Brief Announcement: On Implementing Wear Leveling in Persistent Synchronization Structures

Jakeb Chouinard
Department of Mechanical and Mechatronics Engineering, University of Waterloo, Canada

Kush Kansara
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Xialin Liu
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Nihal Potdar
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab

Department of Electrical and Computer Engineering, University of Waterloo, Canada

Abstract

The last decade has witnessed an explosion of research on persistent memory, which combines the low access latency of dynamic random access memory (DRAM) with the durability of secondary storage. Intel's implementation of persistent memory, called Optane, comes close to realizing the game-changing potential of persistent memory in terms of performance; however, it also suffers from limited endurance and relies on a proprietary wear leveling mechanism to mitigate memory cell wear-out. The traditional embedded approach to wear leveling, in which the storage device itself maps logical addresses to physical addresses, can be fast and energy-efficient, but it is also relatively inflexible and can lead to missed opportunities for optimization. An alternative school of thought, exemplified by "open channel" solid state drives (SSDs), delegates responsibility for wear leveling to software, where it can be tailored to specific applications. In this research, we consider a hypothetical hardware platform where the same paradigm is applied to the persistent memory device, and ask how the wear leveling mechanism can be co-designed with synchronization structures that generate highly skewed memory access patterns. Building on the recent work of Liu and Golab, we implement an improved wear leveling atomic counter by leveraging hardware transactional memory in a novel way. Our solution is close to optimal with respect to both space complexity and measured performance.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases persistent memory, transactional memory, wear leveling, atomic counter, concurrency, fault tolerance, theory

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.38

Funding This research was supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Acknowledgements We thank the anonymous reviewers for their helpful feedback on this work and their insightful suggestions regarding future research directions.

1 Introduction

The last decade has witnessed an explosion of research on persistent memory. Research activities in this area are primarily driven by the performance benefits of persistent memory, which behaves like dynamic random access memory (DRAM) with respect to access latency and yet provides the durability of secondary storage. Thus, persistent memory can be used

© Jakeb Chouinard, Kush Kansara, Xialin Liu, Nihal Potdar, and Wojciech Golab;
licensed under Creative Commons License CC-BY 4.0

directly to store application state during a computation, and its use opens the door to recovering such state efficiently from the memory device after a power failure or system crash. Intel's implementation of persistent memory, called Optane, comes close to realizing the game-changing potential of persistent memory in terms of performance, but it suffers from limited endurance, meaning that the memory cells tend to wear out in response to repeated overwriting [3]. To prevent irrecoverable data loss during the product warranty period, Optane persistent memory modules use a proprietary wear leveling mechanism that remaps logical memory addresses to physical addresses somewhat similarly to a flash translation layer (FTL) in a solid state drive (SSD).

The traditional embedded approach to wear leveling, in which the storage device itself internally performs address remapping, can be fast and energy-efficient. However, this one-size-fits-all solution is relatively inflexible, and it can lead to missed opportunities for optimization when the workload (i.e., data access pattern) generated by an application deviates from the one anticipated by the hardware designer. An alternative school of thought, exemplified by "open channel" solid state drives (SSDs) [8], addresses this inherent limitation by delegating responsibility for wear leveling to software, where it can be tailored more effectively to specific applications. In this research, we consider a hypothetical hardware platform where the same paradigm is applied to the persistent memory device, and ask how the wear leveling mechanism can be co-designed with persistent data structures.

The case for application-managed wear leveling in the context of persistent memory is especially interesting due to stringent design constraints that limit the solution space. Specifically, the physical form factor of the persistent memory module limits how much logical-to-physical (L2P) address translation data can be stored on the device, and the translation algorithm must be extremely fast to enable memory access at DRAM-like latency. To operate within these constraints, the wear leveling algorithm cannot accurately account for the number of write cycles applied to every individual memory word, and so it must operate at a coarser granularity. Details of Intel's Optane persistent memory are not well documented, but it is known that these memory modules are internally organized into blocks of 256 bytes [5, 7]. Because of this, we speculate that wear leveling state is likely tracked on a per-block basis (or even more coarsely). While such a block-based wear leveling scheme could work effectively for workloads dominated by sequential writing, like storing append-only logs, it can lead to severe resource under-utilization in a scenario where a single memory word is repeatedly overwritten. This limitation is particularly relevant for a byte-addressable "write-in-place" storage medium like Intel's Optane memory, whereas a flash-based SSD's entire data block must always be erased before it can be overwritten.

This paper focuses on software-managed wear leveling for synchronization structures, such as shared counters, which generate precisely the kind of skewed memory access pattern that can delude a general-purpose embedded wear leveling solution. Building on the foundations established by Liu and Golab [6], we propose a novel software implementation of an atomic counter that internally harnesses together multiple words of persistent memory to distribute wear. Our implementation uses transactional memory in a new way and vastly outperforms Liu and Golab's algorithm, which is based on ordinary Compare-And-Swap.

2 The Wear Leveling Problem

For the purposes of this paper, wear leveling is the abstract problem of implementing a concurrent object that maintains correctness across many state changes while using base objects that may lose their correctness after relatively few state changes. Liu and Golab [6] formalized this notion as the following *endurance* property, where T can denote a constant or a function of some model-specific parameters like the number of concurrent threads:

▶ **Definition 1.** *An object has* endurance T *if it maintains its safety and liveness properties in all executions where at most T updates (i.e., operations other than reads) are invoked on the object, but not in some execution where $T + 1$ updates are invoked.*

In general, the endurance of an implemented object (e.g., one that is strictly linearizable [1, 2] and lock-free) is limited by the endurance of the base objects from which the implemented object is constructed. We focus in this work on *endurance-oblivious* [6] implementations that treat the endurance of the base objects as an unknown.

3 The Transactional Counter Algorithm

Building on the work of Liu and Golab [6], we seek improved implementations of the atomic counter, also known as a *Fetch-And-Increment* object, in the system-wide crash-recover failure model with persistent main memory and a volatile cache. The abstract state of a counter object is an integer, typically initialized to zero. The object supports a single operation that retrieves the current value of the counter and also increases the value by one. As an example, a strictly linearizable [1, 2] lock-free implementation of an atomic counter using the `FetchAndIncrement` instruction is presented in Figure 1. Although the implementation lacks wear leveling, it illustrates our syntax conventions and the correct use of persistence instructions to manage the volatile cache.[1] The linearization point of the Increment operation is the first (process-initiated or environment-initiated) flush step that persists either the value of the counter established at line 1 or a larger value.

Persistent shared variables:

- B: base object supporting `FetchAndIncrement` operation, initially 0

Procedure Increment().

```
1 ret := FetchAndIncrement(&B)
2 Persist(&B)
3 return ret
```

Figure 1 Baseline counter implementation.

Following [6], we partition the state of the counter across a collection of k base objects $B_0 \ldots B_{k-1}$ such that the value of the implemented object equals the sum of the values of the base objects. In theory, the endurance of the implemented object can be increased by a factor of k as long as two conditions are met. First, each implemented operation must correctly compute the fetched value, which amounts to obtaining a snapshot of the states of the base objects that appears to be atomic with respect to the increment. Second, each implemented operation must not only spread out wear evenly across the base objects, but also limit the number of updates applied to the base objects to avoid undesirable *write amplification*. Ideally, each implemented operation would increment only a single base object.

Both challenges are addressed by maintaining a particular state invariant over the base objects, as illustrated by way of example in Figure 2. The pattern is that the i'th Increment operation on the implemented object (counting starting at zero) updates base object number $\lfloor i/m \rfloor \mod k$, where m is a parameter we call the *bin size*. Intuitively, m increments are

[1] The ampersand symbol (&) means "address of" as in C/C++. `Persist` represents a process-initiated flush step on a base object that is assumed to fit inside a single cache line. It can be implemented on the Intel platform using the function `pmem_persist` in the Persistent Memory Development Kit [9].

state of implemented object	base object B_0	base object B_1	base object B_2	...	base object B_{k-1}
0	0	0	0	...	0
1	1	0	0	...	0
2	2	0	0	...	0
m	m	0	0	...	0
$m+1$	m	1	0	...	0
$2m$	m	m	0	...	0
km	m	m	m	...	m
$km+1$	$m+1$	m	m	...	m
$km+m+1$	$2m$	$m+1$	m	...	m

Figure 2 State representation of wear leveling counter for bin size m and k base objects.

applied to base object B_0, then m to B_1, ..., m to B_{k-1}, then m more to B_0, etc., in round-robin fashion. As long as the base objects have endurance T such that T is a multiple of m, the implemented object can count up to kT, which improves on the baseline technique from Figure 1 by a factor of k. Not only does this strategy amplify endurance, but it also expands the counter's domain of values by the same factor k.

The central technical challenge in maintaining our state invariant under concurrent access is to apply the correct state transition to the correct base object each time the implemented counter is accessed. This is a non-trivial task since the correct base object and state transition depend on the position of an Increment operation in the linearization order, which is not known to processes ahead of time. Liu and Golab [6] solved the problem for bin size $m = 1$ using an algorithm based on the `CompareAndSwap` instruction, which is lock-free but inefficient under high contention. We improve upon this preliminary design by replacing the `CompareAndSwap` instruction with `FetchAndIncrement`. The immediate problem this strategy presents is that incrementing a base object unconditionally can increase its value beyond the threshold permitted by the invariant presented earlier in Figure 2, which we rely on crucially to correctly compute the response of an Increment operation. For example, if more than m processes attempt to increment base object B_0 starting from the initial state, then the final value of B_0 will exceed the value of B_1 by more than m, which violates the invariant. We address this problem by encapsulating the `FetchAndIncrement` instruction in a hardware transaction, and aborting the transaction whenever the invariant is violated. Secondly, we optimize the selection of the base object by introducing static variables that allow processes to remember which object was last accessed.[2] Assuming that processes access the counter frequently and that the bin size m is large relative to the number of processes, this second optimization mostly avoids the costly linear search in Liu and Golab's algorithm.

We present pseudo-code for the algorithm in Figure 3, which borrows syntax from the GCC transactional memory intrinsics [4]. At the beginning, process p computes the boundary between the current bin and the next bin at line 4 based on its recollection of the current bin obtained from static variable bin_p. The transaction then starts at line 6 inside the outer while loop, and its current status is determined at line 7 using the `_xbegin` intrinsic. For the reader unfamiliar with the GCC transactional intrinsics, the algorithm should be interpreted

[2] A static variable retains its value across calls to Increment. Our algorithms do not persist such variables, and function correctly (albeit more slowly) even if the variables hold stale values after a crash.

Persistent shared variables:

- $B[0..(k-1)]$: array of base objects, each element initially 0

Private static variables:

- $index_p$: integer in the interval $[0, k)$, initially 0
- bin_p: integer ≥ 0, initially 0

Private variables:

- $limit_p, prev_p, status_p, bumped_p, temp_p$: integers

Procedure Increment().

```
4  limit_p := (bin_p + 1) × m
5  while true do
6      status_p := _xbegin()
7      if status_p = _XBEGIN_STARTED then
8          prev_p := FetchAndIncrement(&B[index_p])
9          if prev_p ≥ limit_p then
10             _xabort(_ABORT_BIN_EXCEEDED)
11         else
12             _xend()
13             Persist(&B[index_p])
14             return prev_p + m × (bin_p × (k − 1) + index_p)
15     else if _XABORT_CODE(status_p) = _ABORT_BIN_EXCEEDED then
16         bumped_p := false
17         while true do
18             temp_p := B[index_p]
19             if temp_p < limit_p then
20                 break
21             else
22                 index_p := (index_p + 1) mod k
23                 if index_p = 0 then
24                     bin_p := ⌊temp_p/m⌋
25                     limit_p := (bin_p + 1) × m
26                 bumped_p := true
27         if bumped_p then
28             Persist(&B[(index_p + k − 1) mod k])
```

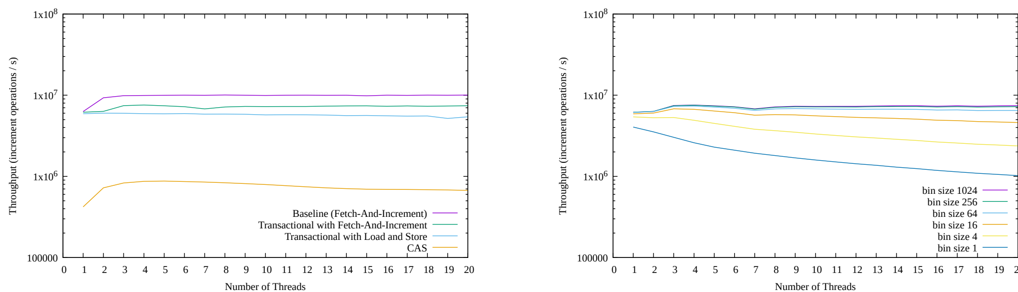
Figure 3 Endurance-oblivious counter implementation using hardware transactions and FetchAndIncrement. Pseudo-code shown for process p, k base objects, and bin size m.

as returning a successful status (_XBEGIN_STARTED) when line 7 is first executed in an iteration of the outer while loop. It then proceeds with the FetchAndIncrement instruction at line 8 and continues onward to the commit point at line 12 and beyond (lines 13–14), unless the transaction aborts. The latter can occur due to an explicit abort at line 10 via the _xabort intrinsic or due to a spontaneous abort, and in either case, the algorithm is rolled back to line 6 where _xbegin is re-executed and returns a special status code different from _XBEGIN_STARTED. The _XABORT_CODE intrinsic (a GCC macro) at line 15 determines the user-defined code (if any) passed to _xabort at line 10. If the transaction aborted

spontaneously then it is restarted at the next iteration of the while loop, otherwise the fallback execution path at lines 16–28 is executed to adjust the values of the static variables $index_p$ and bin_p, and another transaction is attempted.

4 Experiments

We implemented a collection of wear leveling counters in C++ and evaluated their performance on a 20-core Intel Xeon Gold 6230 platform with Optane persistent memory. The Intel Persistent Memory Development Kit (PMDK) [9] was used to access the Optane memory using memory-mapped files. The `Persist` operation featured in our pseudo-code was implemented using the `pmem_persist` function in the PMDK, which internally performs a cache line write-back (clwb) and store fence. Intel's Restricted Transactional Memory (RTM) was accessed using GCC intrinsics [4], which we explained earlier in Section 3. Persistent memory and hardware transactions are typically not used together as the transactions do not guarantee failure-atomicity, and persistence instructions inside a transaction can cause an abort on some platforms. However, the transaction used in our algorithm circumvents these drawbacks by accessing only a single memory word and persisting after committing.



(a) Comparison of counter implementations with cache line write-backs and store fences.

(b) Sensitivity of transactional `FetchAndIncrement` implementation to the bin size parameter.

Figure 4 Scalability experiments.

Figure 4a presents an experimental comparison of the baseline algorithm from Figure 1, our transactional counter algorithm from Figure 3, an alternative implementation of our algorithm that uses load and store instead of `FetchAndIncrement`, and the Liu-Golab algorithm (denoted CAS). The bin size parameter (m in Section 3) was 1 for the baseline and Liu-Golab algorithms, and 1024 for the two transactional algorithms. We observe that the transactional `FetchAndIncrement`-based algorithm is roughly 1.5× slower than the baseline, which lacks wear leveling, and outperforms the alternative transactional algorithm by roughly 2×. It also outperforms Liu-Golab by roughly 15×. Next, we consider the effect of the bin size on performance in Figure 4b, and find that a bin size of $256 \leq m \leq 1024$ works well.

References

1. Marcos K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.
2. Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

3 Frank Hady. Intel Optane technology delivers new levels of endurance, 2019. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html`.

4 Free Software Foundation Inc. Transactional memory intrinsics. [last accessed 5/01/2023]. URL: `https://gcc.gnu.org/onlinedocs/gcc/x86-transactional-memory-intrinsics.html`.

5 Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019. `arXiv:1903.05714`.

6 Xialin Liu and Wojciech Golab. Brief announcement: Towards a theory of wear leveling in persistent data structures. In *Proc. of the 41st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 220–223, 2022.

7 Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel Optane byte-addressable NVM. In *Proc. of the International Symposium on Memory Systems (MEMSYS)*, pages 304–315, 2019.

8 Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel SSD (what is it good for). In *In Proc. of the 10th Conference on Innovative Data Systems Research (CIDR)*, 2020.

9 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit. [last accessed 5/01/2023]. URL: `https://pmem.io/pmdk/`.