Nova: Safe Off-Heap Memory Allocation and Reclamation

Ramy Fakhoury

Nova: Safe Off-Heap Memory Allocation and Reclamation

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

Ramy Fakhoury

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tevet 5783 Haifa December 2022

This research was carried out under the supervision of Prof. Idit Keidar, in the Faculty of Electrical & Computer Engineering.

Acknowledgements

I would like to thank Idit for accepting me as a student of hers, and sharing her knowledge. It was a real fun, meaningful and eye-opening experience (though it got cut short by her new role). I could not have asked for a better advisor. I wish to thank Anastasia Braginsky, with whom I worked closely on these results. I have learned a lot from our time together. You set a very high standard for engineers and researchers. It was truly an honor collaborating with both of you.

During my studies I was fortunate to have learned a lot from great professors, whom I would like to thank: Yoram Moses, Yuval Cassuto, Erez Petrank and Dana Drachsler-Cohen. It was a pleasure meeting you.

Gal Assa, Oded Naor, Shir Cohen, Arik Rinberg and Shaked Elias-Zada, it was fun while it lasted. You all have a place in my heart. Mohammad Nassar thanks for sharing the experience with me.

A special thanks to the administrative staff, Orly Babad-Tamir, Adi Rosu, Reut Maravi, Hovav Gazit, and Roy Mitrany, for the help along the way, and the fun conversation we had.

Last but not least, my family at home Nader, Hibat, Razy, Reem and Reham, thanks for being there.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

In recent years, we begin to see Java-based systems embrace off-heap allocation for their big data demands. As of today, these systems rely on simple ad-hoc garbage-collection solutions, which restrict the usage of off-heap data.

This research introduces the abstraction of *safe off-heap memory allocation and reclamation (SOMAR)*, a thread-safe memory allocation and reclamation scheme for managed environments that additionally support allocating unmanaged memory blocks, referred to as off-heap memory. SOMAR allows multi-threaded Java programs to use off-heap memory seamlessly, while ensuring safety, lock-freedom and robustness.

Though the *safe memory reclamation (SMR)* abstraction exists, we explain the shortcomings of such abstraction, and why it is not sufficient for the case of managed environment.

To realize this abstraction, we present Nova, *Novel Off-heap Versioned Allocator*, a lock-free SOMAR implementation. Moreover, we transformed some of the SMR implementations to be compliment with the SOMAR abstraction in order to evaluate how the different approached measure in comparison to Nova.

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, allowing it to reclaim keys while the data structure is being accessed.

Chapter 1

Introduction

Programmers nowadays are accustomed to managed-memory environments as today's most popular programming languages are managed [Dig21]. 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. Yet despite recent advances, today's leading GC solutions still struggle to scale with the volume of on-heap memory. And this problem is exacerbated in memory-hungry "big data" systems. Figure 1.1 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 shape). 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 on-heap; (for more details see Chapter 4). The only on-heap GC implementation that can sustain 85GB 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 [Jav21, Cim20]. 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 [Ell14] and HBase [Hba22, BBH$^+$18, LSJV17]. Another example is the Druid analytics database, which uses off-heap buffers as temporary space for processing queries (e.g., large table merges) [Dru22]. Nevertheless, these systems use the off-heap memory only in specific use cases where memory management 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.

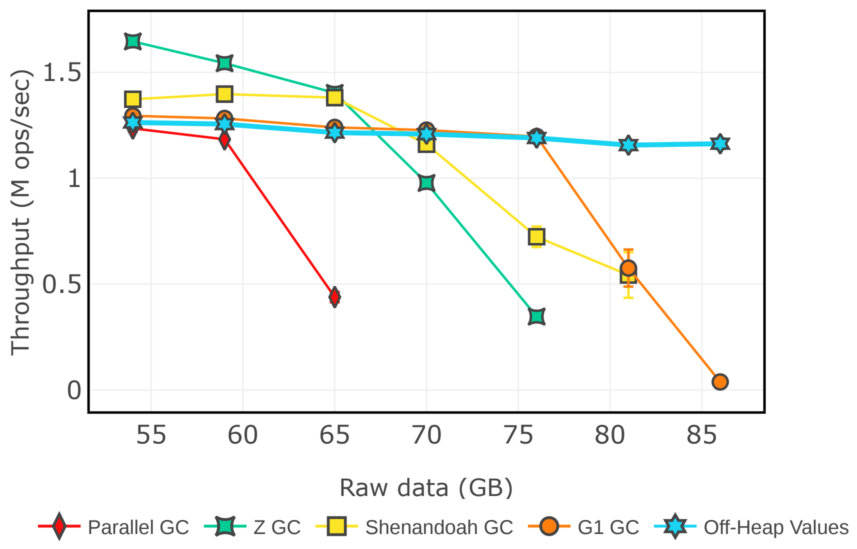


Figure 1.1: Performance of Java's ConcurrentSkipListMap with state-of-the-art Java17 GC implementations versus off-heap allocation managed by Nova.

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 whereby the memory location is reclaimed and then re-used while it is still being accessed by another thread.

The recently-developed Oak key-value map [MBB+20] 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 research, we take off-heap memory management another step forward. In Chapter 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, it might reclaim memory that is still referenced by live threads. A key challenge a SOMAR solution needs to address is ensuring *safety*, namely, detecting every unsafe attempt to access reclaimed (and re-used) memory. When unsafe access is detected the application is informed. Therefore, SOMAR is applicable to data structures that can abort (and possibly restart) an operation upon failure. This includes, for example, lock-free data structures in normalized form [CP15a], and similar approaches [SBM20, SHP21].

In Chapter 3 we present *Nova*, Novel Off-heap Versioned Allocator, a lock-free SOMAR implementation. In addition to safety and lock-freedom, Nova ensures *robustness* [WIC$^+$18]– a stalled thread delays the reclamation of at most one data item. We strive to 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 [CP15c] for read and a variant of hazard pointers [Mic04] for writes. Thanks to the use of version numbers, Nova is applicable to data structures that include traversals, in contrast to traditional hazard pointers.

We implement Nova in Java (as a user-level class) 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 [RC17, Fra04] as well as to on-heap data structures managed by Java's automatic GC. Our results, reported in Chapter 4, show that SOMAR solutions outperform Java's existing (modern) GCs in all workloads. We further show that when primitive types are used to reference off-heap data (instead of Java objects), Nova either outperforms or performs similarly to alternative SOMAR implementations.

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 (where memory is not reclaimed at all).

It is important to point out that SOMAR is different from *safe memory reclamation* (SMR) for unmanaged languages [Mic04], which has received a lot of attention in recent years [RC17, Fra04, Mic04, CP15c, CP15b, SBM20, NR21, HLMM05, GPST09, BKP13, BGHZ16, WIC$^+$18, Bro17, KJ20, SHP21]. The SMR paradigm 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. That said, Nova does employ some techniques that were used to support SMR. We discuss related work in more detail in Chapter 5.

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. Chapter 6 concludes our research.

Chapter 2

SOMAR

We introduce the abstraction of safe off-heap memory allocation and reclamation, SOMAR. SOMAR is intended for managed environments that additionally support allocating unmanaged memory *blocks*, referred to as *off-heap* memory. Such allocation is natively supported in Java versions 14 and up. Specifically, in new Java versions, blocks are allocated off-heap using *memory segments* [Jav21]. Individual blocks allocated within memory segments are not subject to GC and their access is not safe against races between memory access and delete-reuse.

SOMAR allows applications in a managed environment to allocate, use, and deallocate off-heap memory in a safe manner. The allocated unit is called a *slice.* Slices reside within large off-heap blocks. Applications can use their allocated slices for storing data, for instance, the keys and values of a key-value map. An application may further 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. An application 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 lambda functions that operate directly on off-heap memory. Consider for example a slice containing information about a user in serialized form, where the tenth byte encodes its browser type. Algorithm 2.1 shows how a lambda function can be used to update the browser type.

We assume that the lambda functions have no side effects beyond accessing the slice, and furthermore that the one passed to read operations does not update the underlying memory. For simplicity, we assume that there are no nested calls to the safe-pointer

Algorithm 2.1 Writing off-heap data using a lambda function

```
procedure setBrowserCode(long addr, char code)
    set addr+9 to code
write(ptr, setBrowserCode, code)
```

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 API methods may fail upon attempting to access deleted data. In such cases, the read and write functions return $\perp$, whereas the delete method returns true on success and false on failure.

Privatization. Ensuring safety in the presence of potential concurrent deletions incurs a synchronization cost. *Privatization* [SMDS07, KAGR18] 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.

Chapter 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 3.1. In Section 3.2, we present the algorithm's pseudocode, while abstracting away considerations such as data placement, slice allocation and the memory model. These are discussed in Sections 3.3 and 3.4, respectively. Correctness is discussed in Section 3.5.

3.1 Overview

As an implementation of SOMAR, Nova allocates slices with the help of a simple allocator discussed inSection 3.3. 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 within the block right before the slice. This approach is simple, efficient, and avoids indirection. However, it also restricts future allocations within the block and may thus lead to fragmentation in case of 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 off-heap memory and do not reuse *any* memory as long as some allocated data might be referenced [Ell14, Hba22, BBH^{+}18, LSJV17, Dru22]. For applications with frequent

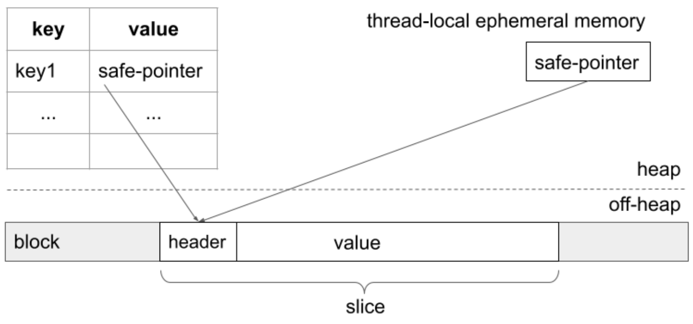


Figure 3.1: Example: A Nova slice with two safe-pointers.

allocation and de-allocation with variable allocation sizes, we present in Section 3.3 below an approach for mitigating fragmentation at the cost of a fixed memory overhead. The cost-effectiveness of this method is workload-dependent.

Figure 3.1 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 another 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.

In order to discover that a slice has been reallocated (after having been deleted), we employ *versions*, which are stored in both headers and safe-pointers, and change whenever a slice is reallocated. Every attempt to access the slice verifies that the version in the safe-pointer is the same as the one in the header.

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 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* [CP15c] and *hazard pointers* [Mic04] for writes, whereby writing threads announce which slices they are accessing to prevent these slices from being reclaimed. Because every access to a slice's data is via a safe-pointer, the scope of each slice access is limited to the

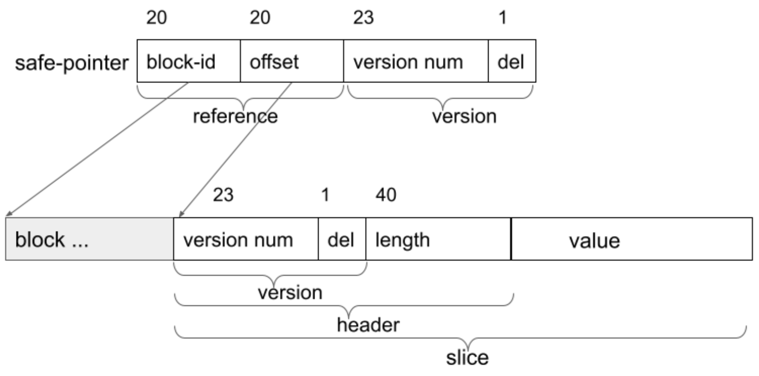


Figure 3.2: An example layout of Nova headers and safe-pointers with 40-bit references and 24-bit versions. The length of each field in bits is indicated above it.

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 nonetheless not difficult 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, a read is allowed to temporarily observe an illegal value, but this situation is detected (via the version) and results in a failure.

2.3 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* consisting of a block identifier and an offset within the block. Each safe-pointer holds a version and a *reference*. An example safe-pointer layout is illustrated in Figure 3.2. In this example, the reference consists of 40 bits, 20 of which specify the offset. With 40 bits, we 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 [GKMZ20]. 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. To allow full flexibility in dividing the 40 bits between the offset and the

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

Table 3.1: Shared data structures used by Nova.

header, we dedicate 40 bits to the length field.

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 3.1. 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 an unused block 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 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.

3.2 Nova Algorithm

We now detail how Nova's API is supported. For clarity, in this section we abstract away some practical considerations: First, as noted above, we assume that the header is static, so it continues to exist in the same location after a slice is freed or even reallocated. Second, we describe the algorithm assuming a sequentially consistent memory model, ignoring the need to define atomic variables or add explicit fences in order for Nova to execute correctly on an architecture with a weak memory model (e.g., x86-TSO). We defer the discussion of these issues to Sections 3.3 and 3.4, respectively.

Algorithm 3.1 Nova safe-pointer read and write methods

```
1: procedure READ(ptr, f)
2:     ⟨ver, slice, header⟩ ← locateSlice(ptr)
3:     if slice = $\perp$ then
4:         return $\perp$
5:     ret ← f(slice)     ▷ read from slice
6:     if header.ver $\neq$ ver then     ▷ validate
7:         return $\perp$
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 = $\perp \lor$ header.ver $\neq$ ver then
13:        set hazard pointer in TAP to $\perp$
14:        return $\perp$
15:    ret ← f(slice, param)     ▷ write to slice
16:    set hazard pointer in TAP to $\perp$
17:    return ret
18: procedure LOCATESLICE(ptr)
      ▷ returns ⟨version, slice, header⟩ triple
      ▷ where version includes version num and deleted bit
19:    myVer ← ptr.ver
20:    if myVer.deleted then return $\langle\perp,\perp,\perp\rangle$
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 3.1. 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 in line 5 and line 15, resp., via the user-provided lambda function f. Their 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 check whether the versions are equal and not which is larger. (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 the previous incarnation exist). The read is optimistic, performing validation after the actual read (line 6). This validation makes Nova suitable for data structures where traditional hazard pointers approaches are not. Nova's write performs the validation before the actual write (line 12).

In both cases, if the versions (with the deleted bit) do not match, this means that the slice has been deleted (and possibly subsequently reallocated), and the operation fails. As an optimization – to prevent future operations from having to check the header in order to detect the same mismatch – 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 does not impact 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.

We do not detail the pseudocode of the privatized write method, as it does nothing more than apply the given lambda function to the slice.

Slice allocation. Pseudocode for Nova's allocation and deletion methods appears in Algorithm 3.2.

Algorithm 3.2 Nova's allocation and deletion methods

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

A new slice is obtained either from a fresh block or from the free list. We assume that the slice is 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 and the deleted bit unset (line 29). We also update the slice's length in the header. The off-heap header is the source of truth regarding the slice's allocation, and so once it's set, the slice is considered to be allocated. The method returns a safe-pointer with a reference to the slice's header and the new version.

Deletion and garbage collection. Like read and write, a delete operation begins by calling locateSlice to retrieve the slice reference and version from the safe-pointer. It then proceeds to update 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.

Note that if additional safe-pointers refer to the same slice, the deleted bits in these safe-pointers remain unset, and threads holding them may continue to attempt to read and write the slice. But after we set the delete bit in the slice's header, such other threads detect the mismatch: If a read is ongoing at the time of deletion, it detects the header change when checking the version at the end of the read, and returns $\perp$. If a write is ongoing at the time of deletion, the slice is registered in the writer's TAP, preventing the slice's garbage collection. Nova does not determine the execution order between a concurrent delete and writes of the same slice. Rather, Nova ensures that (1) a deleted slice is not reallocated as long as ongoing writes might access it; and (2) no future reads or writes that begin after the deletion is complete successfully access the slice. The latter is ensured since reads and writes validate the header version.

Each thread periodically calls the retire procedure (line 42–47) 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. Not shown in the pseudocode is a helping mechanism, whereby threads retire eligible slices from other threads' full release lists to achieve robustness.

3.3 Memory allocation and fragmentation

Nova uses a simple allocator that manages a pool of large pre-allocated off-heap blocks. Slices are allocated with the help of this allocator's free list using a first-fit approach; these slices return to the free list upon being retired. We use a simple allocator, which is satisfactory for achieving good performance in our benchmarks; implementing more sophisticated allocation schemes is orthogonal to Nova's design and is thus beyond the scope of this research.

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 [ZLP08].

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 research.

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 37). 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.

As an aside, we note that it is possible to reduce the number of fences by using epoch-based approaches, in which a fence is issued only when the epoch is updated and not on every access. We experimented with removing fences from the common path and found that their performance impact in the JVM environment was minimal. We therefore decided to forgo this optimization.

3.5 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 line 5, 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.

Chapter 4

Evaluation

We now evaluate Nova and two other SOMAR implementations. The experiment setup is described Section 4.1. In Section 4.2 we use SOMAR to manage off-heap data organized in Java data structures. Next, we seek to evaluate Nova within Java solutions employing off-heap memory. To this end, in Section 4.2.6 we integrate Nova into the Oak library, replacing the naïve grow-only allocator therein. We note that there is no benefit in integrating Nova to systems like HBase and Druid, which use off-heap memory as a temporary space which is released at once when the computation is done and therefore do not require a memory manager.

4.1 Experiment setup

All the code is written in Java 17 (user-level code without modifying the JDK) and built with Apache Maven 3.8.6. The code is available on GitHub [1].

The experiments use a customized synchrobench tool [Gra15]. 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 with error bars (in most cases they are unnoticeable).

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 – AWS instance c6gd.8xlarge. The trends were similar; one exemplary result is shown below. All experiments utilize 32 cores (without hyper-threading) on one NUMA node, the OS is Ubuntu 20.04.

4.2 Off-heap data indexed by Java data structures

4.2.1 Benchmarks.

We measure end-to-end application performance for data structure benchmarks. Our experiments highlight two different scenarios where off-heap allocation can be beneficial.

[1] https://github.com/li0nr/Nova-Safe-off-heap-memory-allocation-and-reclamation.

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 [Har01] 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.

We note that in the LL-map experiment we did not limit the available memory. Thus, this experiment runs with no memory stress.

4.2.2 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. Among these (surveyed in Chapter 5 below), we chose to compare Nova to *Epoch-Based Reclamation (EBR)* [Fra04] and *Hazard Eras (HE)* [RC17]. We also experimented with Interval-Based Reclamation [WIC$^+$18] in some workloads and got similar results to EBR, so we refrained from a detailed study of this approach. Another recent SMR approach is VBR [SHP21]; its read operations are similar to Nova's and resulting in similar performance similarly and its writes are different from Nova's (they are optimistic), but they are restricted to single-word updates, so VBR is not applicable to our setting.

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 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 Nova both with and without magic numbers.

We compare the three SOMAR approaches (Nova, EBR, and HE) to the following baselines: (1) on-heap allocation with Java17's default G1 GC; (2) on-heap allocation with Java17's newly introduced ZGC; (3) and Shenandoah GC; (4) raw Java memory segments 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.

4.2.3 LL-map results.

The throughput results for the LL-map are shown in Figure 4.1(a)–(c). 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 the off-heap solutions.

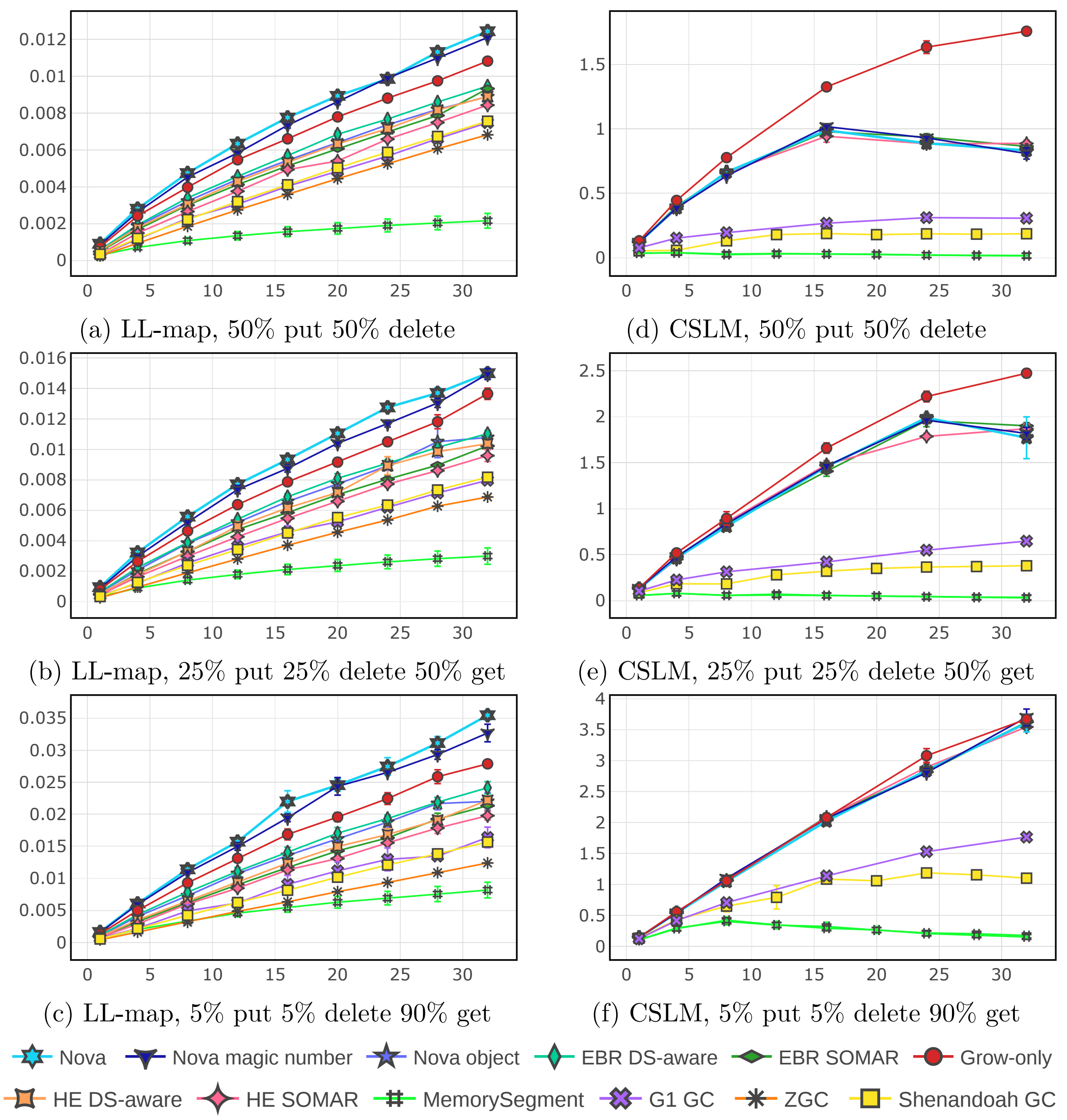
Nova achieves the best performance among SOMAR solutions. Its peak throughput is more than 2x that of the Java GC in all workloads, and it outperforms all other off-heap solutions by 20–30%. It even performs better than the grow-only solution that does not reclaim memory at all. Part of the benefit can 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.

4.2.4 CSLM results.

The CSLM results are shown in Figure 4.1(d)–(f). In all these experiments, ZGC crashes with an OutOfMemoryError exception when allocated the same DRAM budget as the other solutions. Therefore, it is not shown in the graphs. In all workloads, the Java on-heap G1, Shenandoah solutions as well as memory segments fall far behind the off-heap solutions. At the same time, all SOMAR variants behave similarly (less than 1% difference). This can be expected since the heavy work in a CSLM (searching) involves only the 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 an extra 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.

We use a profiler to assess the impact of GC. Figure 4.2 presents a timeline of CPU utilization and GC activity during the 32 thread experiment. We see that Nova (Figure 4.2a) 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 the on-heap solution (Figure 4.2b) 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 4.1 summarize the memory utilization of all solutions (except grow-only) in the read-write workload with 32 threads. We measured the memory consumption both before the benchmark run (after initialization) and after it, and the memory consumption of all algorithms was roughly the same at both times. We see that GC requires slightly more memory than the off-heap approaches, and that the overhead of Nova's magic numbers is small. We note that fragmentation does not occur in this



(a) LL-map, 50% put 50% delete

(d) CSLM, 50% put 50% delete

(b) LL-map, 25% put 25% delete 50% get

(e) CSLM, 25% put 25% delete 50% get

(c) LL-map, 5% put 5% delete 90% get

(f) CSLM, 5% put 5% delete 90% get

Figure 4.1: LL-map and CSLM throughput (Mops/sec) versus number of threads.

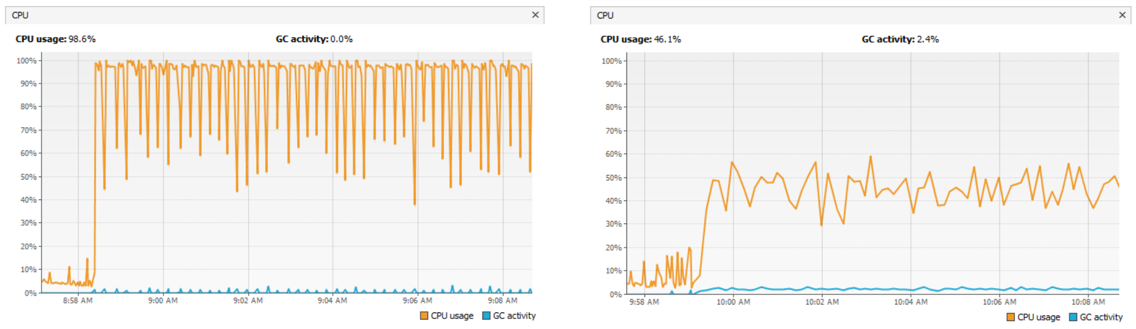
experiment because all allocations are of the same size.

	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
total	12.4	12.5	12.5	12.6	12.7	13.0

Table 4.1: Memory consumption, GB, CSLM, read-write workload, 32 threads.

4.2.5 ARM results.

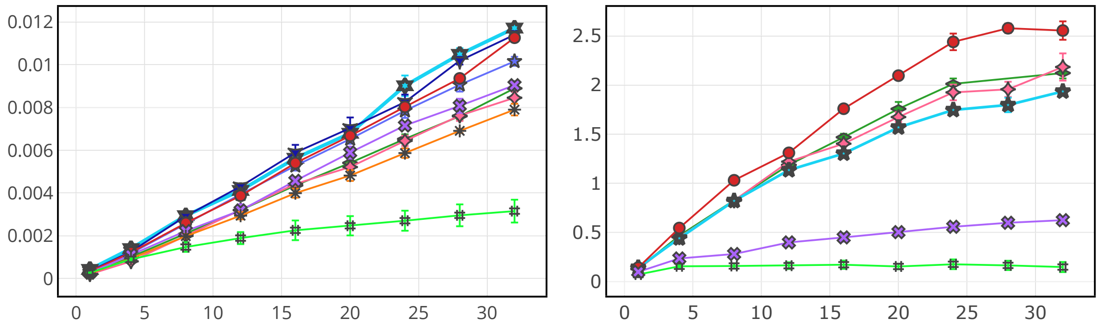
We repeated some experiments on ARM to check for platform-specific effects. In Figure 4.3 we see the read write workload tests for LL-map and CSLM on ARM. We see



(a) off-heap CSLM (b) on-heap CSLM

Figure 4.2: CPU usage and GC activity in the CSLM experiment measured by VisualVM.

that the trends are generally the same, with two differences: In LL-map, Java's G1 GC is not worse than all SOMAR solutions, though it is still outperformed by Nova. And in CSLM, Nova is slightly inferior to other SOMAR implementations. The latter is probably due to the fact that load fences incur a performance penalty in ARM and not in x86, and Nova uses such fences on all reads, whereas HE and EBR reduce the number of such fences using epochs.



(a) LL-map, 25% put 25% delete 50% get (b) CSLM, 25% put 25% delete 50% get

Figure 4.3: LL-map and CSLM throughput on ARM (Mops/sec) versus number of threads, read- write workload.

4.2.6 Nova in Oak

We also integrated Nova into the Oak 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. By default, the former is used for keys and the latter for values.

[2]https://github.com/yahoo/Oak/commit/68485b04ddc020ace5ac33d30c803b97f49e8759.

We experiment with Oak using the following memory managers:

Unreleased keys grow-only keys, lock-based values;

Grow-only grow-only for both keys and values;

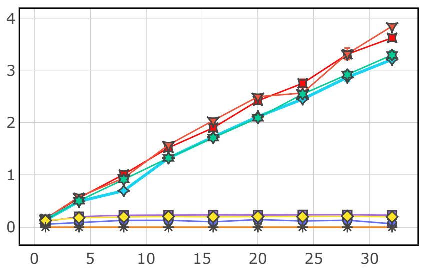
Lock-based lock-based for both keys and values;

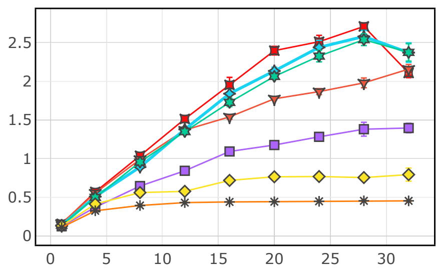
Nova keys Nova for keys, lock-based values;

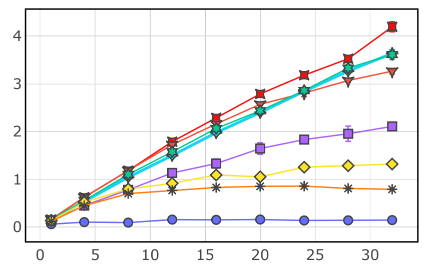
Nova all Nova for keys and values.

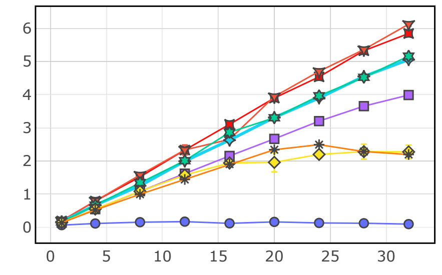
We compare all of these to Java's CSLM, G1 GC, Shenandoah GC and ZGC. All solutions run with a memory budget of 17GB. The off-heap solutions split this budget to 6GB on-heap, and the rest is off-heap. The grow-only variation has no memory restriction.

Our benchmarks generally follow the ones used in Oak's evaluation [MBB+20]. 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. In order to measure memory consumption and demonstrate the effect of the grow-only approach, we ran the experiments for 5 minutes.


(a) 100% put


(b) 50% put 50% delete


(c) 50% get, 25% put, 25% delete


(d) 100% zero-copy get

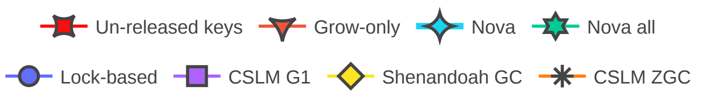


Figure 4.4: Map throughput (Mops/sec): Oak with different reclamation schemes and CSLM.

		Oak					CSLM	
		grow-only	un-released 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
	total	10.7	10.8	10.9	10.9	10.8	12.4	14.8
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	11.0	0	0
	total	43.6	16.5	10.9	11.4	11.0	12.4	15.4

Table 4.2: Memory consumption, GB, Oak read-write workload, 16 threads.

The throughput results appear in Figure 4.4 and the memory consumption in Table 4.2. We see that the lock-based memory manager does not scale due to contention on the lock. 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 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, enhancing it with a functionality that is currently missing, namely, the ability to reclaim memory used for keys while the data structure is being accessed.

Chapter 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)* [RC17, Fra04, Mic04, CP15c, CP15b, SBM20, NR21, HLMM05, GPST09, BKP13, BGHZ16, WIC$^+$18, Bro17, KJ20], 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 the safe access is the DS operation, and data stored in the DS is externally inaccessible. Thus, retrieval operations cannot return pointer, and must instead copy the data, which is costly when data items are large. In addition, this scheme does not accommodate iterators, because they continue to hold a pointer into the DS after a getNext call returns. In contrast to all of these, 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. 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 may need to be retried. Another difference is that SOMAR is designed for a managed environment where only some of the 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 [NR21], and is also used for automatic reclamation in OrcGC and DRC [CRF21, ABW21]. As explained above, Nova is not amenable to reference counting because user code may copy safe-pointers without informing Nova.

Hazard pointers [Mic04, HLMM05, DHK16] and their variants [GPST09] 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 [Bro17]. 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 [CP15c, CP15b, SHP21] and uses them only for writes while reads proceed optimistically.

Epoch-based reclamation [Fra04, MS02] 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+ [Bro17], NBR/NBR+ [SBM20], and ThreadScan [ALMS15]. 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 [Bro17, SBM20], which is not readily applicable in a JVM environment. Other solutions combine HP and EBR to improve EBR's robustness [WIC+18, RC17, BKP13, BGHZ16, KJ20]. 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.

Chapter 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.

Bibliography

[ABW21] 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, page 526–541, New York, NY, USA, 2021. Association for Computing Machinery.

[ALMS15] 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, page 123–132, New York, NY, USA, 2015. Association for Computing Machinery.

[BBH$^+$18] 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.

[BGHZ16] 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, page 349–359, New York, NY, USA, 2016. Association for Computing Machinery.

[BKP13] 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, page 33–42, New York, NY, USA, 2013. Association for Computing Machinery.

[Bro17] Trevor Brown. Reclaiming memory for lock-free data structures: there has to be a better way, 2017.

[Cim20] Maurizio Cimadamore. Bytebuffers are dead, long live bytebuffers! `https://www.youtube.com/watch?v=RYrk4wvar6g&t=172s&ab_channel=FOSDEM`, 2020.

[CP15a] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *SIGPLAN Not.*, 50(10):260–279, oct 2015.

[CP15b] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *SIGPLAN Not.*, 50(10):260–279, October 2015.

[CP15c] 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, page 254–263, New York, NY, USA, 2015. Association for Computing Machinery.

[CRF21] 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, page 205–218, New York, NY, USA, 2021. Association for Computing Machinery.

[DHK16] 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, page 36–45, New York, NY, USA, 2016. Association for Computing Machinery.

[Dig21] Larry Dignan. The most popular programming languages and where to learn them. `https://www.zdnet.com/article/best-programming-language/`, May 2021.

[Dru22] Docs Druid. Basic cluster tuning. `https://druid.apache.org/docs/latest/operations/basic-cluster-tuning.html`, retrived in April, 2022.

[Ell14] Jonathan Ellis. Off-heap memtables in cassandra 2.1. `https://www.datastax.com/blog/heap-memtables-cassandra-21`, 2014.

[Fra04] Keir Fraser. Practical lock-freedom. technical report ucam-cltr-579. 2004.

[GKMZ20] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap, 2020.

[GPST09] 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.

[Gra15] 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, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[Hba22] Reference Guide Hbase. Hbase offheap read/write path. `https://hbase.apache.org/book.html#offheap_read_write`, retrived in April, 2022.

[HLMM05] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Non-blocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

[Jav21] Docs Java. Interface memorysegment. `https://docs.oracle.com/en/java/javase/17/docs/api/jdk.incubator.foreign/jdk/incubator/foreign/MemorySegment.html`, 2021.

[KAGR18] Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. Safe privatization in transactional memory. *SIGPLAN Not.*, 53(1):233–245, February 2018.

[KJ20] 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, page 314–328, New York, NY, USA, 2020. Association for Computing Machinery.

[LSJV17] 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.

[MBB$^+$20] 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, page 17–31, New York, NY, USA, 2020. Association for Computing Machinery.

[Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[MS02] Paul E. Mckenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. 2002.

[NR21] 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, page 987–1002, New York, NY, USA, 2021. Association for Computing Machinery.

[RC17] 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, page 367–369, New York, NY, USA, 2017. Association for Computing Machinery.

[SBM20] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: Neutralization based reclamation, 2020.

[SHP21] 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, page 443–445, New York, NY, USA, 2021. Association for Computing Machinery.

[SMDS07] 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, page 338–339, New York, NY, USA, 2007. Association for Computing Machinery.

[WIC$^+$18] 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, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery.

[ZLP08] 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, February 2008. USENIX Association.

עם איטרטורים (iterators). גם כן חסרון ממש בולט באבסטרקציה הזאת הוא שאי אפשר להחזיר למתכנת מצביע לאזור זיכרון מסוים, או שאנחנו מעבדים הנתונים שלנו תוך ההיקף (scope) של הפעולה או שמחזירים למתכנת העתק של הנתונים. ולכן היה צורך בלהציע אבסטרקציה חדשה. האלגוריתם שאנחנו מציעים, משתמש בגרסאות כדי להתגבר על הבעיות הקיימות באלגורתמים אחרים.

ממשנו האלגוריתם שלנו בשפת ג'אווה בגרסה הכי עדכנית, ואנחנו מראים בניסויים שלנו השיפור בביצועים מול פתרונות ניהול זיכרון אחרים. גם כן אנחנו מראים שיפור משמעותי מול כל פתרונות ניהול הזיכרון הדינמי האוטומטית הכי עדכניים שהשפה מציעה (GC). גם מיזגנו המימוש שלנו לתוך ספריית קוד פתוח (open source) שלא היה שם ניהול זיכרון, השימוש שם בזיכרון off-heap היה בלי שחרור או בעזרת מפתחות. הראינו שבעזרת השיטה שלנו אפשר לנהל זיכרון כזה בלי להתפשר על ביצועם.

תקציר

מתכנתים היום רגילים לשפות תכנות שהם מנוהלות, כגון שפת ג'אווה (JAVA), או במלים אחרות שפות שיש להם שיטת ניהול זיכרון דינמי אוטומטית. בשפות כאילו הזיכרון מוקצה באזור שנקרא הערימה (Heap), והמערכת בדרך כלל מנהלת הזיכרון בעזרת garbage collector (GC) . למרות שפרדיגמת תכנות זאת היא נוחה, היא עדיין סובלת ברגע שיש למערכת כמות גדולה של נתונים. ובפרט שבשנים האחרונים ניכרת גדילה משמעותית בהתמודדות עם נתוני עתק (Big Data), מה שגרם לבעיות של ה GC, להיות יותר דומיננטיות. כתוצאה מגידול זה הרבה מהמערכות שמבוססות על שפות מנוהלות, התחילו להשתמש בזיכרון שהוא לא מנוהל ע"י ה GC, זיכרון זה נקרא זיכרון off-heap. אופן השימוש בזיכרון הזה היה פשוט ולרוב מבוסס על שימוש חד פעמי, כך שצורת העבודה הזאת היא לא הכי אופטימלית ובזבזנית, אבל מצד שני אפשר לשחרר זיכרון בצורה יותר נוחה, כך שהשחרור יתבצע ברגע שכל החוטים סיימו לעבוד על אזור זיכרון זה.

שחרור (ניהול) הזיכרון הדינמי היא לא משימה פשוטה, אפילו משימה קשה ברגע שאנחנו מדברים על סביבה מקבילית: אם מסתכלים על אזור מסוים שחוט אחד שחרר אותו במערכת שיש בה גישות מקביליות - צריך לדאוג ולמנוע מצב שבו אזור זה משוחרר ומוקצה מחדש בזמן שחוט אחר עדיין ניגש אליו.

כדי להתמודד עם הביעה של נתוני עתק בשפות מנוהלות ולשפר הביצועים של מערכות שכבר משתמשות בזיכרון off-heap, ולאפשר שימוש יותר רחב בזיכרון זה (להבדיל מהצורה הסטנדרטית של שימוש חד פעמי). במחקר הזה אנחנו מציעים אבסטרקציה חדשה, עם ממשק (API), לשפות שבהן הזיכרון מנוהל. הממשק שלנו מאפשר למתכנת להקצאות זיכרון off-heap, ולשחרר זיכרון זה. גם כן דרך האבסטרקציה אנחנו מאפשרים קראיה וכתיבה לזיכרון כזה, תוך כדי שמירה על הנכונות של פעולות מקיבליות על אותו קטע זיכרון.

בנוסף להצגת האבסטרקציה אנחנו ממשים אותה ע"י אלגוריתם ניהול זיכרון דינמי חדשני שמתאים גם לשפות מנהלות וגם לשפות שהם לא מנוהלות. אנו מוכיחים שהאלגוריתם שלנו בטוח(safety), עומד בדרישת חוסר-נעילה (lock-freedom) ושתמיד מאפשר מחיקה של זיכרון (robustness). גם כן אנו מתאימים את האלגוריתם שלנו לעבוד עם מודל זיכרון חלש (weak memory model).

קיימים בספרות אלגוריתמים רבים שנותנים פתרון לבעיית ניהול הזיכרון, כך שאפשר לסווג כל הפתרונות תחת אבסטרקציה בשם ניהול זיכרון דימני בטוח (Safe Memory Reclamation). תחת האבסטרקציה הזאת המתכנת צריך להצהיר באופן מפורש על כוונתי להתחיל פעולה מסוימת, ובסוף צריך להצהיר על סיום הפעולה. האבסטרקציה הזאת לא מתאימה לנהל זיכרון off-heap בשפות שהן מנוהלות, מכוון שיש כמה תרחישים שלא מאפשרות ההצהרה על סיום פעולה, למשל עבודה

המחקר בוצע בהנחייתו של פרופסור עידית קידר, בפקולטה להנדסת חשמל ומחשבים.

מחבר חיבור זה מצהיר כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו', נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

גישה חדשה להקצאת זיכרון

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת חשמל

ראמי פאחורי

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
טֵבֵת תשפ"ג חיפה דצמבר 2022

גישה חדשה להקצאת זיכרון

ראמי פאחורי