threads.

barrier (non-tran), waiting for the lock (waitlock), and transactional rollback (rollback). In order to demonstrate the advantages of the HTMLock mechanism, we conducted experiments under 32 threads. The execution time breakdown and transaction commit rate of three HTM systems under each workload are shown in Fig. 9. We can see that in the LockillerTM-RWIL system with the HTMLock mechanism, the execution time under the four workloads of genome, vacation+, vacation, and intruder is significantly reduced. This is because the HTMLock mechanism enables lock transactions and HTM transactions to be executed entirely in parallel, significantly reducing the time of waiting for locks, as shown in the purple bar in Fig. 9. In addition, the HTMLock mechanism significantly improves the commit rate of transactions, as shown in the curve part of Fig. 9, because some transactions do not conflict with transactions executed with locks. As shown in Fig. 9, most transactions in the two applications of labyrinth and yada rely on a fallback path to execute because transactions often abort due to cache overflow or exceptions. Despite the significant improvement in transaction commit rate under labyrinth by using the HTMLock mechanism, the execution time becomes longer since labyrinth is a dynamic routing paths workload, and LockillerTM-RWIL adds more paths, resulting in shorter average effective calculation times per path.

c) switchingMode Mechanism: In order to highlight the usefulness of the switchingMode mechanism, we chose to experiment with two threads since proactive switching mode is more likely to be successful when the number of threads is low. We attribute the causes of a transaction abort to the following six categories, respectively: conflicts with HTM transactions (mc); conflicts with lock transactions (lock); conflicts with the fallback path (mutex); NonTransactional conflicts excludes the lock and mutex (non_tran); cache overflow (of); exception (fault); Two columns are missing in Fig. 10 for kmeans+ due to the fact that kmeans+ has a 100% commit rate under the HTMLock mechanism. Additionally, From Fig. 10, it can be seen that the HTMLock mechanism eliminates transaction aborts due to mutex, while the switchingMode mechanism significantly reduces aborts due to cache overflow because the aborts due to cache overflow can be avoided if the switch to HTMLock Mode is successful.

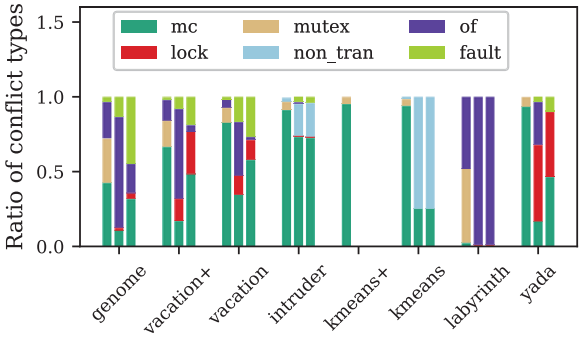

Fig. 10. Percentage of different reasons for the abort of transactions on three evaluated systems under 2 threads. From left to right: Baseline, lockillerTM-RWIL, LockillerTM

Fig. 11 shows the execution time breakdown, with a new category, switchLock, added to Fig. 9, indicating the execution time of the entire transaction under a successful switch to HTMLock mode. As we can see from the graph, especially for yada, the switchingMode mechanism increases the commit rate of transactions, greatly reducing the time wasted on transactions and thus reducing the overall execution time.

d) Performance Comparison with related HTMs: LosaTM [23] is an excellent recent work on performance optimization on best-effort HTM. It generalizes and identifies four specific conflict scenarios, tailors appropriate conflict solutions to each scenario by recording specific messages, and incorporates a prioritized conflict solution to cover the remaining conflict scenarios. In addition, it resolves most false conflicts through fine-grained cache management. Since LasaTM 's solution for false sharing is orthogonal to our

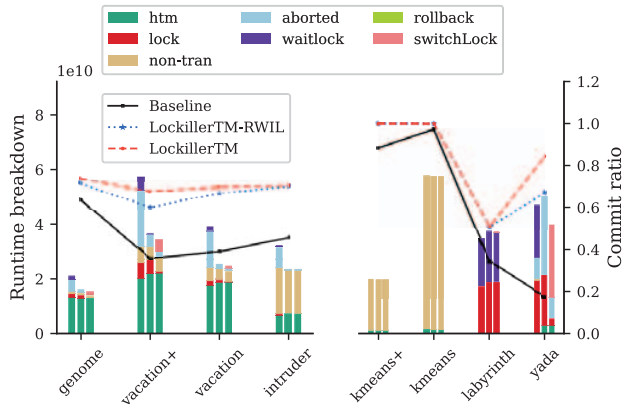

Fig. 11. The breakdown of execution time and the transaction commit rate of the evaluated systems under 2 threads.

scheme, and its optimization scheme for capacity overflow has a little effect due to the limited usage scenarios described in its paper, we choose LosaTM-SAFU as a target for comparison. From Fig. 12, it can be seen that our scheme outperforms LosaTM-SAFU on average under almost any workload except labyrinth because the recovery mechanism and insts-based priority proposed in the paper can cover the friendly-fire scenario and the insts-based priority is more representative than the progression-based priority used by losaTM. Furthermore, the HTMLock mechanism is a fully enhanced solution to the unfair competition scenario mentioned in LosaTM.

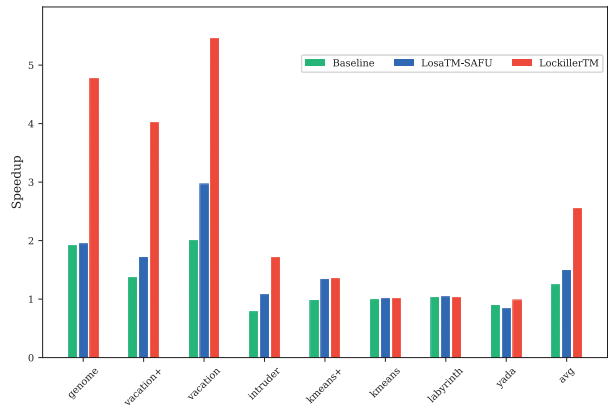

Fig. 12. The average speedup of evaluated systems under five different threads.

e) Sensitivity Analysis for Cache Size: In order to better characterize that our scheme helps improve the performance lower bounds of best-effort HTM, we conducted experiments with a small cache configuration of 8kB L1 cache and 1MB LLC and a large cache configuration of 128kB L1 cache and 32MB LLC, respectively. The results of the experiments are shown in Fig. 13. It can be seen that the average speedup of LockillerTM in both large and small cache configurations is better than that of coarse-grained locking schemes, as well as that of requester-win best-effort HTM.

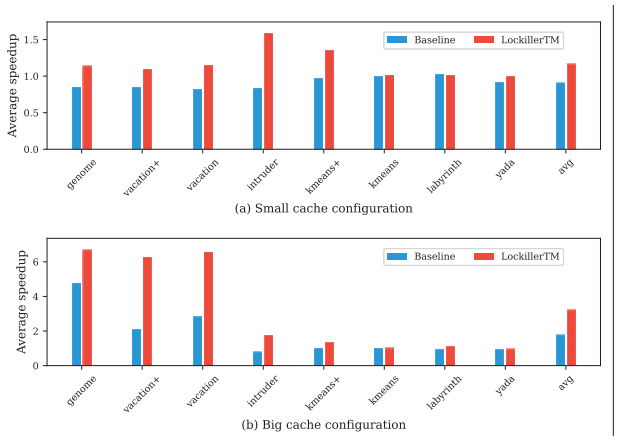

Fig. 13. The average speedup in the large cache configuration and the small cache configuration respectively under five different number of threads.

V. Conclusions

This paper provides a brief overview of previous optimization methods for HTM and analyzes the possible factors that have led vendors to opt for best-effort HTM solutions. In addition, to improve system performance lower bounds, we propose three lightweight mechanisms, which are carefully designed and do not require invasive modifications to the system. Firstly, the recovery mechanism and instruction-based priority contribute to stabilizing system performance in most scenarios by mitigating the occurrence of livelocks. For scenarios where transactions in the system take a fallback path, the HTMLock mechanism further stabilizes system performance by allowing lock transactions and HTM transactions to execute simultaneously. Lastly, serving as a complement to the previous two mechanisms in extreme cases, the switchingMode mechanism saves some transactions from unconditional termination due to cache overflow. Our experiments demonstrate that LockillerTM outperforms traditional coarse-grained locking schemes across almost all scenarios. On average, we achieve speedups of 1.86x and 1.57x compared to best-effort HTM and state-of-the-art HTM, respectively. In extreme cases, these speedups reach 7.79x and 6.73x, respectively. In conclusion, LockillerTM presents a promising solution for overcoming best-effort HTM limitations, offering improved performance and adaptability to diverse workloads.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant 61934002.

References

[1] M. Herlihy, J. Eliot, and B. Moss, “Transactional Memory: Architectural Support For Lock-free Data Structures,” in Proceedings of the 20th Annual International Symposium on Computer Architecture, May 1993, pp. 289–300.

[2] C. Cascaval et al., "Software Transactional Memory: Why Is It Only a Research Toy? The promise of STM may likely be undermined by its overheads and workload applicabilities.," Queue, vol. 6, no. 5, pp. 46–58, Sep. 2008.

[3] D. Makreshanski, J. Levandoski, and R. Stutsman, "To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing," Proc. VLDB Endow., vol. 8, no. 11, pp. 1298–1309, Jul. 2015.

[4] J. Bobba et al., "Performance Pathologies in Hardware Transactional Memory," IEEE Micro, vol. 28, no. 1, pp. 32–41, Jan. 2008.

[5] Rajwar, Ravi, and Martin Dixon. "Intel transactional synchronization extensions." In Intel Developer Forum San Francisco, vol. 2012.

[6] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in Proceedings of the 40th Annual International Symposium on Computer Architecture, in ISCA '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 225–236.

[7] Overview of Arm Transactional Memory Extension. (2022, May 17). https://developer.arm.com/documentation/102873/0100/Overview

[8] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in 11th International Symposium on High-Performance Computer Architecture, Feb. 2005, pp. 316–327.

[9] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in 32nd International Symposium on Computer Architecture (ISCA'05), Jun. 2005, pp. 494–505.

[10] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory," in The Twelfth International Symposium on High-Performance Computer Architecture, 2006., Feb. 2006, pp. 254–265.

[11] L. Hammond et al., "Transactional Memory Coherence and Consistency," SIGARCH Comput. Archit. News, vol. 32, no. 2, p. 102, 2004,

[12] S. Tomić et al., "EazyHTM: eager-lazy hardware transactional memory," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York New York: ACM, Dec. 2009, pp. 145–155.

[13] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in 2008 International Symposium on Computer Architecture, Jun. 2008, pp. 139–150.

[14] M. Lupon, G. Magklis, and A. González, "A Dynamically Adaptable Hardware Transactional Memory," in 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 2010, pp. 27–38.

[15] S. Park, C. J. Hughes, and M. Prvulovic, "Forgive-TM: Supporting Lazy Conflict Detection In Eager Hardware Transactional Memory," in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep. 2019, pp. 192–204.

[16] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "DeTraS: Delaying Stores for Friendly-Fire Mitigation in Hardware Transactional Memory," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 1, pp. 1–13

[17] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," 2008 41st IEEE/ACM International Symposium on Microarchitecture, pp. 246–257, Nov. 2008,

[18] U. Aydonat and T. S. Abdelrahman, "Hardware Support for Relaxed Concurrency Control in Transactional Memory," Micro, pp. 15–26, Dec. 2010,

[19] [1] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar, "Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies," International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 41, no. 1, pp. 521–534, Mar. 2013,

[20] H. Litz, D. R. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "SI-TM: reducing transactional memory abort rates through snapshot isolation," vol. 42, no. 1, pp. 383–398, Feb. 2014,

[21] Ziqi Wang, Z. Wang, Ziqi Wang, M. Kozuch, T. C. Mowry, and V. Seshadri, "Multiversioned Page Overlays: Enabling Faster Serializable Hardware Transactional Memory," pp. 395–408, Sep. 2019,

[22] S. Park, M. Prvulovic, and C. J. Hughes, "PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory," in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Mar. 2016, pp. 285–296.

[23] C. Fu, L. Wan, and J. Han, "LosaTM: A Hardware Transactional Memory Integrated With a Low-Overhead Scenario-Awareness Conflict Manager," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 12, pp. 4849–4862, Dec. 2022,

[24] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, in SPAA '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 169–178.

[25] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," MICRO, 2009.

[26] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," HPCA, 2011.

[27] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic Scheduling for Hardware Transactional Memory," in Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures, Portland Oregon USA: ACM, Jun. 2015, pp. 224–233.

[28] M. M. Waliullah and P. Stenström, "Intermediate checkpointing with conflicting access prediction in transactional memory systems," 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–11, Apr. 2008,

[29] C. Blundell, A. Raghavan, and M. M. K. Martin, "RETCON: transactional repair without replay," SIGARCH Comput. Archit. News, vol. 38, no. 3, pp. 258–269, 2010,

[30] A. Negi, A. Armejach, A. Cristal, O. S. Unsal and P. Stenstrom, "Transactional prefetching: Narrowing the window of contention in Hardware Transactional Memory," 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, USA, 2012, pp. 181-190.

[31] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Concurrent Irrevocability in Best-Effort Hardware Transactional Memory," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 6, pp. 1301–1315.

[32] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," ACM Trans. Archit. Code Optim., vol. 10, no. 4, p. 42:1-42:25

[33] ARM. AMBA AXI and ACE Protocol Specification, March. 2020. version ARM IHI0022H.

[34] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom, "Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell," in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May 2014, pp. 615–624.

[35] Yen, Luke, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. "LogTM-SE: Decoupling Hardware Transactional Memory from Caches." In 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 261–72, 2007.

[36] Lowe-Power, Jason, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, et al. "The Gem5 Simulator: Version 20.0+." arXiv, September 29, 2020.

[37] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. "STAMP: Stanford Transactional Applications for Multi-Processing." In 2008 IEEE International Symposium on Workload Characterization, 35–46. Seattle, WA, USA: IEEE, 2008.

[38] Negi, Anurag, Rubén Titos-Gil, Manuel E. Acacio, José M. Garcia, and Per Stenstrom. "π-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory." In IEEE International Symposium on High-Performance Comp Architecture, 1–12, 2012.