MapperX: Adaptive Metadata Maintenance for Fast Crash Recovery of DM-Cache Based Hybrid Storage Devices

Lujia Yin, *NUDT;* Li Wang, *Didi Chuxing;* Yiming Zhang, *NiceX Lab, NUDT;* Yuxing Peng, *NUDT*

https://www.usenix.org/conference/atc21/presentation/yin

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

MapperX: Adaptive Metadata Maintenance for Fast Crash Recovery of DM-Cache Based Hybrid Storage Devices

Lujia Yin
ylj1992nudt@gmail.com
NUDT

Li Wang
laurence.liwang@gmail.com
Didi Chuxing

Yiming Zhang
zym@nicexlab.com (Corresponding)
NiceX Lab, NUDT

Yuxing Peng
pengyuxing@aliyun.com
NUDT

Abstract

DM-cache is a component of the device mapper of Linux kernel, which has been widely used to map SSDs and HDDs onto higher-level virtual block devices that take fast SSDs as a cache for slow HDDs to achieve high I/O performance at low monetary cost. While enjoying the benefit of persistent caching where SSDs accelerate normal I/O without affecting durability, the current design of DM-cache suffers from long crash recovery times (at the scale of hours) and low availability. This is because its metadata of dirty bits has to be *asynchronously* persisted for high I/O performance, which consequently causes *all* cached data on SSDs to be assumed dirty and to be recovered after the system is restarted.

This paper presents MapperX, a novel extension to DM-cache that uses an on-disk adaptive bit-tree (ABT) to *synchronously* maintain the metadata of dirty bits in a hierarchical manner. Leveraging spatial locality of block writes, MapperX achieves controlled metadata persistence overhead with fast crash recovery by adaptively adding/deleting leaves in the ABT where different levels represent the states of blocks with different granularity. We have implemented MapperX for Linux DM-cache module. Experimental results show that the MapperX based hybrid storage device outperforms the original DM-cache based hybrid device by orders of magnitude in crash recovery times while only introducing negligible metadata persistence overhead.

1 Introduction

SSDs (solid state drives) are preferable to HDDs (hard disk drives) in building cloud storage systems [11, 31, 33, 35, 43, 44, 51, 65, 66] as SSDs significantly outperform HDDs in random small I/O [13, 14, 26, 27, 40] which is dominant in the cloud [44, 48]. Since currently SSDs are still much more expensive than HDDs, in recent years we see a trend of adopting HDD-SSD hybrid storage for high I/O performance at low monetary cost. For instance, URSA [38] is a distributed block storage system which stores primary replicas on SSDs and replicates backup replicas on HDDs; and SSHD [28] integrates a small SSD inside a large HDD of which the SSD acts as a cache.

As the demand of HDD-SSD hybrid storage increases, Linux kernel has supported users to combine HDDs and SSDs to jointly provide virtual block storage service. DM-cache [5] is a component of the device mapper [4] in the kernel, which has been widely used in industry to map SSDs and HDDs onto higher-level virtual block devices that take fast SSDs as a cache for slow HDDs. DM-cache records the mapping between SSDs and HDDs for each cached block in its metadata. When DM-cache adopts the default `writeback` mode, a block write will go only to the SSD cache and get acknowledged after being marked dirty, so that the dirty block could be *demoted* from the SSD cache to the HDD later in a batch for accelerating random small I/O. Linux kernel also provides other modules (Bcache [3] and Flashcache [9]) which have similar functionalities with DM-cache.

While enjoying the performance benefit of persistent caching without affecting data durability [68], in the current design of DM-cache and its variations the metadata of dirty bits has to be *asynchronously* persisted (with a hard-coded period of one second), otherwise the synchronous update overhead of the metadata for each write would be overwhelming. Unfortunately, asynchronous metadata update causes *all* cached data on SSDs to be assumed dirty once the system crashes and gets restarted, which results in long crash recovery times and consequently low availability. For example, in our production storage cluster we use DM-cache to combine SSDs with HDDs for hybrid block storage, and it will take more than two hours (depending on the locality of workloads) to recover from a crash even if most cached blocks are clean. The low availability caused by asynchronous metadata update prevents Linux kernel's hybrid storage mechanisms from being widely applied in availability-sensitive scenarios.

To address this problem, in this paper we present MapperX, a novel extension to DM-cache that uses an on-disk adaptive bit-tree (ABT) to *synchronously* maintain the metadata of dirty bits in a hierarchical manner. Workloads in the cloud usually have adequate write locality [36, 38, 48], which can be exploited to use one bit to represent the state of a range of consecutive blocks and thus effectively reduce the number of actual persistence operations for synchronous metadata update. Leveraging spatial locality of block writes, MapperX achieves controlled metadata persistence overhead with fast crash recovery by adaptively adding/deleting leaves at different levels in the ABT, which represent the states of blocks with different granularity.

We have implemented MapperX for Linux DM-cache module. Experimental results show that for workloads with certain localities the MapperX based hybrid storage device outperforms the original DM-cache based hybrid device by orders of magnitude in crash recovery times while only introducing negligible metadata persistence overhead.

The rest of this paper is organized as follows. Section 2 introduces the background and problem of DM-cache. Section 3 presents the design of MapperX. Section 4 evaluates the performance of MapperX and compares it with the original DM-cache. Section 5 discusses related work. And finally Section 6 concludes the paper.

2 Background

2.1 DM-Cache Overview

DM-cache is a component of the Linux kernel's device mapper, a volume management framework that allows various mappings to be created between physical and virtual block devices. DM-cache allows one or more fast flash-based SSDs (cache devices) to act as a cache for one or more slower mechanical HDDs (origin devices). In this section we briefly introduce the basic caching mechanism of DM-cache.

Like most cache solutions, DM-cache has three operating modes, namely, `writeback`, `writethrough`, and `passthrough`, among which only the default `writeback` mode can accelerate small writes (by asynchronously persisting SSD-cached data to HDDs). In this paper we focus on DM-cache's `writeback` mode. Linux provides DM-cache with various (plug-in) cache policy modules, such as multi-queue (MQ) and stochastic multi-queue (SMQ), to determine which blocks (and when) should be migrated from an HDD to an SSD (a.k.a. *promoted*) or from an SSD to an HDD (a.k.a. *demoted*). The cache policy is orthogonal to this paper, and we simply adopt the default SMQ policy which performs the best for most workloads.

DM-cache can use either the SSD cache device or a separate metadata device to store its metadata, which includes the mapping (between the cached blocks on SSDs and the original blocks on HDDs) and the dirty bits, as well as other policy-related metadata such as per-block hit counts. DM-cache adopts a fixed (but configurable before cache creation)

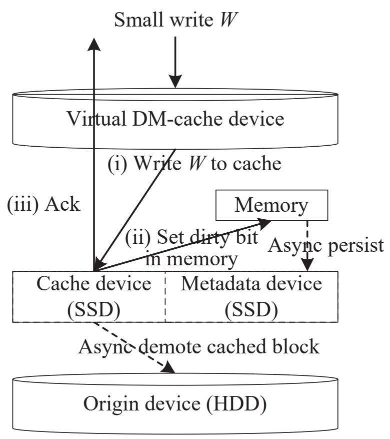

Figure 1: DM-cache maps HDDs and SSDs onto higher level virtual block devices.

block size typically between 32KB and 1MB.

Consider a small write to a virtual DM-cache device in the `writeback` mode.

If the target block is already in the SSD cache, then as shown in Fig. 1, (i) the new data (W) is written to the SSD, (ii) the corresponding bit of the block is set dirty in memory, and (iii) the write is acknowledged. The cached block will be asynchronously persisted to the HDD according to the cache policy. There are two kinds of metadata: the metadata for the cached block's mapping (between SSD and HDD) already exists and thus needs no persistence; and the metadata for the block's dirty bit has to be *asynchronously* persisted (one persistence per second by default), because once being synchronously persisted this update will be on the critical path of writing W which would greatly affect the performance of cached writes, as demonstrated in the next subsection. In original DM-cache the persistence of dirty bits is not for crash recovery but for (e.g., battery-backed) graceful shutdown which happens after dirty bits are (incrementally) persisted, so that up-to-date dirty-bit metadata can be read after reboot.

If the target block is not yet in the cache, then the processing is slightly complex: when the write is not aligned with a block, the block needs to be first promoted to the cache with the first kind of (mapping) metadata being persisted, after which the processing is the same as cached block writes. Note that the mapping metadata must be synchronously updated, in which case the synchronous update of the dirty-bit metadata is less harmful because the number of metadata writes only increases from one to two. In contrast, if the mapping metadata needs no update then the number of metadata writes *sharply* increases from zero to one. Mapping metadata update is largely decided by the locality of the workloads together with the replacement policy. Generally speaking, higher locality of writes causes less changes of mapping metadata which makes synchronous dirty-bit per-

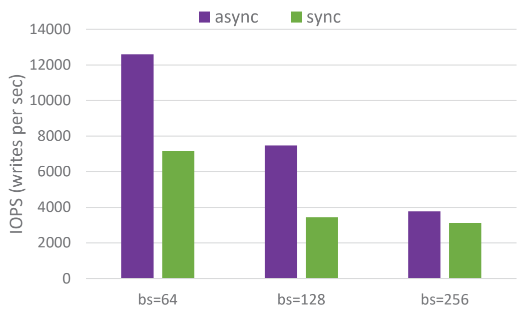

Figure 2: Synchronous updates of dirty bits severely affect the I/O performance, evaluated using one HDD + one SSD.

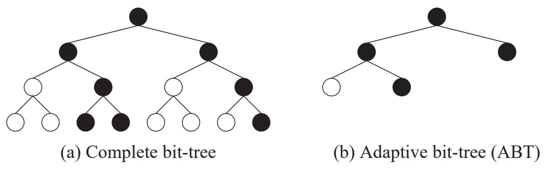

Figure 3: MapperX maintains an on-disk adaptive bit-tree (ABT), a summary of the complete bit-tree (CBT) of which the leaves represent the states of all HDD blocks. Black nodes represent dirty = `true`. ABT is *synchronously* updated for each write, but most updates do not need persistence to SSD since one bit represents a set of consecutive blocks.

sistence have higher negative impact on the performance of cached writes, and vice versa.

2.2 The Main Drawback of DM-Cache

To demonstrate the runtime performance problem of persisting dirty bit for each write, we compare the IOPS of DM-cache with synchronous and asynchronous updates, respectively. We use `fio` to perform random writes (`rw = randwrite`) and evaluate the IOPS (writes per second) with `iodepth`=16, using various cache block sizes ranging from 64 KB to 256 KB. The `fio` write sizes are the same as the catch block sizes. The result (Fig. 2) shows that the performance is severely affected when adopting synchronous update for dirty-bit metadata, causing several times IOPS degradation. The high overhead prevents synchronous update of dirty-bit metadata from being adopted by DM-cache.

Unfortunately, asynchronous metadata update causes *all* cached data on SSDs to be assumed dirty once the system crashes and gets restarted, which results in long crash recovery times and consequently low availability. For example, in our production block storage system where we use DM-cache to combine multiple SSDs with multiple HDDs on each storage machine, it takes more than two hours (depending on the workloads) to recover from a power failure (by demoting all cached blocks) even if most blocks are clean. The low availability (caused by asynchronous metadata update) prevents Linux's HDD-SSD hybrid cache mechanisms (like DM-cache) from being widely applied in availability-sensitive scenarios.

3 MapperX Design

3.1 Synchronous Metadata Update

The timing of dirty-bit metadata update is a dilemma for HDD-SSD hybrid devices. The asynchronous update mechanism periodically updates dirty bits for not affecting normal writes but suffers from long crash recovery time (since all SSD-cached blocks have to be assumed dirty and get recovered even if most of them are clean); while the synchronous update mechanism keeps all dirty bits up-to-date for fast crash recovery but greatly increases the latency of cached writes. Although the-state-of-the-art caching solution (DM-cache) adopts asynchronous update for high I/O performance, in this paper we propose to take the synchronous update mechanism for fast crash recovery.

The challenge is that there is no trade-offs for the timing of dirty-bit metadata update, because once the metadata is asynchronously updated even only one outdated bit state would require demotion of all cached blocks to ensure data durability after crash recovery. To address this challenge, our key idea is to adjust the granularity, instead of the timing, of dirty-bit metadata update to smoothly trade off between normal I/O performance and crash recovery time.

Workloads in the cloud usually have adequate write locality which can be leveraged to use one bit to represent the state of a range of consecutive blocks, as long as we can (roughly) know the effective range of the bit. Note that false positives of the effective range are not critical: if one clean block is wrongly included in the effective range of a dirty bit, the price is simply an unnecessary demotion of that block in crash recovery.

There is an in-memory bitmap that precisely records the state of every block. We first extend the in-memory bitmap to a (logical) hierarchical complete bit-tree or CBT (as shown in Fig. 3(*left*)), where the leaves are the bits of the bitmap and each inner node is the disjunction of its direct children. Then, MapperX maintains a *summary* of the CBT on the metadata device, which we refer to as on-disk *adaptive bit-tree* (ABT), as shown in Fig. 3(*right*). The ABT is synchronously updated for each write request, but most updates do not cause disk writes for metadata persistence, as discussed below. Each inner node of the ABT represents a range of consecutive blocks, and thus only the first dirty block within the range causes a write to the metadata device and subsequent writes of blocks in the range need no persistence. Initially, the ABT only has a root node representing the states of *all* cache blocks, which will change from clean to dirty after the first block write.

MapperX proposes a synchronous ABT update mecha-

Algorithm 1 Synchronous bit-tree update of MapperX

```
1: procedure BitTreeUpdate(Block $b$, Adaptive bit-tree $abt$)
2:     Update in-memory bitmap by $b$
3:     Calculate affected inner nodes of complete bit-tree ($cbt$) ▷ Dirty if any child dirty, clean if all children clean
4:     if $b$ causes leaf node $L$ of $abt$ to become dirty then
5:         Update $abt$ on disk according to $cbt$
6:     end if
7:     return SUCCESS
8: end procedure

9: procedure PeriodicAdjust(Period $p$, SLA $n$, Adaptive bit-tree $abt$)
10:    $W \leftarrow$ total number of client writes during $p$
11:    $N \leftarrow$ total number of metadata writes during $p$
12:    if $N/W \geq 1/10^n$ then   ▷ Too many metadata writes
13:        $parents \leftarrow$ all direct parents of the leaves in $abt$
14:        $target\_parent \leftarrow$ the parent from $parents$ which has experienced the most metadata writes during $p$
15:        Delete all children of $target\_parent$ in $abt$
16:    else
17:        $target\_leaf \leftarrow$ the leaf from all leaf nodes of $abt$ which has experienced the least metadata writes in $p$
18:        Generate $d$ children for $target\_leaf$ in $abt$ and set their states according to $cbt$   ▷ $d$ is the degree of $abt$
19:    end if
20:    return SUCCESS
21: end procedure
```

nism (Algorithm 1), which can adaptively adjust the metadata persistence frequency. The basic idea is to control the effective range of the corresponding bits of the leaves (i.e., the granularity of metadata update) by adding/deleting leaves in the ABT. Leaves at higher levels (farther from the root) in the ABT represent the states of a smaller range of blocks and thus increasing the levels of the ABT will increase the metadata persistence overhead while reducing the expected recovery time, and vice versa. When adding child leaves to an existing leaf node, the information about which children are dirty can be obtained from the logical CBT (calculated from the in-memory bitmap).

Since the user-perceived I/O performance is usually described as tail latencies, e.g., 99.9th percentile latency guarantee requires only one out of 1000 writes can be affected by dirty-bit metadata update, we use the number of nines (n) of the SLA (service-level agreement) to control the summary levels of the ABT.

The BitTreeUpdate procedure first updates the in-memory bitmap and calculates the affected nodes in the complete bit-tree (Lines 2∼3). Then, if the current block write causes a leaf node to change from clean to dirty, we will persist the updated ABT (Lines 4∼6).

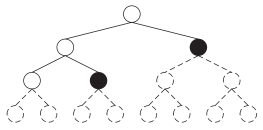
Figure 4: Example of a virtual ABT (degree $d = 2$, level $m = 4$) stored in a flat bit array for the actual ABT (Fig. 3(*left*)). For $d^{m-1} = 8$ leaves (representing 8 blocks), there are totally $\sum_{i=0}^{m-1} d^i = \frac{d^m-1}{d-1} = 15$ nodes in the tree, which can be stored using 15 bits (0010 1000 0000 000).

The PeriodicAdjust procedure decides whether to add or delete leaf nodes in the ABT according to the statistics of the last period (p). The period is configurable and by default set as one second. If there are too many metadata writes compared to the SLA in p, then we will remove all the children of the parent that has experienced the most metadata writes during p (Lines 12∼15). Otherwise we will add children to the leaf node that has experienced the least metadata writes during p (Lines 16∼18). Similar to the original DM-cache, MapperX can synchronously or asynchronously update the ABT (without add/deleting leaves) when the dirty blocks are demoted to the HDD.

3.2 Fast Crash Recovery

Since the ABT is synchronously updated for every write request, the leaf nodes in the ABT has no false negatives for dirty states. That is, when a leaf is not dirty, each of the blocks it represents are guaranteed to be not dirty and we can safely skip these blocks in crash recovery. Therefore, the recovery procedure of MapperX-based DM-cache devices is straightforward: for each dirty leaf L of the ABT, demote all SSD-cached blocks of L to the HDD.

3.3 Implementation

We have implemented MapperX on CentOS 7 by augmenting the original DM-cache with BitTreeUpdate and PeriodicAdjust in Algorithm 1, and realizing the in-memory CBT and the on-disk ABT structures in Fig. 3.

In order not to introduce extra storage overhead, we reuse DM-cache's four-byte dirty-bit metadata structure of each cached block, of which DM-cache uses only the last two bits (a dirty bit and a valid bit) leaving the first 30 bits available for MapperX. We organize the first 30 bits of all cached blocks' metadata structures into a flat bit array. To minimize the update overhead of adding/deleting leaves, we store ABT as the virtual ABT or V-ABT (Fig. 4) which has the same numbers of levels and leaves as the CBT but where only the ABT's *dirty leaves* are 1 and all other inner/leaf nodes are 0. We use the flat bit array for statically representing the states of all inner/leaf nodes in the V-ABT (each bit for one node started from the root in a breadth-first manner).

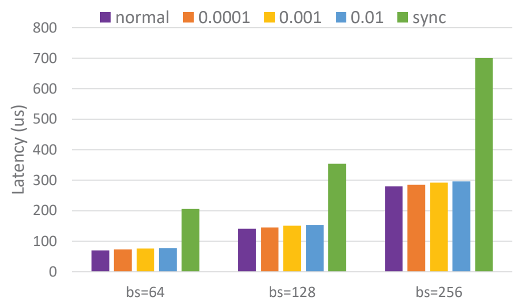

Figure 5: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (`normal`) in mean latency.

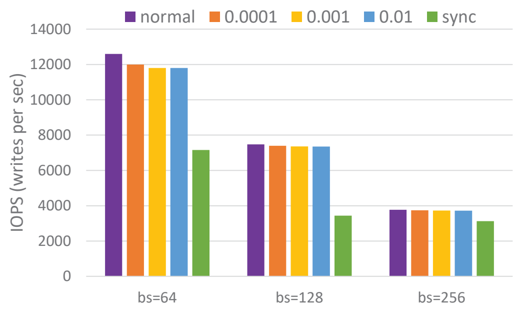

Figure 7: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (`normal`) in IOPS (writes per sec).

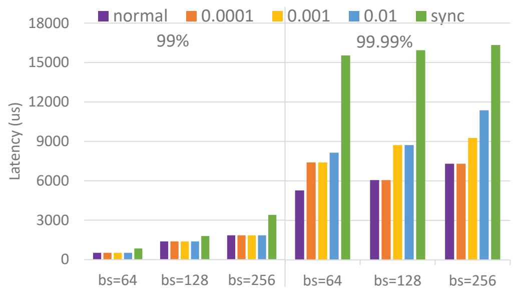

Figure 6: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (`normal`) in tail latency.

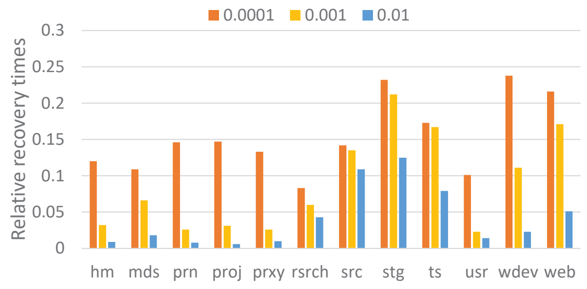

Figure 8: Recovery times of MapperX ($\beta = 0.01, 0.001, 0.0001$) relative to DM-cache for MSR traces.

4 Evaluation

Our test machine has an Intel gold 6240 36-core 2.60GHz CPU and 64GB RAM, running CentOS 7. The machine has one SATA 7200RPM 2TB HDD and one NVMe 400GB SSD. We configure the DM-cache virtual block device with 1TB HDD storage device, 128GB SSD cache device, and 1GB SSD metadata device. The client runs the `fio` benchmark tool [8] to perform random writes (`rw=randwrite`) that all hit the SSD cache for three hundred seconds. Note that cache miss should be avoided in this test because otherwise the poor performance of HDD storage would dominate the overall performance making MapperX and DM-cache have no difference.

4.1 Micro Benchmarks

We first compare the latency of random writes of MapperX and DM-cache. The cache block size (*bs*) is 64KB $\sim$ 256KB, and the `fio` write block size is equal to the cache block size. The degree of the tree is $d = 4$. We evaluate the latency using one `fio` thread with `iodepth` $= 1$. We use β to represent the expected ratio of metadata writes to all writes (i.e., $\beta = 1/10^n$ where n is the SLA) and set $\beta = 0.01, 0.001, 0.0001$, respectively. We set `max_level` $= 7$ (maximum number of levels), which limits the ABT to have $d^{\texttt{max_level}} = 16$K leaves each representing 1024, 512, and 256 blocks for *bs* = 64KB, 128KB, and 256KB (for the 1TB HDD storage), respectively. The original DM-cache adopts asynchronous metadata update (which equals to set `max_level` $= 1$). The results are shown in Fig. 5 and Fig. 6 respectively for the mean and tail latencies. The results show that the latency overhead of MapperX slightly increases as β increases from 0.0001 to 0.01, but the overhead is small compared to the original DM-cache.

We compare the IOPS (number of writes per second) of random writes of MapperX and DM-cache. The configuration is the same as that in the latency test except that we use one `fio` thread with `iodepth` $= 16$. The result is shown in Fig. 7, where MapperX has similar IOPS with DM-cache, which proves that the IOPS overhead of MapperX is small compared to the original DM-cache. Note that higher latency does not necessarily cause lower throughput (and vice versa), because NVMe SSD supports high parallelism which enables it to mask I/O delays with parallel I/O requests flying over the wire and waiting in the pipeline.

4.2 Trace-Driven Evaluation

We compare the recovery performance of MapperX (*bs* = 128) and the original DM-cache, using the public I/O traces from Microsoft Research [1] that capture block-level I/O (below the filesystem cache) of various desktop/server applications running for one week. The result is shown in Fig. 8, where the original DM-cache has to recover all the cached blocks because of its asynchronous dirty-bit metadata update. In contrast, MapperX only needs to recover much fewer blocks owing to its synchronous dirty-bit metadata update. For example, the recovery time of MapperX with $\beta = 0.01$ for the `proj` trace is only 0.6% that of DM-cache. Generally speaking, less locality leads to fewer ABT levels, fewer normal-time ABT updates, and longer crash recovery time (due to higher false-positive rates), with the design of the original DM-cache at the extreme end.

5 Related Work & Discussion

DM-cache [5], Bcache [3] and Flashcache [9] are Linux kernel modules which are used to combine fast SSDs with slow HDDs as a virtual block device. The difference is that Bcache utilizes a btree cache structure, while Flashcache's cache is structured as a set-associative hash table. LVM-cache [12] is built on top of DM-cache using logical volumes to avoid calculation of block offsets and sizes. DM-cache can also be used as the client-side local storage for caching of virtual machines in storage area networks (SANs).

DM-cache and its variations asynchronously update dirty bits and thus cannot recover from crashes with up-to-date dirty-bit information. Consequently, all cached blocks on SSD have to be assumed dirty, which makes cached blocks have to be written to HDD. In contrast, MapperX uses ABT to synchronously update dirty bits, providing flexibility of whether to write dirty data to HDD on recovery. Since recovery from crashes takes time, it is natural for MapperX to take a little more time (usually several minutes depending on the volume of dirty data) for demotion.

In addition to DM-cache/LVM-cache/Bcache/Flashcache provided by Linux kernel, several SSD-HDD hybrid designs use SSD as a cache layer. For example, Nitro [37] designs a capacity-optimized SSD cache for primary storage. Solid State Hybrid Drives (SSHD [28]) integrate an SSD inside a traditional HDD and realize SSD cache in a way similar to Nitro. URSA [38] is a distributed block storage system which stores primary replicas on SSDs and replicates backup replicas on HDDs. Griffin [60] designs a hybrid storage device that uses HDDs as a write cache for SSDs to extend SSD lifetimes. Compared to these studies, MapperX mainly focuses on the tradeoff between normal write performance and recovery times, using the ABT to adaptively adjust the range represented by the leaves.

Journal [38] based metadata write is inefficient [63] for DM-cache because of expensive write ordering [22]. For consistency, journal-based solutions always impose ordering constraint on writes (e.g., data $\rightarrow$ sync $\rightarrow$ metadata $\rightarrow$ sync) [21], which both increases latency and decreases throughput. In DM-cache and MapperX, if the blocks are already in the cache then the writes need not update the mapping metadata, so they only need one SSD write for DM-cache (Fig. 1) and (if ABT unchanged) for MapperX.

Log-structured solutions [24] are inefficient for DM-cache, because log-style writing always causes changes of the SSD-HDD mapping metadata (due to new block allocation) which has to be updated even for cache-hit write requests. In contrast, this can be avoided by DM-cache and MapperX for cache-hit write requests when mapping metadata keeps unchanged, where only one metadata write is needed (Fig. 1). Moreover, garbage collection (GC) [49] for old versions is expensive for log-structured systems.

Hardware-based, out-of-band solutions [17] require special hardware and driver supports with high programming complexity, and recently-emerging open-channel SSDs [7] are not readily available. As far as we know, large cloud providers that own millions of SSDs only have a few thousands SSDs with open-channel customization support. Further, synchronous update of SSD page states can be viewed as a special case of ABT where each leaf represents a page. This is inefficient for workloads with good locality where page states frequently change: writing a page back to HDD requires to synchronously change its state (clean) on SSD, which can be avoided by ABT if the higher-level node state keeps unchanged (dirty). On the other hand, NVM (non-volatile memory) [6] based solutions [15, 19, 20, 62, 71] are promising but expensive. NVM is still uncommon in the cloud, and the relatively-small NVM is expected to be used in more critical scenarios.

6 Conclusion

This paper presents MapperX, a novel extension to DM-cache that uses an on-disk bit-tree to *synchronously* persist the dirty-bit metadata in a hierarchical manner. Experimental results show that MapperX significantly outperforms DM-cache in crash recovery times while only introducing negligible metadata write overhead. In our future work, we will apply EC [34, 39, 45, 55, 70] for ABT, study the impact of MapperX on durability and consistency [30, 32, 46, 52, 58], apply ABT in distributed file systems [21, 23, 29, 36, 50, 53, 56, 59] and object storage systems [2, 16, 47, 64, 67], apply bloom filters [25, 42, 57, 61, 69] and compression [18, 41, 54] in ABT, and improve ABT's adjustment policy.

The source code of MapperX is available at [10].

Acknowledgement

We would like to thank Amy Tai, our shepherd, and the anonymous reviewers for their insightful comments. We thank Shun Gai for helping in the experiments, and we thank the Didi Cloud Storage Team for their discussion. Lujia Yin and Li Wang are co-primary authors. Lujia Yin implemented some parts of MapperX when he was an intern at Didi Chuxing. This research is supported by the National Key R&D Program of China (2018YFB2101102) and the National Natural Science Foundation of China (NSFC 61772541, 61872376 and 61932001).

References

[1] http://iotta.snia.org/traces/388.

[2] https://aws.amazon.com/s3/.

[3] https://bcache.evilpiepirate.org/.

[4] https://en.wikipedia.org/wiki/Device_mapper.

[5] https://en.wikipedia.org/wiki/Dm-cache.

[6] https://en.wikipedia.org/wiki/Non-volatile_memory.

[7] https://en.wikipedia.org/wiki/Open-channel_SSD.

[8] https://fio.readthedocs.io/en/latest/.

[9] https://github.com/facebookarchive/flashcache.

[10] https://github.com/nicexlab/mapperx.

[11] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[12] https://man7.org/linux/man-pages/man7/lvmcache.7.html.

[13] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.

[14] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[15] ANDERSON, T. E., CANINI, M., KIM, J., KOSTIĆ, D., KWON, Y., PETER, S., REDA, W., SCHUH, H. N., AND WITCHEL, E. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 1011–1027.

[16] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 47–60.

[17] BJØRLING, M., GONZALEZ, J., AND BONNET, P. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 359–374.

[18] BORSTNIK, U., VANDEVONDELE, J., WEBER, V., AND HUTTER, J. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput. 40*, 5-6 (2014), 47–58.

[19] CHEN, Y., LU, Y., CHEN, P., AND SHU, J. Efficient and consistent NVMM cache for ssd-based file system. *IEEE Trans. Computers 68*, 8 (2019), 1147–1158.

[20] CHENG, W., LI, C., ZENG, L., QIAN, Y., LI, X., AND BRINKMANN, A. Nvmm-oriented hierarchical persistent client caching for lustre. *ACM Trans. Storage 17*, 1 (2021), 6:1–6:22.

[21] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.

[22] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Usenix Conference on File and Storage Technologies* (2012).

[23] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[24] DAI, Y., XU, Y., GANESAN, A., ALAGAPPAN, R., KROTH, B., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 155–171.

[25] DAI, Z., AND SHRIVASTAVA, A. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier with application to real-time information filtering on the web. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds.

[26] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 25–36.

[27] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[28] DORDEVIC, B., TIMCENKO, V., AND RAKAS, S. B. Sshd: Modeling and performance analysis. *INFOTEH-JAHORINA 15*, 3 (2016), 526–529.

[29] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[30] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 202–210.

[31] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 199–212.

[32] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[33] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)* (2005), IEEE, pp. 18–27.

[34] KOSAIAN, J., RASHMI, K. V., AND VENKATARAMAN, S. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019* (2019), T. Brecht and C. Williamson, Eds., ACM, pp. 30–46.

[35] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[36] LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference* (2008), vol. 1, pp. 2–5.

[37] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (2014), pp. 501–512.

[38] LI, H., ZHANG, Y., LI, D., ZHANG, Z., LIU, S., HUANG, P., QIN, Z., CHEN, K., AND XIONG, Y. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 15.

[39] LI, H., ZHANG, Y., ZHANG, Z., LIU, S., LI, D., LIU, X., AND PENG, Y. Parix: speculative partial writes in erasure-coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX Association, pp. 581–587.

[40] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[41] Liu, Y., and Schmidt, B. Lightspmv: Faster cuda-compatible sparse matrix-vector multiplication using compressed sparse rows. *J. Signal Process. Syst. 90*, 1 (2018), 69–86.

[42] Lu, J., Wan, Y., Li, Y., Zhang, C., Dai, H., Wang, Y., Zhang, G., and Liu, B. Ultra-fast bloom filters using SIMD techniques. *IEEE Trans. Parallel Distributed Syst. 30*, 4 (2019), 953–964.

[43] Meyer, D. T., Aggarwal, G., Cully, B., Lefebvre, G., Feeley, M. J., Hutchinson, N. C., and Warfield, A. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 41–54.

[44] Mickens, J., Nightingale, E. B., Elson, J., Gehring, D., Fan, B., Kadav, A., Chidambaram, V., Khan, O., and Nareddy, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.

[45] Mitra, S., Panta, R. K., Ra, M., and Bagchi, S. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 30:1–30:16.

[46] Mohan, J., Martinez, A., Ponnapalli, S., Raju, P., and Chidambaram, V. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 33–50.

[47] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., and Kumar, S. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 383–398.

[48] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS) 4*, 3 (2008), 10.

[49] Nguyen, K., Fang, L., Xu, G., Demsky, B., Lu, S., Alamian, S., and Mutlu, O. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 349–365.

[50] Niazi, S., Ismail, M., Haridi, S., Dowling, J., Grohsschmiedt, S., and Ronström, M. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017), pp. 89–104. 00091.

[51] Nightingale, E. B., Elson, J., Fan, J., Hofmann, O., Howell, J., , and Suzue, Y. Flat datacenter storage. In *OSDI* (2012).

[52] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[53] Pan, S., Stavrinos, T., Zhang, Y., Sikaria, A., Zakharov, P., Sharma, A., P, S. S., Shuey, M., Wareing, R., Gangapuram, M., Cao, G., Preseau, C., Singh, P., Patiejunas, K., Tipton, J. R., Katz-Bassett, E., and Lloyd, W. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 217–231. 00000.

[54] Pligouroudis, M., Nuno, R. A. G., and Kazmierski, T. Modified compressed sparse row format for accelerated fpga-based sparse matrix multiplication. In *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020* (2020), IEEE, pp. 1–5.

[55] Rashmi, K. V., Chowdhury, M., Kosaian, J., Stoica, I., and Ramchandran, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 401–417.

[56] Ren, K., Zheng, Q., Patil, S., and Gibson, G. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE, pp. 237–248. 00136.

[57] Santiago, L., Verona, L. D., Rangel, F. M., de Faria, F. F., Menasché, D. S., Caarls, W., Jr., M. B., Kundu, S., Lima, P. M. V., and França, F. M. G. Weightless neural networks as memory segmented bloom filters. *Neurocomputing 416* (2020), 292–304.

[58] Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., and Bronson, N. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 407–423.

[59] Shvachko, K. V. Hdfs scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE 35*, 2 (2010), 6–16.

[60] Soundararajan, G., Prabhakaran, V., Balakrishnan, M., and Wobber, T. Extending ssd lifetimes with disk-based write caches. In *FAST* (2010), vol. 10, pp. 101–114.

[61] Stanciu, V. D., van Steen, M., Dobre, C., and Peter, A. Privacy-preserving crowd-monitoring using bloom filters and homomorphic encryption. In *EdgeSys@EuroSys 2021: 4th International Workshop on Edge Systems, Analytics and Networking, Online Event, United Kingdom, April 26, 2021* (2021), A. Y. Ding and R. Mortier, Eds., ACM, pp. 37–42.

[62] Venkatesan, V., Wei, Q., and Tay, Y. C. Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines. In *2014 IEEE International Conference on High Performance Computing, HPCC/CSS/ICESS 2014* (2014), IEEE, pp. 966–973.

[63] Wang, L., Xue, J., Liao, X., Wen, Y., and Chen, M. LCCFS: a lightweight distributed file system for cloud computing without journaling and metadata services. *Sci. China Inf. Sci. 62*, 7 (2019), 72101:1–72101:14.

[64] Wang, L., Zhang, Y., Xu, J., and Xue, G. MAPX: controlled data migration in the expansion of decentralized object-based storage systems. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020* (2020), S. H. Noh and B. Welch, Eds., USENIX Association, pp. 1–11.

[65] Wang, Y., Kapritsos, M., Ren, Z., Mahajan, P., Kirubanandam, J., Alvisi, L., and Dahlin, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.

[66] Warfield, A., Ross, R., Fraser, K., Limpach, C., and Hand, S. Parallax: Managing storage for a million machines. In *HotOS* (2005).

[67] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[68] Zhang, Y., Guo, C., Li, D., Chu, R., Wu, H., and Xiong, Y. Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems. In *NSDI* (2015), pp. 529–542.

[69] Zhang, Y., Li, D., Chen, L., and Lu, X. Collaborative search in large-scale unstructured peer-to-peer networks. In *2007 International Conference on Parallel Processing (ICPP 2007), September 10-14, 2007, Xi-An, China* (2007), IEEE Computer Society, p. 7.

[70] Zhang, Y., Li, H., Liu, S., Xu, J., and Xue, G. PBS: an efficient erasure-coded block storage system based on speculative partial writes. *ACM Trans. Storage 16*, 1 (2020), 6:1–6:25.

[71] Zhu, G., Lu, K., Wang, X., Zhang, Y., Zhang, P., and Mittal, S. Swapx: An nvm-based hierarchical swapping framework. *IEEE Access 5* (2017), 16383–16392.