Vol.2015-HPC-150 No.25
2015/8/5

Design of concurrent B+Tree index for native KVS on Non-Volatile-Memories (Unrefereed Workshop Manuscript)

Mohamed Amin Jabri[1] Osamu Tatebe[2]

Abstract: We present our preliminary results about our design of a highly concurrent B+Tree index for Key-Value Stores (KVS) that are natively running on Non-Volatile-Memories (NVM). This work is beneficial in providing and enabling range-search capabilities to KVS natively running on flash devices like Fusion-io's NVMKV.

1. Introduction

In this work we present our preliminary results about our design of a highly concurrent B+Tree index for Key-Value Stores (KVS) that are natively running on Non-Volatile-Memories (NVM). This work is beneficial in providing and enabling range-search capabilities to KVS natively running on flash devices like Fusion-io's NVMKV [1].

In the reminder of this paper, we present, first in section 2, our design for a highly concurrent B+Tree index enabling range-queries on hash-based KVS running natively on flash. Then, in section 3, we provide an evaluation of our B+tree index design.

2. Design

In this section, we describe our design for a highly concurrent B+Tree index, which enable range queries and prefix-search features on hash-based key-value Store running natively on flash like Fusionios NVMKV key-value store.

As depicted in **Fig. 1**, our B+Tree index leverages NVMKV API [2] and use them as an underlying layer to do manage, persist and retrieve data from the underlying flash device. Our B+Tree could be thought as an in-memory tree-based KVS except that each key after being inserted into the index is persisted into the flash device through a transparent NVMKV calls. Keys and the corresponding metadata about the key-value pair are also stored in the leaf nodes of the B+Tree index. Each node of the tree index, except the tree's root node, contains a number of entries ranging from a minimum threshold named min-order, below which the corresponding tree node have to be merged with a neighboring node located at the same height, and a maximum threshold named branching-factor (generally equal to $2 * min - order$), above which the tree node have to be split in two different new node.

In general, key insertion will cause full (having a number of entries greater or equal to the branching factor of the tree) leaf nodes to be split, which in turn will lead to an additional entry insertion in the parent node. This latter, in turn, will need to be split if the additional entry insertion causes an overflow (entry number is greater or equal to the branching factor of the tree). Thus key insertion could trigger a node split operation, which could propagate from the bottom of the tree up to the tree root. Similarly key deletion will cause under-full nodes (entry count below the min-order of the tree) to be merged with a neighboring node. As in case of key insertion, tree nodes merging could also propagates from the bottom of the tree where the actual key-value pair deletion is performed through the intermediate internal nodes up to the root of the tree.

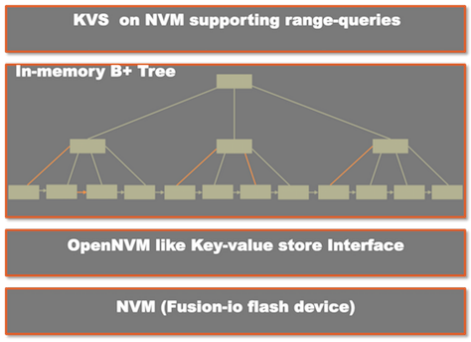

Fig. 1 B+tree in-memory index atop native flash running KVS architecture

Given that such a balancing operations (split/merge) are very costly, our design tries to minimize them by having at most one split operation or merge operation per key insertion or deletion. To achieve this goal, our B+Tree index uses a pro-active top-down approach for keeping the tree structure balanced all the time. Each time a key-value pair is inserted or deleted from the index, as we traverse the tree is starting from its root node in search for appropriate leaf node in which the key would be inserted into or from which the key entry is to be deleted, we check

[1] Center for Computational Sciences, University of Tsukuba
[2] Faculty of Engineering, Information and Systems, University of Tsukuba

the encountered intermediate internal node on the traversed path and proactively split or merge those nodes whose entry count is close to the branching factor (for a split) or the min-order (for a merge).

Search operations and range queries are not delayed when a concurrent key-value pair insertion or deletion causes the tree to perform a rebalancing operation (nodes split or merge). Only tree structure modifying operations (insert or delete) are delayed when the corresponding tree traversal accesses a node that is being split or merged. Additionally, for each range query, the tree structure is traversed only once. This is because all the leaf nodes in our Tree index are chained together so as to form an ordered linked list with all the keys entries.

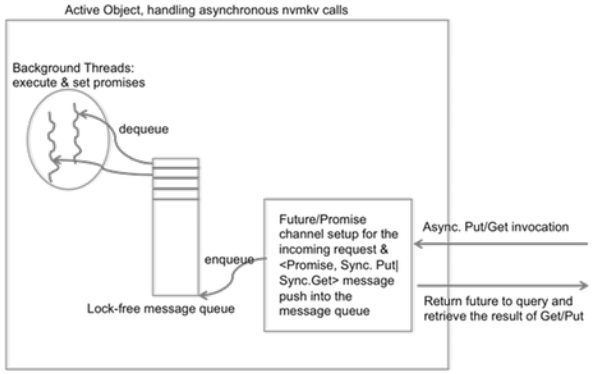

Fig. 2 Asynchronous NVMKV calls design principle: turning synchronous calls into asynchronous calls at the expense of occupying some threads.

As NVMKV API calls are synchronous and blocking for the calling thread, calls to our B+tree index needs also to block as we transparently issue NVMKV calls for data persistence on the underlying flash device. To maximize the concurrency of our B+tree we design an asynchronous non-blocking equivalent calls for NVMKV with respect to the calling thread.

Our design for the asynchronous equivalent of NVMKV calls, leverage the new *c++11/c++14* concurrency and asynchrony tools: *std::future*, *std::promise* and *std::async*. A *std::future* and *std::promise* constitute a communication channel in which *std::promise* is used to set and write the result of a given function call whereas a *std::future* is the reading or retrieving end of the communication channel. A *std::async* executes a given function call concurrently with a calling thread. The idea behind our asynchronous NVMKV calls, as depicted in **Fig. 2**, stems from the fact that we could turn any synchronous blocking call into an asynchronous non blocking call at the cost of occupying a thread. Our asynchronous equivalent calls for NVMKV wraps the blocking calls into a function object that will be put in a lock-free message queue and returns a *std::future* immediately to the calling thread. A set of background threads keeps popping any available function objects from the message queue and executes them in the background. When the result of the execution is ready, the background thread executing the function object uses the *std::promise* corresponding to the *std::future* given to the calling thread when the object was queued in the message queue, to write the result. At this time only, the *std:future* becomes ready and the calling thread could retrieve the result of the execution. This principle is depicted in Fig. 2.

3. Evaluation

In this section, we will provide an evaluation to both: our preliminary design for in-memory concurrent B+tree index which would enable range-queries and prefix search features on NVMKV key-value store, and our wrapper around NVMKV synchronous and blocking calls providing an asynchronous and non-blocking interface for the calling thread.

Our benchmark consists of a stress test in which we execute 10000 key-value pair insertions followed by 10000 random key-value retrieval. The Key size is fixed in our experiment to 40 bytes, which is the secure hash (sha-1) of integers ranging from 1 to 10000. The corresponding value is equal to the 40-bytes with a padding to make the values size multiple of a sector size (512 bytes). In our experiment, we measure the latency for each of the individual 10000 calls.

In a first experiment, we compare key-value pair insertion and retrievable with respect to size of the value varying from 1 sector up to 1024 sectors using NVMKV calls only (without our B+Tree index) as a baseline and then using our in-memory B+Tree index on top of NVMKV. In this setting, our tree index has minimum order of 128. **Fig. 3** depicts the overage latencies over a10000 operations with respect to the value size, whereas in **Fig. 5** we take a closer look on how the latency evolve with respect to each individual iterations. From those two graphs, we could say that our B+tree index introduces a very little overhead on top of NVMKV calls.

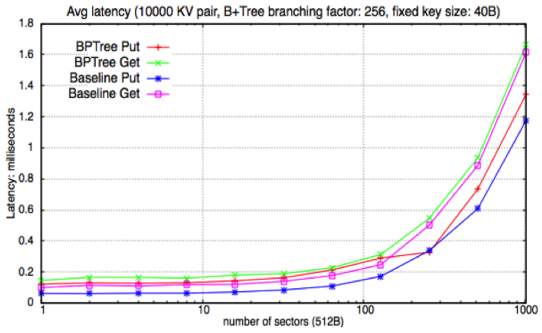

Fig. 3 Average key-value pair Insertion/retrieval overhead with respect to the value size in number of sectors: using our B+tree index versus the baseline NVMKV calls.

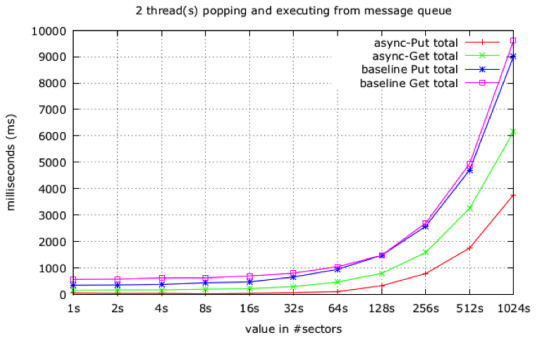

Fig. 4 Asynchronous NVMKV calls (AsyncPut/AsynGet) and synchronous NVMKV calls (Put/Get) overhead with respect to the value size in number of sectors.

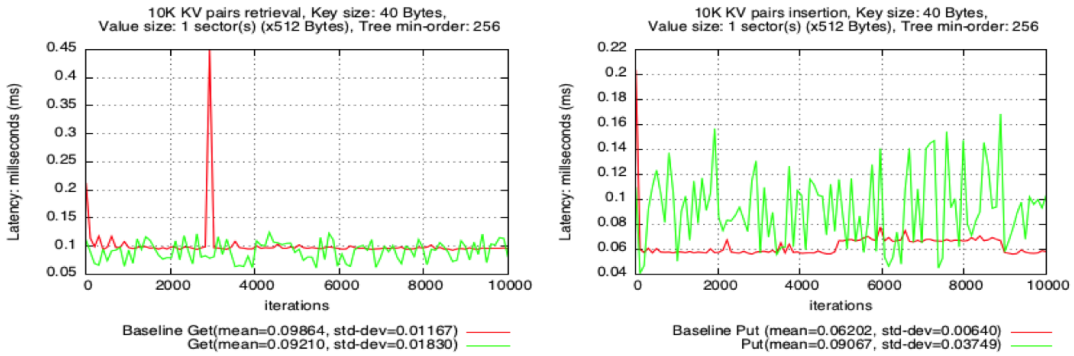

Fig. 5 Detailed timing for 10000 key-value pair insertion and retrieval: using our B+tree versus the baseline NVMKV calls.

In a second experiment, we try to compare our asynchronous wrapper interface around NVMKV to their synchronous and blocking equivalents. For the asynchronous calls, first, we issue 10000 asynchronous call and measure their latency (time needed for the call to return an *std::future*), and then at an immediately following stage we measure the time needed for each of the 10000 *std::future* to become ready and get their result. **Fig. 4** shows the average total latency, which includes the latency for the asynchronous calls to return an *std::future* and the waiting time required for the latter to become ready and deliver its result, with respect to the used value size.

As shown in Fig. 4, our asynchronous wrapper around NVMKV calls outperforms their synchronous equivalent for all the different value sizes. **Fig. 6** compares the individual latencies needed for each of the returned 10000 *std::future* to become ready with the baseline NVMKV synchronous calls, when the key-value pair's value size is two sectors long.

In a last experiment, we compare the overhead of our B+Tree index when using our asynchronous wrapper for NVMKV calls on one hand and our B+Tree index when using the synchronous and blocking NVMKV calls on the other hand.

Fig. 7 and **Fig. 8** depict the detailed timing, when the value size is fixed to two sectors long, for a 10000 Key-value pair insertion and retrieval for both cases: the case where we use our B+Tree index with our asynchronous NVMKV (AsyncPut/AsyncGet) calls in the underlying layer and the case where we use the synchronous and blocking (Put/Get) calls in the underlying layer.

4. Conclusion

In this paper, we presented our design for a highly concurrent in-memory B+Tree index that will be used as an intermediate layer on top of hash-based key-value stores running natively on flash, like fusionio's ioMemory devices, to enable prefix search and range-queries. Also, we presented our wrapper around NVMKV API calls, which turn them into asynchronous call for the calling thread. Additionally, we provided an evaluation of our in-memory B+tree index and our asynchronous non-blocking wrapper around NVMKV calls.

References

[1] Mármol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B. and Ganesan, S.: NVMKV: a scalable and lightweight flash aware key-value store.
[2] NVMKV: NVM key-value store API library, Fusionio (online), available from ⟨https://github.com/opennvm/nvmkv⟩ (accessed 2015-06-27).

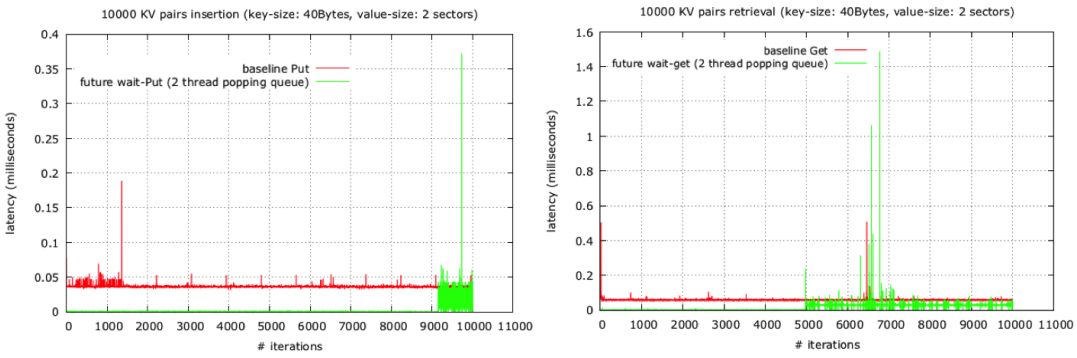

Fig. 6 Detailed timing for 10000 key-value pair insertion and retrieval using our asynchronous NVMKV calls versus the baseline synchronous NVMKV calls.

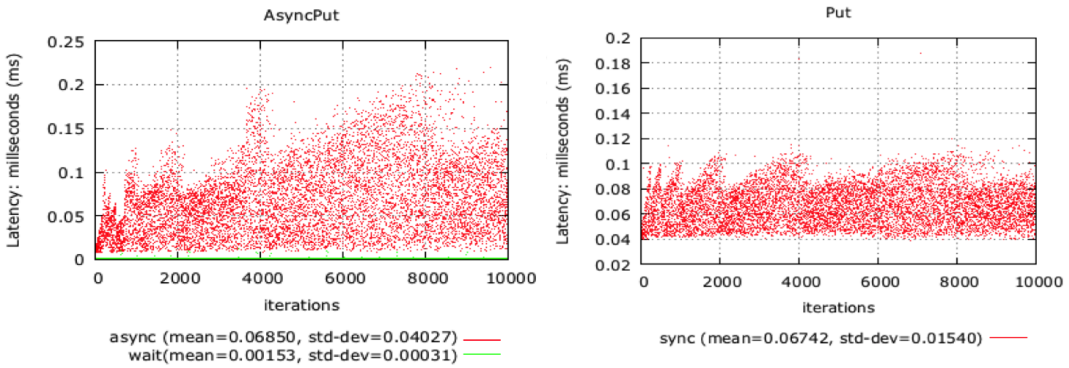

Fig. 7 Detailed timing for 10000 key-value pair insertion using our B+tree. AsyncPut uses our NVMKV asynchronous calls whereas Put uses only the synchronous NVMKV calls.

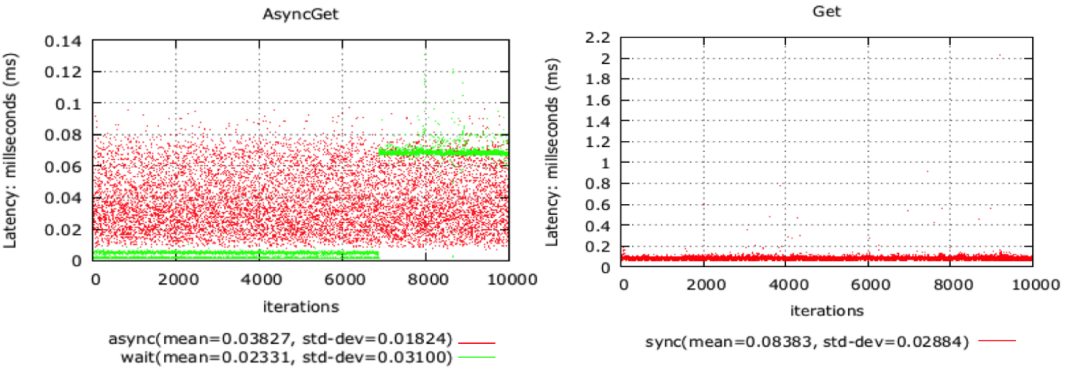

Fig. 8 Detailed timing for 10000 key-value pair retrieval using our B+tree. AsyncGet uses our NVMKV asynchronous calls whereas Gett uses only the synchronous NVMKV calls.