Bounding Speculative Execution of Atomic Regions to a Single Retry

Eduardo José Gómez-Hernández
eduardojose.gomez@um.es
Computer Engineering Department
University of Murcia
Murcia, Spain

Juan M. Cebrian
jcebrian@um.es
Computer Engineering Department
University of Murcia
Murcia, Spain

Stefanos Kaxiras
stefanos.kaxiras@it.uu.se
Department of Information Technology
Uppsala University
Uppsala, Sweden

Alberto Ros
aros@ditec.um.es
Computer Engineering Department
University of Murcia
Murcia, Spain

Abstract

Mutual exclusion has long served as a fundamental construct in parallel programs. Despite a long history of optimizing the lower-level lock and unlock operations used to enforce mutual exclusion, such operations largely dictate performance in parallel programs. Speculative Lock Elision, and more generally Hardware Transactional Memory, allow executing atomic regions (ARs) concurrently and speculatively, and ensure correctness by using conflict detection. However, practical implementations of these ideas are best-effort and, in case of conflicts, the execution of ARs is retried a predetermined number of times before falling back to mutual exclusion.

This work explores the opportunities of using cacheline locking to bound the number of retries of speculative solutions. Our key insight is that ARs that access exactly the same set of addresses when re-executing can learn that set in the first execution and execute non-speculatively in the next one by performing an ordered cacheline locking. This way the speculative execution is bounded to a single retry.

We first establish the conditions for ARs to be able to re-execute under a cacheline-locked mode. Based on these conditions, we propose cleAR, cacheline-locked executed AR, a novel technique that on the first abort, forces the re-execution to use cacheline locking. The detection and conversion to cacheline-locking mode is transparent to software.

Using gem5 running data-structure benchmarks and the STAMP benchmark suite, we show that the average number of ARs that succeed on the first retry grows from 35.4% in our baseline to 64.4% with cleAR, reducing the percentage of fallback (coarse-grain mutual exclusion) execution from 37.2% to 15.4%. These improvements reduce average execution time by 35.0% over a baseline configuration and by 23.3% over more elaborated approaches like PowerTM.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0391-1/24/04...$15.00
https://doi.org/10.1145/3622781.3674176

ACM Reference Format:
Eduardo José Gómez-Hernández, Juan M. Cebrian, Stefanos Kaxiras, and Alberto Ros. 2024. Bounding Speculative Execution of Atomic Regions to a Single Retry . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3622781.3674176

1 Introduction

Shared-memory parallel thread coordination is dominated by two models: First, historically, is the mutual-exclusion (ME) model epitomized by critical sections (CS) that are protected by locks. The second is the transactional memory (TM) model epitomized by transactions (TX) that rely on conflict detection and re-tries.

There is an inherent tension in the mutual exclusion model regarding the granularity of the CS protected by locks. Using coarser locking is easier (compared to, for example, using multiple nested locks), and it means fewer locks and (expensive) lock operations overall, but perhaps with higher contention. The transactional model resolves this tension by relieving the programmer from specifying the locking granularity, instead relying on detecting conflicts in the dynamic set of addresses that are accessed in a transaction. Yet, the speculative nature of a transaction and the need to bound its re-tries, inevitably leads us back to treating it as a CS with coarse-grain locking as the fallback path.

In this work, we take a unified view of critical sections and transactions and discuss them simply as *atomic regions*:

An Atomic Region (AR) is a section of code that needs to be protected to prevent data races that produce an undesired behavior or result. For us, it makes no difference if the high-level code is critical-section-based or transaction-based. We treat both in a unified manner. The duality of critical sections executed speculatively and transactions executed as critical sections is apparent in their historical development:

Speculative lock elision (SLE) [35, 36] proposed that locks can be elided if no *conflicts* occur, that is, none of the concurrently executing ARs write data that is either read or written by another AR. Later, hardware transactional memory (HTM) [17, 19] generalized the idea of speculative execution of ARs with the minimum programmer effort. In case of conflicts, threads discard the work done, restore a checkpoint prior to the start of the AR, and restart execution. This process can repeat indefinitely since these techniques suffer from an unbound number of retries. SLE and currently available HTM implementations prevent livelocks by having a threshold for the number of retries to mitigate performance loss. When the threshold is reached, the fallback path is taken: a costly serialized mutual-exclusive execution of the conflicting ARs [6].

An alternative solution to speculative execution of atomic regions relies on knowing a priori all cachelines that will be accessed within the region [16, 33]. First, new instructions are added to the architecture in order to convey the addresses that will be accessed in the AR to the hardware. Then, the core executing the atomic region requests exclusive access for the accessed cachelines and locks them in a predetermined order using available hardware mechanisms (*cache locking* Vol 3 - Chapter 9.1.4 [21]). Once all cachelines have been locked, the execution is guaranteed to be atomic, and therefore, no rollback mechanisms nor fallback path are necessary. While this solution can significantly improve performance over SLE and TM in high-contention scenarios by avoiding retries, it comes with important disadvantages. First, it requires adding new instructions, making it incompatible with existing shared-memory binaries. Second, it can degrade performance in low-contention scenarios, since (i) execution cannot start until all cachelines have been locked in order and (ii) exclusivity is requested also for cachelines that are only read, thus causing extra invalidation events.

Still, in high-contention scenarios it is desirable to guarantee the success of an AR after a limited number of retries, such that the cost of falling back to coarse-grain locking is never paid. In this work we aim to answer the following question: Is it possible to dynamically learn (in hardware) the addresses that will be accessed in an AR and effectively use this knowledge to execute the AR with at most one retry?

Our insight is that there is a non-negligible fraction of ARs whose particular memory footprint (i) does not change (i.e., are *immutable*) on retries and (ii) is bounded to a small number of cachelines. In such cases, the hardware can learn the memory footprint when speculatively executing the AR

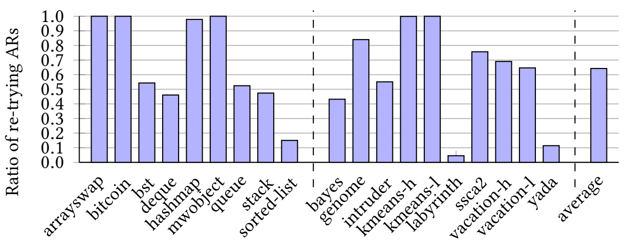

Figure 1. ARs that do not change their accessed cachelines on the first retry

for the first time. When a conflict happens, the AR can re-execute locking the learned cachelines so that completion is guaranteed with a single retry. To motivate our approach, we measure in our benchmark set the mutability properties of ARs (see section 3). Figure 1 shows a runtime view of the ARs that access a memory footprint lower than 32 cachelines and remains immutable on the first retry, representing on average the 60.2% of the ARs that abort after the first attempt. However, even with such dynamic knowledge enforcing an "at most single-retry execution" of an AR without reverting to coarse-grain locking is not trivial. This is where the contributions of this work are focused.

Contributions: We propose Cacheline-locked executed Atomic Region (cleAR), a technique that bounds the number of AR retries to one, as long as their memory footprint is kept immutable between an aborted and a re-tried execution. (Recall here that it does not matter to us if the AR is coded as a critical section or as a transaction.) cleAR performs an optimistic execution of the AR expecting no thread conflicts while gathering critical information about the memory footprint and feasibility of executing the AR non-speculatively using cacheline locking. This optimistic execution serves two purposes:

- *Like in speculative approaches*, it avoids the overhead of cacheline locking in low-contention scenarios.
- It learns the cachelines that have to be locked to execute the AR, *like in cacheline-locking approaches*, in case a retry is needed.

Of course, the immutability of the memory footprint of an AR cannot always be dynamically guaranteed. We must decide whether to (i) try the transaction's re-execution using fine-grain locking but being alert of possible alterations in the memory footprint or (ii) revert to the baseline approach (SLE or HTM).

We further discuss cleAR with only in-core speculation, i.e., within the confines of a speculative instruction window delimited by a reorder buffer (ROB), in section 4 and extend this to out-of-core speculation in subsection 4.2 if HTM facilities are available. Note that HTM facilities are not a requirement for having cleAR; but if they are available, cleAR can optimize the execution of transactions.

Lastly, we evaluate cleAR with full-system gem5 (32 cores, Intel Icelake-like), on a wide range of benchmarks (including all STAMP benchmarks [30] among others). Compared to a gem5 TSX-like implementation (Vol 1 – Chapter 16 [21]), cleAR improves performance on average by 35.0% and by 23.3% when implemented over a PowerTM baseline [9]. Performance improvement comes mainly from a reduction in the average retry count per committed AR (from 7.9 to 1.6). Indeed, cleAR increases the ARs committed after the first try by 29.1%, reducing the need to fall back to mutual exclusion by 21.7%.

2 Background

Based on conflict detection, many approaches have proposed concurrent execution of ARs. Most of them are speculative and rollback in the case of conflicts. Other more recent approaches are non-speculative and guarantee forward progress. This section covers both solutions since our proposal relies on both of them.

2.1 Speculative approaches

The Oklahoma update [39] set the basis for speculative solutions based on conflict detection. It arose as a generalization of load-linked/store-conditional instructions but for multiple memory locations. It uses a two-phase commit protocol, and in case of conflicts, retries execution.

Speculative lock elision (SLE) [35], targets traditional critical sections protected with locks. It starts speculative execution by omitting the lock acquisition and release operations. Misspeculation is also detected based on conflicts. Upon failure, the region undergoes re-execution with lock elision, up to a predefined retry limit. Exceeding this limit, the execution enters non-speculative mode with coarse-grain locking.

Hardware transactional memory (HTM) [17] aims to simplify programming by using large ARs, and offloading the responsibility of achieving concurrency to the hardware. In HTM, each access to a memory address within the transaction is recorded in either a read or write set, depending on the operation type. Upon encountering a conflict, i.e., a potential violation of atomicity, in currently available implementations the system attempts the execution of the transaction again, until a certain limit is achieved. Exceeding this limit, HTM defaults to a non-speculative "fallback" execution mode.

In both SLE and HTM the fallback path is based on acquiring a lock, either the original lock protecting the critical section (SLE) or a global lock (HTM). When the thread no longer executing speculatively starts the fallback path, exclusive permissions are acquired for the lock variable, and it is set to *busy*. When starting the speculative execution, the lock address is read and if free, the execution can start. Threads in speculative mode must include the lock address in their read set to be notified if any thread starts running non-speculatively. This mechanism is needed to prevent unprotected accesses of the fallback path from clashing with protected accesses of the speculative path.

2.2 Non-speculative approaches

Read-modify-write atomic instructions emerge as the most efficient protection mechanism for single-address atomic regions. Such instructions rely on cacheline locking, which guarantees that, between the read and the write, the cacheline is kept exclusively in the cache of the core executing the atomic instruction and no other thread can access it. Cacheline locking offers potential performance improvements over speculative methods (such as load-linked/store-conditional [22]), albeit with reduced generality.

Paving the way to non-speculative solutions, a hardware-based implementation of the Multiple Compare and Swap (MCAS) synchronization primitive was developed by Patel et al. [33]. Their implementation utilizes two distinct architectural instructions: a first one to populate a table of accessed addresses and a second one for initiating the cacheline locking mechanism and MCAS execution. While effective for data-structure benchmarks using MCAS constructs, their initial version is susceptible to deadlocks. To address this limitation, they enable a back-off protocol when directory conflicts are detected. The paper does not specify the exact back-off strategy, suggesting the probable use of a retry mechanism, especially in the absence of an alternative fallback execution method.

In the same line, MAD atomics [16] introduce a set of individual architectural instructions designed for atomically updating a small number of memory locations. The solution achieves deadlock freedom by locking the accessed cachelines in a specific order, such as lexicographical order [38]. New deadlocking scenarios are also addressed in the paper [16]. However, the number of memory addresses that can be updated is limited by the processor's architecture, since both accessed addresses and operating data must be placed in registers.

A key limitation of the described multi-address atomic constructs is that accessed addresses need to be known a priori. As a consequence, they can only be directly applied to MCAS-friendly applications, so their scope is limited. Other applications, e.g., STAMP benchmarks, cannot be ported to use such multi-address atomic constructs without completely redesigning the benchmarks and reducing the size of transactions.

3 Analyzing Atomic Regions

Our goal is to extend the scope of non-speculative approaches making them suitable to a larger number of ARs and without requiring changes in the programming model. This way, we can obtain the best of both worlds by dynamically selecting

```
register uint64_t* a = array[posa];
register uint64_t* b = array[posb];
atomic {
  uint64_t elem_a = *a;
  uint64_t elem_b = *b;
  *a = elem_b;
  *b = elem_a;
}
```

Listing 1. Inmutable AR. From arrayswap.

```
User *users;
atomic {
  users[from].bitcoins -= amount;
  users[to].bitcoins += amount;
}
```

Listing 2. Conditionally Inmutable AR with indirections. From bitcoin.

```
atomic {
  auto curr = head->next;
  while (curr != tail) {
    if (curr->data == val) n_val++;
    curr = curr->next;
  }
}
```

Listing 3. Mutable AR. From sorted-list.

between rollback-on-conflict and cacheline-locking mode: low-contended ARs can enjoy the benefits of speculative execution, while high-contended ARs can bound the number of retries. All this, transparently to software.

Performing cacheline locking in a predetermined address order requires a previous identification of all cachelines accessed within the AR. Furthermore, to extract the full potential of this approach it is important to know if the accessed footprint change between retries. This section analyses the ARs found in our wide range of evaluated benchmarks and characterizes them based on their likeliness of footprint mutations. We first offer some examples of ARs and discuss their mutability properties.

A simple example with an immutable set of addresses (extracted from arrayswap) is presented in Listing 1. It accesses two memory locations: a and b. Furthermore, the locations are directly accessed in the AR, so no other memory accesses are performed to compute the locations (i.e., no indirections). Hence, this AR is immutable, since it will always access the same cachelines when retrying execution on conflicts.

Listing 2 shows a similar code (extracted from bitcoin), but requires access to the array `users` in order to compute the target addresses *inside the same AR* (i.e., an indirection). If the indirection value can be modified by a concurrent AR,

Benchmark	# of ARs	Immutable	Likely immutable	Mutable
arrayswap	2	2	0	0
bitcoin	1	0	1	0
bst	3	0	0	3
deque	2	0	1	1
hashmap	3	0	0	3
mwobject	1	1	0	0
queue	2	0	1	1
stack	2	0	1	1
sorted-list	3	1	0	2
bayes	14	0	5	9
genome	5	0	0	5
intruder	3	0	2	1
kmeans-h	3	1	2	0
kmeans-l	3	1	2	0
labyrinth	3	0	0	3
ssca2	3	2	1	0
vacation-h	3	0	1	2
vacation-l	3	0	1	2
yada	6	1	0	5

Table 1. Characterization of ARs

the set of addresses may mutate on re-tries; otherwise, the set of addresses can be considered as immutable. We consider these cases, where the indirection values are not modified by concurrent ARs, as *likely immutable*. Note that control dependencies are treated similarly to data dependencies. That is, if a branch depends on a value loaded inside an AR, it can lead to a different set of addresses depending on the loaded value; again if that value is not modified by concurrent ARs then the address footprint is immutable on re-tries.

On the other hand, Listing 3 (extracted from sorted-list) shows a scenario where a linked list is traversed. Addresses are computed by an indirection (`curr->next`). However, the indirection values will change when the list is modified. Therefore, we consider this kind of ARs as *mutable*, i.e., the set of accessed cachelines can change across AR executions. Note that regions that modify their own indirection, are also classified as mutable.

Based on the previous examples we characterize[1] the benchmarks of our evaluation according to the aforementioned properties. Table 1 presents for each benchmark (column 1) the number of ARs that are executed at least once[2] (column 2). The table shows then the subset of these ARs that remain immutable on re-executions in column 3, those that are likely immutable in column 4, and finally, those that are mutable in column 5. As Table 1 shows, there is good potential for an approach to exploit the immutable and likely immutable ARs.

[1]These results are obtained from runs with 32 cores and medium size inputs.
[2]bayes and ssca2 have ARs that are never executed with the execution path determined by the input.

4 cleAR Overview

cleAR aims to optimize the execution of ARs by taking advantage of information gathered during the first speculative execution attempt regarding the immutability of the set of addresses accessed in an AR.

In the absence of conflicts, an AR executes unobstructed similarly to a speculative critical section (e.g., with SLE) or a transaction (e.g., with HTM). But it is in the presence of conflicts that cleAR differs from both SLE and HTM, aiming to discover an efficient way to re-execute the AR after an abort. To explain the cleAR approach we start with in-core speculation (SLE), where the speculative state is captured in (and limited by) the ROB and the store queue (SQ) and then we proceed to explain the differences with out-of-core speculation (HTM).

4.1 Discovery with in-core speculation (SLE)

Each new invocation of an AR also forms a *discovery phase*. The AR starts in speculative execution (e.g., speculatively eliding a critical section entry lock), and cleAR tracks the cachelines accessed within the AR (up to a limit) and detects indirections in the AR's executed code. Since there is no guarantee that two different invocations of the same AR will use the same set of addresses, the discovery phase runs for each of the AR invocations, unless the AR has been already marked as non-convertible (explained below). Two scenarios are possible during the discovery phase of an AR:

- No conflicts occur: the AR commits as in the baseline. This is the common case.
- A conflict occurs. This is where cleAR differs from previous work. Instead of aborting the execution, the discovery phase continues, in a *failed mode*, until either the core's speculative resources are exhausted or the end of the AR is reached, whichever comes first. The reason for this behavior is that discovery needs to see the whole execution of the AR in order to make an informed decision on how to attempt a retry. While this delays the abort of a failed AR, the benefits, as we show, outweigh the cost.

A discovery phase in failed mode does not alter the memory state (stores are kept in the SQ) and, furthermore, tries to limit damage to other ARs by flagging its accesses (coherence transactions) as coming from a failed mode, and therefore not constituting new conflicts.

Discovery makes a series of assessments that drive the decision on how the AR executes after abort. In particular, discovery hierarchically assesses the following:

1. *Does the AR fit the speculation window?* If the core's speculative resources (ROB, SQ, etc.) are exhausted before reaching the end of the AR, it is hopeless to continue discovery. Discovery ends and marks the AR as *non-convertible*. If this is already a failed AR, there is no reason to continue to its end and the AR is immediately aborted.
2. Assuming that the failed AR reaches its end, *can we simultaneously lock the cachelines accessed within the AR?* In discovery, we do not lock cachelines but given a set of accessed addresses it is straightforward to see if there are any cache or directory conflicts among them. This test tells us whether we can rely on efficient, fine-grain, cacheline locking to enforce atomicity (rather than the default coarse-grain locking).
3. *Is the set of addresses accessed by the AR immutable?* This determination is simply based on the *absence* of indirection and the *absence* of conditional branches dependent on values accessed *inside* the AR. Upon sources of non-determinism (e.g., rdtsc instruction), affected registers should also be marked as indirections, although we did not find such instructions in our workloads.

4.2 Discovery with out-of-core speculation (HTM)

With HTM capabilities, cleAR is no longer restricted by the in-core speculation window. In contrast to a failed-mode discovery supported only by in-core speculation, a failed-mode discovery with HTM capabilities can retire instructions from the ROB. Speculation extends beyond the ROB (or other in-core structures), potentially delaying the abort further.

Speculative memory accesses are tracked at a private cache level and stores are not allowed to exit the SQ and go to memory, to avoid causing conflicts with other ARs, if write permissions were requested. Thus, the SQ becomes the limiting factor for the failed-mode discovery in HTM.

4.3 cleAR with in-core speculation (SLE)

Once discovery has made its assessment, a decision can be taken on how to proceed with the re-execution of a failed AR. (A committed AR simply discards the discovery assessment without the need to make a decision.) cleAR introduces two new modes of execution, specifically designed for the re-execution of an AR:

- **NS-CL**: Non-Speculative Cacheline-Locked Execution
- **S-CL**: Speculative Cacheline-Locked Execution

Going in the reverse order of the hierarchical discovery assessment, cleAR selects an execution mode (Figure 2) as follows:

3. **NS-CL**: Execute *non-speculatively* with *cacheline locking*. Discovery determines that the AR accesses an immutable set of addresses that can all be simultaneously held locked in the cache. Given that we can lock (in a deadlock-free order) the set of known addresses in the cache, before they are used, this means that the AR can be re-executed entirely in *non-speculative* mode. This is the best-case scenario where the AR is re-executed

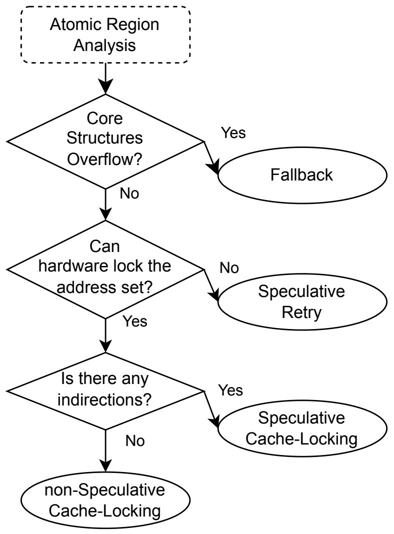
Atomic Region Analysis
Core Structures Overflow?
Yes
Fallback
No
Can hardware lock the address set?
No
Speculative Retry
Yes
Is there any indirections?
Yes
Speculative Cache-Locking
No
non-Speculative Cache-Locking

Figure 2. Decision tree and execution modes of CLEAR

with minimal effort (cacheline locking) and is guaranteed to complete non-speculatively without the need to maintain a speculative (transactional) state that can be rolled-back.

2. **S-CL**: Execute *speculatively* with *cacheline locking*. During discovery, the set of accessed cachelines *can* be locked in the cache, but the set is not guaranteed to be immutable. In this scenario, cacheline locking is attempted in hopes that it can get us through the whole AR (without the need for coarse-grain locking), but a speculative state is maintained and conflict detection is active in case i) there is a deviation from the learned set of addresses, ii) a conflict happens for a non-locked access (only in S-CL –all– mode), or iii) a request gets nacked (only in S-CL –writes– mode).
1. **Speculative Retry**: Discovery cannot even guarantee that the set of accessed cachelines can be simultaneously locked in the cache (or a S-CL execution aborted). In this case, cacheline locking is not even tried, and a speculative retry based on conflict detection is attempted, as is typical for SLE (or HTM).
0. **Fallback**: Since the speculative resources are not even enough to attempt a speculative retry (or a limited number of retries has been reached), the fallback path of the critical section lock acquisition in SLE (or coarse-grain locking in HTM) is taken.

Both NS-CL and S-CL, before proceeding to lock cachelines, ensure that no other AR is in fallback mode by acquiring a read lock on the AR's mutex lock.

4.4 CLEAR with out-of-core speculation (HTM)

For the rest of this work, we use HTM as our baseline. More specifically, we use an HTM implementation similar to Intel's TSX, that provides the necessary mechanisms for CLEAR.

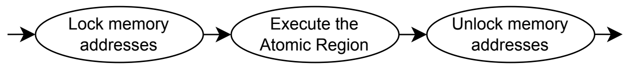

Figure 3. Execution flow of an AR running in NS-CL mode

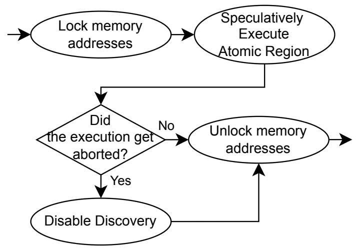

Figure 4. Execution flow of an AR running in S-CL mode

Using a transaction as our baseline AR speculative execution, the two CLEAR execution modes NS-CL and S-CL are adapted as follows.

4.4.1 NS-CL in HTM. Figure 3 shows the flow diagram for the NS-CL execution with HTM. Recall that with only in-core speculation, NS-CL is free to retire instructions from the ROB due to the non-speculative completion guarantee. With HTM the same guarantee means that there is no need for a checkpoint and no need for the conflict manager, which is deactivated. To prevent conflicts with fallback execution in other ARs, the fallback lock is read-locked before proceeding. Subsequently, the set of all discovery-learned addresses is locked into the cache (recall that discovery already determined that the cache can hold all the locked cachelines simultanously) in a deadlock-free, pre-determined lexicographical order [38], and at the same time the AR begins execution (without HTM). If the AR tries to access an address that has not been cacheline-locked, it must wait for it. At the end of the AR, the cacheline locks are released.

4.4.2 S-CL in HTM. Figure 4 shows the flow diagram for the execution of S-CL with HTM. S-CL starts in the same way as NS-CL. The fallback lock is read-locked to prevent conflicts with potential fallback execution in other ARs. Discovery-learned addresses are also locked into the cache. A question that arises regarding ARs that contain indirections is whether to lock solely the memory addresses that are written in the AR or to also lock the addresses that are only read. Locking all addresses would, on one hand, lead to successfully executing this AR, but on the other hand, this would require exclusivity for addresses shared by multiple ARs (increasing coherence traffic and the execution time of other ARs). Hence, we opt for locking both the write set and any read that suffered a conflict in a previous execution of the AR, in order to avoid suffering again this conflict.

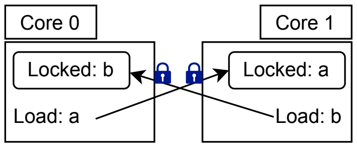

Figure 5. Deadlock scenario involving two cores

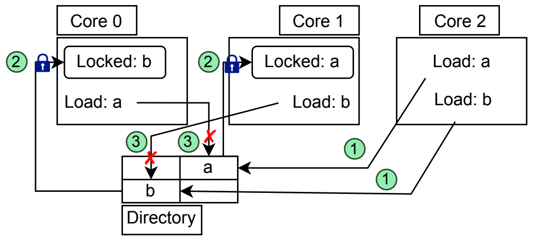

Figure 6. Deadlock scenario involving three cores

In S-CL execution mode recovery is necessary in the event of an unexpected conflict (to a non-locked cacheline). For this, S-CL relies on HTM capabilities: it creates a checkpoint and checks for conflicts. If an abort is triggered by any other reason than memory conflicts, the section is marked as non-discoverable to prevent future retries that, probably, will not succeed.

Avoiding deadlocks in S-CL. Loads that do not cacheline-lock in S-CL mode can prevent the AR from committing, and, consequently, not releasing locked addresses (a potential deadlock scenario). The deadlock scenario (illustrated in Figure 5) happens as follows: i) *Core 0* has cacheline b locked and wants to read cacheline a. ii) *Core 1* has cacheline a locked and wants to read cacheline b. This scenario that creates a cycle can happen because loads are not constrained by a predetermined deadlock-free order (only the cacheline locks are). The loads will never complete because of a remote cacheline locking. We break the deadlock cycle, by allowing the requests of the non-locking loads to be "*nack-ed*" if they reach locked cachelines (e.g., the request of `Load: a` to the cacheline `[Locked: a]`). Loads that receive a nack, abort the AR.

This problem escalates as we increase the core count (Figure 6). Imagine that a third core, *Core 2*, is trying to access cachelines a and b ①, and the request is held (these requests are not nackable because they come from a non-cacheline-locking AR). The request to a waits in *Core 1* and the request to b in *Core 0* ②. While the requests are waiting the directory is blocked (in a transient state) for both a and b. If now the reads of *Core 0* and *Core 1* start, even if nackeable, they will remain forever in the blocked directory ③. ARs in *Core 0* and *Core 1* cannot finish and a and b will never be unlocked. To solve this deadlock, requests to locked cachelines are retried instead of just delayed: that is, signaling the requester cache to send the request again, unlocking the directory entry in the process, and allowing the nack-able read requests to progress.

5 Implementation Details

In this section, we delve into the implementation details of CLEAR, depicted in Figure 7, including the structures required and their size. The changes to a baseline HTM architecture can be categorized into three different groups: i) support for detecting the immutability of the memory footprint (orange) ①, ii) support for deciding on the retry execution mode (purple) ②④, and iii) support for tracking the cacheline footprint and performing multiple cacheline locking (blue) ③.

To track immutability during the discovery phase, each entry in the register file is extended with an indirection bit ①. This bit is set when the (physical) register is the destination of a load, or of any other instruction whose source registers have the indirection bit set. Hence, the bit propagates along with register dependencies. When a memory operation or a branch retires, the indirection bit of their source registers is checked. If any bit is set to one, the AR is identified as not immutable.

The Explored Region Table (ERT) ② stores one transaction (AR) per entry. Transactions are identified by the address of their first instruction (*Program Counter field*). The remaining information stored in the ERT consists of a *Is Convertible* bit, indicating if cacheline locking can be employed on a retry and a *Is Immutable* bit, indicating that a retry can start in NS-CL mode (or in S-CL mode if the transaction is convertible but not immutable). The *SQ-Full Counter* field is a 2-bit saturating counter used to track the times a failed discovery has run out of SQ resources. The counter is increased on that event, which can appear on several retries of a transaction, and is decreased when the transaction commits. When the counter saturates, discovery is disabled for that transaction. By default, on each new AR, its entry is initialized with *Is Convertible* to one, *Is Immutable* to one, and the *SQ-Full Counter* to zero.

The cache controller is extended with an Address to Lock Table (ALT) ③. For each cacheline address learned in the discovery phase, the ALT stores the address, a Needs Locking field, indicating if the cacheline needs to be locked, and a Locked bit, indicating if it has already been locked. The ALT is used to perform the cacheline locking in lexicographical order [38], and it requires two extra fields labeled as *Hit* and *Conflict*.

To take into account the possible limited hardware resources [16], the lexicographical order is defined as the set index of the smallest shared structure, in our case the directory cache. Addresses are inserted in the ALT sorted by this lexicographical order. Multiple addresses having the same lexicographical order, which belong to the same directory

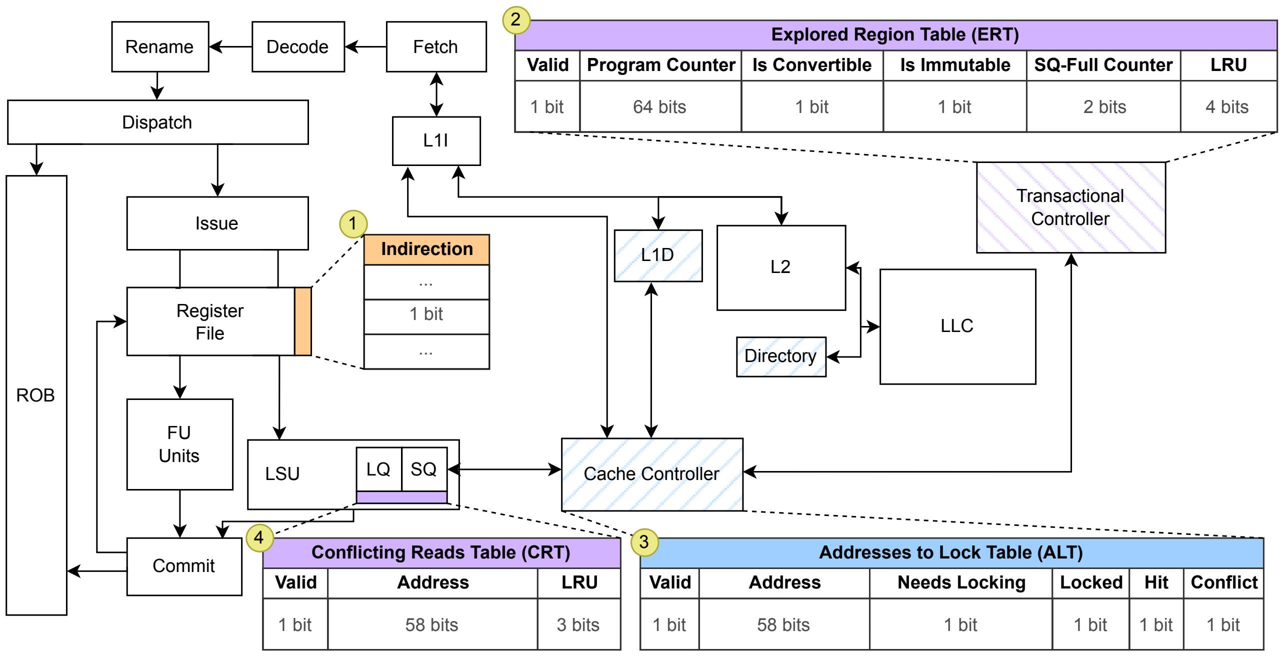
Rename
Decode
Fetch
Dispatch
L1I
ROB
Issue
Register File
Indirection
...
1 bit
...
FU Units
LSU
LQ
SQ
Commit
L1D
L2
Directory
LLC
Cache Controller
Transactional Controller
1
2
3
4
Explored Region Table (ERT)
Valid | Program Counter | Is Convertible | Is Immutable | SQ-Full Counter | LRU
1 bit | 64 bits | 1 bit | 1 bit | 2 bits | 4 bits
Conflicting Reads Table (CRT)
Valid | Address | LRU
1 bit | 58 bits | 3 bits
Addresses to Lock Table (ALT)
Valid | Address | Needs Locking | Locked | Hit | Conflict
1 bit | 58 bits | 1 bit | 1 bit | 1 bit | 1 bit

Figure 7. CLEAR architecture with extra field and tables highlighted with colors

set, create a lexicographical conflict. All the conflicting addresses for a specific lexicographical order form a group. While previous research breaks these groups into smaller independent pieces to elide the conflict [38], this is not a possibility in this case, as all addresses must be locked to correctly execute the atomic region. Note that all entries in a group are marked with the *Conflict* bit except the last one, delimiting the end of the group.

At the time of locking addresses, when an entry with the *Conflict* bit is found, the locking mechanism changes. First, every entry on the group probes the private cache. If the cacheline is present and it has exclusive permission, the *Hit* bit is enabled. If all entries enable the *Hit* bit, they are locked without requiring any communication with the rest of the cache hierarchy. In the case of missing entries, a request is sent to lock the specific set in the shared structure (in this case the directory), such that permission will be eventually acquired by all cachelines [16].

Finally, the S-CL execution mode relies on an additional structure that tracks addresses that: i) are not written by the AR during the discovery phase; and ii) in a previous execution received an invalidation that caused a conflict and abort. These addresses are stored in the Conflicting Reads Table (CRT) 4, and before starting S-CL execution mode will be set as *Needs Locking* in the ALT.

The storage overhead introduced by all these structures is as follows. The indirection bits require 22.5 bytes since we model 180 physical registers. ERT contains 16 entries and it is fully associative (146 bytes). ALT has 32 entries and it is organized as a CAM with a priority search (276 bytes). Finally, CRT has 64 entries and it is 8-way associative (544 bytes). The total storage overhead is less than 1KiB (988.5 bytes).

5.1 CLEAR implementation in action

In our implementation using HTM, when `XBegin` is executed, the ERT is searched for a matching transaction (AR). If found and the *Is Convertible* bit is not set, discovery is not initiated and the transaction follows the baseline execution. Otherwise, discovery is initiated.

If the execution reaches `XEnd` without encountering conflicts, the transaction simply commits. In case of conflicts, the transaction continues performing discovery (entering failed mode) by holding the abort signal until reaching `XEnd` or `XAbort`. In failed mode, stores do not exit the SQ to go to the cache. Loads, on the other hand, are allowed to read from cache, but in case of a miss, their coherence requests are marked as *non-aborting* to prevent damage to other transactions. All cacheline addresses accessed during discovery are inserted into the ALT. Cachelines that are written are definitely going to be locked so their *Needs Locking* bit is set. Cachelines that have not been written in discovery, will not be locked under S-CL (their *Needs Locking* bit is cleared), unless they have caused conflicts in the past (and therefore are already present in the CRT). In the latter case, their *Needs Locking* bit is also set so that they will be locked. On retirement, loads, stores, and branches update the *Is Immutable* field of the current AR in the ERT.

The failed-mode discovery phase ends when encountering `XEnd`, `XAbort`, a non-memory-conflict HTM abort, or when

Core	32-core out-of-order Icelake-like. Fetch/Decode/Rename width: 5 instructions per cycle; Issue/Commit width: 10 instructions per cycle; ROB: 352 uops; LQ: 128 entries; SQ: 72 entries; RAS: 64 entries; Branch predictor: LTAGE
L1 Cache	Instructions: 32KiB, 8-way, 1-cycle access latency; Data: 48KiB, 12-way, 1-cycle access latency.
L2 Cache	512KiB, 8-way, 10-cycle access latency.
L3 Cache	4MiB, 16-way, 45-cycle access latency.
Memory	80-cycle access latency.
Coherence	Three-level MESI protocol interconnected with a crossbar. Directory has 800% coverage.
HTM	Intel TSX-like requester wins, and Power-TM. Best of 1 to 10 retries before taking the fallback lock.

Table 2. Baseline system configuration

the SQ overflows. In the case of reaching the XEnd, an informed decision about the retry mode takes place. If the SQ overflows, the *SQ-Full Counter* is increased. In the other two cases, no further action is taken.

When running in NS-CL or S-CL modes, the fallback lock is tested. If free, execution continues by locking it as read-only. Otherwise, the read of the lock spins until the lock is free. All the addresses that require locking are sent to memory following a deadlock-free lexicographical order: Recall that for NS-CL, *all* ALT addresses are locked; for S-CL, only the ones with a set *Needs Locking* bit. While this locking is taking place, the region starts executing but requests to yet-unlocked cachelines must block. At any point during the S-CL execution mode, addresses that do not require locking and cause a conflict are added to the CRT. When reaching the XEnd, all addresses are unlocked (with a bulk operation) and the read-only lock is released, completing the AR execution.

5.2 clearAR interaction with PowerTM

clearAR can be used in systems that implement different HTM designs. A state-of-the-art design that has proven to yield good performance with minimum changes is PowerTM [9]. PowerTM increases the priority of a transaction that has already failed once, entering power mode, but only one transaction can be executed in power mode at the same time. With this priority update, cycles are broken earlier, delivering an increase in performance. However, as a side effect the number of aborts in other transactions can increase.

clearAR does not need modifications to work correctly along with PowerTM, but some enhancements should be made to further improve performance. In particular, S-CL and power transactions should not abort each other on a conflicting request but they should answer with a nack message, causing the requester to abort.

6 Methodology

Simulation environment. Our evaluation is performed using *gem5* [5] modeling an x86 full-system environment. We simulate a 32-core processor using the detailed out-of-order core model. We use Ruby to model the memory hierarchy and the coherence protocol and GARNET [1] to model the interconnection network. The simulated system runs Ubuntu 18.04 with Linux kernel 5.4.49. We model energy consumption using McPAT [26, 40] using the most advanced technology available (22nm) and the default clock gating scheme for the core.

The processor parameters, similar to an Intel Icelake processor, are shown in Table 2. Execution and issue latency is modeled as measured on real hardware by Fog [14]. We optimized the implementation by preserving the Return Address Stack (RAS) at the beginning of the transaction and restoring it when aborting. Otherwise, when a function encapsulates the *xbegin* instruction, the return executed at the end of the function will get a target misprediction. Besides the cost of a pipeline flush, the transaction may start requesting addresses, potentially aborting other threads and poisoning its own read/write sets. We faithfully model the overheads incurred by the extra resources and the tables required by clearAR.

Benchmarks. We evaluated the proposal using the following benchmarks:

- Multiple implementations of data structure benchmarks that include: arrayswap [15], binary search tree (BSTree [20, 33]), deque [7, 11, 20, 24, 25], hashmap [8, 18], queue [20, 33], sorted-list [20] and stack [20].
- Two individual applications: *Bitcoin*, which emulates operations often made in the bitcoin network [23] over a set of bitcoin wallets and *Mwobject* [12, 13], which performs 4 additions to 4 different values that fall into the same cacheline.
- The STAMP benchmark suite [30] with the recommended medium inputs.

We measure performance within the region of interest, that is, the parallel phase that starts after initialization and ends before output generation. All applications are executed on 32 threads for a total of 10 times with different seeds and the trimmed mean is used to remove 3 outliers.

Configurations tested. Our evaluation includes an HTM implementation with requester-wins, PowerTM, clearAR over requester-wins, and clearAR over PowerTM. We have performed a design space exploration for the optimal number of retries for each application. In our evaluation, we run from 1 to 10 retries for all benchmarks and select the best-performing one in each case.

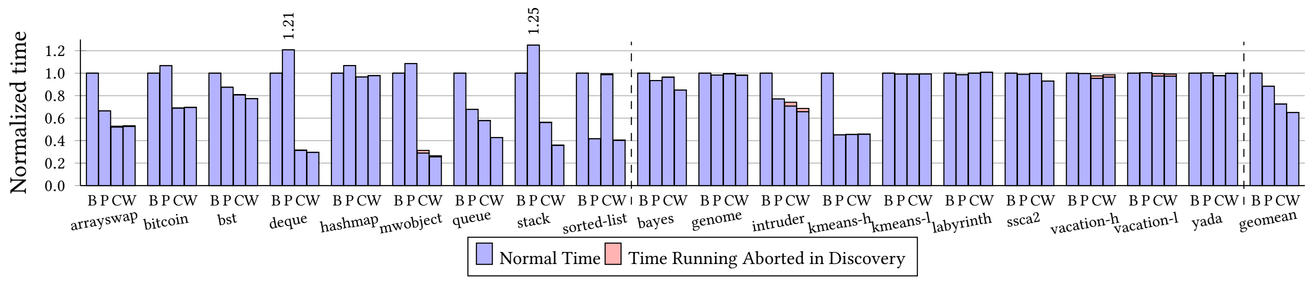

Figure 8. Normalized execution time

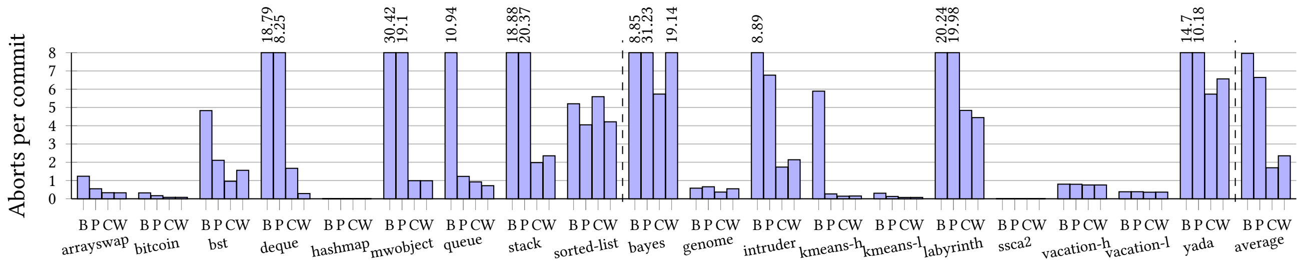

Figure 9. Aborts per Committed Transaction

7 Results

This section evaluates the performance and energy implications of clear for the simulated configurations: requester-wins (*B*), PowerTM (*P*), clear over requester-wins (*C*) and clear over PowerTM (*W*).

Execution time. Figure 8 shows the execution time normalized to requester-wins. PowerTM shows an average improvement over requester-wins of 12.7%. clear improves performance when combined with requester-wins (27.4%) and PowerTM (35.0%). From the STAMP benchmarks, *intruder* and *kmeans-h* benefit the most from clear, driven by their reduction of aborts. In most STAMP benchmarks, the size of the read and write sets is too big to allow for discovery without running out of resources. This is not the case for data-structure benchmarks, that greatly benefit from clear in terms of performance.

The overhead introduced from running discovery is usually negligible (under 1% of the execution time). The notable exception is *intruder* where this overhead peaks at 3.4% and 2.9% for requester-wins and PowerTM, respectively. The main reason is that the ARs are large, but still able to be executed in S-CL. This means that for each execution of an AR it must continue running in discovery if aborted. Following this concept, *labyrinth* and *yada* should have a similar behavior. However, most of the time, *yada* is running either in fallback mode or commits on the first try, therefore, discovery is not used or disabled quickly.

Aborts. Figure 9 shows the number of aborts per committed transaction. While PowerTM can reduce the number of aborts per committed transaction from 7.9 to 6.6, clear reduces the aborts per committed transaction when combined with requester-wins to 1.6 and 2.3 when combined with PowerTM.

In general, the reduction of aborts correlates with a reduction in the execution time, but there are exceptions. In *bayes*, even though it triples the number of aborts, there is an improvement in execution time. This is related to a significant reduction in the amount of fallback executions. On the other hand, *labyrinth* suffers from a "serialization" effect in the execution of fallback transactions due to the fallback lock cacheline being held locked by the speculative S-CL execution. This means it can save energy due to a reduction in aborted transactions, but does not improve performance.

While there should be a maximum number of retries of 10 (since we optimally select the number of retries between 1 and 10 per application), some applications reach higher values. This is because certain types of aborts do not increase the counter to take the fallback path. An example would be aborting because another thread took the fallback lock.

Energy consumption. The reduction in aborts not only translates into performance benefits but also energy reduction, as plotted in Figure 10. clear improves energy, on average, by 26.4% over requester-wins and 30.6% when combined with PowerTM, respectively. Energy reduction comes from two sources: i) clear executes faster, therefore reducing the static energy component, and ii) clear executes fewer instructions, since it aborts fewer times, reducing the dynamic energy component.

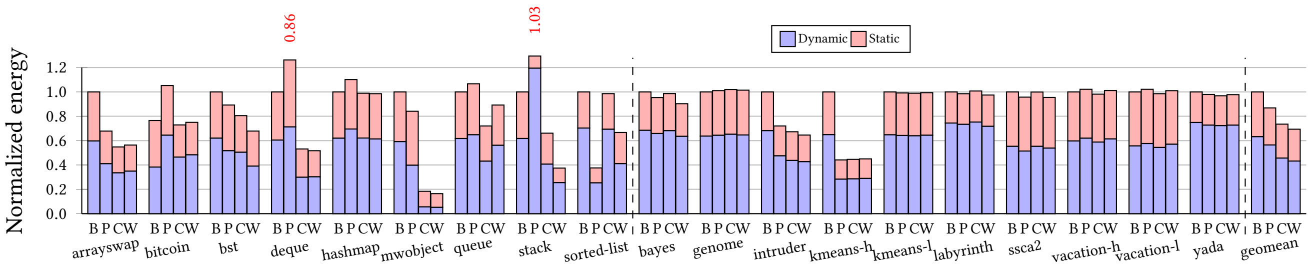

Figure 10. Normalized energy consumption

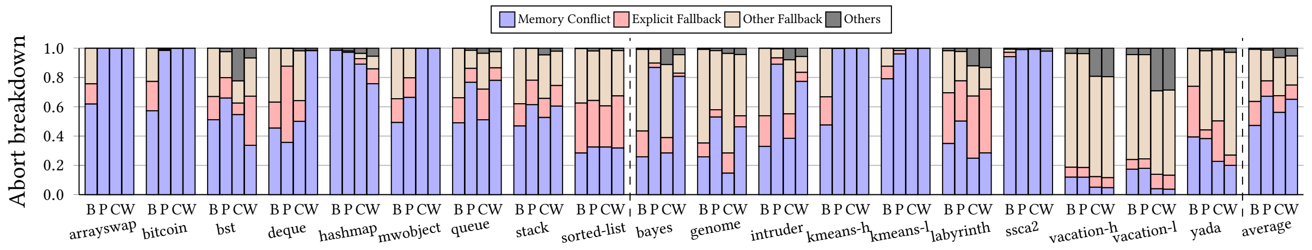

Figure 11. Abort breakdown per type

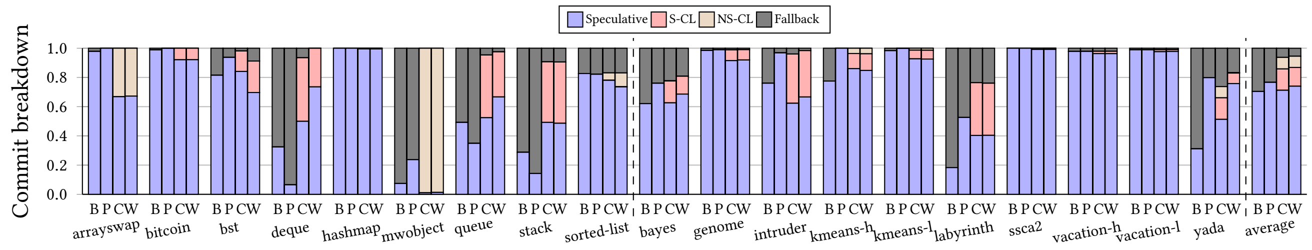

Figure 12. Commit breakdown per mode

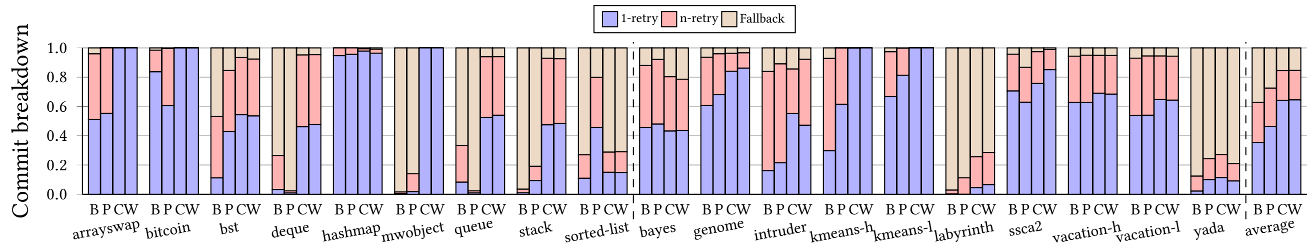

Figure 13. Commit breakdown per number of retries (excluding commits at 0 retries)

Moreover, not all aborts have the same performance and energy penalties. Figure 11 shows the types of aborts grouped into four categories, from cheap to expensive aborts in terms of performance: i) *Memory Conflict*, ii) *Explicit Fallback*: the thread attempts to start speculative AR execution but finds the fallback lock taken, iii) *Other Fallback*: a thread is executing the AR speculative and another thread takes the fallback lock), and iv) *Others*: the rest of scenarios, including instructions causing exceptions, interrupts, etc. *Explicit Fallback* and *Other Fallback* imply that the ARs are ending into fallback execution quite often. This usually translates into higher contention or bigger ARs.

NS-CL versus S-CL execution mode. Figure 12 shows a breakdown of the running mode an AR was when it committed. This includes the original speculative execution, NS-CL or S-CL, and fallback. *mwobject* is the only applications that can run mostly in NS-CL mode, while *arrayswap* runs about 33.1% and 32.7% in NS-CL mode. Some transactions may change drastically between different executions. Sometimes, they are eligible to be run in S-CL mode, but at some point their read and writes sets can become so large that they are no longer eligible. *Bst* shows this surprising behaviour, as all of its transactions were classified as mutable in Table 1, but while the data structure is small, they can still run in S-CL.

Bounding ARs to a Single Retry. Finally, the goal of cleAR is to limit the number of retries to a single one. Figure 13 shows the number of retries it took for a transaction to commit. For our baseline, on average, applications exceed the maximum number of retries 37.2% of the time, while finishing on the first retry only 35.4% of the time. PowerTM improves performance by completing 46.4% of transactions in the first retry while leaving only 27.4% in the fallback path. With cleAR running on requester-wins, a new peak of 64.2% of transactions is completed in the first retry, with only 15.5% reaching the fallback path. When combined with PowerTM we obtain similar numbers (64.4% and 15.4%). We can conclude that, regardless of the HTM implementation, cleAR provides substantial improvements in both energy and performance, while minimizing the number of aborts and maximizing the number of single-retry executions.

8 Related Work

Lock-free programming is a paradigm where atomic instructions are employed to avoid the use of software locking mechanisms [11]. Atomic instructions offer an efficient means for implementing ARs with fine-grain locking, albeit limited to a single address. Idempotent atomic regions eliminate the need for locking by encouraging threads to assist each other rather than compete for resources. As discussed by Ben-David et al. [4], this method involves executing the same Atomic Region multiple times to achieve consistent results, but at the expense of efficiency. This approach is particularly useful when a thread is interrupted by the operating system or another application. However, developing and debugging lock-free algorithms is often time-consuming and challenging, and transforming algorithms into lock-free variants may not always be feasible with current hardware. Transactional Lock Removal (TLR) [36] introduces a hardware mechanism that optimistically converts lock-based critical sections into lock-free transactions. However, this technique risks performance degradation in scenarios characterized by high resource contention, large or complex transactions, inadequate hardware support, irregular workload patterns, or significant system overhead.

Snapshot isolation [27, 28] improves TM systems reducing abort rates by providing each transaction with a consistent view of memory, allowing them to operate independently of concurrent updates and minimizing the need for rollbacks. However, it can increase the memory overhead due to the necessity of maintaining multiple data versions and complicates the system design, especially in environments with high write activity. Lu et al. [29], propose a N-Retry TM model that delays transactions instead of running them in fallback. The proposal adds a queue of tasks (similar to the OpenMP task model [3]), and after N-retries, instead of using the fallback lock, the task is added back to the task queue for a future (unbounded) retry. Diegues et. al [10] introduce a software scheduler that establishes a dynamic locking scheme to serialize transactions in a fine-grained manner. They perform a statistical analysis by sampling data, and by requesting locks, transactions are forced to be reordered in a specific order. Other proposals effectively reduce the number of aborts by reordering or speculating on data [9, 32, 34, 37]. The problem with these approaches is that the amount of retries continues to be unbound, and none of them introduce any guarantee of succeeding or limiting the number of retries.

Nord et al. [31], recently proposed a methodology to optimize the worst-case scenario in software TM for real-time systems: grouping transactions at compile time to decide which global lock to use on each specific resource group.

Asgharzadeh et al. [2] propose Free Atomics, a mechanism that removes the need for memory fences on atomic read-modify-write (RMW) instructions, thus requiring to manage several locked cachelines. cleAR also tracks multiple cachelines using a queue, in our case inspired by the "lock queue" implementation of MAD atomics [16]. Cachelines are not locked until all locks are inserted in the queue, allowing to establish a deadlock-free ordering for locking the addresses.

In summary, our approach has a unique set of features that advance beyond the current state-of-the-art techniques: i) it is agnostic to the speculation method used to handle the AR; ii) takes advantage of the cacheline-locking support of modern CPUs for atomic instructions, iii) locks the memory addresses accessed in the AR, reducing the amount of possible conflicts to zero in the first retry.

9 Conclusion

In this paper, we introduce cleAR, a hardware technique that bounds the number of speculative retries, of a significant portion of the ARs found in many applications, to one. cleAR captures the memory footprint and mutability properties of an AR on its first speculative execution. Based on this information, it makes an informed decision that leads to three different re-execution modes: 1) a new non-speculative execution mode, for small immutable ARs, using cacheline-locking for all accessed addresses that guarantees the success of the atomic execution while allowing concurrency; 2) a new speculative execution mode, for small mutable ARs, that locks the critical part of the AR memory footprint, reducing the risk of conflicts, and guaranteeing success when the footprint does not change; 3) a (baseline) speculative retry, for larger ARs; As a result, the amount of ARs that can be completed in a single retry increase by 28.8% when cleAR is on top of requester-wins and by 18.0% when is on top of PowerTM. The number of aborts per committed AR is reduced from 7.9 to 1.6. Overall, with a minimum storage impact of less than 1KiB per core, average execution time is reduced by 35.0%.

Acknowledgments

This paper has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), and Grant TED2021-130233B-C33 funded by MCIN/ AEI/10.13039/501100011033 and by the European Union Next GenerationEU/PRTR, and the Grant PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU.

References

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj 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)*, pages 33–42, April 2009.

[2] Ashkan Asgharzadeh, Juan M. Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. Free atomics: hardware atomic operations without fences. In *49th Int'l Symp. on Computer Architecture (ISCA)*, pages 14–26, 2022.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed systems*, 20(3):404–418, 2008.

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

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[6] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop. ACM*, page 54, 2014.

[7] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 21–28, New York, USA, 2005. Association for Computing Machinery.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[9] Dave Dice, Maurice Herlihy, and Alex Kogan. Improving parallelism in hardware transactional memory. *ACM Trans. Archit. Code Optim.*, 15(1), mar 2018.

[10] Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. *ACM Trans. Comput. Syst.*, 35(3), nov 2017.

[11] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 216–224, 2004.

[12] Steven Feldman, Pierre Laborde, and Damian Dechev. A practical wait-free multi-word compare-and-swap operation. 2013.

[13] Steven Feldman, Pierre Laborde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, August 2014.

[14] Agner Fog. Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns. Available at http://www.agner.org/optimize/instruction_tables.pdf (accessed on 30 Nov. 2023).

[15] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 46–61, June 2018.

[16] Eduardo José Gómez-Hernández, Juan M. Cebrian, J. Rubén Titos Gil, Stefanos Kaxiras, and Alberto Ros. Efficient, distributed, and non-speculative multi-address atomic operations. In *54th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 337–349, October 2021.

[17] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *11th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 1–13, October 2004.

[18] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 289–300, May 1993.

[20] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, USA, 2008.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-081US. September 2023.

[22] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[23] Daniel Kondor. Bitcoin network dataset. Available (archived) at: https://web.archive.org/web/20200502094144/https://senseable2015-6.mit.edu/bitcoin/ (accessed on 30 Nov. 2023).

[24] Shubham Lagwankar. A lock-free work-stealing deque. https://github.com/ssbl/concurrent-deque.

[25] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices*, 48(8):69–80, 2013.

[26] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[27] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 383–398, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Heiner Litz, Ricardo J. Dias, and David R. Cheriton. Efficient correction of anomalies in snapshot isolation transactions. *ACM Trans. Archit. Code Optim.*, 11(4), jan 2015.

[29] Kun Lu, Changhao Yan, Hai Zhou, Dian Zhou, and Xuan Zeng. A novel n-retry transactional memory model for multi-thread programming. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages

814–821. IEEE, 2017.

[30] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 35–46, September 2008.

[31] Claire Nord, Shai Caspin, Catherine E Nemitz, Howard Shrobe, Hamed Okhravi, James H Anderson, Nathan Burow, and Bryan C Ward. Tortis: Retry-free software transactional memory for real-time systems. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481. IEEE, 2021.

[32] Sunjae Park, Christopher J. Hughes, and Milos Prvulovic. Forgive-tm: Supporting lazy conflict detection in eager hardware transactional memory. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–204, 2019.

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

[34] Christian Piatka, Rico Amslinger, Florian Haas, Sebastian Weis, Sebastian Altmeyer, and Theo Ungerer. Investigating transactional memory for high performance embedded systems. In *Architecture of Computing Systems–ARCS 2020: 33rd International Conference, Aachen, Germany, May 25–28, 2020, Proceedings 33*, pages 97–108. Springer, 2020.

[35] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 294–305, December 2001.

[36] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. 37(10):5–17, oct 2002.

[37] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, 2008.

[38] Alberto Ros and Stefanos Kaxiras. Non-speculative store coalescing in total store order. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 221–234, June 2018.

[39] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(4):58–71, November 1993.

[40] Sam Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 577–589, 2015.