Efficient Hardware Primitives for Immediate Memory Reclamation in Optimistic Data Structures

Ajay Singh
University of Waterloo
ajay.singh1@uwaterloo.ca

Trevor Brown
University of Waterloo
trevor.brown@uwaterloo.ca

Michael Spear
Lehigh University
spear@lehigh.edu

***Abstract*—Safe memory reclamation (SMR) algorithms are crucial for preventing use-after-free errors in optimistic data structures. SMR algorithms typically delay reclamation for safety and reclaim objects in batches for efficiency. It is difficult to strike a balance between performance and space efficiency. Small batch sizes and frequent reclamation attempts lead to high overhead, while freeing large batches can lead to long program interruptions and high memory footprints. An ideal SMR algorithm would forgo batching, and reclaim memory immediately, without suffering high reclamation overheads.**

To this end, we propose *Conditional Access*: a set of hardware instructions that offer immediate reclamation and low overhead in optimistic data structures. *Conditional Access* harnesses cache coherence to enable threads to efficiently *detect* potential use-after-free errors without explicit shared memory communication, and without introducing additional coherence traffic.

We implement and evaluate *Conditional Access* in Graphite, a multicore simulator. Our experiments show that *Conditional Access* can rival the performance of highly optimized and carefully tuned SMR algorithms while simultaneously allowing immediate reclamation. This results in concurrent data structures with similar memory footprints to their sequential counterparts.

***Index Terms*—safe memory reclamation, concurrent data structures, cache-aware safe memory reclamation, immediate safe memory reclamation and cache-aware data structures.**

I. Introduction

Current *safe memory reclamation* (SMR) [1]–[13] algorithms used in many optimistic data structures *delay reclamation* and free nodes in batches to trade-off space in favor of high performance and safety. When the batches are too small, the data structure's *throughput* suffers due to more overhead from frequent reclamation. On the other hand, when the batches are too large, though the reclamation overhead is amortized due to reduced frequency of reclamation, the occasional freeing of large batches causes long program interruptions and dramatically increases *tail latency* for data structure operations.

Larger batch sizes also increase the *memory footprint* of applications, which makes memory utilization and allocation challenging in virtualized environments [14]. For example, increased memory footprints of virtual machines (VMs) or processes due to large batch sizes preclude the host machine from taking advantage of *Memory Overcommitment* where the available dynamic memory could otherwise be shared amongst multiple VM instances (or other processes).

Besides requiring programmers to find an acceptable batch size (i.e., *reclamation frequency*), most fast epoch-based SMR algorithms [1], [8], [9], [15] also have to determine an optimal increment frequency of a global timestamp (sometimes referred to as *epoch frequency*). The values of these parameters in tandem influence the time and space efficiency of SMR algorithms. Choosing an optimal value for these parameters can be quite challenging since they vary depending on the type of data structure, workload, and machine characteristics.

In this work, we turn the traditional SMR paradigm on its head. Whereas SMR algorithms usually ensure that reclaimers delay reclamation until a node can no longer be accessed by readers, we allow reclaimers to free immediately, and put the onus on readers to avoid inconsistency. Inspired by the recent *Memory Tagging* proposal of Alistarh et al. [16], we propose a new hardware mechanism called *Conditional Access* to allow readers to efficiently determine whether a node they are trying to access has been freed. It allows a thread to *conditionally read* (`cread`) a new location *only if* a set of programmer defined *tagged* locations have not changed since they were previously read. Similarly, threads can conditionally write (`cwrite`) a tagged location by `validating` that no tagged location has changed. Locations are tagged by invoking `cread`, and are manually untagged by invoking `untagOne` or `untagAll`.

Conditional Access is ideal for implementing data structures for which one can prove a read is safe if a small set of previously read locations have not changed since they were last read. For example, in a linked list that sets a *marked bit* in a node before deleting it, if a thread reads the next pointer of an unmarked node, and at some later time its marked bit and next pointer have not been changed, then it is still safe to dereference its next pointer. In such a data structure, once a node is unlinked and marked, it can *immediately* be freed, since doing so will merely cause subsequent `cread`s or `cwrite`s on the node to fail, triggering a *restart* (an approach common to many popular SMRs [2], [7], [9], [17]).

Conditional Access can be thought of as a generalization of load-link/store-conditional (LL-SC) where the load is also conditional, and the store can depend on many loads. Whereas an LL effectively tags a location, and an SC untags the location, in *Conditional Access*, locations are not automatically untagged when they are written. So, multiple `cread`s and `cwrite`s can be performed on the same set (or a dynamically changing set) of tagged locations.

Conditional Access also has some similarities to a restricted form of transactional memory. However, whereas hardware transactional memory (HTM) is increasingly being disabled

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00021

due to security concerns, we believe *Conditional Access* can be implemented more securely. For example, since a thread becomes aware of concurrent updates to its tagged nodes only when it performs a `cread` or `cwrite` and then checks a status register, we can avoid some timing attacks that are made possible by the immediacy of aborts (as a result of conflicting access by the other threads) in current HTM implementations.

Much of the information needed to efficiently implement *Conditional Access* is already present in modern cache coherence protocols. We propose a simple extension where tagging is implemented at the L1 cache level without requiring changes to the coherence protocol. At a high level, each L1 cache line has an associated *tag* (a single bit). This *tag* is set by a `cread` on any location in that cache line, and unset by an `untagOne` on a location in that cache line or an `untagAll` instruction. Each core tracks invalidations of its own tagged locations. (In SMT architectures, where k hyperthreads share a core, each hyperthread tracks invalidations of its own tagged locations).

While a detailed implementation at the microarchitectural level is beyond the scope of this paper, the extensions we require to the cache, and between the cache and processor pipeline, are a strict subset of those needed to implement HTM. This strongly suggests that *Conditional Access* implementations can be practical and efficient.

Conditional Access enables memory footprints similar to those of sequential data structures. This is desirable in modern data centers, to save costs related to memory over-allocation and to facilitate Memory Overcommitment [14]. Further, immediate reclamation can help in avoiding exploits that use the extended lifetime of unlinked objects in delayed reclamation algorithms to leak private data. It also has the potential to prevent denial of service attacks in which threads induce a schedule that causes batches of unreclaimed memory to grow unboundedly, leading to out-of-memory errors. Such attacks have been reported in RCU implementations in the Linux kernel [18].

This paper makes the following contributions. (A) We introduce *Conditional Access*, a set of hardware instructions that enable efficient immediate memory reclamation. (B) We prototype *Conditional Access* on an open source multicore simulator, Graphite. (C) We implement a benchmark comprised of multiple state-of-the-art memory reclamation techniques and data structures. (D) We show how *Conditional Access* can be used to avoid *use-after-free* errors for several data structure design patterns, including optimistic two-phased locking.

The remainder of the paper is organized as follows. Section II explains the core idea of *Conditional Access* and the semantics of the proposed instructions. Section III sketches a straightforward hardware implementation of *Conditional Access*. Section IV discusses how *Conditional Access* can be used with optimistic data structures, using a stack and a lazy linked list as examples, and briefly discusses correctness. Section V benchmarks *Conditional Access*, illustrating its efficiency and low memory usage. Related work appears in Section VI, followed by a conclusion in Section VII.

II. Conditional Access

A. Key Idea

A *use-after-free* error can be considered a special case of a read-write data race, where a shared memory location is accessed after it has been freed by a different thread. In modern systems with coherent caches, *use-after-free* errors are always preceded by events in the cache coherence protocol. Consider a traditional MESI protocol: To store a value at a location `X` that is currently in the shared state, a core C first invalidates copies of `X` at other cores by sending *invalidation* messages to all other cores. Upon receiving such a message, a core *invalidates* its copy of that location and responds with an acknowledgement message. Once C has received acknowledgements from all other cores, it has exclusive access to `X`. A thread that reads `X` after it is freed will respond to such invalidation messages before reading `X`. This reveals that, at the level of the coherence protocol, readers are aware that the memory location they are trying to access may have been concurrently modified. Moreover, a subsequent read of `X` must begin with a cache miss— an avoidable overhead if the information about concurrent modification could be harnessed.

Read-write data races and *use-after-free* errors are indistinguishable at the architectural level, which makes it difficult to identify *use-after-free* errors by looking solely at events in the cache coherence protocol. If we are willing to accept false positives, we can interpret each invalidation message as a sign of a possible *use-after-free* error. To that end, *Conditional Access* monitors the exchange of messages at the architecture level between an updating and a reading thread, and exposes these interactions to the program through specialized instructions to enable safe memory reclamation.

We expect the programmer to know which memory accesses *might* result in *use-after-free* errors, and we require the programmer to use our new instructions to perform these accesses. The hardware can then *tag* the corresponding cache lines, indicating to the hardware that *invalidation* of such a cache line is an event of interest. Subsequent loads and stores of a tagged location only complete if the location has not been *invalidated* since the location was tagged. Note that in order for this technique to work, reclaimers must do a store on some tagged location before freeing a node, so they can be sure to trigger a cache event that revokes other threads' access to that node. The reclaimer can then immediately free the node: Any thread that has tagged this node before it was freed, and subsequently tries to access the tagged node, will observe that the corresponding cache line has been *invalidated*, and will not perform the access.

Since the memory accesses are *conditioned* upon whether a memory location has been invalidated since it was previously tagged, we call our technique *Conditional Access*.

B. New State and Instructions

Additional Storage: (A) Each core *tags* an address for which it intends to monitor invalidation requests. Abstractly, the set of tagged addresses can be represented by a *tagSet*. (B)

Additionally, each core also maintains an *accessRevokedBit*, which is initially clear, and is set when its access is revoked for any of the addresses in its *tagSet*. For brevity, in this section, we assume that *tagSet*'s capacity is not bounded. Efficiently approximating this set is the subject of Section III.

Remote Events: For each entry in a core C's *tagSet*, the hardware is required to detect whether any other core has invalidated that cache line since C tagged it. If another core invalidates this cache line, the hardware must set C's *accessRevokedBit*.

Having described the *tagSet* and *accessRevokedBit*, we now describe the proposed instructions:

(1) `cread addr, dest`: Similar to a `load` instruction, `cread` updates register `dest` with the value at the address in register `addr`, but with two key differences: *tagging* and *conditional access*[a]. More specifically, `cread` atomically checks if `addr` is in *tagSet*, and if not, adds it to *tagSet*. It also checks if *accessRevokedBit* is set, and if so, skips the load, and updates some other processor state, such as a flag register, to indicate that there may have been a *use-after-free* error. In this case, we say the `cread` has *failed*. Otherwise, it loads the value at `addr` into `dest`, indicating that the memory access was safe. In this case, we say the `cread` has *succeeded*.

(2) `cwrite addr, v`: Unlike `cread`, `cwrite` does not update *tagSet*. Atomically: `cwrite` checks if the *accessRevokedBit* is set or `addr` is not in the *tagSet*, in which case the store is skipped and a processor flag is set to indicate that the `cwrite` has *failed* (suggesting there may have been a *use-after-free* error). Otherwise, it stores `v` at `addr`, and we say the `cwrite` has *succeeded*.

It is worth discussing here why `cwrite` fails when it executes on an `addr` which is not in the *tagSet*. This design decision rules out uses where programmers may invoke `cwrite` before invoking a `cread` (or in other words before first tagging a location). This helps to avoid tagging during a `cwrite`, which could incur significant delays if the access misses in the L1 Cache, making it easier to avoid tricky time-of-check to time-of-use (TOCTOU) issues. In particular we would prefer to avoid scenarios where a `cwrite` misses in the L1, waits for the data, and takes exclusive ownership of the line, only to discover that the *accessRevokedBit* has been set during the wait, thus eventually failing the `cwrite`. By requiring `cread` to be performed first, we move the high latency parts of this operation into a shared mode access, potentially reducing invalidations and coherence traffic.

(3) `untagOne addr`: The `untagOne` instruction does not access memory. Its purpose is to allow the programmer to remove an address from the *tagSet*. If `addr` is not in *tagSet*, `untagOne` has no effect. Once an address is removed from a core's *tagSet*, subsequent remote invalidations of the address will *not* set the core's *accessRevokedBit*.

(4) `untagAll`: `untagAll` clears the *tagSet* and unsets the value of *accessRevokedBit*. It is intended to be used in two cases: (1) when a `cread` or `cwrite` fails, at which point a data structure operation will need to be retried; and (2) before returning from a successful data structure operation.

Note, for SMT architectures with multiple hardware threads *additional storage* and *remote events* are required per hardware thread, instead of per core.

[a]For simplicity of presentation, we do not parameterize `cread` by the number of bytes to read from memory, or consider different addressing modes. In a practical system, several opcodes will be needed for these purposes.

III. Implementation

Conditional Access can be implemented by a straightforward extension of existing caches, such that modifications are only introduced between a processor and its primary cache, e.g., the L1 data cache. Based on our prototyping on a multicore simulator, we believe these changes are a strict subset of those required to implement HTM, which implies *Conditional Access* is practical and efficient to implement.

Implementing *Conditional Access* requires realizing the *tagSet* and *accessRevokedBit*. The proposed instructions use those data structures to track relevant invalidation messages, which are generated by the underlying cache coherence protocol.

(A) The *tagSet* can be approximated by adding one `tag` bit to each cache line of a core's L1 data cache. This is similar to how hardware transactional memory approximates its read and write sets. (B) The *accessRevokedBit*, which tracks the invalidations of the addresses in a thread's *tagSet*, requires adding one bit for each core. One way the *accessRevokedBit* could be implemented is by adding it to the condition code or flag registers of the host architecture (e.g., EFLAGS on x86).

Note, in SMT architectures, where k hyperthreads share a core, each hyperthread will track which of its cache lines are tagged and tracks invalidations of its tagged locations. For instance, on a 2-way SMT architecture, two *tag* bits and two *accessRevokedBit*s, one for each hardware thread, will be required.

Given these changes, we can now harness the cache coherence protocol to detect unsafe accesses. When a `cread` adds an address to the *tagSet*, it loads that line into the cache and sets the *tag bit* for that line. There are two ways in which the line can subsequently depart the cache: remote invalidation or a local associativity conflict. In either case, the cache must notify the hardware thread that its *accessRevokedBit* must be set, so that its subsequent `cread` or `cwrite` will fail. For remote invalidations, doing so must be atomic with acknowledging the remote request. For associativity conflicts, doing so must be atomic with fetching new data from the memory hierarchy. The atomicity requirements for `untagOne` and `untagAll` are simpler: they cannot be reordered with respect to loads and stores by the same hardware thread. Furthermore, `untagAll` must clear the *accessRevokedBit* for future operations.

Besides the aforementioned two ways, In SMT architectures, a thread's *accessRevokedBit* can be set upon a write to a shared cache line by another thread (in case of hyperthreading) or on a context switch. Setting the bit on a context switch is more straightforward to implement since it enables the Operating system to avoid keeping track of invalidations on behalf of

switched-out thread. These properties provide a foundation for the *Conditional Access* to be used in multiuser systems.

Intuitively, tagging in CA facilitates a kind of local protection of shared memory locations that does not trigger any additional coherence traffic. This is contrary to popular paradigms like hazard pointers [2] or other reservation-based [9] techniques, which always trigger global cache traffic between threads.

The aforementioned way of implementing *tagSet* means that the *tagSet* size is bounded by the associativity of the cache and therefore *tagSet* could overflow. This would lead to eviction of tagged addresses (in *tagSet*), causing *accessRevokedBit* to be set. This, in turn, could lead to spurious failures of subsequent `cread`s or `cwrite`s which could stall progress. However, in practice it is not an issue because in most cases the *tagSet* is small. Our experiments (Section V) show associativity does not have any significant impact on progress for the workloads we consider.

IV. Using Conditional Access with Optimistic Data Structures

In this section we discuss how *Conditional Access* can be used to achieve safe memory reclamation with optimistic data structures such as lists [19] and external binary search trees [20]. Operations of many such data structures have a search phase consisting of multiple reads, wherein a thread continuously traverses the next fields of nodes until it has visited a set of nodes it is interested in, where the operation eventually takes effect. After reaching the nodes of interest, the operation may perform zero or more writes. For example, in a linked list, a thread might traverse multiple links to find a predecessor and current node where an operation should take effect.

For ease of exposition we assume that each node fits in a single cache line, and a cache line contains only one node. Thus, adding a node to a core's *tagSet* implies adding a cache line containing the node to the core's *tagSet*.

We start by stating the following high level directives required for all data structures to be able to correctly use *Conditional Access*.

(DI) **Replace and Analyse:** *Replace*: all read/write accesses to nodes that can be freed should be substituted by the corresponding `cread` and `cwrite` instructions. This enables *Conditional Access* to tag a node and monitor it for concurrent modification and notify programmers by updating a flag register. *Analyze*: If a `cread` or `cwrite` fails, the operation should immediately `untagAll` and retry. A failed instruction implies a node could have been concurrently freed, therefore any future access will not be safe.

(DII) **Validate Reachablility:** A node is tagged when it is first `cread`. In order to ensure that the tagged node is valid, it should be verified it was reachable in the data structure after the fact.

We now demonstrate how these directives can be applied to use *Conditional Access* in different classes of optimistic data structures. Depending upon the data structures, DI could be partially relaxed, as we will see in the example of a lazy list or DII may not be needed as we will see in the example of a lock free stack. The lazy list requires some more rules which are detailed in the Section IV-B.

A. In Data Structures with Single Writes

Data structures with a single write in their update phase include some list based stacks [21] and queues [22], both of which we have implemented. For the purpose of illustration, we will consider a list-based unbounded lock-free stack. In such a stack, a `push` operation involves reading a `top` pointer, allocating a new node for a key value to be pushed, and then doing a Compare-and-Swap (CAS) to set the node as new `top`. Likewise, a `pop` operation consists of reading the `top`, and then atomically setting the `top` to its next node. After a `pop`, the unlinked node cannot be freed if a concurrent thread might still access it.

The original operations of the stack could be upgraded to enable *Conditional Access* by simply replacing every read with `cread` and the CAS with `cwrite` (DI). Then the `pop` operation could immediately free the unlinked node as shown in Algorithm 1. Note, in our pseudocode `CAFAIL` is set when a `cread` or `cwrite` fails. This is similar to updating a flag register.

Linearizability of the upgraded operations. The correctness follows from the fact that the `top` is read using `cread`, which adds it to the corresponding thread's *tagSet*, upon which the thread starts monitoring for any subsequent modifications to `top`. Since the top itself is never deleted it is guaranteed to be always in the data structure at the time it is added to the *tagSet* (DII). At the beginning of an operation the *accessRevokedBit* is clear and it is only set when the thread receives an invalidation request for the `top` when it is modified elsewhere. Later, the thread attempts to change the `top` using `cwrite` which atomically checks the *accessRevokedBit* for any interfering memory access. It fails if the *accessRevokedBit* is set. This causes the thread to remove the top from its *tagSet*, clear the *accessRevokedBit* using `untagAll`, and then retry the operation. Otherwise, the thread succeeds by changing `top` to another node. The push and pop operations can be linearized on the successful `cwrite` at line 9 and line 17 in Algorithm 1, respectively.

Note, the call to free at line 18 is safe because whenever a core C1 modifies the `top` (either for push or pop), any other core C2 having access to `top` will fail its `cwrite` because C1 will invalidate C2's tag by setting its *accessRevokedBit*.

Conditional Access is ABA-safe despite the fact that it allows immediate reuse of freed objects. Suppose a thread T1, in order to insert a new node, reads an address A from the `top` into a local variable t. Then just before it executes a CAS to set `top` to t's next, some other thread T2 removes A by setting a node at address B as the new `top`, frees A, and then pushes a new node at this recycled address A, making it the new `top`. Now T1 would succeed its CAS (based on address comparison) as the `top` still contains the address A, which matches the expected address stored in its local variable t. Thus it incorrectly succeeds when it should have failed—a typical ABA case. *Conditional Access* prevents this error as

Algorithm 1 Using *Conditional Access* with unbounded lock free stack. Lines annotated as LP are the linearization points.

```
1: type Node {Key key, Node *next}
2: class Stack {Node *top}
3: #define CA_CHECK if CAFAIL then untagAll(); goto retry;

4: procedure PUSH(key)
5:     newtop = new node(key);
6:     retry:
7:     t ← cread(top); CA_CHECK
8:     newtop->next = t;
9:     cwrite(&top, newtop); CA_CHECK                    ▷ LP
10: end procedure

11: procedure POP( )
12:     retry:
13:     t ← cread(top); CA_CHECK
14:     if NULL == t then
15:         untagAll(); and return;
16:     end if
17:     cwrite(&top, t->next); CA_CHECK                   ▷ LP
18:     free(t)
19: end procedure
```

`cwrite`, unlike a CAS, is not based on comparing two values. Instead, it relies on the underlying cache invalidation messages to detect that a location has been modified since it was last read.

B. In Data Structures having Multiple Writes with Locks

Another category of linked concurrent data structures have update operations wherein threads optimistically traverses a sequence of nodes ending in multiple updates within a critical section guarded by locks. One example is the lazy list [19].

Operations of data structures with such design patterns could be upgraded to enable the proposed technique's *safe memory reclamation* using the following broad guidance:

1) In the search phase, use DI to replace all reads with `cread`. Use `untagOne` to remove previously traversed nodes from current thread's *tagSet* when they are no longer required to prove that a node, to be accessed in the future, is reachable in its data structure at the time it is tagged. If a `cread` fails during the traversal then do `untagAll` and retry the search. For read only operations this will suffice. Update operations require the following steps:
2) Use try locks designed using `cread`/`cwrite` (Algorithm 2) to lock all the nodes identified at the end of the search. This marks the beginning of a critical section to execute the updates atomically. If lock acquisition fails on any of the nodes then unlock the previous nodes (if any), do `untagAll` and retry the operation.
3) Within the critical section use normal writes to execute intended updates. This is safe because the nodes are guarded by the critical section and therefore cannot be concurrently updated or reclaimed (partial relaxation of DI).
4) If the update is a delete, mark the node before unlinking it. This satisfies the *no unlinked access rule*.
5) Finally, unlock any locked nodes and execute `untagAll` before exiting the operation. Note that unlock may use regular stores instead of `cwrite`, since locked nodes cannot be freed by other threads.

By the way of example of a lazylist (in reference to Algorithm 3) we will demonstrate how we can easily upgrade it to use *Conditional Access*.

Using D1, all the regular reads are replaced by `cread`s in searches, as shown in Algorithm 3. As explained in the specification, the `cread`s atomically: add a node to current thread's *tagSet*, if it is not in it already, check that the *accessRevokedBit* is clear, and complete a normal read, if the condition succeeds. However, if the `cread` fails (when `CAFAIL` is set) then it could be the case that a subset of the nodes in the *tagSet* have been modified (potentially deleted) since they were last accessed, therefore it may not be safe to read them. In such a case the *tagSet* is emptied, the *accessRevokedBit* is unset using `untagAll`, and the search is retried. Otherwise it continues and eventually stops when some `pred` and `curr` nodes of interest are found (line 22 in Algorithm 3).

Note, if we do not untag previous nodes during searches, then since `cread`s tag nodes, we will have all the nodes in the search path added to a thread's *tagSet*. This could cause `cread`s to fail when any node (relevant to current access or not) in the search path is modified, which forces operations to retry repeatedly. In other words, certain updates will be serialized, as if threads acquired a global lock, which will inhibit concurrency. As a remedy to this problem, threads can untag previous nodes using `untagOne` and are only required to keep two consecutive nodes tagged at any given time, which is equal to the number of nodes required to carry out updates, much like hand-over-hand locking.

Furthermore, in order to guarantee searches are safe we need to ensure that at the time a node is tagged it is reachable in the list (DII). To see why, assume a case where a thread accesses content of an arbitrary node using `cread`. During this `cread`, atomically: a cache line containing the node will be tagged and then its content will be loaded. Now, if the node was already marked before it was tagged by the `cread`, then a subsequent `cread` would succeed even though the node is marked (logically deleted), which is not safe as the node could be reclaimed (a *use-after-free* error). This is resolved by validating that a node is not marked, immediately after the `cread` that tagged it. If the node is found to be marked, validation fails and the corresponding operation untags all nodes and retries. For example, in the lazy list a node is first tagged during `cread` at line 5 in Algorithm 3, due to a `validate()` invoked from line 11, 13, or 19. If `validate()` returns False due to the node being marked then the operations untags all nodes and retries. This way DII is satisfied.

One may further ask, what if the node was marked (already logically deleted) and also freed before it is tagged? In that case a subsequent `cread` could succeed as its *accessRevokedBit* will not be set since no update will occur after the node was tagged. This could cause a *use-after-free* error. However, this cannot happen because, in order to free the node, a reclaimer has to unlink it by modifying the next field of its predecessor,

Algorithm 2 *Conditional Access* based lock. Precondition: the node containing the lock field should be `cread` so that it is tagged.

```
1: procedure TRYLOCK(bool *lock)
2:     lockVal ← cread(lock);
3:     if CAFAIL or 1 == lockVal then return False;

4:     cwrite(lock, 1); if CAFAIL then return False;
5:     return True;
6: end procedure

7: procedure UNLOCK(bool *lock)
8:     *lock ← 0;            ▷ safe as a node can only be mutated by owner.
9: end procedure
```

which is already in the thread's *tagSet*. Thus, if the predecessor node is modified the thread's *accessRevokedBit* will be set and the `cread` will fail, preventing unsafe access. This invariant is maintained during a search that eventually yields a `pred` and `curr` that were reachable in list at the time they were tagged.

Later, before starting the updates, locks on the `pred` and `curr` nodes are acquired (line 33 & 36 for `insert()` and line 49 & 52 for `delete()`). However, it may happen that after the search returns the nodes and before the locks are acquired some thread may delete these nodes. In that case if the lock is accessed with normal reads and writes then the thread may attempt acquiring lock on a freed node which could lead to undefined behaviour: unlike `cread`s, regular reads do not have the ability to check whether the nodes have been modified. Thus, to resolve this issue we provide `cread`/`cwrite` based try locks which only acquire the lock on a node if it has not been modified (deleted) concurrently.

Algorithm 2 depicts the implementation of this lock. It has a precondition that the node containing the lock field should have been previously accessed using `cread` so that it gets added to its thread's *tagSet*, enabling a `cread`/`cwrite` to verify through *accessRevokedBit* whether the node has been modified since then. In further detail, a thread does a `cread` on the lock variable. If it sets `CAFAIL`, the node of which the lock is part might have been deleted; if it returns 1, it means that lock is busy. In both the cases, the lock acquisition fails. Otherwise, a thread proceeds to acquire the lock by setting the lock field to 1 using a `cwrite`, which again checks if the node containing the lock field has not been modified (possibly deleted). If the check succeeds it writes 1 to the lock field and returns `True`, indicating that lock acquisition is successful. Otherwise if the `cwrite` fails (by setting `CAFAIL`) it returns `False` indicating that lock acquisition has failed, and the operation which invoked the lock untags all nodes and retries.

The insert operation(line 29, Algorithm 3), first executes `locate`, which returns tagged `pred` and `curr` nodes along with `currkey` (key field of `curr`). If the key to be inserted is already present in the list then the operation returns false. Otherwise, the key is not present and needs to be inserted. To insert the key, first the *Conditional Access* based trylocks on the `pred` and `curr` nodes are acquired, then a new node is

Algorithm 3 Using *Conditional Access* with lazylist [19]. Lines annotated as LP are the linearization points.

```
1: type Node {Key key, lock, mark, Node *next}
2: class Lazylist {Node *head}
3: #define CA_CHECK if CAFAIL then untagAll(); goto retry;

4: procedure VALIDATE(Node *node)
5:     isMarked ← cread(node->mark); if CAFAIL then return False;
6:     if isMarked then return False; else return True;
7: end procedure

8: procedure LOCATE(Key key)
9:     retry:
10:    pred ← cread(head); CA_CHECK
11:    VALIDATE(pred); if False then untagAll(); goto retry;
12:    curr ← cread(pred->next); CA_CHECK
13:    VALIDATE(curr); if False then untagAll(); goto retry;
14:    currkey ← cread(curr->key); CA_CHECK
15:    while currkey < key do
16:        untagOne(pred);
17:        pred ← cread(curr); CA_CHECK
18:        curr ← cread(curr->next); CA_CHECK
19:        VALIDATE(curr); if False then untagAll();goto retry;
20:        currkey ← cread(curr->key); CA_CHECK
21:    end while
22:    return ⟨ pred, curr, currkey ⟩;
23: end procedure

24: procedure CONTAIN(key)
25:    ⟨ pred, curr, currkey ⟩ ← locate(key); ▷ LP:When currkey was read.
26:    untagAll();
27:    return (currkey == key);
28: end procedure

29: procedure INSERT(key)
30:    retry:
31:    ⟨ pred, curr, currkey ⟩ ← locate(key);        ▷ LP when insert fails.
32:    if currkey == key then untagAll; & return False;
33:    if False == tryLock(&pred->lock) then      ▷ attempt locking pred.
34:        untagAll & retry;
35:    end if
36:    if False == tryLock(&curr->lock) then      ▷ attempt locking curr.
37:        unlock(&pred->lock);
38:        untagAll & retry;
39:    end if
40:    node ← new Node(key, curr);
41:    pred->next ← node;                          ▷ LP when insert succeeds.
42:    unlockAll() and untagAll;
43:    return True;
44: end procedure

45: procedure DELETE(key)
46:    retry:
47:    ⟨ pred, curr, currkey ⟩ ← locate(key);        ▷ LP when delete fails.
48:    if currkey != key then untagAll; & return False;
49:    if False == tryLock(&pred->lock) then      ▷ attempt locking pred.
50:        untagAll; & retry;
51:    end if
52:    if False == tryLock(&curr->lock) then      ▷ attempt locking curr.
53:        unlock(&pred->lock);
54:        untagAll; & retry;
55:    end if
56:    curr->mark ← true;                          ▷ LP when delete succeeds.
57:    pred->next ← curr->next;
58:    unlockAll(); & untagAll;
59:    free(curr);
60:    return True;
61: end procedure
```

created and inserted between pred and curr. Following that all locks are released, nodes are untagged, and then the operation

returns true.

Note, because the pred and curr nodes are already locked, no other thread could ever modify them without acquiring lock first. Therefore, validation to check whether the node has been concurrently freed is not needed. This allows us to use normal reads and writes instead of cread/cwrite within the critical section.

The delete operation(line 45, Algorithm 3) invokes locate, which returns tagged pred and curr nodes along with currkey. If the key to be deleted is not present in the list then the operation untags the nodes and returns false. Otherwise, similar to the insert operation, it acquires the trylocks on both nodes, does a write on the curr node to set its mark field, unlinks the curr node, unlocks and untags both the nodes, frees the curr node and then returns true.

1) Correctness: If all the aforementioned rules are followed to enable *Conditional Access* in the lazylist then the list is linearizable and all access in it are safe. Contains or unsuccessful inserts and deletes, which behave like contains, can be linearized at the time when the key of curr was read. Whereas, successful inserts and deletes can be linearized when a new node is linked (line 41) or a node is marked (line 56), respectively. Also, because creads never dereference an unreachable (unlinked) node, *use-after-free* errors do not occur. Therefore the lazy list with *Conditional Access* is safe. We state the following key lemmas and theorems here. Detailed proof will be available as a companion technical report [23].

Lemma 1. *A node's predecessor is unmarked and its next field points to the node at the time it is tagged.*

Theorem 2 (Safety)**.** *Threads cannot access reclaimed nodes in the Conditional Access based list.*

V. Experiments

We prototype *Conditional Access* (CA) using the Graphite multicore simulator [24]. Our modifications were restricted to the L1 data cache level; we did not change the cache coherence protocol. Graphite is configured to use a directory based MSI cache coherency protocol with a private 32K L1 and a shared inclusive 256K L2 cache. Each cacheline is 64 bytes and each thread runs on a dedicated simulated core with a basic branch prediction mechanism and an out-of-order memory subsystem.

We evaluate the scalability and memory efficiency of CA using microbenchmarks that stress test the lazy list and an external binary search tree (extbst). We also use stack and hash table microbenchmarks to evaluate CA at different contention levels. The keys in the lazy list, stack and hash table range from 0 to 1K; the extbst keys range from 0 to 10K. The hash table has 128 buckets, where each bucket is a lazy list.

Each of these data structures are made to use the following safe memory reclamation techniques: a leaky implementation(**none**:), *Conditional Access* (**ca**), the 2geibr variant of IBR (**ibr**), **rcu**, quiescent state based reclamation (**qsbr**), hazard pointers(**hp**), and hazard eras (**he**). CA reclaims each deleted node immediately and requires no other parameters. The other reclamation schemes were configured to attempt reclamation after every 30 successful remove operations (*reclamation frequency*). For epoch based schemes (ibr, rcu, qsbr and he) the epoch were configured to change after every 150 allocations (*epoch frequency*). These values are the default in the IBR benchmark [9].

Each trial in each experiment prefills its data structure to 50% full and executes 3K operations per thread. The number of threads varies from 1 to 32. Each time a thread invokes a data structure operation, it randomly chooses an operation with a random key. In our experiments threads choose insert or delete with equal probability of 0%, 5% or 50%, allowing us to run experiments with 0% (read only), 10% and 100% updates, respectively. Because the insert and delete probabilities are equal in all our workloads the data structure size remains roughly constant, storing half the elements in the key range. For each workload configuration we report the average of three runs. There was no significant variance across the runs.

Throughout the experiments in Figure 1 and Figure 2; hp, he and ibr are generally slower than the other algorithms. This mainly can be attributed to high per-read overheads, as these algorithms have read/write fences to access or update reservations and epochs, respectively. Additionally, these algorithms have reclamation overhead which requires scanning of reservations to determine which records are safe to free. In general this results in poor cache behaviour and high operation latency.

On the other side, rcu and qsbr have no per-read overhead. Their main overhead arises from their reclamation events, where batches of retired objects are freed after scanning the epochs of all the processes. This is amortized over multiple operations. As a result these algorithms are faster and perform similar to the baseline none, across workloads and data structures.

In read-only workloads, CA is comparatively slower than rcu, qsbr and none. This is due to the increased latency: checking the *accessRevokedBit* after each cread increases the instruction count. Since there are no conflicts, these checks are superfluous. However, in workloads with updates, CA is closer to or faster than rcu, qsbr and none. It even outperforms these algorithms in high contention scenarios (i.e., high updates and high thread counts). This is due to the fact that CA avoids read-write fences for both readers and reclaimers. CA brings additional benefits. Immediate reclamation improves cache and TLB locality, especially relative to none; it discovers failures earlier than other algorithms, which enables it to restart without wasting as much work; and it avoids some cache miss latencies. All these contribute to low latency and higher throughput. The low cost of cache misses is due to a property that unlike regular reads, in creads the impact of cache misses remains confined to its core [16]. We explain this in following paragraphs.

In data structure operations with normal reads and writes, all threads that share memory locations experience latency due to cache misses. Consider a lazy list, and suppose that thread T1 is about to acquire locks on its pred and curr nodes. Suppose that another thread T2 has read the pred and is about to re-read it. At this point, at the cache level, both T1 and T2 will have copies of the cache lines corresponding to these

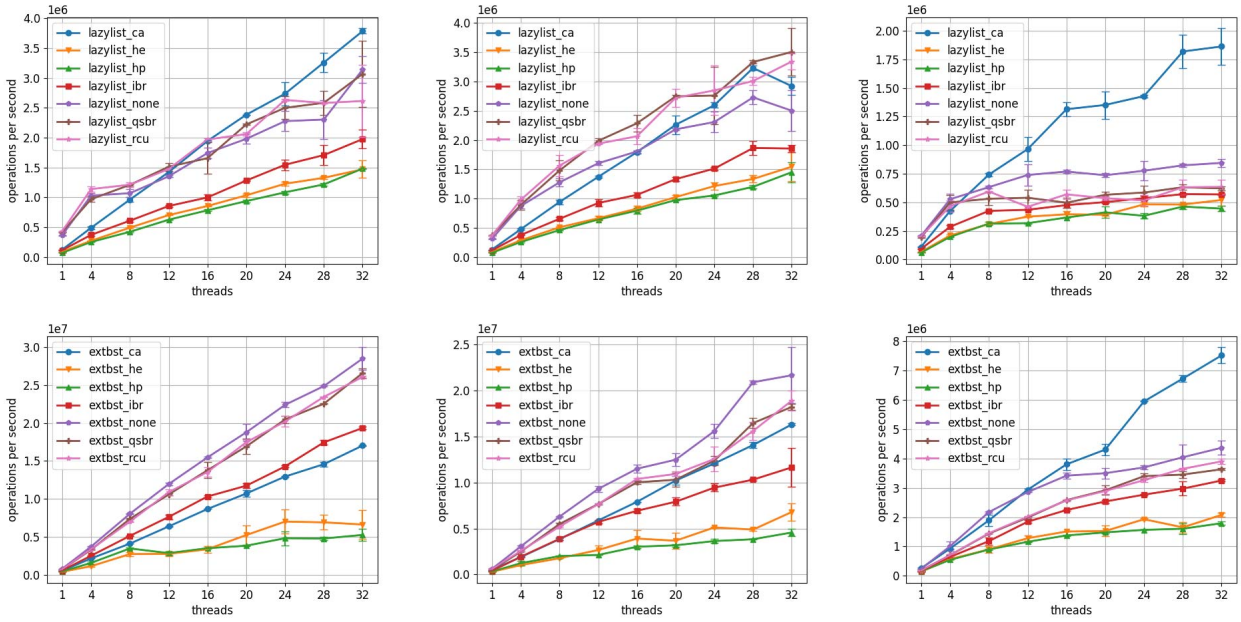

Fig. 1: Evaluation of throughput. Y axis: throughput. X axis: #threads. Left: 0i-0d. Middle: 5i-5d. Right: 50i-50d. (Top Row) Lazy linked-list, size:1K. (Bottom Row) External BST, size:10K.

addresses in the shared state. When T1 acquires a lock on `pred` it does a write. This triggers coherence traffic: all other readers of `pred` must invalidate their copies of the cache line (here T2). When T2 reads `pred` again it will suffer a cache miss as its copy of the cache line is invalid. In order to serve the cache miss:

- T2 triggers a cache level transaction to fetch the latest copy of the cache line and waits for a response.
- T1, which has the cache line in M state may be forced to write its copy of the cache line back to the memory hierarchy, and also supply it to T2.

This wastes T2's compute time, because T2 will ultimately see that the line has changed, necessitating that it restart its operation. If it had not waited, it could have already restarted and executed multiple instructions. Furthermore, since T1 acquired the lock on `pred` it is likely to write to `pred` again. T2's request caused the line to downgrade from M to S in T1's cache, so a subsequent write by T1 will need to begin with an ownership request that causes T2 to re-invalidate the line. Such frequent downgrading to shared state and upgrading to modified state interferes with the gains made by write buffering and makes it difficult to hide the cache latency. These overheads worsen with increases in contention on shared locations.

On the other hand, in data structures designed using `cread` and `cwrite`, T2's second `cread` will fail validation and retry its operation by detecting that the line is no longer present, *without requesting a new copy of the line*. Unlike the aforementioned issues with regular reads/writes, CA allows T2 to skip requesting the value of `pred`, which prevents global cache traffic. This avoids read latency for T2, and also helps T1 to avoid a cache state upgrade transactions. In other words, unlike regular read/write based data structures, the impact of failure to access cache lines in data structures with CA remains confined to a local core [16].

Thus, in all the data structure implementations, the lower L1 data cache latencies that result from the aforementioned properties allow CA to be as good as the other algorithms, if not faster, when contention is high. In read only workloads, CA is slower or comparable to the baseline (none) and other fast algorithms (`qsbr` and `rcu`) mainly due to overhead of its higher instruction count.

Figure 3 looks at memory overheads: For each of the reclamation schemes, we measure the number of nodes that were allocated but not yet freed (Y axis) during execution of the lazy list data structure after every 1000 operations (X axis). This test exposes the amount by which the memory footprint of a data structure increases when paired with different reclamation schemes. For this experiment, we use a lazy list with values in the range of 0 to 1000, initially pre-filled with 500 nodes. The experiment has 16 threads operating on the list. During the measured part of the experiment all threads execute insert and delete operations with equal probability of 50% (100% update workload). Each thread runs 5000 ops.

In the ideal case, at any time during the experiment the list size should be roughly 500 due to the workload characteristics, and the number of nodes deleted but yet not freed should be zero. The CA scheme has a consistent reading of roughly 500 nodes that are allocated but not freed; these are the nodes that are still reachable in the list. This confirms that we are achieving immediate reclamation and keeping the memory footprint low.

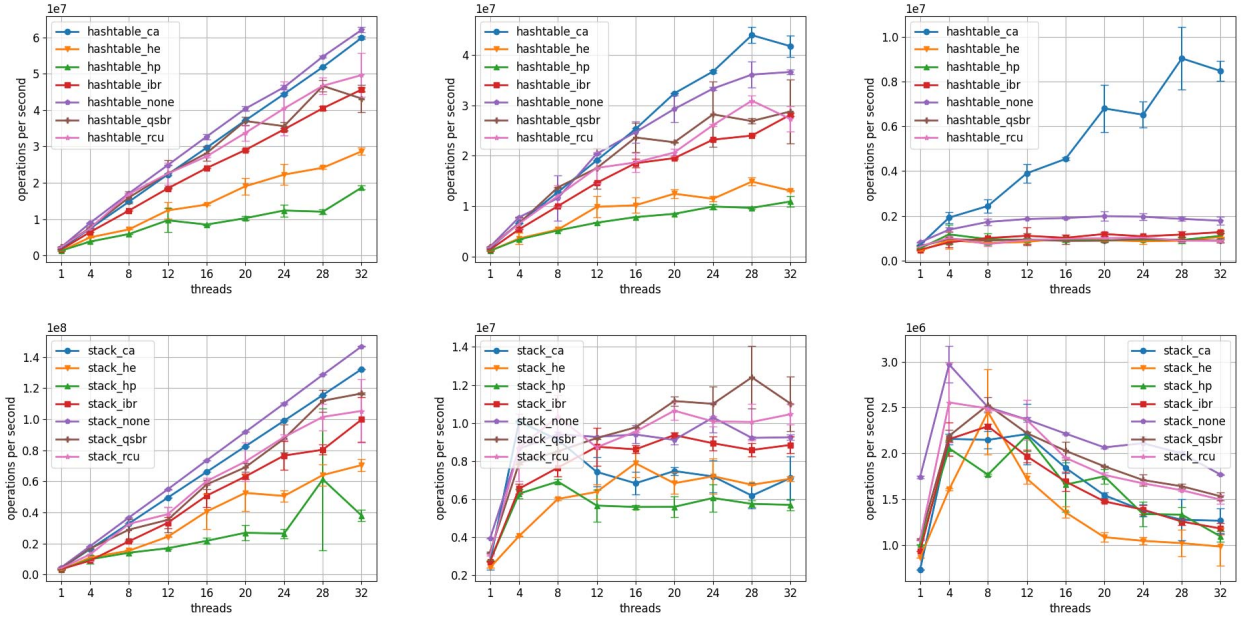

Fig. 2: Evaluation of throughput. Y axis: throughput. X axis: #threads. Left: 0i-0d. Middle: 5i-5d. Right: 50i-50d. (Top Row) Chaining Hash table, #buckets:128, MAX size 128K. (Bottom Row) Stack.

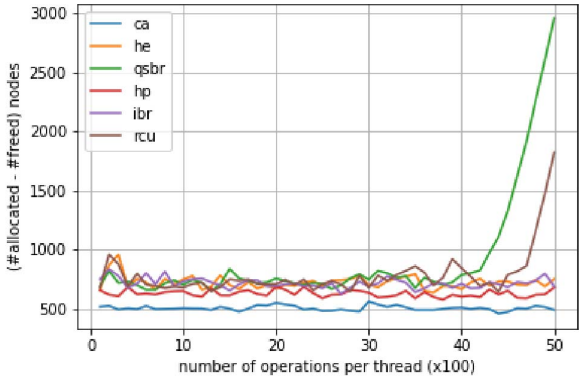

Fig. 3: Memory Consumption: Shows Number of nodes allocated but not yet freed for a list of size ∼500. Insert and delete percent is 50 each. For 16 threads.

Since the other reclamation schemes defer memory reclamation of deleted nodes by collecting them in a local retired list, the number of nodes that have not been reclaimed increases, which in turn leads to increased memory footprint. This intuition is verified in the chart, as hp, he, ibr, rcu and qsbr all report a higher number of un-reclaimed nodes It is worth mentioning that, since in qsbr and rcu a delayed thread could prevent reclamation of all threads, the number of unreclaimed nodes could increase without bound. Had we run the experiment for longer, the number of unreclaimed nodes for these schemes would be expected to balloon as soon as any thread context switched.

VI. Related Work

A. Discussion on Reclamation Techniques

Many existing safe memory reclamation techniques delay reclamation of unlinked nodes, which could be broadly categorised as epoch based reclamation (EBR), hazard based reclamation (HBR), reference counting based reclamation (RCBR) and hybrid reclamation (HYR): using a combination of prior techniques or specialized hardware support. EBR [1], [11], [25] schemes are fast but could have an unbounded number of unreclaimed nodes. HBR [2], [26], [27] and RCBR [8], [10], [26], [28] can bound the number of unreclaimed nodes but are generally slow due to high per-read overhead or node instrumentation overhead. HYR techniques have achieved both speed and bounds on unreclaimed number of nodes with varying success, but require assumptions pertaining to specialised hardware or operating system and memory allocators [3]–[9], [12], [13], [15], [17], [29]–[36]. Nevertheless, these techniques still prefer to reduce the reclamation algorithm's overhead by delaying reclamation using batches that increase the memory footprint. Surveys of these batch-based reclamation techniques appear in [1], [13], [33]. In this section we focus on techniques which could provide immediate reclamation [33], [37] and therefore are most closely related to *Conditional Access*.

Zhou et al. [37] make use of a sequence of short hardware transactions which execute in hand over hand fashion to design concurrent data structures that retain the property of immediate memory reclamation. The technique relies on augmenting the

data structure with a table of metadata, which can be a source of false conflicts. Consequently, it does not appear to be as general as *Conditional Access*. Moreover, we found that the frequent starting and committing of transactions for read-only operations introduced significant latency.

VBR [33] attaches metadata to each mutable field of each node in a concurrent data structure. It also requires a type preserving allocator, where unlinked nodes can never be returned to the operating system. Threads can detect use-after-free errors through the per-field metadata, which is updated atomically with the corresponding field. While VBR can support immediate reclamation, it is most efficient when it waits until it has a batch of nodes to reclaim in a single operation.

On the other hand, *Conditional Access* does not require any metadata to achieve safe reclamation and only makes use of the implicit book-keeping of the underlying cache-coherence protocol. In addition, since it does not make any assumptions about the number of threads present in the system, it is fully adaptive [38]. Furthermore, whereas HTM can accelerate timing-based attacks by leveraging the immediacy with which a thread is aborted upon a memory conflict [39], in particular, transaction rollbacks could lead to data leaks [40], we believe *Conditional Access* is less risky, since threads must poll to learn of remote coherence events.

B. Discussion on Similar Synchronization Techniques

Conditional Access is inspired by, but quite different from, the Memory Tagging proposal of Alistarh et al. [16]. Perhaps the most significant difference is that *Conditional Access* solves the safe memory reclamation problem (and moreover offers immediate reclamation), in addition to providing useful synchronization primitives for designing concurrent data structures. In contrast, Memory Tagging does not address the memory reclamation problem, and it requires a data structure designer to rely on separate safe memory reclamation algorithms, which come with their own tradeoffs.

In reference to the programming interface, *Conditional Access* offers `cread`, which is critical to our immediate memory reclamation technique. The `cread` instruction has no equivalent instruction in Memory Tagging, and it is not clear how one could implement `cread` using memory tagging. We also streamlined tagging by integrating it into `cread`, whereas Memory Tagging requires a programmer to use an explicit *AddTag* instruction before reading.

From an implementation standpoint, *Conditional Access* does not require changing the underlying coherence protocol, whereas Memory Tagging's Invalidate and Swap (IAS) instruction does, as this single instruction can invalidate many (potentially non-contiguous) remote cache lines (potentially spanning many pages). Moreover, *Conditional Access* requires only 1 bit per cache line (2 bits per cache line in case of 2-way hyperthreading), whereas Memory Tagging needs to additionally maintain a set of addresses to invalidate with IAS.

It is worth noting that at the outset *Conditional Access* may appear similar to HTM with early release (ER) [41], [42]. Possibly, one could achieve many aspects of our work by using HTM with early release. However, this would introduce various downsides. HTM defaults to putting all reads and writes into the read/write sets. This includes the stack, the allocator, library code, etc. In data structures, many reads and writes would need to be released, which would increase the instruction count significantly. This could reduce performance and might yield a less convenient interface than *Conditional Access*.

Practically, some commercial HTMs have a region based (not per access) *disable tracking* feature, for instance, Intel's new TSXLDTRK [43], and IBM's TSUSPEND/TRESUME [44], but this does not *release* load tracking of already-read locations, it only prevents tracking of future accesses. This is different from *Conditional Access*'s proposed `untag` instruction which allows the release of any previously accessed location. Among Early Release proposals, we are not aware of any that release writes, although AMD's 2008 ASF proposal allowed per-access decisions about whether or not to track [45]. However, ASF remains unimplemented.

Unlike HTM, *Conditional Access* does not need a write set at all, which admits a simpler implementation in hardware, as well as simpler conflict tracking and resolution.

VII. Conclusion

In this paper, we introduced *Conditional Access*, a hardware extension that enables concurrent data structures to reclaim memory immediately, without introducing new inter-thread coordination. *Conditional Access* is fast. Unlike its competitors, it does not require tuning to achieve high performance, and is tailored to the needs of modern optimistic data structures.

To date, we have used *Conditional Access* for simple non-blocking data structures, as well as optimistic lock-based data structures. In the future, it would be interesting to determine whether *Conditional Access* can also be used for more complex lock-free data structures. We also believe that there are exciting opportunities at the interface between *Conditional Access* and non-volatile main memory technologies.

Acknowledgments

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Program grant: RGPIN-2019-04227, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Launch Supplement: DGECR-2019-00048, National Science Foundation under Grant No. CNS-CSR-1814974, and the University of Waterloo. The findings and opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. We also thank anonymous reviewers for helping us improve the manuscript.

References

[1] T. A. Brown, "Reclaiming memory for lock-free data structures: There has to be a better way," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, 2015, pp. 261–270.

[2] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[3] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[4] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Forkscan: Conservative memory reclamation for modern operating systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 483–498.

[5] "Threadscan: Automatic and scalable memory reclamation," D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, Eds., vol. 4, no. 4. ACM New York, NY, USA, 2018, pp. 1–18.

[6] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi, "Fast and robust memory reclamation for concurrent data structures," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016, pp. 349–359.

[7] N. Cohen, "Every data structure deserves lock-free memory reclamation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–24, 2018.

[8] R. Nikolaev and B. Ravindran, "Hyaline: fast and transparent lock-free memory reclamation," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 419–421.

[9] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott, "Interval-based memory reclamation," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 1–13, 2018.

[10] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr, "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, 2002.

[11] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1270–1285, 2007.

[12] R. Nikolaev and B. Ravindran, "Universal wait-free memory reclamation," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 130–143.

[13] A. Singh, T. Brown, and A. Mashtizadeh, *NBR: Neutralization Based Reclamation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 175–190. [Online]. Available: https://doi.org/10.1145/3437801.3441625

[14] E. VMware, "Understanding memory resource management in vmware esx 4.1."

[15] R. Nikolaev and B. Ravindran, "Crystalline: Fast and memory efficient wait-free reclamation," *arXiv preprint arXiv:2108.02763*, 2021.

[16] D. Alistarh, T. Brown, and N. Singhal, "Memory tagging: Minimalist synchronization for scalable concurrent data structures," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 37–49.

[17] "Automatic memory reclamation for lock-free data structures," N. Cohen and E. Petrank, Eds., vol. 50, no. 10. ACM New York, NY, USA, 2015, pp. 260–279.

[18] P. E. McKenney, D. Sarma, I. Molnar, and S. Bhattacharya, "Extending rcu for realtime and embedded workloads," in *Ottawa Linux Symposium, pages v2*, 2006, pp. 123–138.

[19] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," in *International Conference On Principles Of Distributed Systems*. Springer, 2005, pp. 3–16.

[20] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010, pp. 131–140.

[21] R. K. Treiber, *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.

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

[23] A. Singh, T. Brown, and M. Spear, "Efficient hardware primitives for immediate memory reclamation in optimistic data structures," 2023. [Online]. Available: https://arxiv.org/abs/2302.12958

[24] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[25] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, vol. 509518, 1998.

[26] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 2, pp. 146–196, 2005.

[27] D. Dice, M. Herlihy, and A. Kogan, "Fast non-intrusive memory reclamation for highly-concurrent data structures," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, 2016, pp. 36–45.

[28] G. E. Blelloch and Y. Wei, "Concurrent reference counting and resource management in wait-free constant time," *arXiv preprint arXiv:2002.07053*, 2020.

[29] P. Ramalhete and A. Correia, "Brief announcement: Hazard eras-non-blocking memory reclamation," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017.

[30] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir, "On the power of hardware transactional memory to simplify memory management," in *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2011, pp. 99–108.

[31] N. Cohen and E. Petrank, "Efficient memory management for lock-free data structures with optimistic access," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 254–263.

[32] R. Nikolaev and B. Ravindran, "Snapshot-free, transparent, and robust memory reclamation for lock-free data structures," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 987–1002.

[33] G. Sheffi, M. Herlihy, and E. Petrank, "Vbr: Version based reclamation," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 443–445.

[34] A. Correia, P. Ramalhete, and P. Felber, "Orcgc: automatic lock-free memory reclamation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[35] D. Anderson, G. E. Blelloch, and Y. Wei, "Concurrent deferred reference counting with constant-time overhead," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 526–541.

[36] N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, Y. Sun, and Y. Wei, "Space and time bounded multiversion garbage collection," *arXiv preprint arXiv:2108.02775*, 2021.

[37] T. Zhou, V. Luchangco, and M. Spear, "Hand-over-hand transactions with precise memory reclamation," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017.

[38] M. Herlihy, V. Luchangco, and M. Moir, "Space-and time-adaptive nonblocking algorithms," *Electronic Notes in Theoretical Computer Science*, vol. 78, pp. 260–280, 2003.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[40] Intel, "Intel transactional synchronization extensions (intel tsx) asynchronous abort," Tech. Rep. CVE-2019-11135, 2019.

[41] N. Sönmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero, "unreadtvar: Extending haskell software transactional memory for performance." *Trends in Functional Programming*, vol. 8, 2007.

[42] T. Skare and C. Kozyrakis, "Early release: Friend or foe," in *Workshop on Transactional Memory Workloads*, vol. 77, 2006.

[43] Intel, "Intel®64 and ia-32 architecturessoftware developer's manual," Tech. Rep. Document Number: 252046-070, Dec, 2022 (online). [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[44] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 8–1, 2015.

[45] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier *et al.*, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 27–40.