Persistent Memory in EOS Namespace: Designing Transactional Hash Tables for NVRAM

Elena Vasileva and Ryan Thompson

Department of Computer Science, University of California, Los Angeles (UCLA); and Department of Electrical and Computer Engineering, Carnegie Mellon University

Abstract

EOS[1] provides fast and reliable disk-only storage for data produced by the LHC experiments. In order to ensure fast access, EOS keeps a representation of the namespace in RAM along with a change-log file on disk. As a consequence, restoring the namespace from disk after a restart or a crash can take a relatively long time. Migration of the namespace to persistent memory (non-volatile RAM or NVRAM) will ensure persistence of data even in the event of a sudden power cut, as well as a considerable improvement on boot-up time. However, this migration requires that changes to the memory are done in a *transactional* fashion; in the event of a crash, we would like to recover with a consistent view of the namespace.

We present a hashtable that can be used to store contents persistently and transactionally, for use with the Mnemosyne[2] toolchain. To benchmark and validate this hashtable, an extensible benchmarking tool was written. We show that, outside Mnemosyne, our hashtable has a performance similar to the implementation currently in use. We furthermore integrate our own hashtable into the EOS code base.

1

2

1 Introduction

The experiments involved in the LHC make use of EOS to store data. To ensure that users are able to find files quickly, a representation of the file metadata, called the *namespace*, is kept in RAM. When a node containing such metadata crashes, this in-memory representation needs to be reconstructed using the modification log written to disk, taking on the order of tens of minutes to recover.

In an effort to address this problem, CERN openlab is researching the possibility of moving the namespace to persistent memory, in collaboration with the Data Storage Institute[3] (DSI) in Singapore, represented by Sergio Ruocco and Le Duy Khanh [1]. Using persistent memory, one could instantly recover the namespace state upon boot. However, this also entails that the memory needs to be updated in a transactional fashion, in order to ensure that the recovered view of the namespace is consistent even if the program aborts while an update is in progress. Throughout this report, we assume a persistent memory implementation similar to that of Mnemosyne [2], although a proposal to integrate transactional memory into the C++ language exists [3]. In particular, we assume that updates to 64-bit aligned memory positions can occur in an atomic fashion.

Since a significant part of the EOS namespace is stored using hashtables, we implemented a hashtable that was amenable to use with Mnemosyne. Furthermore, a benchmark tool had to be written to compare the performance of the system currently in use versus a hypothetical implementation that uses persistent memory with transactions implemented in software. We report on an extensible benchmarking tool, which can be used to assess the possible migration. Furthermore, we describe the changes necessary to integrate our hashtable into EOS.

The remainder of this report is organised as follows. In Section 2 we introduce the programming model for persistent memory. We subsequently review the hashtable implementations that can be benchmarked by the tool in Section 3. The tool itself is presented in Section 4, along with some measurements. We discuss the changes necessary to integrate the hashtable into EOS in Section 5. Lastly, we finish with some suggestions for future work in Section 6.

2 Persistent memory

To use persistent memory, one needs software support. First of all, a mechanism is needed to indicate that memory ought to be allocated persistently. Moreover, to ensure consistency, updates to such memory need to be done in a transactional fashion. Mnemosyne provides instrumentation for these in its toolchain. This instrumentation includes[4] the following.

[3] http://www.a-star.edu.sg/dsi/

[4] Originally, support for a `persistent` keyword, which would allow a programmer to label pointers that point into persistent memory, was planned. It would function like `const` in that it would not change the compiler's binary output, but rather serve as a tool to find cases where persistent

- Blocks with the keyword `atomic` give transactional control over updates to memory. The actions on persistent memory within these blocks are performed either as a whole or not at all. Doing so makes sure that memory will be consistent even in the event of a shutdown while an atomic block is being executed.
- The functions `pmalloc` and `pfree` are variants of `malloc` and `free` that allocate and deallocate memory on the persistent heap. Naturally, the state of the memory allocator is stored persistently as well.
- The `pstatic`[5] keyword instructs the linker to place these static variables in the persistent area of the process memory map. This keyword can be used for pointers that function as an entry point to persistent memory.

3 Hashtable implementations

In this section, we will briefly review the hashtables under consideration in the benchmark tool. We included the hashtable implementations currently in use in EOS, namely `google::dense_hash_map` [4] and `google::sparse_hash_map` [4]. For reference, we also included an interface to `std::map`. Additionally, a simple hashtable implementation (dubbed `PersistentHashtable`) was written that is easily portable to NVRAM-backed storage with software transactional memory, such as Mnemosyne.

3.1 Google's dense hash map

Google's dense hash map, part of the SparseHash package, claims to be a fast hashtable implementation with approximately 78% memory overhead. It uses an array of buckets that are either full or empty (marked with a special value 'stolen' from the key space). In case of collision, it uses quadratic probing to search for the next empty slot: first, the slot at offset 1 is probed, then offset 3, then 6, then 10, et cetera. The implementation is aimed to have modification complexities of approximately $O(1)$.

The dense hash map allocates a chunk of memory statically. When it notices that approximately 80% of buckets are full, it will opt to resize, re-inserting all entries into a fresh hash map of larger size. When growing the table, the size is usually doubled. This resizing may cause a slight speed bump if one wants to grow the table by inserting many entries without allocating the appropriate amount of space first.

3.2 Google's sparse hash map

Google's sparse hash map is also part of the SparseHash package. The way it works is very similar to the dense hash map described above, except that it uses a *sparse* array to store the buckets. The sparse array is intended to eliminate large consecutive swaths of unused buckets. Therefore, the sparse hash map ought to be more memory-efficient at the expense of speed.

memory is expected but not provided. This feature, however, did not make it to the Mnemosyne release.

[5] Found in the final release of Mnemosyne as `MNEMOSYNE_PERSISTENT`.

3.3 The standard library implementation

In the GNU implementation of the standard template library (STL), the `std::map` container is a red/black tree [5] [6]. Entries are retrieved by finding the node in the tree associated to the key value. After insertion, the nodes are re-painted and rotated to restore the balance of the tree. By virtue of this additional structure, the tree remains (approximately) balanced, ensuring worst-case insertion, deletion and lookup complexities of $O(log\ log\ n)$. Updating the tree after modification can also be done within $O(log\ log\ n)$.

The use of the red/black tree means that `std::map` is not strictly a hashtable, as entries are not ordered based on their hashed value but rather on the total order of their key type. This approach, however, does mean that one can iterate over the entries in order of their key quite easily.

3.4 Our own implementation

To compare against NVRAM, we implemented a simple hashtable that combines the characteristics of the hashtables described above. Our `PersistentHashtable` contains an array of buckets, indexed by their hash value. Every bucket is the root of an AVL tree [7], which (like the red/black tree) is a self-balancing binary tree with logarithmic complexity for insertion, deletion and lookup. All keys that hash to the same value are inserted into the AVL tree at the corresponding bucket.

Our implementation makes use of the `pmalloc` and `pfree` primitives in Mnemosyne to allocate persistent memory for storage. It copies the given keys and values (using an assumed copy constructor) for persistent storage. The consuming code is tasked with ensuring that all pointers within that data also point to persistent memory.

Though not impossible, `PersistentHashtable` does not contain a facility to resize to fewer or more buckets. Instead, we trust the consuming code to give a reasonable estimate of the necessary amount needed for a typical instance of the hashtable.

Except for the constructor, the API of PersistentHashtable is designed to be exactly like that of `std::map` and `google::dense_hash_map`. In doing so, we hoped to make as much of a drop-in replacement as possible for the maps currently in use.

Because CERN does not yet have a copy of the DSI's version of the Mnemosyne toolkit, we did not test our implementation. However, Sergio Ruocco and Le Duy Khanh (of DSI) have recently succeeded in modifying the current implementation to be compatible with Mnemosyne and recover persistent memory between program runs.

4 Benchmarks

4.1 Benchmarking tool

The benchmarking tool consists of two parts. The first part is a small utility (`genlogfile`) that can generate a synthetic workload for the hashtable based on a brief description format in a configuration file. One can choose the number of entries to reserve in the hashtable, as well as the number and type of operations to generate. The workload is written in a concise binary format.

The second part of the benchmarking tool simulates the log file on the hashtable of the user's choosing. This interface is loaded from a shared library file using `dlopen`. After the simulation is complete, the code outputs the elapsed time (both in kernel- and in user space) as well as the memory consumption at the end of the program.

In case of `PersistentHashtable`, the tool can optionally output a CRC32 fingerprint of the memory state after each operation. This mode of operation is only intended to verify the atomicity of operations and significantly slows down the benchmarks.

This benchmarking tool, along with the code for `PersistentHashtable`, is available today in the `dss/map-benchmarks` repository on GitLab.

4.2 Speed

As a preliminary test, we plotted the results of the benchmark tool for a workload[6] where N entries are inserted into the hashtable, followed by the deletion of those N entries (in arbitrary order). An entry consisted of a key-value pair, both strings, of lengths respectively 100 and 1000 characters. We let N vary from 1000 to 100000 with steps of 1000 and averaged the total elapsed time of 50 runs.

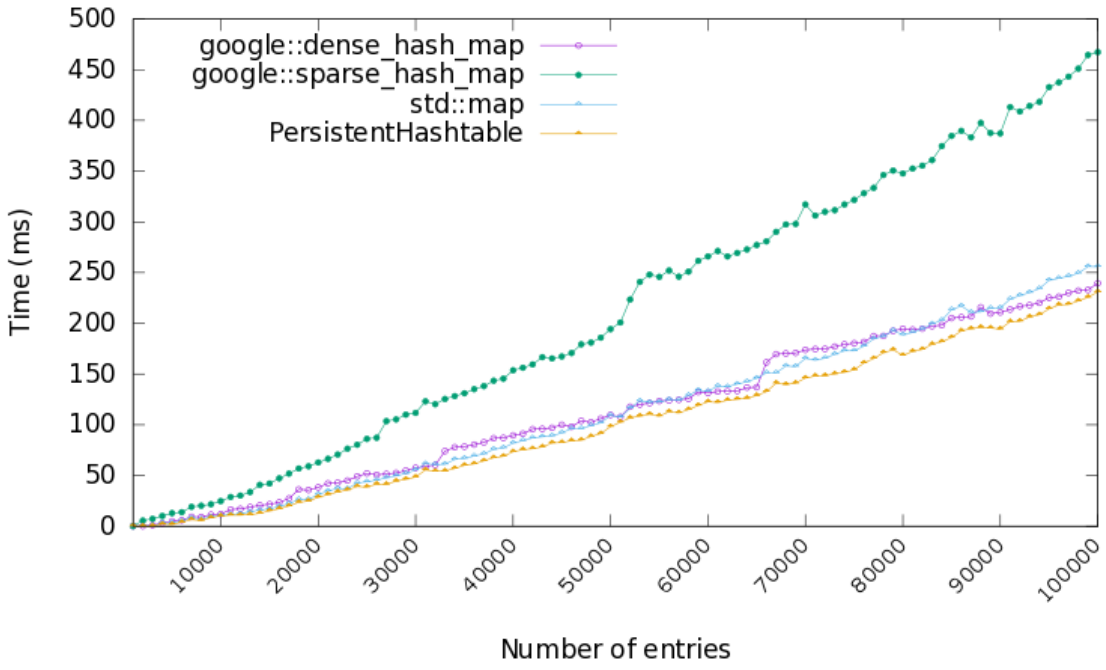

Figure 1: Speed of the hashtable implementations.

[6] It should be noted that this is by no means intended as a representative workload for the hashtables used within EOS. Instead, one can regard this as a first check to see if our new implementation has feasible performance at all. Indeed, for actual integration into EOS it would be a good idea to run the benchmark with a more representative workload.

The results are plotted in Figure 1. We see that Google's dense hash map outperforms `std::map`, as well as the noticeable steep inclines around powers of two for `google::dense_hash_map`. When subjected to the workload under consideration, `PersistentHashtable` appears to perform similar to the hashtables currently in use, *for now* (that is, without Mnemosyne's software transactional memory instrumentation in place).

4.3 Memory usage

As for memory consumption, we performed a similar benchmark. Here, N entries (of the same type and size as the previous benchmark) were inserted into the hashtable, after which the memory consumption was measured. Since memory usage is deterministic, this measurement was only performed once for each N.

The results can be found in Figure 2. In it, we see that `PersistentHashtable` uses roughly 7% more memory to store its entries. This is most likely due to the overhead necessary to store the balance factors and pointers of the AVL tree.

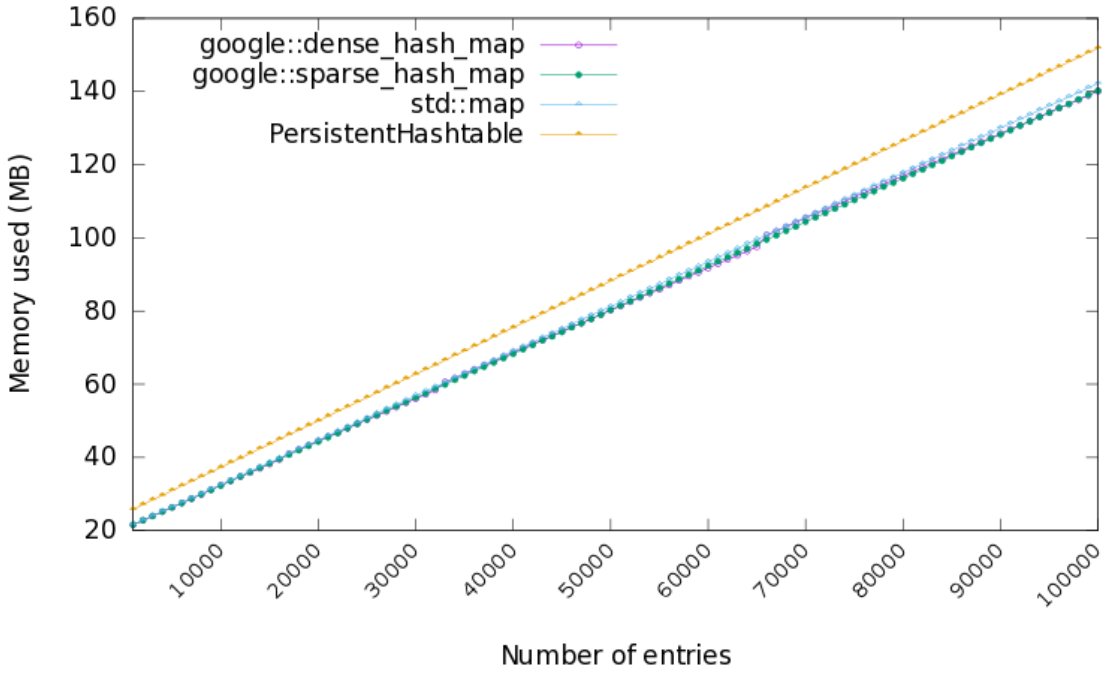

Figure 2: Memory consumption of the hashtable implementations.

5 EOS integration

In an effort to work towards actually moving the EOS namespace into persistent memory, we made an attempt at integrating `PersistentHashtable` into the EOS code base. In the process, a number of bugs and API mismatches in `PersistentHashtable` were resolved. The following instances of `google::dense_hash_map` were replaced by `PersistentHashtable`:

1. `pIdMap` in `ChangeLogContainerMDSvc`.

2. `pSubContainers` in `ContainerMD`
3. `pFiles` in `ContainerMD`

Moreover, `pIdMap` was changed to be a pointer to an instance of `PersistentHashtable`. In doing so, future adaptations of the code can first check if the namespace is already present in persistent RAM and restore it using some `pstatic` pointer, or opt to proceed with booting the namespace from serialized disk content as before.

The changes described are currently available in the `beryl_aquamarine_nvram` branch of the EOS repository.

6 Future work

Further integration of persistent memory into EOS can be broken up into two phases, both of which are detailed in this section.

6.1 First phase: Tooling

The current Mnemosyne compiler as used by DSI makes use of relatively old versions of the GNU Compiler Collection (`gcc`) as well as the Intel C++ Compiler (`icc`). For EOS to actually make use of persistent memory, Mnemosyne will have to be ported to use the most recent versions of both compilers.

Moreover, the EOS namespace is not limited to using hashtables. For instance, it also uses `std::string` to store filenames and paths. As such, more data structures will have to be implemented or modified to use transactions and allocate all of their (relevant) memory in the persistent heap. For this, a library of generic, transactional and persistent data structures that mimic the API of the standard template library could be very useful.

6.2 Second phase: Migration

Provided that all necessary utensils for using persistent memory are in place, the second phase would first involve deciding which parts of EOS memory ought to be stored persistently. Then, the types of the variables involved will have to be modified to the persistent versions. This migration might seem trivial, but bear in mind that there should be no pointers to volatile memory from these data structures. As an alternative, one might also consider the option of identifying parts of the namespace that are stored redundantly and can be quickly recalculated upon recovery.

Lastly, EOS will have to be aware of persistency in the sense that its boot-up code will need to detect whether a namespace is still present in persistent memory. If so, the namespace will need to be restored and possibly validated; if not, the namespace will need to be restored from disk as is currently the case.

7 Bibliography

[1] S. Ruocco and D. K. Le, "Efficient Persistence of Financial Transactions in NVM-based Cloud Data Centers," *Proceedings of International Conference on Cloud Computing Research and Innovation (to appear),* 2015.

[2] H. Volos, A. J. Tack and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGPLAN Notices,* vol. 46, pp. 91-104, 2011.

[3] H. Boehm, J. Gottschlich, J. Maurer and others, "Transactional Memory Support for C++," 2014.

[4] C. Silverstein, D. Hide and G. Pike, "Sparsehash: an extremely memory-efficient hash_map implementation," 2012. [Online]. Available: http://code.google.com/p/sparsehash.

[5] R. Bayer, "Symmetric binary B-Trees: Data structure and maintenance algorithms," *Acta Informatica,* vol. 1, no. 4, pp. 290-306, 1972.

[6] L. J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proceedings of the 19th Annual Symposium on Foundations of Computer Science,* pp. 8-21, 1978.

[7] G. Adelson-Velsky and E. M. Landis, "An algorithm for the organisation of information," *Proceedings of the USSR Academy of Sciences,* vol. 146, pp. 263-266, 1962.