DENKV: Addressing Design Trade-offs of Key-value Stores for Scientific Applications

Safdar Jamil[1], Awais Khan[2], Kihyun Kim[1], Jae-Kook Lee[3], Dosik An[3]
Taeyoung Hong[3], Sarp Oral[2], Youngjae Kim[1]
[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, Korea
[2]Oak Ridge National Laboratory, [3]Korea Institute of Science and Technology Information
{safdar, realltd16, youkim}@sogang.ac.kr, {khana, oralhs}@ornl.gov, {jklee, dsan, tyhong}@kisti.re.kr

***Abstract*—High-performance computing (HPC) facilities have employed flash-based storage tier near to compute nodes to absorb high I/O demand by HPC applications during periodic system-level checkpoints. To accelerate these checkpoints, proxy-based distributed key-value stores (PD-KVS) gained particular attention for their flexibility to support multiple backends and different network configurations. PD-KVS rely internally on monolithic KVS, such as LevelDB or RocksDB, to exploit the KV interface and query support. However, PD-KVS are unaware of the high redundancy factor in checkpoint data, which can be up to GBs to TBs, and therefore, tend to generate high write and space amplification on these storage layers. In this paper, we propose DENKV which is deduplication-extended node-local LSM-tree-based KVS. DENKV employs asynchronous partially inline dedup (APID) and aims to maintain the performance characteristics of LSM-tree-based KVS while reducing the write and space amplification problems. We implemented DENKV atop BlobDB and showed that our proposed solution maintains performance while reducing write amplification up to $2\times$ and space amplification by $4\times$ on average.**

***Index Terms*—High Performance Computing, Key-Value Stores, Log-Structures Merge Tree, Deduplication**

I. INTRODUCTION

HPC applications are often compute and data intensive and frequently run simulations for longer times. They perform periodic checkpointing to store their internal states to high I/O bandwidth parallel file systems (PFS) [1]. However, storing checkpoints on PFS presents significant difficulties when scaling to meet the demands of large scientific applications due to the considerable overhead induced. To absorb the high I/O demand, it has become common to deploy additional storage layers, often flash-based, to handle high-bandwidth workloads such as checkpointing. These storage layers have a variety of deployment models such as compute node-local (CN) storage as in Summit [2]–[4] or near-node storage, often known as Burst Buffers (BB) as in Sierra [3], [5].

Most HPC applications use system-level checkpointing libraries, which generate a huge amount of redundant checkpoint data [6], [7]. To verify this claim, we investigated the redundancy ratio of the 10 HPC applications at Nurion supercomputer [8] and analyzed their sample data using an in-house deduplication analysis tool and present the results in Table I. Furthermore, several studies [9]–[12] tend to accelerate checkpointing by adopting a distributed KV interface atop CN-local and BBs. Notably, distributed KVS

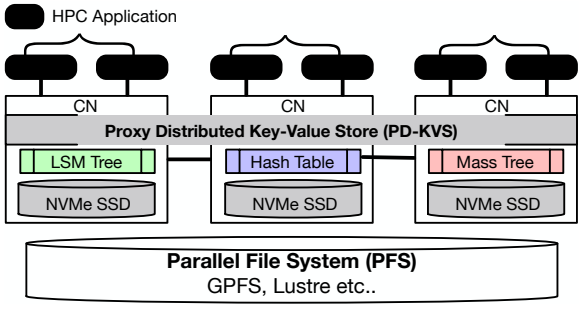

Figure 1: Target HPC environment with compute-node local NVMe SSDs and KVSs.

are categorized as either natively-distributed or proxy-based KVS [10]. Proxy-based distributed KVS (PD-KVS) [9]–[15] has received special attention for their flexibility to support multiple backends and different network configurations.

PD-KVS relies internally on monolithic KVS[1], such as LevelDB [16] or RocksDB [17], to exploit the KV interface and query support. These KVS are unaware of high redundancy factor in checkpoints, ranging from GBs to TBs, and therefore tend to generate high write amplification (WA) and space amplification (SA) on these storage layers, where the storage capacity is limited. Furthermore, log-structured merge (LSM)-tree-based KVSs are unfit for node-local storage due to high internal WA and SA, which degrades performance. A way around this problem is to integrate redundancy reduction techniques such as deduplication (dedup) in the HPC application. However, it requires modifying the application and cannot control the internal WA and SA of KVS. For instance, on average internal WA of LSM-tree-based KVS ranges from $10\times$ - $30\times$ on average, primarily affecting KVS throughput [18], [19]. Integrating deduplication in the KVS backend offers several benefits, such as reduced checkpoint data, minimal WA and SA, and intelligent storage utilization [20]–[23].

Thus, we propose a deduplication-extended node-local LSM-tree-based KV-Store (DENKV), to guarantee high performance for HPC checkpointing applications. DENKV eliminates redundancy to smartly utilize the storage of space-constrained node-local devices. We carefully identified the level of LSM-tree to integrate the APID and aim to main-

[1]Hereafter, we will refer to monolithic KVS to simply KVS.

978-1-6654-7562-4/22/$31.00 ©2022 IEEE
DOI 10.1109/PDSW56643.2022.00009

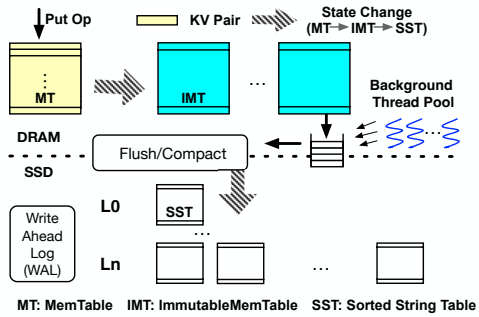

Figure 2: Description of LSM-tree data structures.

tain the performance properties of LSM-tree-based KVS. APID leverages transparent asynchronous dedup using the *FLUSH* operation by the background thread pool. We extended BlobDB [24] to implement the proposed ideas. BlobDB [17] is based on vanilla RocksDB with key-value separation design. Evaluation with the in-house simulation of checkpointing data of HPC applications revealed that DENKV reduces the WA and SA by upto $2\times$ and $4\times$ without jeopardizing the performance.

II. Background and Motivation

A. Proxy-based Distributed KVS (PD-KVS) in HPC

Figure 1 depicts an HPC environment where each compute node (CN) has a local NVMe SSD, and on top of it, a PD-KVS can be adopted to fully exploit the performance characteristics of these NVMe SSDs and provide significant query performance. PD-KVS support multiple backends and network configurations, making them a suitable candidate to be adopted in HPC facilities. With different backend storage engines in PD-KVS, applications can benefit according to their requirements. For instance, applications that support range queries will benefit by using a MassTree-based storage engine [25] and applications dealing with write-intensive applications can take advantage of LSM-tree-based storage engines [16], [17], [26].

B. Log-Structured Merge (LSM)-Tree

Figure 2 illustrates the architecture and operational flow of the representative LSM-tree-based KV store, RocksDB at Facebook [17]. LSM-tree consists of memory and storage-based components. Memory components include in-memory skiplist-based MemTables, which are mutable, and applications can write/update KV pairs directly in MemTables. When the MemTable reaches a certain size threshold, it is converted into an Immutable Memtable (IMT) and eventually written to level 0 by employing asynchronous threads from the background thread pool, as shown in Figure 2. Storage components are formed by Sorted String Tables (SSTs) files and stored in terms of increasing size levels. The SSTs in level 0 are unsorted and can have overlapping key ranges; however, SSTs in higher levels are sorted and do not have overlapping key ranges. A compaction operation is triggered when a level reaches a size threshold to maintain the sorted order of SSTs and lookup performance of the LSM-tree. The compaction operation merges and sorts the SSTs from particular level n to level $n+1$ and is performed by the asynchronous threads from the background thread pool. RocksDB employs Write-Ahead-Log (WAL) for failure consistency of memory components.

LSM-tree-based KVSs suffer from high WA and SA. WA occurs when KVS performs more write operations than the application intended. It happens due to internal operations of LSM-tree such as compaction. Meanwhile, SA occurs when KVS occupies more space than the application requires. It mainly happens when the workload is update-intensive, and higher levels of the LSM-tree still store the stale data. As LSM-tree stores data in terms of levels, lower levels always have the latest/valid KV pairs, which makes the KV pairs stored at the higher level invalid/stale and compaction operation is responsible for reclaiming those stale KV pairs. WA shortens the lifespan of SSD due to the high number of writes, and SA requires unnecessary SSD space, so they limit the adoption of SSD as CN-SSD in the HPC environment.

C. Motivation

As shown in Figure 1, HPC applications store their intermediate state and checkpoint data in the compute-node local SSD (CN-SSD) by exploiting PD-KVS. PD-KVS rely on monolithic KVS to exploit performance characteristics of CN-SSD and query support. PD-KVS can deploy multiple independent instances of KVSs in the HPC environment to support the distinct needs of the HPC applications. The HPC application generates bursty write patterns when checkpointing the state, and to absorb these write patterns, LSM-tree-based KVSs are adopted due to being highly write optimized. However, LSM-tree-based KVSs suffer from high WA and SA, which limits their adoption due to the limited capacity of the CN-SSD. Thus in this work, we aim to incorporate data dedup in CN-SSD-based PD-KVSs only to store the unique checkpoint data.

This paper solves the aforementioned WA and SA by running dedup in the storage engine of the LSM-tree-based KVS. We focus on LSM-tree-based KVS as they are highly write optimized and argue that they are a suitable candidate to be adopted in PD-KVS to store intermediate and checkpoint data on KVSs. However, the incorporation of dedup in LSM-tree comes with its own challenges. First, identifying a suitable component to adopt dedup plays a vital role in the performance of the LSM-tree. If dedup is incorporated in memory components, it impedes the performance of the LSM-tree due to additional dedup overhead. Second, integrating dedup at the higher levels of storage components, level 1 onward, will not be efficient in reducing WA and breaking the structural constraint of the LSM-tree. LSM-tree only allows a single valid instance of the KV pair at a particular level, and storing multiple valid instances of KV pair based on the value chunks will lead to the increased complexity of the compaction operation and results in write stalls.

III. Redundancy in HPC Application

HPC applications often generate massively repeated data (checkpoints) during execution and as an output. This dupli-

Table I: Deduplication analysis of 10 applications on Nurion Supercomputer hosted at KISTI facility. DR denotes duplication ratio.

App.	Size	DR	App.	Size	DR
Abaqus	386 GB	41.8%	CESM	273 GB	25.7%
Charmm	382 GB	23.1%	Gaussian	293 GB	20.4%
Lammps	24 GB	42.5%	MOM	323 GB	53.9%
MPAS	197 GB	81.7%	Siesta	566 GB	52.1%
VASP	1 TB	27.3%	ANSYS	544 GB	23.8%

cate data limits the usage of CN-SSD, which is constrained by limited storage capacity. For example, a compute node in Summit supercomputer has only 1.6 TB of storage capacity [3]. Whereas HPC applications, including physics and scientific applications, generate several TBs of data during the simulation lifetime. To investigate the duplicate ratio of the HPC application, we analyzed the 10 HPC applications consuming the most CPU cycles at the Nurion Supercomputer hosted at the Korea Institute of Science and Technology (KISTI) for one year and captured a sample of their generated data. We analyzed the application data using an in-house dedup analysis tool that we implemented based on FS-C [7]. Nurion has a total theoretical performance of 25.7 petaflops, which was ranked 11th in the world in June 2018 [3]. Table I shows the sample size collected within 10 minutes from each application's generated data. Meanwhile, HPC applications execute in terms of days, which can lead to a vast amount (TBs-PBs), while the CN-SSDs have limited capacity [3].

Table I also shows the ratio of duplicates in terms of percentage for the data generated by the HPC applications. This duplicate ratio typically ranges between 20% to 81%. Furthermore, it has been studied that system-level checkpointing within HPC systems generates over 80% of duplicate data [6]. Note that storing duplicate data limits not only the capacity of the CN-SSD but also affects the performance of the HPC application. For instance, when the capacity constraint is reached in the CN-SSD, the system will flush the data to the parallel file system (PFS), which is usually employed on top of slow storage devices, typically over HDDs. On the other hand, only storing the unique data at CN-SSD will help achieve efficient utilization of these storage devices while reducing the flush time over PFS.

IV. DenKv: Design and Implementation

A. System Overview

We proposed DenKv, a dedup-enabled LSM-tree-based KVS, which employs asynchronous partially inline dedup (APID). We carefully identified the level of LSM-tree to integrate the APID without compromising the performance. APID leverages the transparent asynchronous *FLUSH* operation by the background thread pool to perform dedup operation when IMTs are written to the level 0 of the LSM-tree. APID uses the fixed-size chunking mechanism for value partitioning and employs SHA1 for fingerprinting. To manage the dedup metadata, APID introduces a chunk information table (CIT), which helps identify the duplicate value chunks. Furthermore, DenKv adopted the KV-separation approach to maintaining

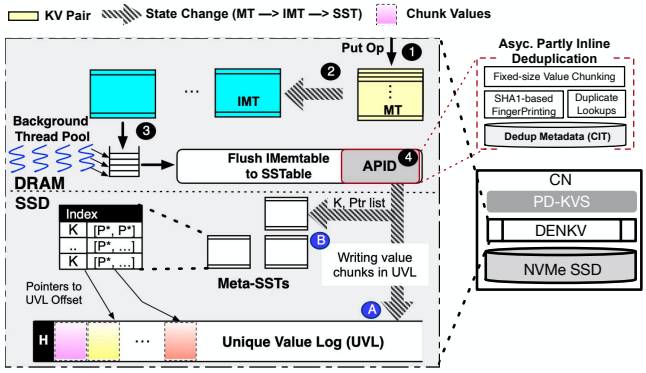

Figure 3: Design overview of DenKv and operational flow of *PUT* and *FLUSH* operation in DenKv.

the structural constraint of the LSM-tree and introduced Meta-SSTs and unique value log (UVL). In the KV-separation technique, the keys and values are stored separately. The reason for adoption of KV-separation design is manifolds: First, the value chunks are stored separately based on the chunk size; meanwhile, the corresponding key can store the list of value chunk pointers in Meta-SSTs. Second, with the KV-separation technique, SSTs are relatively smaller in size, thus reducing the data movement during compaction operation and resulting in less WA.

B. Data Structures

Figure 3 unboxes the internals of DenKv within a single compute node of HPC system. DenKv consists of memory and storage components. DenKv introduces chunk information table (CIT), which manages the dedup metadata. CIT is a hash table that stores the value chunks' fingerprint (20 bytes), offset (8 bytes), and reference count (4 bytes). A single entry in CIT is comprised of 32 bytes and the total size of CIT will depend on the workload size. CIT is updated during the *FLUSH* and compaction operations by the background thread pool. Additionally, DenKv introduces Meta-SSTs and the UVL at storage layer. Meta-SSTs store the keys and list of value chunk pointers that points at the corresponding value chunk associated with the key. The format of Meta-SSTs is similar to the traditional SSTs and comprised of filter, metadata, and data blocks. The filter block is a bloom filter which helps in GET queries, while the metadata blocks store the additional metadata related to the SSTs such as number of KV pairs, size of the SSTs, starting and ending range, and so on. The data blocks stores the keys and list of value chunk pointers as shown in Figure 3. The UVL is a log-based file which stores the value chunks based on dedup chunk size and comprised of value chunk and the reference count, represented as H in Figure 3.

C. Put Operation

The operational flow of the PUT and *FLUSH* operations of our proposed design is shown in Figure 3. When the HPC application places a PUT request, it follows the path of the

PUT operation of the traditional LSM-tree and writes the KV pair to MT, as shown by step ❶ of Figure 3. When MT reaches a certain size threshold, it is converted to IMT (step ❷). At step ❸, the background thread pool selects the IMTs and passes them to the APID, step ❹. During *FLUSH* operation, APID takes each KV pair from IMT and chunks the value part of the KV pair, based on the chunk size, and the fingerprint is computed. When the fingerprint is obtained, the CIT is traversed to match the fingerprint. If a unique value chunk is encountered, it is written to the UVL (step Ⓐ), and an entry is created in the corresponding Meta-SST at level 0 (step Ⓑ). When the unique value chunk is written to UVL and Meta-SST, the dedup metadata is updated at the CIT. If a duplicate value chunk is detected, the offset stored at the CIT is attached to the pointer list of the key and added to the Meta-SST. DENKV supports large KV pairs for dedup by chunking the values into fixed size chunks and storing unique chunks at the UVL as shown in Figure 3. This mechanism obtains the offset of each chunk maintained at the CIT and LSM-tree with the corresponding key. The P^* in Figure 3 represents an offset pointer of the value chunk in Meta-STT.

D. Get Operation

The GET operation also follows the same path as the traditional LSM-tree-based KVS. When a GET request is placed by the HPC application, it starts by looking for the requested KV pair in the MT, and if found, it returns. Otherwise, the GET operation will move to the IMTs. If the KV pair is not found in the memory components, the GET request will traverse the Meta-SSTs to another level. Each Meta-SST employs a bloom filter to identify if it contains the requested KV pair or not. If the bloom filter returns true, then the particular Meta-SST is traversed, and the corresponding KV pair is reconstructed by fetching the value chunks from the UVL in the same order the value chunk pointers are stored on the list. If the bloom filter returns false, the GET request moves to another Meta-SST and reads its bloom filter. During the GET operation, there is no interaction with CIT for reconstructing the KV pair. The GET operations are based on keys, and APID does not store any information regarding keys.

E. Garbage Collection

Although DENKV only writes the unique chunks at the UVL, if a chunk is not being referred by any of the keys in the LSM-tree, that chunk needs to be removed and the space utilized by that chunk must be reclaimed. We introduce garbage collection in DENKV to reclaim the obsolete chunk. We integrated garbage collection with compaction operation as proposed in BlobDB [24]. During compaction operation, BlobDB reclaims obsolete values from the blob files if an update operation has been encountered for the corresponding key of the value. However, in DENKV, a single chunk in UVL can be pointed by several keys in the LSM-tree. Therefore, during compaction, we only update the reference count of the chunk maintained within the UVL (H in UVL of Figure 3). At the end of compaction, an asynchronous garbage collection thread traverses the UVL, reclaims the chunks with zero reference count, and updates CIT by deleting the entries of reclaimed chunks.

F. Performing Analysis on Node-local KV

Key-value and NoSQL columnar data stores are better suited for several graphs and simulation-based analytical workloads and applications. Performing search-rich querying on such KV stores is easier and much faster than traditional file system approaches. The CN-SSDs offer opportunities to analyze and gain insight into intermediate data to optimize the HPC applications. For instance, various HPC applications perform computation and analysis in the Map-Reduce `<key, value>` style [27] to obtain insights into the data generated during simulations. Therefore, by adopting the KV interface, HPC applications gain immediate access to perform queries on the intermediate data and get the specific insights required.

V. EVALUATION

A. Experimental Setup

We performed all the experiments on a Linux machine with 4 Intel Xeon(R) E5-4640 v2 CPUs @ 2.20 GHz with 10 physical cores per CPU node, 80 MiB last level cache, and 256 GiB DDR3 DRAM. The operating system is 64-bit Linux 5.15.0.33, with an EXT4 file system. The machine is equipped with 1 TB Samsung 970 EVO SSD.

1) Implementation: DENKV is implemented on top of BlobDB [24] 7.3.0 which implements the key-value separation in RocksDB [17]. BlobDB is optimized for write and read operations and shows better performance than the traditional RocksDB. Similar to BlobDB, DENKV maintains foreground and background threads where foreground threads perform operation on mutable data which can be directly manipulated by the HPC applications. In DENKV, the background thread pool is divided into *FLUSH* threads and compaction threads. *FLUSH* threads perform KV separation, dedup operation and then write the keys and value pointers to the Meta-SSTs of LSM-tree, while only unique value chunks are written to the UVL. We compare the three implementations below.

- **RocksDB**: Traditional LSM-tree-based KVS where storage components are composed of SSTs which store KV pairs together. Additionally, we utilize traditional RocksDB to compare the performance difference between baseline LSM-tree structure and KV separated design.
- **BlobDB**: KV separated design of RocksDB where keys and value pointers are stored at the SSTs, while the values are stored at the value-log, named Blobs. BlobDB maintains the basic design of the LSM-tree means, it has memory and storage components where *FLUSH* operation is responsible for separating keys and values.
- **DENKV**: Our proposed design which introduces APID. Similar to BlobDB, DENKV also employed KV separation technique while instead of storing the complete values of the corresponding KV pair, our solution only stores the unique value chunks at the UVL.

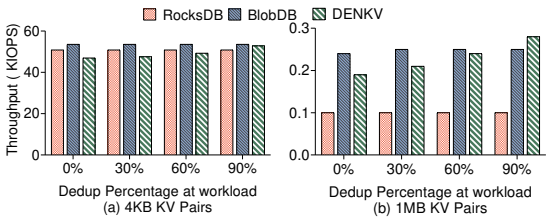

Figure 4: Performance analysis of DENKV with varying KV pair sizes and dedup ratio.

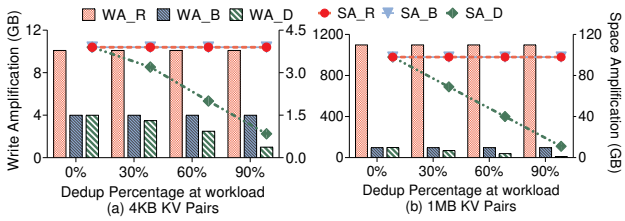

Figure 5: WA and SA analysis with varying KV pair sizes and dedup ratio. WA_R and SA_R presents WA and SA of traditional RocksDB, WA_B and SA_B presents BlobDB, and WA_D and SA_D presents DENKV.

2) Benchmarks and Workload: We built an in-house benchmark which simulates the dedup patterns of HPC applications as shown in Table I. Specifically, with our benchmark we can adjust the dedup ratio, the number of KV pairs and value size. The benchmark uses single application thread to perform requests. For workloads, we generated two categories of KV pairs: small (4 KB KV pairs) and large (1 MB KV pairs) where the key size is fixed to 16 bytes while the value size varies. For small workload, we used 1 Million KV pairs while for the large workload, we only used 100 thousand KV pairs for the evaluation. The dedup ratio evaluated in these experiments include 0%, 30%, 60%, and 90%. We selected these dedup ratio considering the dedup ratios presented by the HPC applications in Table I.

B. Results

We present the performance analysis of proposed DENKV in comparison to baseline RocksDB and BlobDB. We also show the WA and SA evaluation of the compared KVSs.

1) Performance analysis: Figure 4 shows the evaluation results in terms of throughput of 4KB and 1MB sized KV pairs with varying dedup ratios. It can be observed that DENKV has the worst performance in comparison to baseline RocksDB and BlobDB for small KV pairs with a 0% dedup ratio as shown in Figure 4(a). It is due to extra dedup operation. With additional dedup operations, the background flush threads have larger critical sections and take longer time than baseline RocksDB and BlobDB. The dedup operations (fingerprinting and dedup metadata update) are the major cause of additional execution time for background *FLUSH* threads. However, these overheads can be overcome by adopting parallelized fingerprinting methods and an optimized concurrent-friendly dedup metadata table (CIT).

Furthermore, it can be seen that with an increasing dedup ratio, DENKV improves the performance and outperforms baseline RocksDB when the dedup ratio is 90% with small KV pairs. This is due to less number of I/O operations being performed by DENKV and only CIT and Meta-SSTs are updated, which reduces the overall I/O size and thus improves the performance. Additionally, DENKV outperforms RocksDB regardless of the dedup ratio with large KV pairs. This is because DENKV is implemented atop BlobDB, and due to its KV separation design both DENKV and BlobDB are optimized for large KV pairs. Notably, DENKV also outperforms BlobDB when the dedup ratio is 90% as explained above as shown in Figure 4(b).

2) WA and SA analysis: Figure 5 shows the WA and SA analysis. The right side Y-axis of Figure 5 shows the WA, while the left side presents the SA results. We measured the WA during the execution of each workload using the Linux system-stat command. It can be observed that traditional RocksDB suffers from huge WA in all workload meanwhile DENKV and BlobDB has significantly less WA due to KV separation design. DENKV and BlobDB both do not include the value parts for the compaction operation, which helps mitigate the WA. Note that DENKV has even less WA than BlobDB with an increasing dedup ratio in both workloads. With a high dedup ratio, a smaller number of writes are performed to the UVL of the DENKV, thus less WA.

We measured the SA as the total storage space occupied by all the KVSs after the execution of the workloads. Note that DENKV has the least storage requirement with an increasing dedup ratio while RocksDB and BlobDB have constant storage utilization regardless of the dedup ratio. With increasing dedup ratio, DENKV only stores the unique value chunks in the UVL while storing all the keys and associated value chunk pointers in the Meta-SSTs. Since the keys and value chunk pointers are small, only a few Meta-SSTs are created and thus less storage utilization.

VI. CONCLUSION

In this work, we showed that HPC applications generate highly redundant data by a thorough dedup analysis of the 10 HPC applications at the Nurion supercomputer at KISTI. We argue that adopting storage optimization techniques at PD-KVS based KVSs would help reduce the storage utilization and improve the performance of HPC applications. Thus, we proposed DENKV, deduplication extended node-local key-value store to guarantee high-performance for HPC checkpointing applications. DENKV introduces asynchronous partly inline deduplication (APID) at *FLUSH* operation. We designed and implemented DENKV atop BlobDB and showed that our proposed solution maintains performance while reducing WA up to 2× and SA by 4×. For future work, we plan to deploy DENKV in an HPC setting to further evaluate its performance and recovery process.

ACKNOWLEDGEMENT

This work was supported in part by the Korea Institute of Science and Technology Information (Grant No.J-22-NB-C03-S01) and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2014386). This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). Y. Kim is the corresponding author.

REFERENCES

[1] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[2] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea, "End-to-end i/o portfolio for the summit supercomputing ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[3] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, "An analysis of system balance and architectural trends based on top500 supercomputers," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 11–22.

[4] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing i/o bottleneck for large-scale deep learning applications," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022.

[5] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, and J. Yin, "The design, deployment, and evaluation of the coral pre-exascale systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18, 2018, pp. 661–672.

[6] J. Kaiser, R. Gad, T. Süß, F. Padua, L. Nagel, and A. Brinkmann, "Deduplication potential of hpc applications' checkpoints," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. Cluster '16, 2016, pp. 413–422.

[7] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[8] "Nurion," https://www.ksc.re.kr/eng/resource/nurion, 2018, accessed: 2021-08-16.

[9] C. Cugnasco, Y. Becerra, J. Torres, and E. Ayguadé, "Exploiting key-value data stores scalability for hpc," in *Proceedings of the 46th International Conference on Parallel Processing Workshops (ICPPW)*, ser. ICPPW '17. IEEE, 2017, pp. 85–94.

[10] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. Butt, "Bespokv: Application tailored scale-out key-value stores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18, 2018, pp. 14–29.

[11] H. Greenberg, J. Bent, and G. Grider, "MDHIM: A parallel key/value framework for hpc," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '15, 2015.

[12] T. Wang, A. Moody, Y. Zhu, K. Mohror, K. Sato, T. Islam, and W. Yu, "Metakv: A key-value store for metadata management of distributed burst buffers," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '17. IEEE, 2017, pp. 1174–1183.

[13] J. Kim, S. Lee, and J. S. Vetter, "Papyruskv: A high-performance parallel key-value store for distributed nvm architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, 2017, pp. 1–14.

[14] Z. W. Parchman, F. Aderholdt, and M. G. Venkata, "Sharp hash: A high-performing distributed hash for extreme-scale systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 647–648.

[15] S. Eilemann, F. Delalondre, J. Bernard, J. Planas, F. Schuermann, J. Biddiscombe, C. Bekas, A. Curioni, B. Metzler, P. Kaltstein *et al.*, "Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 608–617.

[16] "LevelDB," https://github.com/google/leveldb, 2010, accessed: 2021-10-20.

[17] "RocksDB," http://rocksdb.org, 2012, accessed: 2021-10-20.

[18] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, Mar. 2017.

[19] F. Pan, Y. Yue, and J. Xiong, "Dcompaction: Delayed compaction for the lsm-tree," *Int. J. Parallel Program.*, vol. 45, no. 6, p. 1310–1325, Dec. 2017.

[20] H. Kwon, Y. Cho, A. Khan, Y. Park, and Y. Kim, "DeNOVA: Deduplication extended nova file system," in *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '22, 2022, pp. 1360–1371.

[21] A. Khan, P. Hamandawana, and Y. Kim, "A content fingerprint-based cluster-wide inline deduplication for shared-nothing storage systems," *IEEE Access*, vol. 8, pp. 209 163–209 180, 2020.

[22] A. Khan, C.-G. Lee, P. Hamandawana, S. Park, and Y. Kim, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," in *Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 87–93.

[23] P. Hamandawana, A. Khan, C.-G. Lee, S. Park, and Y. Kim, "Crocus: Enabling computing resource orchestration for inline cluster-wide deduplication on scalable storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1740–1753, 2020.

[24] "BlobDB," http://rocksdb.org, 2018, accessed: 2022-02-14.

[25] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European conference on Computer Systems*, ser. Eurosys '12, 2012, pp. 183–196.

[26] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based Ultra-Large Key-Value store for small data items," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 71–82.

[27] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, Jan 2008.