NVMW 2018

Dalí: A Periodically Persistent Hash Map

Faisal Nawab*‡, Joseph Izraelevitz†‡,
Charles B. Morrey III‡, Dhruva R. Chakrabarti‡, Michael L. Scott†

*University of California, Santa Cruz
fnawab@ucsc.edu

†University of Rochester
{jhi1,scott}@cs.rochester.edu

I. Introduction

In current real-world processors, instructions to control the ordering, timing, and granularity of writes-back from caches to NVM main memory are rather limited. Even in the best of circumstances, however, "persisting" an individual store (e.g., using CLFLUSH) and ordering it with respect to other stores (e.g., using MFENCE) is likely to take at least tens of cycles. Additionally, due to power constraints, we expect that writes into NVM will be guaranteed to be failure-atomic only at increments of eight bytes—not across a 64-byte cache line.

We use the term *incremental persistence* to refer to the strategy of persisting store w_1 before performing store w_2 whenever w_1 occurs before w_2 in the happens-before order of the program during normal execution. Given the expected latency of even an optimized persist, this strategy seems doomed to impose significant overhead on the operations of any data structure intended to survive program crashes.

As an alternative, this work introduces a strategy we refer to as *periodic persistence*. The key to this strategy is to design a data structure in such a way that modifications can safely leak into persistence *in any order*, removing the need to persist locations incrementally and explicitly as an operation progresses. To ensure that an operation's stores eventually become persistent, we periodically execute a *global fence* that forces all cached data to be written back to memory. The interval between global fences bounds the amount of work that can ever be lost in a crash (though some work may be lost). To avoid depending on the fine-grain ordering of writes-back, we arrange for "leaked" lines to be ignored by any recovery procedure that executes before a subsequent global fence. After the fence, however, a known set of cache lines will have been written back, making their contents safe to read. Like naive uninstrumented code, periodic persistence allows stores to persist out of order. It guarantees, however, that the recovery procedure will never use a value v from memory unless it can be sure that all values on which v depends have also safely persisted.

As an example of periodic persistence, we introduce Dalí,[1] a transactional hash map for nonvolatile memory. Dalí demonstrates the feasibility of using periodic persistence in a nontrivial way, and is provably correct under buffered durable linearizability [2], an extension of traditional linearizability that accommodates whole-system crashes. Experience with a prototype implementation confirms that Dalí can significantly outperform alternatives based on either incremental or traditional file-system-based persistence. Our prototype implements the global fence by flushing (writing back and invalidating) all on-chip caches. Performance results would presumably be even better with hardware support for whole-cache write-back without invalidation. Dalí was previously published at DISC'17 [4].

‡This work was supported in part by the US Department of Energy under Cooperative Agreement no. DESC0012199 while the indicated authors were members of Hewlett Packard Labs. At the University of Rochester, the work was supported in part by NSF grants CNS-1319417, CCF-1337224, and CCF-1422649, and by a Google Faculty Research award.

[1]The name is inspired by Dalí's painting *The Persistence of Memory*.

II. Dalí

Dalí is our prepend-only transactional hash map designed using periodic persistence. Dalí consists of an array of buckets, each of which points to a singly-linked list of *records*. Each record is a key-value pair. For the sake of simplicity, each list is prepend-only: records closer to the head are more recent.

Dalí uses a periodic global fence to guarantee that changes to the data structure have become persistent. We say that the initiation points of the global fences divide time into *epochs*, which are numbered monotonically from the beginning of time (the numbers do not reset after a crash). Each update is logically confined to a single epoch, and the fence whose initiation terminates epoch E serves to persist all updates that executed in E. The execution of the fence, however, may overlap the execution of updates in epoch $E+1$. As a result, in the absence of crashes, we are guaranteed during epoch $E+1$ that any update executed in epoch $E-1$ has persisted. If a crash occurs in epoch F, however, updates from epochs F and $F-1$ cannot be guaranteed to be persistent (they *failed*, and should therefore be ignored. Failed epochs are maintained in a persistent *failure list*, updated during the recovery procedure.

In Dalí, hash map records are classified according to their persistence status. Assume that we are in epoch E. *Committed records* are ones that were written in a non-failed epoch at or before epoch $E-2$. *In-flight records* are ones that were written in epoch $E-1$ if it is not a failed epoch. *Active records* are ones that were written during the current epoch E. Records that were written in a failed epoch are called *failed records*. By steering application threads around failed records, Dalí ensures consistency in the wake of a crash.

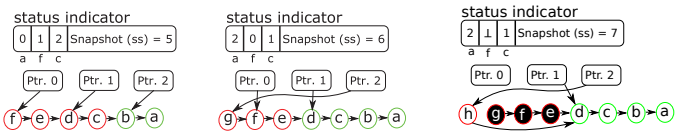
Dalí adds metadata to each bucket to track the persistence status of the bucket's records and to avoid persisting records incrementally. Specifically, a Dalí bucket contains not only a singly-linked list of records, but also a 64-bit *status indicator* and, in lieu of a head pointer for the list of records, a set of three *list pointers*. The status indicator comprises a *snapshot* (*SS*) field, denoting the epoch in which the most recent record was prepended to the bucket, and three 2-bit *role IDs*, which indicate the roles of the three list pointers (i.e. active, in-flight, or committed). A single STORE suffices to atomically update the status indicator on today's 64-bit machines.

Figure 1a shows an example bucket. In the figure *SS* is equal to 5, which means that the most recent record was prepended during epoch 5. The active pointer is Pointer 0. It points to record e, which means that e was added in epoch 5, even if we are reading the status indicator during a later epoch. Pointer 1 is the in-flight pointer, which makes d the most recently added record in epoch 4. Finally, Pointer 2 is the committed pointer. This makes record b the most recently added record before or during epoch 3. By transitivity, the earlier record a was also added before or during epoch 3. Both record b and the earlier record a are therefore guaranteed persistent (shown in green) as of the most recent update (the time at which e was added), while the remainder of the records may not be persistent (shown in red). It is important to note that the status indicator reflects the bucket's state *at SS* (the epoch of the most recent update to the bucket) even if a thread inspects the bucket during a later epoch.

Reads. A reader begins by using a hash function to identify the appropriate bucket for its key, and locks the bucket. It then consults the bucket's epoch number (*SS*) and the global failed epoch list to identify the most recent, yet valid, of the three potential pointers into the bucket's linked list (Figure 1a). We call this pointer the *valid head*.

Updates. Updates in Dalí prepend a new version of a record. Like the `read` method, `update` locks the bucket. An update to a Dalí bucket comprises several steps: (1) Determine the most recent, valid, head. (2) Create a new record with the key and its new value. (3) Determine the new pointer roles (if the new and old epochs are different). (4) Retarget the new active pointer to the new record node. (5) Update *SS* and the role IDs by overwriting the status indicator.

Step 3 is the most important part of the update algorithm, as it is the part that allows the update's component writes to be reordered. The problem to be addressed is the possibility that writes from neighboring epochs might be written back and become mixed in the persistent state. We might, for example, mix the snapshot indicator from the later epoch with the pointer values from the earlier epoch. Given any combination of update writes from bordering epochs, and an indication of epoch success or failure, the `read` procedure must find the valid head, and the list beyond that head must be persistent. In order to maintain this invariant, the new pointer roles are rotated based on their previous values and the success (or failure) of epochs SS and $SS - 1$.



(a) Initial state in epoch 5.

(b) Adding record g in epoch 6.

(c) Adding record h in epoch 7; epochs 5 and 6 have failed.

Fig. 1: A sequence of Dalí updates.

III. Experiments

We have implemented a prototype version of Dalí in C/C++. We implemented the global fence by exposing the privileged WBINVD instruction to user code. As a representative workload for a hash map, we chose the transactional version of the Yahoo! Cloud Serving Benchmark (YCSB) [1] where each transaction consists of 75% reads. We compare against: **Silo** [6] an in-memory database configured to use NVM for persistent storage; **FOEDUS** [3] an online transaction processing engine, explicitly designed for heterogeneous machines with both DRAM and NVM; and **IP**, an incrementally persistent hash map [5]. Figure 2 shows the transaction throughput of Dalí and the comparison systems while varying the number of worker threads from 1 to 60; transactions here comprise three reads and one write. Dalí achieves a throughput improvement of 2–3× over Silo and FOEDUS across the range of threads.

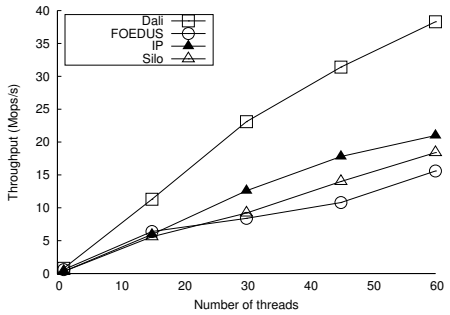


Fig. 2: Scalability (75% reads).

References

[1] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. *In: 1st ACM Symp. on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA, 2010.

[2] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. *In: 30th Intl. Conf. on Distributed Computing*. DISC '16. Paris, France, 2016.

[3] H. Kimura. Foedus: OLTP engine for a thousand cores and NVRAM. *In: 2015 ACM SIGMOD Intl. Conf. on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia, 2015.

[4] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. *In: 31st Intl. Symp. on Distributed Computing*. DISC '17. Vienna, Austria, 2017.

[5] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. *In: 3rd VLDB Wkshp. on In-Memory Data Mangement and Analytics*. IMDM '15. Kohala Coast, HI, USA, 2015.

[6] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *In: SOSP*. Farmington, PA, USA, 2013.