Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation

Michael Haubenschild
Salesforce
mhaubenschild@salesforce.com

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

High-performance data systems running on modern many-core CPUs should not require readers to acquire physical locks. There exist multiple synchronization schemes that allow this, but they all require a solution for concurrent memory reclamation. In this paper, we show that one can eschew such a scheme when a few basic requirements are fulfilled. We exploit this insight in our high-performance buffer manager implementation in LeanStore, which as a consequence got simpler, more robust, and more performant.

ACM Reference Format:
Michael Haubenschild and Viktor Leis. 2023. Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation. In *1st Workshop on Simplicity in Management of Data (SiMoD '23), June 23, 2023, Bellevue, WA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3596225.3596228

1 INTRODUCTION

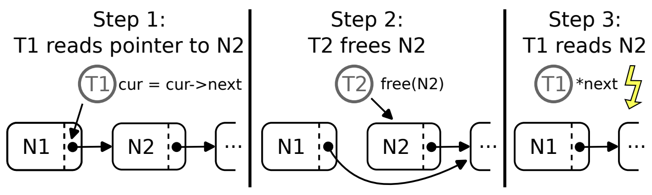
Efficient Reads on Shared Data Structures. For high-performance systems, concurrent access to shared data structures by multiple threads is crucial. In particular, readers should not acquire a lock (not even a shared lock), as it can severely degrade scalability [12]. Different synchronization schemes exist where readers do not need to write to shared memory at all, e.g., lock-free data structures [8] and optimistic locking [4]. Conventional wisdom is that all schemes that avoid pessimistic locking require some form of concurrent memory reclamation to prevent node deletion from causing crashes. Memory reclamation increases implementation complexity and adds runtime overhead or makes the system less robust. In this paper, we show that – remarkably – one can avoid memory reclamation altogether if two conditions, which we describe in Section 3, are fulfilled. These conditions hold for our high-performance buffer manager in LeanStore, which got simpler and more robust once we had this insight.

Background: Lock-Free Reads. Lock-free data structures [8, 13, 15, 16] allow access with multiple concurrent threads without them acquiring locks. Modifying operations (insert, update, and delete) employ atomic read-modify-write (RMW) primitives, e.g., compare and swap (CAS), to manipulate the data structure. Read operations only perform loads, and depending on the guarantees of the concrete data structure (wait-free, lock-free, or obstacle-free), can always finish in a bounded numbers of steps or may require a restart if they detect any inconsistency, e.g., a broken chain in a linked list. An alternative are optimistic locks, which consist of a normal lock and an additional atomic version counter [4–6, 12]. A read operation does not acquire any locks. Instead, it reads the version before and after the operation. If it is the same, and in addition the lock was unlocked before and after the read, it can be certain that no in-between operation modified the object. On the other hand, if the version counter changed, then the operation needs to restart. Write operations increment the version counter while they hold the lock. Optimistic locks can be combined with lock coupling [10, 11], which yields a powerful building block for implementing scalable synchronization protocols, e.g., for B-trees or tries.

Table 1: Qualitative comparison of concurrent memory reclamation strategies.

	Epochs	Hazard Pointers	Stable Buffer Frames & Version Counters
efficiency	+	~	++
robustness	-	+	++
impl. complexity	+	~	++

Concurrent Memory Reclamation. Both lock-free data structures and optimistic locks need to solve the challenge of freeing allocations while concurrent readers, which do not hold any lock at all and are thus "invisible" to the deleting thread, might be still accessing them. The following example illustrates how this can lead to a segmentation fault in thread T1:



In the figure, after T1 has read a pointer to N2, but before it can dereference it, another thread T2 frees N2. If T1 now dereferences that pointer, a crash may occur. A naive solution is to never physically free allocations. But since memory is a scarce resource, leaking memory is generally not feasible. We describe the two common existing solutions that allow to eventually free unused allocations in Section 2. Unfortunately, both come with disadvantages.

Synchronization in LeanStore. In LeanStore [9], we use an optimistic lock for each page to synchronize buffer pool accesses. The B-tree avoids locking pages for inner node traversals and leaf page point lookups using an *optimistic* mode – making reads lock-free.

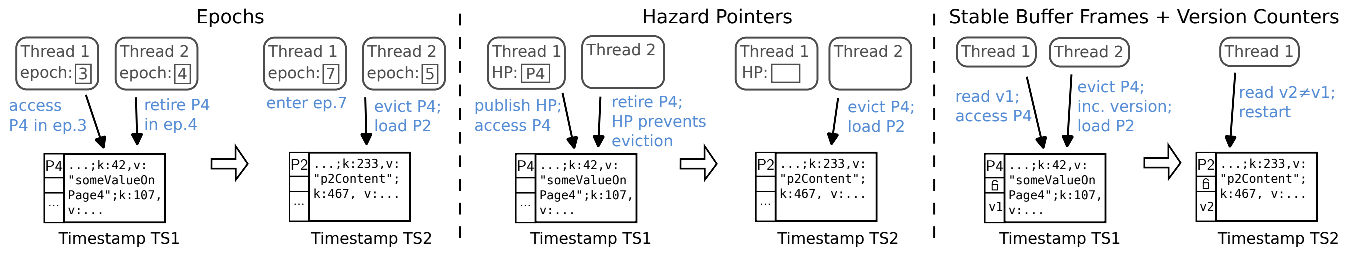


Figure 1: Three strategies for how a buffer frame occupied by page P4 can be eventually reused for another page P2 in the presence of a lock-free reader (Thread 1). Both *epochs* and *hazard pointers* delay the physical deallocation until some timestamp TS2 when there are no more concurrent readers. *Stable buffer frames & version counters* allow reusing the page immediately with the reader detecting the inconsistency.

LeanStore also supports an *exclusive* mode for all modifying operations and a *shared* mode for long-running read operations (e.g., range scans). The first version of LeanStore used epoch-based reclamation for evicting pages and reusing buffer frames [9]. In this paper, we describe how we were able to get rid of concurrent memory reclamation completely by adhering to two basic requirements. As shown in Table 1, this resulted in a simpler, more robust and efficient system.

2 EXISTING SOLUTIONS FOR MEMORY RECLAMATION

Retired List. The existing approaches for concurrent memory reclamation decouple the logical and the physical delete of an object. The logical delete unlinks the object from the data structure so that it is no longer reachable by new operations. Then, it is put into a so-called *retired list* of objects that are to be freed at a later point in time. The approaches differ in the mechanism how they detect that there are no lock-free concurrent readers on those objects anymore. The actual deallocation of that object can then be done either by the thread that originally unlinked it, or by other threads in a collaborative fashion.

2.1 Epoch-Based Reclamation

Algorithm. Each thread has a counter, called *local epoch*, that all other threads can read. Threads regularly update their local epoch with the current *global epoch*. The global epoch is a monotonically increasing counter, which is frequently incremented. So all epoch counters can only ever increase, with local epochs potentially lagging behind the global epoch. When an allocation is put into the retired list, it is tagged with the current global epoch. The retired list is frequently scanned for allocations that are tagged with an epoch that is smaller than the minimum of all local epochs, which can then be physically freed. An example for this is shown in Figure 1.
Discussion. Epochs are fairly easy to understand and implement. They are also efficient in the hot path, as individual object accesses require no additional overhead. The work to update the local epoch and increment the global epoch can be amortized over multiple operations. On the downside, the frequency of the updates is a tuning parameter that needs to be chosen carefully: Incrementing epochs too frequently incurs overhead, but if they are not updated often enough, the retired list may grow large. Worst of all, a single straggler thread may halt the entire memory reclamation, leading to an unbounded amount of garbage accumulating.

2.2 Hazard Pointers

Algorithm. Hazard pointers [14] are a more precise way of tracking concurrent accesses by other threads than epochs. Each thread has a small number of *hazard pointer* slots which it uses to announce to other threads which objects it is currently reading. It does so by first reading a pointer, then storing that pointer in its array of hazard pointers, and finally reading the pointer again, checking if it is still the same. If that is the case, the thread can safely dereference the pointer. Multiple slots are required to achieve some kind of overhand locking. When a thread wants to delete an object from the retired list, it has to check that no hazard pointer from any of the threads currently points to that object. In the hazard pointer example in Figure 1, thread 2 cannot immediately evict P4 at timestamp TS1, since thread 1 holds a hazard pointer to P4.
Discussion. The main benefit of hazard pointers is that the size of the retired list is bounded by the total number of hazard pointers in the system. But compared to epoch-based reclamation, hazard pointers are more complex to implement, and a mistake in the precise sequence of operations can lead to subtle bugs. The biggest issue, however, is their overhead. For every object being accessed, a sequentially consistent atomic store to a hazard pointer is required in order to flush the write buffer of the CPU core.

3 IDEA: CONCURRENT READERS DETECT EVICTION AND RESTART

Requirements. As discussed above, both of the existing memory reclamation schemes have disadvantages. We realized that our buffer manager in LeanStore fulfills two requirements that together allow us to get rid of memory reclamation altogether [1]:

(1) Our buffer manager *never actually releases memory* back to the operating system. All memory for the buffer pool is pre-allocated at application startup. When a page is evicted, its memory is simply put into a free list.
(2) The version counter of each optimistic lock is *strictly monotonically increasing*. It is stored in the buffer frame and not the page itself, and is not reset between page evictions. Pages are evicted while holding the optimistic lock exclusively, which increases the version counter before the lock is released.

Table 2: Performance metrics for the reclamation schemes for random OLC reads on a B-tree with 8 byte keys and values.

	B-tree size	ops / sec	scale w/ 32 thr.	instr. / op	cycles / op
Epochs	160kb	13.4M	31.0	181	260
Hazard Pointers	160kb	10.1M	30.9	203	345
Stable Buffer Frames & Version Counter	160kb	13.6M	30.8	175	257
Epochs	1.6gb	1.8M	30.8	363	1930
Hazard Pointers	1.6gb	1.4M	31.1	393	2448
Stable Buffer Frames & Version Counter	1.6gb	1.9M	30.0	358	1842

The first requirement guarantees that there is no segmentation fault when a thread tries to access the memory of a page after it has been evicted. However, it may read the data of a completely different page that has been loaded into that memory location in the meantime. But the second requirement ensures that the version validation will fail in that case, and the reader will restart its operation, which is shown in Figure 1 at timestamp `TS2`. When traversing the tree again, the thread will (re)load all required pages into the buffer pool. Thus, there is no need for delaying page eviction until there are no more potential concurrent readers.

Implementation. Putting this idea into practice is very straightforward. We simply remove all epoch related code from LeanStore, and instead of resetting the version counter when a page gets evicted, it is now incremented, which is a trivial change. The optimistic locking code already needed to be able to handle restarts before, and thus did not require any adaptation. Remarkably, this change is also orthogonal to the persistency scheme [7], the concurrency control scheme [3], and contention management [2], all of which naturally integrate with it via optimistic locking.

4 MICROBENCHMARKS

Table 2 shows the performance characteristics of the different memory reclamation schemes for random B-tree lookups. The experiments are conducted on an AWS EC2 `c6i.16xl` instance which has 32 physical CPU cores. Table 2 shows that stable buffer frames and epochs reach similar single-threaded performance, while hazard pointers are ~20% slower. All competitors use optimistic lock coupling (OLC) for synchronization, so they scale very well (30 – 31×) with 32 threads. While epochs are almost as fast as stable buffer frames, their main disadvantage is that an unbounded amount of allocations can accumulate in the retired list when there is a straggler thread, which does not show up in the microbenchmark. Hazard pointers are robust, but require additional instructions on the hot path, including sequentially consistent atomic stores which flush the write buffer of the CPU core. Note that the performance difference between the schemes is not that pronounced in this benchmark, as B-tree lookups consist of relatively expensive binary searches on a few large nodes. However, it is more representative of a real workload in LeanStore than, e.g., a simple linked list traversal.

5 FUTURE WORK: GENERALIZATION

In this paper, we have shown that LeanStore does not require concurrent memory reclamation despite employing scalable optimistic reads. We believe that this scheme is directly applicable to many existing buffer-managed systems. A remaining question is whether our idea can be generalized to arbitrary lock-free data structures. A general purpose allocator for data structures synchronized with optimistic locks can be implemented without concurrent memory reclamation by adhering to the following principles:

(1) When freeing an allocation, mark its memory with `MADV_DONTNEED` instead of unmapping it from the virtual address space. This way, no physical memory is consumed for that memory range anymore. Subsequent accesses to that address would be safely redirected to the zero page by the OS. Reads will consequently also return 0s.
(2) The version counter validation that is part of the optimistic locking protocol must treat "`0`" always as an invalid version, as it might be the result of an optimistic reader accessing the zero page after the memory has been deallocated.
(3) The version counter must always be increasing (or be "`0`"). This can be achieved either by
    (a) remembering the version counter, e.g., in a hash table, and restoring it on the next allocation or by
    (b) storing the version counter separately from the actual allocation, e.g., in a global array.

The granularity of allocations with this approach is multiples of the page size. An open challenge is how to reuse the memory for allocations of a different sizes, as the version counter might be overwritten with arbitrary payload data.

REFERENCES

[1] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI, Vol. P-331)*. 259–281.
[2] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*. www.cidrdb.org.
[3] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (2023), 1426–1438.
[4] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*.
[5] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *PPoPP*. 257–268.
[6] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*. 181–190.
[7] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage. *Proc. VLDB Endow.* 16, 9 (2023).
[8] Alex Kogan and Erez Petrank. 2011. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP*. 223–234.
[9] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*.
[10] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.
[11] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.
[12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[13] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. 73–82.
[14] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distributed Syst.* 15, 6 (2004), 491–504.
[15] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. 267–275.
[16] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *J. ACM* 53, 3 (2006), 379–405.