Nova: Safe Off-Heap Memory Allocation and Reclamation

Ramy Fakhoury
Technion, Haifa, Israel

Anastasia Braginsky[1]
Red Hat Research, Ra'anana, Israel

Idit Keidar
Technion, Haifa, Israel

Yoav Zuriel
Technion, Haifa, Israel

Abstract

In recent years, we begin to see Java-based systems embrace off-heap allocation for their big data demands. As of today, these system rely on simple ad-hoc garbage-collection solutions, which restrict the usage of off-heap data. This paper introduces the abstraction of *safe off-heap memory allocation and reclamation (SOMAR)*, a thread-safe memory allocation and reclamation scheme for off-heap data in otherwise managed environments. SOMAR allows multi-threaded Java programs to use off-heap memory seamlessly. To realize this abstraction, we present Nova, *Novel Off-heap Versioned Allocator*, a lock-free SOMAR implementation. Our experiments show that Nova can be used to store off-heap data in Java data structures with better performance than ones managed by Java's automatic GC. We further integrate Nova into the open-source Oak concurrent map library, which allows Oak to reclaim keys while the data structure is being accessed.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases memory reclamation, concurrency, performance, off-heap allocation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.15

Supplementary Material *Software (nova open source code)*:
`https://github.com/li0nr/Nova-Safe-off-heap-memory-allocation-and-reclamation/`
archived at `swh:1:dir:f756390e070c9f32bbbfee2e514cdc5f141a591b`

1 Introduction

Programmers nowadays are accustomed to managed-memory environments as provided by today's most popular programming languages [12]. In such environments, memory is allocated from a region called *heap*, which is managed by the system, and is reclaimed by an automatic *garbage collection (GC)* algorithm. This paradigm is attractive as it reduces programming complexity, bugs, and memory leaks.

Today's leading GC solutions still struggle to scale with the volume of on-heap memory, especially in memory-hungry "big data" systems.

Figure 1 shows the throughput of Java's ConcurrentSkipListMap with a read-write workload (25% put, 25% delete, 50% get), where values are on heap managed by Java17's state-of-the-art GC solutions (star-square, diamond, square and circle shapes), compared to off-heap values managed by our solution (hexagram). Each experiment utilizes 16 threads. Of the machine's 128GB of DRAM, 110GB are allocated to Java's heap in the on-heap experiments, whereas in the off-heap experiment, 80GB are allocated off-heap and 30GB

[1] Work done before joining Red Hat Research.

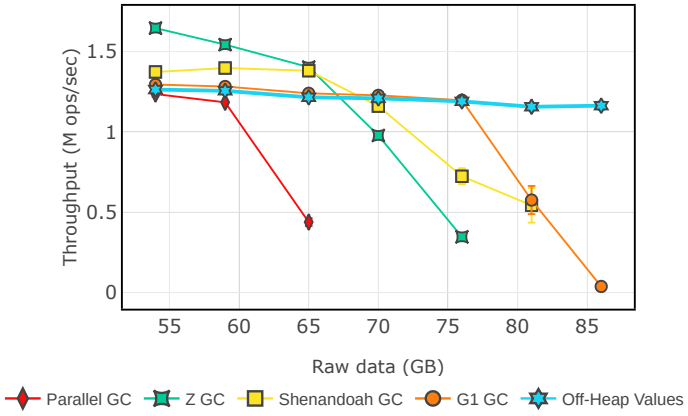


Figure 1 Performance of Java's ConcurrentSkipListMap with state-of-the-art Java17 GC implementations versus off-heap allocation managed by Nova.

on-heap (for more details see Section 4). The only on-heap GC implementation that can sustain 80GB of raw data is G1, and its performance degrades as the amount of ingested data increases, whereas our off-heap solution continues to perform well under memory duress. This conundrum has led Oracle to offer an off-heap allocation API in JDK as of version 14 [7, 22]. And indeed, a number of systems developed in Java now employ off-heap memory buffers alongside on-heap objects. For example, off-heap buffers are used to store in-memory tables in databases like Cassandra [14] and HBase [4, 20, 25]. Another example is the Druid analytics database, which uses off-heap buffers as temporary space for processing queries (e.g., large table merges) [13]. Nevertheless, these systems use the off-heap memory only in specific use cases where GC is straightforward. In particular, the off-heap buffer is reclaimed all at once – when an entire memory table is flushed to disk or when the query computation completes – and, by design, it is assured that all the threads that might have accessed the buffer have terminated. In essence, the off-heap memory usage in these use cases is *grow-only* as long as it might be referenced.

Reclaiming memory while data is still being accessed is challenging: Consider a memory location released by some thread in a system with concurrent access – we must prevent a situation where it is reclaimed and re-used while it is still being accessed by another thread.

The recently-developed Oak key-value map [27] pushes the use of off-heap allocation further: Oak is an open-source Java library that allows programmers to store keys and values off-heap, while managing meta-data on-heap. Oak currently uses a simple grow-only (release-at-once) memory manager for keys and a naïve lock-based memory manager for values, which locks off-heap objects on every access.

In this paper, we take off-heap memory management another step forward. In Section 2, we introduce the abstraction of *safe off-heap memory allocation and reclamation (SOMAR)*. Its API supports allocation, read/write access of allocated data, and deletion of previously allocated data. A SOMAR service reclaims and re-allocates off-heap data in a manner that does not restrict concurrent access: User code may freely copy off-heap objects and hold them for unknown periods. For example, multiple threads might retrieve the same data element from a map and then use it in lengthy computations. Similarly, an iterator might spend some time processing a map element before proceeding to the next one. Java threads do not generally "release" or "close" an object or iterator once it is no longer in use, and so a SOMAR implementation has no way of tracking the allocated objects' accurate reference

counts. Thus, depending on reference count for GC is not an option. A key challenge a SOMAR solution needs to address is ensuring *safety*, namely, detecting every unsafe attempt to access reclaimed (and re-used) memory. Providing such safe access facilitates a familiar programming experience in a managed environment.

In Section 3 we present *Nova*, Novel Off-heap Versioned Allocator, a lock-free SOMAR implementation. Nova may be used in lock-free data structures, as well as lock-based ones. In addition to safety and lock-freedom, Nova ensures *robustness* [34]– a stalled thread delays the reclamation of at most one data item. We keep the design simple so it is easy to implement and understand. In a nutshell, Nova uses version numbers to detect unsafe access. We use optimistic access [9] for read and a variant of hazard pointers [28] for writes.

We implement Nova in Java and use it to manage off-heap data indexed via on-heap data structures. Though no previous work provides the SOMAR functionality, we compare Nova to alternative SOMAR implementations using known reclamation approaches [15, 30] as well as to on-heap data structures managed by Java's automatic GC. Our results, reported in Section 4, show that SOMAR solutions outperform Java's GCs in all tested workloads. We further show that when primitive types are used to reference off-heap data (instead of Java objects), Nova outperforms all alternative SOMAR implementations, and otherwise performs similarly to them.

We further integrate Nova into the Oak open-source library and compare it to Oak's original grow-only and lock-based reclamation schemes. Our results show that the lock-based solution, when used also for keys, does not scale with more than one thread, whereas the Nova-based Oak scales linearly with the number of threads. The Nova-based Oak outperforms Java's ConcurrentSkipListMap and in fact performs as well as Oak with grow-only memory management (in which memory is not reclaimed at all).

It is important to note that SOMAR is different from *safe memory reclamation* (SMR) for C/C++ data structure libraries, which gained substantiated attention in recent years [3, 5, 6, 8, 9, 15, 16, 21, 23, 28–30, 32, 34] this because SMR assumes that the scope of a reference is limited to an explicitly protected part of the code, for example, the execution of the data structure operation. Thus, SMR solutions do not allow programs to *externally* access the memory they manage from outside the data structure library. This means, for example, that a map's get operation must return a copy rather than a reference to a data item that resides in the map. Additionally, iterators are not supported, as these continue to reference the data structure between calls. In contrast, SOMAR allows references to be used externally and to be copied freely. In Section 5, we go over related work in more detail.

In summary, we provide efficient memory management for off-heap data that can be used outside the scope of a given data structure, a functionality that is missing in today's managed environments. Section 6 concludes our paper. Correctness proofs are given in Appendix A. Appendix B includes an extension of Nova that mitigates fragmentation.

2 SOMAR

We introduce the abstraction of safe off-heap memory allocation and reclamation, SOMAR. SOMAR is intended for managed environments that support allocating unmanaged *off-heap* memory. Such allocation is natively supported in Java versions 14 and up. In new Java versions, it is possible to allocate off-heap memory using *memory segments* [22]. Such memory is not subject to GC and its access is not safe against races between memory access and delete-reuse.

SOMAR allows applications in a managed environment to allocate, use, and de-allocate off-heap memory in a safe manner. The allocated unit is called a *slice*. Applications can use their allocated slices for storing data, for instance, the keys and values of a key-value map. An application may invoke a slice delete operation, allowing SOMAR to reclaim its memory for reuse in future allocations.

Slices are accessed via on-heap *safe-pointers*. Applications can store safe-pointers in data structures as well as in ephemeral objects and can copy them without informing SOMAR. Thus, multiple safe-pointers may address the same slice, and the service has no way of counting how many safe-pointers reference a given slice. User code might invoke a delete operation on a slice via one safe-pointer, while additional safe-pointers continue to reference the now deleted slice. Importantly, SOMAR ensures safety in case the slice's memory is reused: all attempts to access slices that have been deleted and reallocated result in errors. In other words, SOMAR protects against access-delete-reuse races.

Reading and writing slice data is done using user-defined functions (lambdas) that operate directly on off-heap memory. We assume that the lambda functions have no side effects beyond accessing the slice, and the function passed to read operations does not update the underlying memory, moreover we assume that lambdas do not fail upon reading illegal values (e.g., infinite loop, dividing by zero). For simplicity, we assume that there are no nested calls to the safe-pointer from within lambda functions, though it is not difficult to relax this assumption, as explained below.

SOMAR supports the following API:

- *safe-pointer allocateSlice(length)* allocates a slice of the requested length;
- *buf read(safe-pointer ptr, lambda f())* allows the application to read the slice using f;
- *buf write(safe-pointer ptr, lambda f(), param))* allows the application to write to the slice using f;
- *boolean delete(safe-pointer ptr)* de-allocates the slice associated with the safe-pointer.

The methods may fail upon attempting to access deleted data.

Privatization. Ensuring safety in the presence of potential concurrent deletions incurs a synchronization cost. *Privatization* [24, 33] is a technique for reducing this cost when it is known that a slice is not accessible by more than one thread. Consider, for example, a thread that allocates a new slice for inserting a new element into a shared data structure. The thread writes its data into the slice when it is still *private*, namely inaccessible to other threads. In this case, the write does not need to be protected from concurrent deletions of the same slice. Similarly, if a thread allocates a slice and then fails to insert it into a shared data structure and deletes it, the delete occurs when the slice is still private and so concurrency races are not possible. To support unprotected operations for private slices, we add the following methods to the API:

- *writePrivate(safe-pointer ptr, lambda f(), param)* ;
- *deletePrivate(safe-pointer ptr)* ;

It is the responsibility of the programmer to call the private methods only when it is ensured that the slice is private.

3 Algorithm

Nova is an algorithm for implementing SOMAR. We give a high level view of Nova in Section 3.1 and describe its data layout in Section 3.2. In Section 3.3, we present the algorithm's pseudocode, while abstracting away the memory model. Implementing the algorithm in weak-memory models is discussed in Section 3.4. Correctness is discussed in Appendix A.

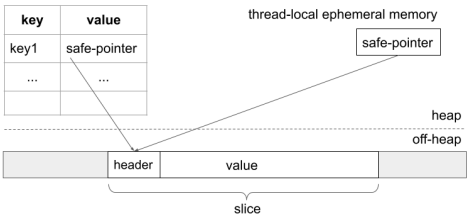


Figure 2 Example: A Nova slice with two safe-pointers.

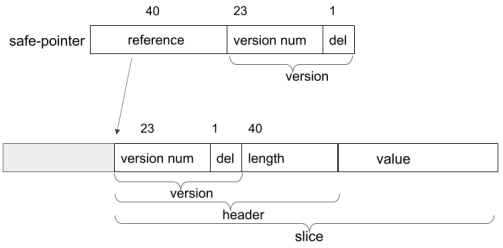


Figure 3 An example layout of Nova headers and safe-pointers.

3.1 Overview

As an implementation of SOMAR, Nova allocates slices within a large memory area, which it requests from the OS via the JDK and never released. It uses on-heap safe-pointers to refer to slices. User code can copy these safe-pointers freely without informing Nova, and might delete a slice when there exist additional safe-pointers addressing it. A key challenge we address, therefore, is detecting access to obsolete slices via old safe-pointers. Nova must ensure that every access via a safe-pointer to a slice that had been deleted and subsequently reallocated results in a failure. In other words, it must prevent access to the memory location that used to pertain to that slice before its deletion and now pertains to another. To this end, each slice has an off-heap *header*, which is used to track its validity as we explain next. A slice's header is unique while multiple on-heap safe-pointers may refer to the same header.

For simplicity, we associate each slice with a static header. Whenever a slice is re-used, the header's location does not change. We place the header right before the slice. This approach is simple, efficient, and avoids indirection. However, it also restricts future allocations and may thus lead to fragmentation when using variable allocation sizes. In many cases, this can be a reasonable compromise: It is cheaper than automatic GC, which relocates data to avoid fragmentation. And it allows released memory to be reused whenever allocation sizes recur, for example, in a map with fixed-size keys. This is in contrast to existing Java-based systems that use grow-only off-heap memory without reusing *any* memory as long as some allocated data might be referenced [4, 13, 14, 20, 25]. For applications with frequent allocation and de-allocation with variable allocation sizes, we present in the Appendix B an approach for mitigating fragmentation at the cost of a fixed memory overhead.

Figure 2 illustrates a Nova slice allocated within a block and two safe-pointers pointing to the slice's header. The slice holds a value stored in a key-value map, and the safe-pointer depicted on the left is an entry in the key-value map. The safe-pointer on the right is ephemeral – it has been retrieved from the map by some thread.

The header includes a *deleted* bit, which is set when the slice is deleted, and the slice's *length*. The deleted bit is also stored in the safe-pointer. Every attempt to access the slice checks the deleted bit in the safe-pointer and fails if it is set. As long as a deleted slice is not re-used, these bits suffice for checking its validity, because the slice remains deleted and so the deleted bit remains set.

To discover that a deleted slice has been reallocated, we employ *versions*, which are stored in both headers and safe-pointers, and change whenever slices are reallocated. Attempts to access the slice verify that the versions in the safe-pointer and in the header are the same.

While the header and safe-pointer suffice to ensure safe access in a sequential execution, an additional mechanism is needed to ensure correct concurrent access. Consider a slice deleted by some thread – we must prevent a situation whereby the slice is reclaimed and re-used while it is still being accessed by another thread.

One simple way to address such concurrency races is using a per-slice read-write lock. We forgo this approach for two reasons. First, we have designed Nova to be lock-free so that it may be used as a building block in lock-free applications, e.g., concurrent lock-free data-structures. Second, using locks can degrade performance, as we show in Section 4 when comparing a Nova-based implementation of Oak to Oak's lock-based reclamation mechanism.

Instead, we opt for a lock-free solution using a combination of *optimistic reads* [9] and *hazard pointers* [28] for writes, whereby writing threads announce which slices they are accessing to prevent these slices from being reclaimed. As every access to a slice's data is via a safe-pointer, the scope of each slice access is limited to the execution of one safe-pointer method. Our assumption that safe-pointer calls are not nested allows us to keep one hazard pointer per writing thread. (It is straightforward to extend the solution to allow nesting of safe-pointer calls by keeping multiple hazard pointers.) For reads, we improve performance by eliminating hazard pointers altogether. Rather, reads are allowed to temporarily observe illegal values, but such situation are detected (via the version) and result in failure.

3.2 Data structures and layout

Both the safe-pointer and the header hold a version, which includes a version number *num* and a *deleted* bit. Versions are assigned using a global monotonically (infrequently) increasing *NovaEra* counter, an atomic integer initialized to 1. For simplicity, we assume that the counter does not wrap around. In practice, it is acceptable for it to wrap around as long as it takes sufficiently long to repeat the same counter value so that no safe-pointers holding the version used in its previous incarnation exist in the system. Slices are addressed via *references*. Each safe-pointer holds a version and a *reference*. An example safe-pointer layout is illustrated in Figure 3. In this example, the reference consists of 40 bits, which can address up to 1TB of memory.

In our implementation, the entire safe-pointer occupies a single (long) memory word, and so can be stored in a primitive long type rather than a Java object. Furthermore, its reference and version may be updated atomically together via a single CAS instruction. Using bigger safe-pointers is possible, but would degrade performance as (1) they would need to be stored in Java objects; and (2) since as of today, Java does not support multi-word CAS, we would need to use a user-level multi-word CAS solution [18]. Note that the length field in the header cannot be larger than the offset, and therefore the header also fits into one memory word that can be updated using a single CAS instruction.

Because multiple safe-pointers may reference the same slice, the off-heap header is always the source of truth regarding the version and deletion state. To this end, on slice deletion, the header is updated before the safe-pointer.

Nova uses a number of shared data structures, as summarized in Table 1. To manage hazard pointers, it holds a dedicated *Thread Array of Pointers (TAP)*, with references to slices accessed by all ongoing writes. The TAP has one slot per thread, holding a reference to the slice that the thread is writing to (if any).

New slices are allocated either from unused memory or from a *free list* holding available (free) slices. The free list is managed on-heap, as a skiplist sorted by size. When a slice is deleted, it is not immediately added to the free list. This is because the slice should not be reallocated as long as it might be written by a concurrent thread. Instead, a deleted slice is temporarily added to a *release list*, which is managed as an on-heap array. To avoid contention, each thread manages its own release list, and infrequently (e.g., after adding 1000 items to the release list) promotes eligible slices from its release list to the shared free list. To ensure robustness, threads also periodically (but infrequently) check if there are other

Table 1 Shared data structures used by Nova.

Data Structure	Access Pattern
NovaEra	shared, updated using atomic increment
TAP	each entry written by one, read by all
free list	shared by all threads
release lists	each owned by one thread, accessible to helping threads

threads whose release lists are full, and if so, *help* them by promoting items from their release list to the free list. Before adding a slice to the free list, the de-allocation process verifies that no TAP entries refer to it. In addition, to ensure that reallocation uses a different version number, the NovaEra is incremented before slices are migrated to the free list. We increment NovaEra using an atomic increment instruction to address races among concurrent threads.

3.3 Nova algorithm

We now detail how Nova's API is supported. For brevity, we describe the algorithm assuming a sequentially consistent memory model, and defer the discussion on its correctness in other memory models to Section 3.4.

Algorithm 1 Nova safe-pointer read and write methods.

```
1:  procedure READ(ptr, f)
2:      ⟨ver, slice, header⟩ ← locateSlice(ptr)
3:      if slice = ⊥ then
4:          return ⊥
5:      ret ← f(slice)                                   ▷ read from slice
6:      if header.ver ≠ ver then                          ▷ validate
7:          return ⊥
8:      return ret
9:  procedure WRITE(ptr, f, param)
10:     set hazard pointer in TAP to ptr
11:     ⟨ver, slice, header⟩ ← locateSlice(ptr)
12:     if slice = ⊥ ∨ header.ver ≠ ver then
13:         set hazard pointer in TAP to ⊥
14:         return ⊥
15:     ret ← f(slice, param)                             ▷ write to slice
16:     set hazard pointer in TAP to ⊥
17:     return ret
18: procedure LOCATESLICE(ptr)          ▷ returns ⟨version, slice, header⟩ triple.
    ▷ version includes version num and deleted bit
19:     myVer ← ptr.ver
20:     if myVer.deleted then return ⟨⊥, ⊥, ⊥⟩
21:     header ← address referred to by ptr.ref
22:     slice ← address of header's slice (header + 8)
23:     return ⟨myVer, slice, header⟩
```

Reading and writing. Pseudocode for Nova's reads and writes appears in Algorithm 1. Both operations use the safe-pointer's locateSlice method in order to extract from the safe-pointer the reference to the accessed slice and its header, as well as its version. Reading and writing occur, via the user-provided lambda function f, and return value is the user-defined return value of f.

In addition, every operation performs *validation* by comparing the version in the safe-pointer to the one in the header (along with the deleted bit) and returns failure in case the version is no longer valid. To allow NovaCounter to wrap-around, we only check whether the versions are equal (As noted above, we assume that the cycle is long enough to ensure that by the time a version is re-used, no obsolete safe-pointers holding its value from previous incarnations exist). Reads are optimistic, performing validation after the actual read (Line 6), whereas writes perform validation before the actual write (Line 12).

In both cases, if the versions (and deleted bits) do not match, it means that the slice has been deleted (and possibly subsequently reallocated), and the operation fails. As an optimization – to expedite future operations, mismatch detection – it is possible to mark the safe-pointer's version as deleted in such cases (using a CAS). This optimization is omitted from the code as it has no effect on correctness.

To synchronize with concurrent deletion and allocation operations, writes proceed through TAP: a writing thread first writes the accessed reference to its TAP entry. As long as the write is ongoing, the TAP entry holds its reference, ensuring that the slice is not retired into the free list. When writing is done, the writing thread sets its TAP entry to $\perp$, allowing the slice to be garbage collected (this write can be lazy because it is not required for correctness).

To make reads lightweight, we allow them to proceed without accessing TAP. Thus, nothing prevents a read slice from being deleted and reallocated before the read operation ends. This means that validation must occur after the value is read; ensuring this execution order in a weak memory architecture is discussed in the next section.

The pseudocode of the privatized write method is not detailed, as it simply applies the given lambda function to the slice.

Slice allocation and deletions. Algorithm 2 describes Nova's allocation and deletion methods. A new slice is obtained either from unused memory or from the free list. We assume that slices are obtained atomically, namely, each slice is assigned to a single thread. We initialize the new slice's header with a version num from the current NovaEra, the deleted bit unset (Line 28), and the slices's length. The off-heap header is the source of truth regarding the slice's allocation, and so once it is set, the slice is considered as allocated. The method returns a safe-pointer with a reference to the slice's header and the new version.

After locating the slice, a delete operation updates the delete bit in the header and the safe-pointer to true. The header is updated first, using a CAS, so all races with concurrent deletions are resolved by the order in which the CASes are scheduled (only the first succeeds). Note that in case delete is called for an already deleted slice, the CAS keeps it unchanged, and delete returns false. If the CAS succeeds, we add the deleted slice to the thread's release list. Either way, the safe-pointer's delete bit is set. The privatized delete method is the same except in that it updates the header without a CAS and always succeeds. If additional safe-pointers refer to the same slice, the deleted bits in these safe-pointers remain unset, but future attempts to access the slice via these safe-pointers detect the mismatch: reads concurrent with a deletions detect the header change when checking the version at the end of the read. While, in the case of a write being concurrent with a deletion, the slice is registered in the writer's TAP, preventing the slice's garbage collection. Nova does not determine the

Algorithm 2 Nova's allocation and deletion methods.

```
24: procedure ALLOCATESLICE(length)
    ▷ atomically get new slice of size length
25:     newSlice ← get reference to new slice header
26:     newPtr ← new safe-pointer
27:     newPtr.ref ← newSlice
28:     newPtr.ver ← ⟨NovaEra, false⟩
29:     newSlice.ver ← newPtr.ver
30:     newSlice.length ← length
31:     return newPtr
32: procedure DELETE(ptr)
33:     ⟨ver, slice, header⟩ ← locateSlice(ptr)
34:     if slice = ⊥ then return false
35:     len ← header.len
36:     flag ← CAS ( ⟨header.ver, header.length⟩, ⟨ver, len⟩,
                     ⟨⟨ver.num, true⟩, len⟩ )
37:     ptr.ver.delete ← true
38:     if flag then
39:         add slice reference to (thread-local) release list
40:     return flag
41: procedure RETIRE
42:     atomically increment NovaCounter
43:     for all ref in (thread-local) release list do
44:         if no TAP entry contains ref then
45:             append ref to (shared) free list
46:             remove ref from release list
```

execution order between concurrent deletes and writes of the same slice. Rather, it ensures that a deleted slice is not reallocated as long as ongoing writes can access it and that no future reads or writes that begin after the deletion is complete successfully access the slice.

Each thread periodically calls the retire procedure (Algorithm 2, Lines 41–46) to move slices from its release list to the shared free list. It increases the NovaVersion and then migrates all slices that have no pending accesses registered in TAP. This ensures that whenever a slice is reallocated, it is assigned a different version than it had before. We omit a helping mechanism from the pseudocode, whereby threads retire eligible slices from other threads' full release lists to achieve robustness.

3.4 Synchronization over weak memory models

For simplicity, we presented Nova's pseudocode above assuming the memory is sequentially consistent. We now identify the points in our pseudocode where synchronization instructions must be used for correct execution on x86-TSO. To ensure that a read operation checks the version *after* reading the data, it issues a load fence after Line 5. All other steps of the read operation can be safely reordered. A write must guarantee that the TAP entry is set before the version check that precedes the actual write. This is ensured by adding a full fence after the TAP update in Line 10. In addition, a store fence is required after Line 15, before setting the TAP entry to $\perp$, to ensure that the $\perp$ is not visible before the write.

A delete operation needs to set the delete bit in the off-heap slice before modifying the safe-pointer. This update is already done using a CAS instruction in order to deal with potential races among deleting threads Line 36. The retire procedure must ensure that the incremented NovaEra is visible before adding slices to the free list, so that older versions will not be used in new allocations of these slices. This is ensured by using an atomic increment instruction to update the counter.

4 Evaluation

We now evaluate Nova and two other SOMAR implementations. In Section 4.1 we use SOMAR to manage off-heap data organized in Java data structures and in Section 4.2 we integrate Nova into the Oak library.

Experiment setup. All the code is written in Java 17 and built with Apache Maven 3.8. In Section 1, we compared four state-of-the-art Java GC solutions. Here, we experiment with – G1, which is best under memory stress according to our experiments, along with the latest GC releases ZGC and Shenandoah, which show the best performance when ample memory is available, in our experiments. We used a customized synchrobench tool [17]. For each data point, we start a new JVM, warm it up for one test, and then repeat the test three times and plot the average.

Experiments were run on an AWS instance c5ad.16xlarge, AMD EPYC 7R32 with 128GB RAM. To check whether there were any platform-specific effects, we repeated some of the experiments on ARM architecture. The trends were similar and the results presented in the Appendix C. All experiments utilize 32 cores (without hyper-threading) on one NUMA node and the OS is Ubuntu 20.04.

4.1 Off-heap data indexed by Java data structures

Benchmarks. Our experiments highlight two different scenarios where off-heap allocation can be beneficial. The first is a user-defined data structure that can store primitive types, reducing GC overhead and eliminating a level of indirection in referencing actual data. The second is a big data scenario, where the system runs under memory stress. Specifically, we use SOMAR to manage off-heap data organized in the following on-heap Java data structures:

LL-map A key-value map using Harris's linked list [19] with on-heap nodes referencing keys and values off-heap (using safe-pointers or Java objects).

CSLM Java's ConcurrentSkipListMap, where keys are on-heap, and values are Java objects referencing off-heap slices. (CSLM does not support primitive types.)

In the LL-map, a put overwrites the slice data if the key is present, and creates a new slice otherwise. In the latter case, the privatized write is used. If the insertion fails (due to a race), the slice is then deleted using the privatized delete method. Because CSLM values are Java objects, every put allocates a new slice and safe-pointer and deletes the old one if it exists. Hence here, all writes are privatized.

In both data structures, we experiment with write-heavy (50% put 50% delete), read-heavy (5% put 5% delete 90% get), and read-write (25% put 25% delete 50% get) workloads. Each test lasts 30 seconds.

Because the linked list has a linear search time, the LL-map size cannot scale, so we keep it to ∼100MB. We initialize it with ∼65K entries holding 512 byte keys and 1KB values. We experimented with other key and value sizes (both larger and smaller), and the trends were similar. Keys are drawn uniformly at random from a sub-range consisting of ∼130K keys, to allow roughly 50% of the deletions to succeed and 50% of the insertions to insert new values.

In order to keep the CSLM experiments tractable, we run them with ∼10GB of raw data and scale down the available RAM budget accordingly. To this end, the CSLM is initialized with 10 million entries holding 128 byte keys and 1KB values (Here too, experiments with different key and value sizes with the number of entries scaled to consume the same overall memory size showed similar trends). Keys are drawn from a sub-range consisting of 20 million keys. The total RAM budget is 14GB. For off-heap (SOMAR) solutions, this is split into 4GB on-heap and 10GB off-heap. Since the LL-map size does not scale, this data structure is not intended for use under memory stress. Hence, we do not limit the available memory in LL-map experiments.

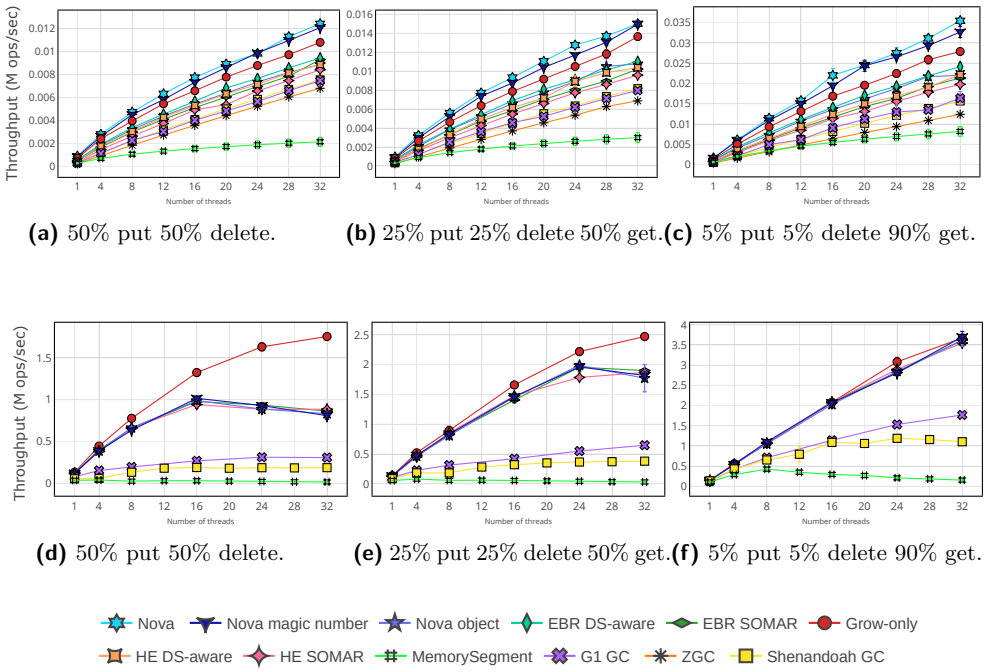
Compared solutions. While no previous work has implemented SOMAR, we can implement a similar service by adapting known safe memory reclamation techniques to work in our managed environment. We chose to compare Nova to *Epoch-Based Reclamation (EBR)* [15] and *Hazard Eras (HE)* [30]. We also experimented with Interval-Based Reclamation [34] in some workloads and got similar results to EBR, so we refrained from a detailed study of this approach.

To use EBR and HE in our context, we need to reference each piece of off-heap data (key/value) with an on-heap representation similar to Nova's safe-pointer. Whereas the safe-pointer can be implemented as a primitive long, the data required by EBR and HE does not fit into one word, and so we use Java objects. As noted above, in the CSLM experiment we must use Java objects also for Nova's safe-pointers. To evaluate the impact of using objects, we run the LL-map experiments with Nova safe-pointers implemented both as primitive longs and as objects.

Additionally, we need to explicitly specify the scope of the safe access, i.e., to wrap all accesses to off-heap data with explicit *protect* and *un-protect* calls. We do this in two ways: (1) The SOMAR approach, where the off-heap representation wraps each access to an off-heap data item, in order to comply with our API. This way, at every node in a linked list traversal, we call protect before reading the key and un-protect afterward. (2) A *data-structure-aware (DS-aware)* approach, where protect is called at the beginning of the data structure operation and un-protect is called at the end. In the latter case, the protected region spans the entire linked list traversal, which accesses all keys along the way. For Nova, we use only the SOMAR approach, as Nova was explicitly designed for SOMAR. Finally, we implement a variant of Nova that reduces fragmentation at the cost of some memory overhead; this technique is called *magic numbers*, and for space limitations, is deferred to Appendix B.

We compare the three SOMAR approaches (Nova, EBR, and HE) to five baselines: (1) on-heap allocation with Java17's default G1 GC; on-heap allocation with Java17's newly introduced (2) ZGC; (3) Shenandoah GC; (4) raw Java memory segments [22] for fine-grain allocations; and (5) grow-only off-heap memory without reclamation with safe-pointers implemented as Java objects holding the length and offset of the off-heap data.

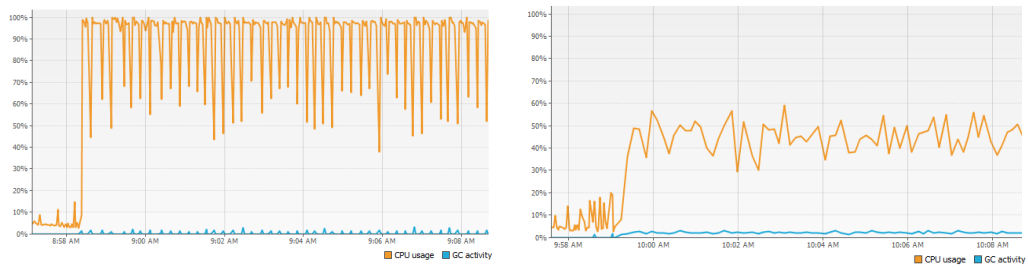
LL-map results. Figure 4(a)–(c) depicts the throughput results for the LL-map. We see that all SOMAR solutions outperform Java's GC solutions and memory segments across all workloads. For memory segments, this is expected since they are not designed for fine-grain allocation scenarios. In these experiments, memory stress is not a factor, and GC activity is a performance bottleneck. We presume that this occurs due to the larger heap space that Java needs to manage. Using a profiler, we learned that the GC (fully on-heap) solution has 7 time more GC activity than Nova and 3x the allocation-churn rate of Nova. Moreover, Nova has the lowest GC time and the lowest allocation churn rate among all off-heap solutions.



(a) 50% put 50% delete. **(b)** 25% put 25% delete 50% get. **(c)** 5% put 5% delete 90% get.

(d) 50% put 50% delete. **(e)** 25% put 25% delete 50% get. **(f)** 5% put 5% delete 90% get.

Figure 4 LL-map (top-row) and CSLM (bottom-row) throughput (Mops/sec) versus number of threads.



(a) Nova CSLM with off-heap data. **(b)** On-heap Java CSLM.

Figure 5 CPU usage and GC activity in the CSLM experiment measured by VisualVM.

Nova achieves the best performance among SOMAR solutions. Its peak throughput is more than 2x that of the Java GC in all workloads, and it outperforms all other off-heap solutions by 20–30%. It even performs better than the grow-only solution that does not reclaim memory at all. Part of the benefit can be attributed to implementing the safe-pointer as a long, saving a level of indirection. When Nova uses object safe-pointers it performs similarly – or only slightly better than – alternative SOMAR solutions, and slightly (up to 10%) worse than the DS-aware HE and EBR implementations. The SOMAR solutions of EBR and HE perform worse than their DS-aware counterparts because they require many protect/un-protect calls during a list traversal whence many (off-heap) keys are read. Finally, magic numbers have a minimal performance impact.

CSLM results. The CSLM results are shown in Figure 4(d)–(f). In all these experiments, ZGC crashes with an OutOfMemoryError exception when allocated the same DRAM budget as other solutions and hence is not shown in the graphs. The Java on-heap GC solutions

Table 2 Memory consumption in GB, CSLM, read-write workload, 32 threads.

	Nova object	Nova magic	EBR	HE	memory segments	G1 GC
on-heap	2.8	2.8	2.9	3.0	3.2	13.0
off-heap	9.6	9.7	9.6	9.6	9.5	0

and memory segments fall far behind the off-heap solutions. At the same time, all SOMAR variants behave similarly (less than 1% difference). This is expected since the heavy work in a CSLM (searching) involves only keys, which are managed on-heap in all variants. By moving the (large) values off-heap, we reduce the load on the GC and improve overall performance. We also experimented with off-heap keys, but because we cannot store primitive types in a CSLM, this necessitated another level of indirection on every key access, deteriorating performance. This tradeoff between GC cost and indirection cost underscores the benefit of allowing programmers to use SOMAR alongside on-heap allocation in a managed environment. The cost of ensuring safe access is manifested in the difference between SOMAR and the grow-only solution, which is significant in updates and negligible in reads. Moreover, in the results shown in Figure 4, we can notice a large gap between the grow-only solution and the pure on-heap approaches, this gap depict the tradeoff between pure-on-heap approaches and leaky-off-heap approaches.

We use a profiler to assess the impact of GC. Figure 5 presents a timeline of CPU utilization and GC activity during the 32 thread experiment. Nova (Figure 5a) incurs periodic GC events (for managing the on-heap keys), which reduce its CPU utilization for short periods. In contrast, because this experiment runs under memory duress, the on-heap solution (Figure 5b) constantly engages in GC activity and its CPU utilization is constantly around 50%. This explains why the GC throughput is roughly half the off-heap solutions in this experiment.

Table 2 summarizes memory utilization of all solutions (except grow-only) in the read-write workload with 32 threads. We measure the memory consumption before the benchmark run (after initialization) and after it. The memory consumption of all algorithms was similar at both times. Note that GC requires slightly more total memory than the off-heap approaches, and that the overhead of Nova's magic numbers is small. Fragmentation does not occur in this experiment because all allocations are of the same size.

4.2 Nova in Oak

We integrated Nova into the Oak library [27] code.[2] Oak is designed to reference off-heap data using a primitive long, which we saw is beneficial in our LL-map experiment above. Because Oak stores primitive longs in the data structure, other SOMAR solutions are inapplicable. Oak currently offers two memory management options – (1) a grow-only manager that cannot reclaim memory concurrently with data access, and (2) a lock-based manager that uses locks to handle concurrency between data access and deletion.

We experiment with Oak using the following memory manager combinations:

Unreleased keys grow-only keys, lock-based values;
Grow-only grow-only for both keys and values;
Lock-based lock-based for both keys and values;

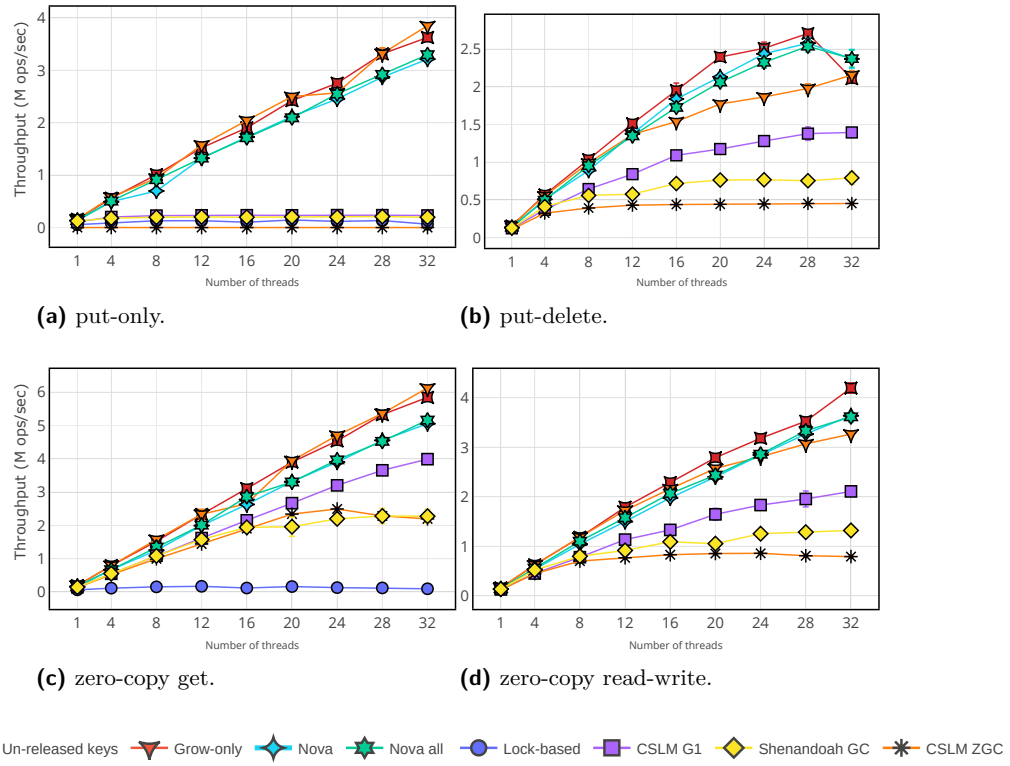
[2] `https://github.com/yahoo/Oak/pull/200`

Table 3 Memory consumption in GB, Oak read-write workload, 16 threads.

		Oak					CSLM	
		grow-only	unreleased keys	lock-based	Nova keys	Nova all	G1	ZGC
after init	on-heap	0.4	0.4	0.4	0.4	0.4	12.4	14.8
	off-heap	10.3	10.4	10.5	10.5	10.4	0	0
after run	on-heap	0.5	0.4	0.4	0.4	0.4	12.4	15.4
	off-heap	43.1	16.1	10.5	11.0	11.0	0	0

Nova keys Nova for keys, lock-based values;
Nova all Nova for keys and values.
We compare all of these to Java's CSLM, G1 GC, ZGC, and Shenandoah GC. All solutions run with a memory budget of 17GB. The off-heap solutions split this budget to 6GB on-heap and the rest off-heap. The grow-only variation has no memory restriction.

Our benchmarks generally follow the ones used in Oak's evaluation [27]. The map consists of 10 million key-value pairs with 100 byte keys and 1000 byte values, and the workloads are (a) put-only; (b) put-delete; (c) Oak's zero-copy get, where the application accesses the off-heap buffer directly using a lambda function; (d) a read-write workload with zero-copy gets. We ran the experiments fro 5 minutes, to demonstrate the effect of grow-only approach.



(a) put-only. **(b)** put-delete.

(c) zero-copy get. **(d)** zero-copy read-write.

Figure 6 Map throughput (Mops/sec): Oak with different reclamation schemes vs. CSLM.

The throughput results appear in Figure 6 and the memory consumption in Table 3. We see that the lock-based memory manager does not scale due to lock contention. Not surprisingly, the solutions that do not release keys (grow-only and unreleased-keys) are the fastest, since neither deals with concurrent access to keys where most of the data structure's work is done. Somewhat surprisingly, the remaining solutions, which have much smaller memory footprints, are no more than 7% slower than the grow-only solution. Moreover,

Nova performs similarly to grow-only in most workloads. We note that the small dip in performance in Figure 6b is duo to thread contention, as many threads compete to obtain a new slice from the allocator.

We see a small increase in Nova's memory usage over time, which stems from internal fragmentation in Oak data structures. We conclude that Nova offers a viable reclamation solution for Oak, allowing it to reclaim memory used for keys concurrently while accessing tha data-structure.

5 Related Work

SOMAR versus SMR. In recent years, many efforts have been dedicated to designing safe memory reclamation for concurrent programs in environments with manual garbage collection, most notably C/C++. Most of these works focus on concurrent *data structures (DSs)* [3, 5, 6, 8, 9, 15, 16, 21, 23, 28–30, 32, 34], and are designed to support safe deletion of DS elements. The main idea is to provide safe access guaranteeing that the accessed data is not concurrently reclaimed. For instance, in a linked listed traversal, protected access ensures that as long as the traversal holds a pointer to some node in the list, that node's memory is not reclaimed.

Typically, the scope of safe access is the DS operation, and data stored in the DS is externally inaccessible. Thus, retrieval operations cannot return pointers, and must instead copy the data, which is costly when data items are large. In addition, this scheme does not accommodate iterators, as they continue to hold a pointer into the DS after a getNext call returns. SOMAR does not limit the scope of the safe access, and instead allows memory to be reclaimed while there are still live references to it. In contrast SOMAR ensures that unsafe attempts to access data after it has been reclaimed fail. This approach is suitable for optimistic concurrency control, where computations may fail and need to be retried. Also, SOMAR is designed for a managed environment where only some data is off-heap, for instance, only the keys and values stored in a map and not the indexing pointers.

Techniques. Although the aforementioned works address a different problem, some do employ similar techniques. The main techniques used in memory reclamation are *reference counting*, *hazard pointers (HP)*, and *epoch-based reclamation (EBR)*. Modern schemes use different combinations of these, and some also rely on OS signals in order to allow progress in the presence of stalled threads.

Reference counting is used for lock-free reclamation in Hyaline [29], and is also used for automatic reclamation in OrcGC and DRC [2, 10]. As explained above, Nova is not amenable to reference counting because user code may copy safe-pointers without informing Nova.

Hazard pointers [11, 21, 28] and their variants [16] work as follows: before accessing data, a thread announces the pointer to the data is about to access, makes this announcement globally visible (using a CAS or fence), and then verifies that the pointer it is about to access has not been deleted from the data structure. The latter is tricky, requiring significant programmer effort and incurring high overhead (potentially a repeated DS traversal), as discussed in [6]. Nova's TAP entries are essentially HPs. Yet Nova eliminates the major difficulty (and source of overhead) in HPs as it does not rely on the DS to check whether the accessed slice had been deleted. Rather, we rely on versions (and the delete bit) to detect deletion. In addition, unlike data structures in manual GC environments, Nova HPs are required only for actual data updates and not for meta-data traversal (e.g., linked list pointers) and so one HP per thread suffices. This allows us to store a single TAP array with

entries for all threads, supporting efficient scans. Originally, HPs were used both for read and write access, but Nova follows the approach in Optimistic Access [8, 9, 31] and uses them only for writes while reads proceed optimistically.

Epoch-based reclamation [15, 26] was introduced in order to avoid synchronizing on every access as in HP. A number of variants of this approach exist in the literature, e.g., Debra/Debra+ [6], NBR/NBR+ [32], and ThreadScan [1]. Indeed, EBR has been shown to achieve better performance than HP by reducing the synchronization cost. Under this approach, threads share a global epoch and an announcements array. A thread wishing to access the data structure must declare its intent by publishing the current epoch in its entry in the announcements array. A block can be reclaimed if it was deleted from the DS before the lowest epoch in the announcements array. The disadvantage of EBR is that it delays reclamation and is not robust – a single stalled thread prevents reclaiming entire epochs (including memory it did not access). In some cases, OS signals are used to address such stalls [6, 32], which is not readily applicable in a JVM environment. Other solutions combine HP and EBR to improve EBR's robustness [3, 5, 23, 30, 34]. We looked into using epochs in Nova in order avoid a fence in the main execution path, but we found that in our setting, the performance penalty of fences was offset by the delay in reclamation introduced by epochs. Hence, we decided to forgo this solution.

6 Conclusions

The push for big data processing increases memory stress, while automatic GC solutions cannot always keep up with this growth. In recent years, we see a clear trend towards off-heap allocation of large chunks of data alongside the managed programming experience for other data. To facilitate this paradigm, we presented SOMAR, a safe memory allocation and reclamation scheme for off-heap data. SOMAR allows multi-threaded Java programs to use off-heap memory seamlessly, freely passing references to such data between threads. We presented Nova, an efficient lock-free SOMAR implementation. We used Nova for managing off-heap data organized in Java data structures, and showed that it performs better than fully on-heap solutions and state-of-the-art safe reclamation alternatives. We further integrated Nova into the open-source Oak concurrent map library, providing efficient support for reclamation of off-heap keys while the map is being accessed, a functionality that is not readily available today.

References

1 Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 123–132, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2755573.2755600.

2 Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 526–541, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454060.

3 Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935764.2935790.

4 Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better memory organization for LSM key-value stores. *PVLDB*, 11(12):1863–1875, 2018. doi:10.14778/3229863.3229873.

5 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486159.2486184.

6 Trevor Brown. Reclaiming memory for lock-free data structures: there has to be a better way, 2017. arXiv:1712.01044.

7 Maurizio Cimadamore. Bytebuffers are dead, long live bytebuffers! https://www.youtube.com/watch?v=RYrk4wvar6g&t=172s&ab_channel=FOSDEM, 2020.

8 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *SIGPLAN Not.*, 50(10):260–279, oct 2015. doi:10.1145/2858965.2814298.

9 Nachshon Cohen and Erez 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*, SPAA '15, pages 254–263, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2755573.2755579.

10 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: Automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 205–218, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437801.3441596.

11 Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 36–45, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2926697.2926699.

12 Larry Dignan. The most popular programming languages and where to learn them. https://www.zdnet.com/article/best-programming-language/, may 2021.

13 Docs Druid. Basic cluster tuning. https://druid.apache.org/docs/latest/operations/basic-cluster-tuning.html, retrived in April, 2022.

14 Jonathan Ellis. Off-heap memtables in cassandra 2.1. https://www.datastax.com/blog/heap-memtables-cassandra-21, 2014.

15 Keir Fraser. Practical lock-freedom. technical report ucam-cltr-579. In *Technical Report, UCAM-CL-TR-579 ISSN 1476-2986*, 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

16 Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009. doi:10.1109/TPDS.2008.167.

17 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2688500.2688501.

18 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap, 2020. arXiv:2008.02527.

19 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. doi:10.1007/3-540-45414-4_21.

20 Reference Guide Hbase. Hbase offheap read/write path. https://hbase.apache.org/book.html#offheap_read_write, retrived in April, 2022.

21 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, may 2005. doi:10.1145/1062247.1062249.

22 Docs Java. Interface memorysegment. https://docs.oracle.com/en/java/javase/17/docs/api/jdk.incubator.foreign/jdk/incubator/foreign/MemorySegment.html, 2021.

23 Jeehoon Kang and Jaehwang Jung. A marriage of pointer- and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 314–328, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385978.

24 Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. Safe privatization in transactional memory. *SIGPLAN Not.*, 53(1):233–245, feb 2018. doi:10.1145/3200691.3178505.

25 Yu Li, Yu Sun, Anoop Sam John, and Ramkrishna S Vasudevan. Offheap read-path in production the Alibaba story. https://blog.cloudera.com/blog/2017/03/, 2017.

26 Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, 1998.

27 Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. Oak: A scalable off-heap allocated key-value map. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 17–31, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374526.

28 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, jun 2004. doi:10.1109/TPDS.2004.8.

29 Ruslan Nikolaev and Binoy 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*, PLDI 2021, pages 987–1002, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454090.

30 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras - non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 367–369, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087556.3087588.

31 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 443–445, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3409964.3461817.

32 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: Neutralization based reclamation, 2020. arXiv:2012.14542.

33 Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281161.

34 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178487.3178488.

35 Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, feb 2008. USENIX Association. URL: https://www.usenix.org/conference/fast-08/avoiding-disk-bottleneck-data-domain-deduplication-file-system.

A Correctness

Nova guarantees the following properties:

Safety once a slice is retired, no operation writes to it or returns a value read from it.

Lock-freedom if threads attempt to perform Nova operations, eventually one succeeds.

Robustness a stalled thread prevents reclamation of at most one slice.

Lock-freedom is guaranteed because every Nova operation completes within a bounded number of its own steps, and fails only if another thread successfully deletes the same slice. Robustness is ensured because only slices referenced in TAP are prevented from being reclaimed, and helping is used to reclaim slices released by stalled threads.

We now discuss safety. First, we examine reads. If a slice is retired, its delete bit is set, so it has been deleted. If the delete occurs before the read begins, then the read either finds the delete bit set in the safe-pointer or a mismatch between the slice header version (which is either marked deleted or already reallocated with a new version) and the safe-pointer version. If the slice is deleted during or before the actual read in the lambda function in Algorithm 1, then the check after the read fails, and the read returns $\perp$. Otherwise, the slice is not retired before the read returns.

As for writes – if they begin after a slice is deleted, they fail in the same manner. If a deletion occurs simultaneously with the write, then the write may still succeed, but in this case, the slice will not be retired until the write completes. This is ensured because the write first sets the TAP entry and then verifies the slice's version against the safe-pointer. If this check succeeds, it means that the delete bit in the slice was not set before the writer's validation. And because a slice can be reclaimed by retire only after its delete bit is set, any attempt to reclaim the slice must begin after the write sets its TAP entry. Thus, the reclaim process sees the TAP entry set and does not reclaim the slice as long as the write is ongoing.

B Mitigating fragmentation

As noted above, we locate headers immediately before their respective slices within the same block. This provides locality of access and avoids the need for additional indirection. However, this restricts memory allocation because headers must remain static – they can be re-used when a released slice is reallocated, but must remain in the same physical location so that future accesses from old safe-pointers will detect the version change. This restriction prevents merging reclaimed slices and may therefore lead to fragmentation.

We note that it would have been possible to instead allocate headers in a dedicated headers block, but this would entail another level of indirection on every slice access so is less desirable. Similarly, it could have been possible to track occupied locations in a separate data structure (e.g., bitmap), but this would require a redundant access to the bitmap before every header access. We therefore forgo these options. Instead, we offer a probabilistic solution based on random *magic numbers* as we now discuss.

Our solution keeps headers located immediately before their slices but allows merging multiple freed slices to form a larger slice. When a slice is deleted, as before, the header remains in the same location with its deleted bit set while the slice is added to the free list. Allocation from the free list first attempts to find an appropriate-size free slice. If a slice of the required length is re-used, then its header is also re-used with a new version, and everything works correctly as described above.

However, if we get an allocation request for a larger slice than all available free ones, we might need to merge two (or more) consecutive free slices. Such a merge will cause the slice's data area to "cover" the old header in the second slice . The problem with allowing the application to use (and possibly over-write) the old (de-allocated) header is that old safe-pointers might still refer to it.

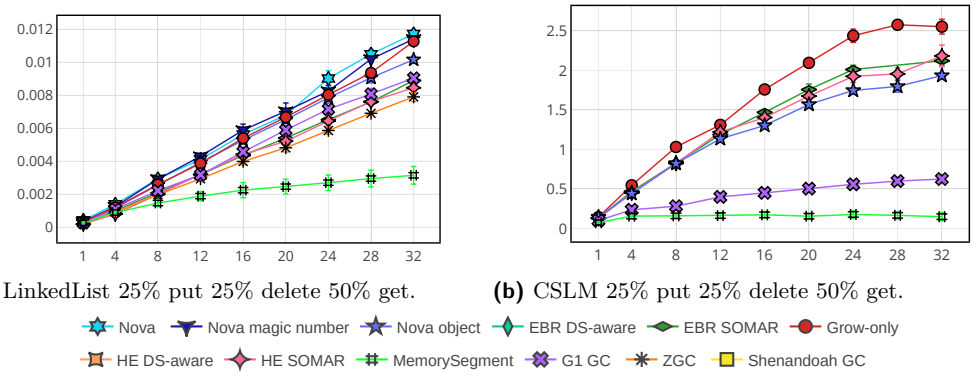
To address this, we add to each header a *magic number*, which is a unique string that is unlikely to occur "in the wild". Specifically, our magic number is a fixed bit string (randomly generated). In our implementation, we use 8 bytes. The magic number is stored in all valid slice headers. When accessing a header, we first verify that the magic number is correct; this is done in addition to validating the version. When we de-allocate a slice, we set its header's magic number field to zero, causing future accesses via this header to fail. The magic number reset can be lazy – we must make it visibly zero only in cases when the slice is merged with a preceding one and the previous header location no longer holds a header.

The magic number approach is probabilistic. It may induce false positives in case (1) a slice is reused and merged with another when an active thread still holds an old reference to it; and (2) the magic number and valid version both occur in the location of a former header by chance. As both fields together comprise $64 + 24 = 88$ bits, the probability for collision is far smaller than the hardware failure rate. This approach is similar to fingerprint-based comparisons used in deduplication techniques [35].

The magic number induces space overhead (it doubles the header size), yet our experiments in the next section show that it has no impact on throughput. The benefit from using magic numbers depends on the workload. If the workload uses uniform allocations and/or entails infrequent deletions, fragmentation is minimal, and the extra overhead for storing magic numbers is not justified. Conversely, if the workload leads to high fragmentation, then the magic number might be beneficial. A study of the fragmentation induced by different workloads is beyond the scope of this paper.

C ARM benchmark results

To check whether there were any platform-specific effects, we repeated some of the experiments on ARM – AWS instance c6gd.8xlarge. The trends were similar to those of x86; results are shown below.



(a) LinkedList 25% put 25% delete 50% get. **(b)** CSLM 25% put 25% delete 50% get.

Figure 7 Throughput (Mops/sec) versus number of threads, on ARM machine.