Crail-KV: A High-Performance Distributed Key-Value Store Leveraging Native KV-SSDs over NVMe-oF

Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, Yang-suk Kee
Samsung DSA
E-mail: {t.bisson, ke.chen1, changho.c, vijay.bala, yangseok.ki}@samsung.com

***Abstract*—A Key-Value SSD (KV-SSD) is a new type of storage device that natively exposes a key-value interface. In this paper, we leverage KV-SSDs to develop new techniques to remove unnecessary layers of indirection traditionally imposed by block devices on distributed storage systems. Specifically, we extend the Crail distributed system [1] to leverage the KV-SSD's native key-value interface exposing it directly to clients through the NVMe-oF protocol. This architectural change simplifies key-value metadata management, as the metadata manager need only track key-value tuples rather than files comprised of blocks. These changes enable fewer RPCs and require less memory for metadata management, resulting in a performance improvement of up to 5x.**

***Index Terms*—distributed systems, storage, key-value, NVMe-oF**

I. Introduction

Apache Crail is a distributed data storage system designed for fast data sharing in big-data processing applications [1], similar to HDFS [2]. Crail's primary object is performance, by leveraging the full capability of the networking and storage hardware it runs on. To achieve this, it leverages a zero-copy user-space IO library over RDMA [3] networking protocol for control and data path communication. Crail supports several tiers of storage, such as memory and block storage. To achieve close to bare-metal performance of block SSDs, Crail leverages NVMe-oF [1].

In a traditional distributed storage system that presents a key-value interface to clients, keys and values must eventually be mapped onto a block interface as that is the primary interface exposed by traditional SSDs. In HDFS, management of individual blocks is performed by DataNodes; while with Crail the NameNode is responsible for that. The fundamental limitation of both approaches is that a key-value interface exposed to clients in these distributed systems requires implementation of an additional indirection layer to map key-value data structures onto a block interface exposed by traditional block storage devices, such as SSDs.

This block mapping is unnecessary if the underlying storage devices are comprised of KV-SSDs, where the key-value data structure exposed to the client can be preserved down to the individual KV-SSD. A KV-SSD is a new storage device that exposes a key-value interface [4], and we discuss it further in Section II. In this paper, we leverage KV-SSDs and introduce *Crail-KV*, an extension to the Crail distributed storage system architecture, which eliminates the block-layer altogether, and provides a distributed KV store with native access to a KV-SSD from client-facing key-value interfaces. Crail-KV achieves both better performance and better scalability by offloading file data management to the KV-SSDs themselves. Crail-KV leverages the thin IO software stack that Crail already offers, and improves scalability by reducing the overall number of RPCs and minimizing the NameNode's metadata overhead.

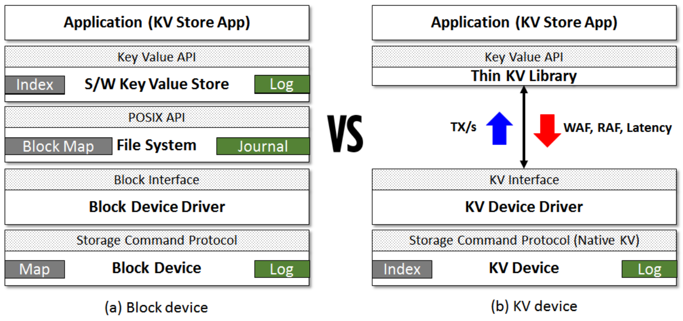


Fig. 1: Software stack for block device and KV device (KV-SSD).

This paper makes three main contributions: First, we introduce KV-DiSNI, an extension to the Java-based zero-copy NVMe-oF protocol [5] by exposing native key-value interfaces (e.g., put(), get(), delete()) to the initiator and a target implementation that operate directly on KV-SSDs. Second, we have extended Crail with a new storage plug-in that provides native key-value accesses as part of Crail's primary I/O path, enabling clients to issue key-value operations directly to a KV-SSD through KV-DiSNI. Finally, we have enhanced the NameNode (i.e., master node) of Crail to manage distributed storage residing on KV-SSDs.

II. KV-SSD

Key-value store applications such as MongoDB [6]) can run on top of key-value storage engines like WiredTiger [7] and RocksDB [8]; such a key-value stack is shown in Figure 1(a). These storage engines are built upon a file system, which eventually map and operate on a block device with a fixed sector size (typically 512 bytes). In this scenario, there are multiple mapping layers: a mapping from keys to files, and another from files to blocks. With a KV Device (KV-SSD), we can consolidate these mapping tables, eliminating the software mapping of files to blocks and move physical space management into the SSD itself, as seen in Figure 1(b). Moreover, by eliminating the file system and its associated overheads, previously over-provisioned space can be used by the application.

Additionally, a KV-SSD can mitigate the read and write amplification typically associated with SSD performance degradation [9] by leveraging the semantic information provided to it through key-value operations. When a KV-SSD is explicitly told to delete key-tuples, it can free up the associated NAND blocks referenced by said key-value tuple. Whereas, with a block-device based key-value store implementation, the block device may not be notified when a block is freed after key-value tuple deletion.

The prototype KV-SSD we used to build Crail-KV on is an extension to a Samsung SSD. Largely, implementing the KV-SSD consisted of device firmware changes to support a larger key-space; block devices already support fixed-length keys, in the form of logical block addressing, which also requires a mapping to physical page locations through the SSD's flash translation layer. The KV-SSD exports native key-value interfaces: *put*, *get*, *delete*, and *exist*. The current host-interface supports key-lengths of up to 16 bytes, while the value size can range from 64 bytes to 28 kilobytes. Support for larger key and value sizes are being developed. Additionally, we used a version of the KV-SSD that relies on a user-level driver through SPDK that uses polling to provide low-latency host operations.

III. Architecture and Design

In traditional master-slave distributed storage systems, the master (NameNode) is responsible for managing file system metadata. Directories store entries inside files and directories, while files store user content. Files are comprised of an ordered set of blocks, each block residing on a storage node in the distributed storage system. Crail is an implementation of this architecture. However, it is different from HDFS by managing every physical block's location in the entire distributed system within the NameNode. Crail does this to enable direct low-latency access to storage. This high-speed I/O path comes at the cost of increased metadata management in the NameNode. In the following subsection, we describe Crail-KV's design and how it improves the overall Crail architecture by leveraging KV-SSDs.

A. NameNode

By leveraging KV-SSDs as the fundamental building block for storage, our Crail-KV design offloads metadata management to the KV-SSDs while still providing clients low-latency access to the storage device. The high-level architecture of Crail-KV is shown in Figure 2. Similar to most other master-slave distributed storage systems, the NameNode in Crail-KV is responsible for managing the namespace and data locations that reside on DataNodes. The clients communicate with the

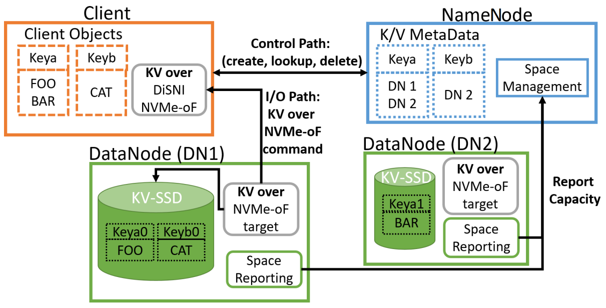


Fig. 2: Crail-KV architecture: NameNode performs space management for key-value tuples, while clients have low-latency I/O access to KV-SSDs through KV-DiSNI over NVMe-oF.

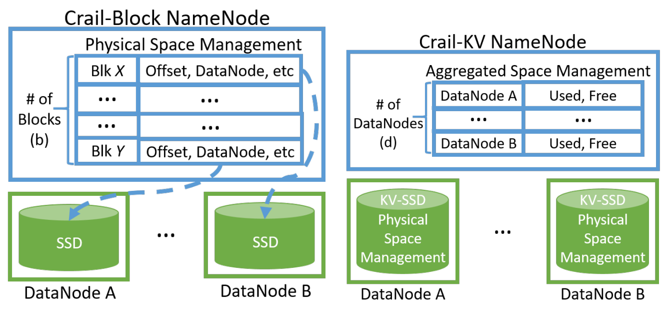
NameNode to retrieve the location of the key-value tuples they are interested in.

1) NameNode Metadata Management: The main architectural change enabling Crail-KV is a re-structure of metadata management: instead of having a list of "BlockInfos" (Crail data structure pointing to the physical location of a block on remote storage) compose a client's value, a Crail-KV client's value is an ordered list of one or more DataNodes, that together store a client's value. Each DataNode stores its portion of the client's key-value tuple natively in a KV-SSD, thus offloading physical storage management to the KV-SSDs. This data structure provides flexibility and control: a key-value tuple can reside entirely in a single KV-SSD, or be stripped across multiple KV-SSDs, as shown in Figure 2 with *Keya*, trading off fewer RPCs for more data distribution and aggregation. The amount of data stored per KV-SSD is determined by the striping factor, instead of block size, which no longer exists in Crail-KV.

When mapping a client's key-value tuple to internal key-value tuples on the KV-SSD, we choose to use the same key id for each internal key-value pair on the KV-SSD in a DataNode. We also append the index to the key stored in the KV-SSD, also shown in Figure 2. The index of the DataNode in the key-value metadata file determines the order of that partial value in the client's key-value tuple. This index reflects the internal key-value tuple's location in the client's tuple. With this explicit index appended to the key, we can store multiple internal key-value tuples on the same DataNode for the same client key-value tuple. Finally, the index also enables the system to perform additional data verification and recovery during NameNode data loss; all of the keys residing on all DataNodes can be scanned, and the indexes used to determine its location in a client key-value tuple, making reconstruction of all key-values possible.

Additional information can be appended to the internal key enabling more storage management features, such as CRC values to ensure the correctness of internal value content, or replication information.

2) Space Management: In a distributed storage system based on blocks, a software layer must manage individual blocks. With Crail, the NameNode manages the physical blocks residing on all storage nodes. This is shown in Figure 3a. Block management is no longer necessary in a distributed system based entirely on KV-SSDs. Crail-KV still requires space management, but is far simpler. Only two scalar values are required per DataNode to account for space in Crail-KV: free and used capacity. This is shown in Figure 3b. When the NameNode receives a request to store a key-value tuple of a certain size, it consults its map of KV-SSDs and their available capacity to make a decision about which DataNode(s) should store that client's key-value tuple. The decision is policy based, for example, round-robin. After telling a client to store a key-value tuple on a particular DataNode, the NameNode will then reduce the free capacity of that DataNode by the corresponding value size.



(a) Crail baseline: NameNode manages individual blocks residing on SSDs.

(b) Crail-KV: NameNode manages aggregate KV-SSD space, while each KV-SSD internally manages its own physical space.

Fig. 3: Space management for baseline Crail and for our Crail-KV design, where ($b \gg d$).

3) Control-Path Interfaces: We have introduced four new interfaces in the NameNode, enabling the primary key-value client support in Crail-KV. *Register Space* is the only interface specific to DataNode. The rest are driven by clients and manipulate the NameNode's key-value metadata. The four interfaces are:

- *Register Space*: Register space for a KV-SSD residing on a DataNode.
- *CreateKV*: Create a key-value tuple with a specific size.
- *LookupKV*: Find and return the key-value metadata to a client.
- *DeleteKV*: Delete a key-value tuple.

B. DataNode

Like Crail, Crail-KV DataNodes are light-weight: a client issues I/O directly to targets exported by a DataNode, freeing up the DataNode's CPU. The new contribution in a Crail-KV DataNode is that it exposes key-value based interfaces for clients to issue I/O against. The new DataNode interfaces are:

- *put*: store a key-value tuple.
- *get*: return the value for given key.
- *delete*: delete a key-value tuple.

Delete is issued by the NameNode, while *put* and *get* are called from clients. The DataNode remains extremely lightweight while executing these operations because the targets themselves handle I/O operations without invoking the DataNode's JVM run-time.

1) Space Management: An advantage of a KV-SSD based DataNode over a block-based Crail DataNode is that a Crail-KV DataNode only needs to register its total capacity rather than each individual block; the KV-SSD manages internal physical storage representing the content of key-value tuples. This reduces the amount of time for a DataNode to come online.

2) Low-latency Key-Value I/O Path: The existing I/O path with the NVMe-oF protocol only supports transferring data in blocks with a fixed size and a pre-determined address on the target device (e.g., SSD) [10]. To transfer key-value pairs remotely, we extended the NVMe-oF protocol to understand arbitrary sized keys and values, not just a fixed block address and length. DiSNI is a Crail module that provides a Java-based storage IO interface and integrates multiple APIs onto the NVMe-oF protocol. We built KV-DiSNI by extending DiSNI to add key-value interfaces supported by the KV-SSD: get(), put(), and delete(). These new NVMe-oF commands ultimately terminate at the KV-SSD. Our extension maintains the zero-copy data transfer between the client and remote KV-SSD.

On the client-side, this transformation from Java to the c-based NVMe-oF APIs only involves pointer passing between the JVM and the client-side NVMe-oF driver. At the target-side, the DataNode JVM does not need to process the payload; the user-space NVMe-oF target receives the key-value tuple directly, then issues the corresponding KV-SSD operation using the received pointers, again without a data copy.

C. Crail-KV Operation Flow

Tying the design together, Figure 4 shows an example flow of control and data-path operations in Crail-KV. Initially, each KV storage-based DataNode registers its total storage capacity (free space) to the NameNode with "setKVInfo" call. When a client creates a KV tuple during runtime, it sends "createKV" command with specified value size to the NameNode. Correspondingly, the NameNode will choose a DataNode to store the key-value tuple, update the metadata for the KV tuple as well as free/used storage space for that DataNode. The NameNode then replies to the client with the file metadata ("KVInfo") including the DataNode's information. Finally, the client issues a KV put to the assigned DataNode through native KV NVMe-oF, directly to KV-SSD.

D. Advantages of Crail-KV

Integrating new KV-SSD functionality into a distributed storage system design provides several architectural benefits:

Minimize overhead of individual block management on NameNode: With a distributed file system like Crail, whose NameNode individually manages blocks on storage nodes, its scalability is limited by management of this metadata, and exacerbated by the necessary bookkeeping for each block's physical location. However, by using a key-value SSD, the NameNode needs not manage individual block information.

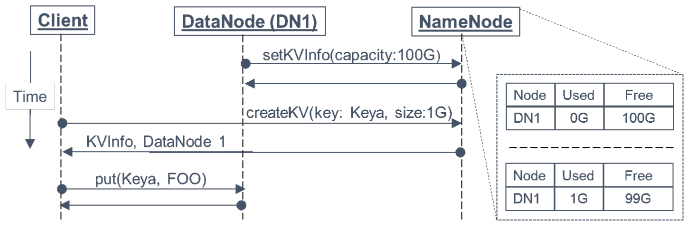


Fig. 4: The process of an I/O: 1) a DataNode registers KV-SSD capacity to the NameNode, 2) a client creates a KV tuple, 3) free capacity decrements by corresponding value size, and 4) the client issues a KV put.

The NameNode is only responsible for storing the location (i.e., DataNode) of each key-value tuple, substantially reducing the metadata maintained in a large distributed file system.

Eliminate fixed block size: The fixed block size is an artifact of choosing a block size at initialization, resulting in runtime trade-offs between space efficiency and communication overhead. With the native key-value store, that is unnecessary; the Crail-KV DataNode can rely on the KV-SSD to persistently store and manage arbitrarily sized data in its key-value tuples.

Fewer RPCs: Since there are no blocks in the storage system and a KV-SSD can store the entire key-value tuple for an application, fewer RPCs are necessary. In block-based distributed storage, a trade-off must be made when choosing the block size: smaller block size incurs less fragmentation but leads to more RPCs for larger files along with additional NameNode metadata; larger block size results in less NameNode metadata and fewer RPCs, but increases fragmentation for small key-value sizes.

IV. Results

A. Crail-KV Performance Measurement Methodology

In this section we describe the evaluation methodology for our Crail-KV performance characterization. The benchmarks we developed include KV *put* and *get* in both synchronous and asynchronous modes. We use Dell Broadwell servers, and the server configuration is shown in Table I. We use separate servers for client, NameNode, and DataNode, and measure I/O throughput and latency to the remote SSDs in the Crail and Crail-KV cluster. The cluster setup is the same for both Crail ad Crail-KV with the exception that Crail-KV uses a KV-SSD. For Crail-KV, the I/O path is zero-copy to remote KV-SSD through KV-DiSNI, SPDK and the NVMe-oF interface. From now on, we refer to the baseline Crail as "Crail-Block" to highlight its difference from Crail-KV.

Server	Broadwell Dell PowerEdge R730XD
Processor	Intel E5-2670 v3 @ 2.30GHz (12cores)
Storage	Samsung PM983 SSD, KVSSD, 3.84TB
Memory	512GB
Network	25 GbE, RoCE
OS, kernel	Ubuntu 16.04, kernel v4.9.16

TABLE I: Configuration of a single server.

B. NameNode MetaData Efficiency

Crail-KV does not require a block abstraction, thus does not need to store associated block metadata. As a result, Crail-KV provides better NameNode scalability with respect to memory usage. Figure 5 shows the existing NameNode memory overhead in Crail-Block system for storing block and file metadata. The memory consumption results from the NameNode keeping a reference to every block (along with metadata describing its size and physical location) in memory. Accordingly, the overhead is proportional to the ratio of total storage capacity registered to the NameNode divided by the block size.

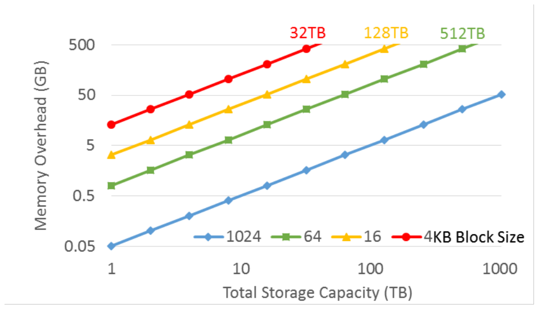


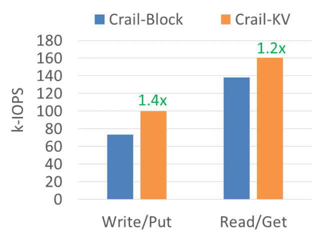
Fig. 5: Crail-Block system's NameNode memory overhead for storing block metadata. Note both x and y axis are in logarithm scale.

This memory overhead grows drastically as the total storage capacity in the system increases, and eventually exceeds the memory capacity available on the NameNode (512GB). For example, to enable total capacity of 32TB, it will consume 416GB of NameNode memory to manage all blocks with 4KB size. Similarly, the maximum capacity a NameNode with 512GB memory can support is 128TB and 512TB for 16KB and 64KB block sizes, respectively. Note that there is additional memory consumption — namely the NameNode JVM process — that also limits available memory to store block metadata.

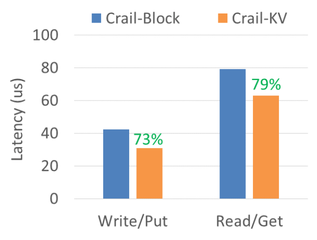
Crail-KV completely eliminates the overhead of managing individual blocks, enabling minimal memory usage and scalability independent of the block count. As a result Crail-KV requires $O(1)$ space usage to support any capacity because only two scalar values (used and free storage) are required per DataNode for space management. In other words, the memory consumption shown in Figure 5 will significantly decrease with Crail-KV.

C. Performance Evaluation of Crail-KV vs. Crail-Block

1) Random IO Results: First we measure the random IO performance of a single client to a single remote DataNode, including latency and throughput for both block read/write and KV put/get. The results are shown in Figure 6, where the file/value size is 4KB. For IOPS, as in Figure 6a, the queue depth is set to 128 to ensure a maximum IOPS. As we can see, for random IO with 4KB granularity, Crail-KV put and get has 1.4x and 1.2x of IOPS, and 27% and 21% better latency than Crail-Block, respectively. This is mostly because Crail-KV has a thin software stack in the IO path, and eliminates the overhead of file opening, closing, and streaming from the Crail-Block system. Particularly, the performance of asynchronous KV put and get is similar to that of a raw KV-SSD when measured on the host (within 3% difference). This means that Crail-KV can fully expose the bare-metal hardware throughput to the application.



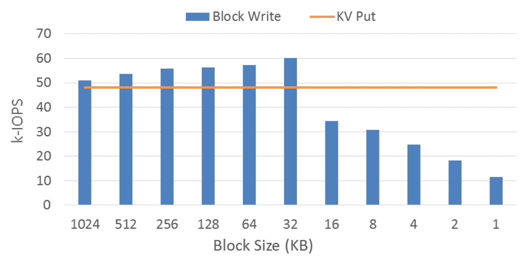
(a) IOPS of random IO (4KB, asynchronous with 128 queue depth).



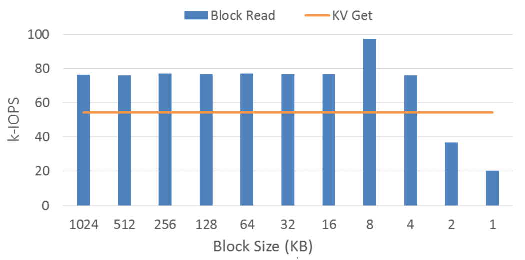
(b) Latency of random IO (4KB, synchronous).

Fig. 6: Random IO performance comparison between Crail-Block and Crail-KV.

2) Block Size Independence: Crail-Block is inflexible at mapping block size to file size at run-time as the block size is determined at initialization time; however, Crail-KV does not have this issue. To demonstrate this, we extended our experiments and show the performance of Crail-Block with multiple block sizes. We choose a file size of 28KB, and sweep the block size from 1MB to 1KB. The file size of 28KB is chosen because that is the current maximum KV-SSD value size, though future KV-SSD versions will have larger maximum value.



(a) Client-side file operations/sec of large files with Crail-Block write vs. Crail-KV put.

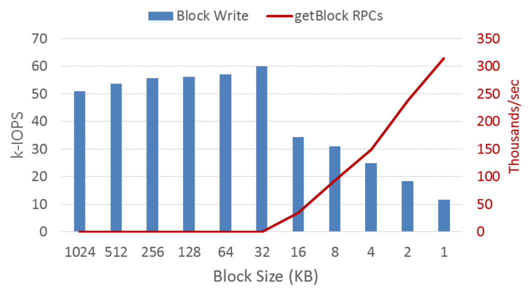


(b) Client-side file operations/sec of large files with Crail-Block read vs. Crail-KV get.

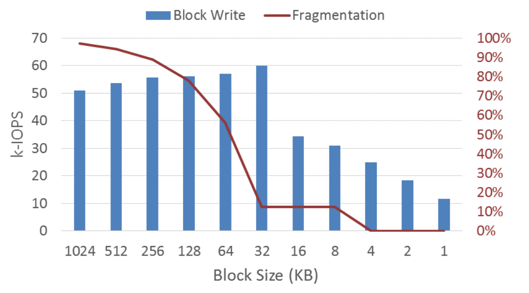
Fig. 7: IOPS of Crail-KV compared against baseline Crail-Block (over different block sizes).

As Figure 7a shows, Crail-KV performance is constant because there is no block abstraction in the system. On the other hand, when the block size is greater and can fit the file size, Crail-Block performs slightly better than Crail-KV due to better sequential read and write performance from the block SSD drives. Contrarily, when the block size is small (16KB - 1KB), a file composes multiple blocks and Crail-Block performs significantly worse due to the overhead incurred in both hardware and software to map a file to multiple blocks. The Crail-Block performance degrades down to the lowest block size we used - 1KB. The results together show that Crail-KV has better IOPS than Crail-Block at a large file-size/block-size ratio.

Figure 7b shows a similar trend of performance reduction with smaller block sizes for block read. Because the block SSD we used has hardware performance optimized at 4KB and 8KB block sizes, there is a performance increase for these two block sizes, offsetting the performance reduction. However, at 2KB and 1KB block sizes, Crail-KV still outperforms Crail-Block significantly.



(a) Crail-Block overhead in client-NameNode RPCs to get block metadata.



(b) Crail-Block overhead in fragmentation due to mismatched file and block sizes.

Fig. 8: Overhead of Crail-Block in performance and storage space because of the artifact of blocks.

3) Overhead of Crail-Block: One drawback of a block-based system like Crail is that it incurs a large number of RPCs between the client and the NameNode. In Crail-Block, when a client needs to write to a file, it must first retrieve the metadata for the blocks belonging to the file by issuing "getBlock" RPC request to the NameNode. The number of getBlock RPCs issued is a function of the file size over the block size; RPC count increases when decreasing the block size or increasing the file size. In addition, these getBlock

RPCs to the NameNode are serialized in the Crail-Block system, forcing clients to wait for RPC completion, throttling I/O performance. Figure 8a shows that with a decreasing block size, RPC count increases proportionally (on the second axis), and is partially responsible for the lower performance at small block sizes of Crail-Block. This problem will become more prominent in a large cluster and/or with lower-speed networking when concurrent RPC requests are queued and delayed in the NameNode.

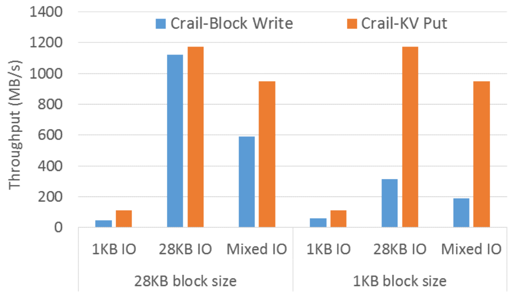
Moreover, another disadvantage of the block abstraction is the storage efficiency trade-off in the existing Crail-Block design. On the second axis, Figure 8b shows the ratio of storage space wasted in Crail-Block due to internal fragmentation from mismatched block and file size (lower is better). Note that the same conclusion applies to read/get scenario in Figure 7b as well. As we can see, the larger the block size, the more internal fragmentation, up to 97% for 1MB block size. Without knowing the block and file size *a priori*, Crail-Block must incur internal fragementation in order to accomodate the largest file size to out-perform Crail-KV.

Predicting the optimal block and file size is difficult as applications and their workloads vary over time [11]. Therefore, in Crail-Block, a significant trade-off must be made: sacrifice potentially multiple orders of magnitude in space for performance, or performance for space. However, with Crail-KV, this trade-off is not necessary – it enables performance and storage efficiency by eliminating the block abstraction independence and accommodating applications with varying file sizes.

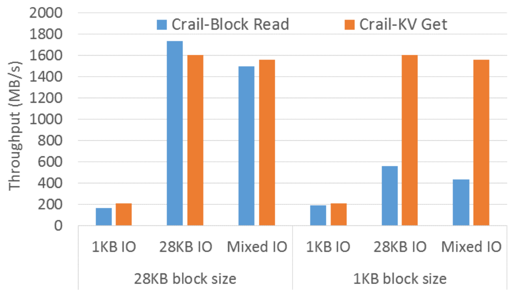
4) Mixed IO Size Results: We are also interested in the performance of Crail with mixed workloads in a real cluster. Figure 9 shows that for two clients with mixed IO sizes, Crail-KV provides higher aggregated throughput than Crail-Block. We use representative IO sizes of 28KB and 1KB for each of the two clients – the two extreme file sizes offered by our KV-SSD. Although file sizes vary across workloads, 28KB can capture up to 90% of the files in server-based workloads [12]. In the Crail-Block case, we also use two block sizes: 1KB and 28KB. The file-size I/O for both Crail-Block and Crail-KV is either 1KB, 28KB, or a mix of both.

As we can see, with 1KB block size, Crail-Block performance is significantly worse than that of Crail-KV in all scenarios. Particularly, for mixed-IO workloads, Crail-KV's throughput is as high as 5x that of Crail-Block with 1KB block size. This is because there is large overhead associated with writing/reading multiple small blocks. Although tuning the block size to match the larger 28KB file size allows Crail-Block read to outperform Crail-KV get, when we have mixed IO sizes from different clients, the aggregated throughput of Crail-KV still outperforms Crail-Block. The throughput degradation results from the interference of mixed-size operations and Crail-Block's inability to provide both clients with an optimal block size.

Our results demonstrate that a single block size does not fit all application file sizes. Specifically, common mixed-IO-size workloads show the impacts that a fixed block-size has on



(a) Mixed-IO-size block write vs. KV put.



(b) Mixed-IO-size block read vs. KV get.

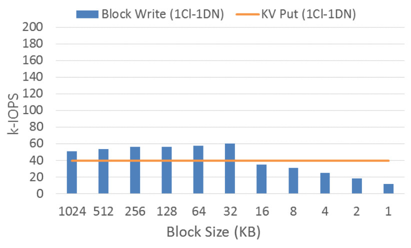
Fig. 9: Mixed-IO-Size results of Crail-Block vs. Crail-KV.

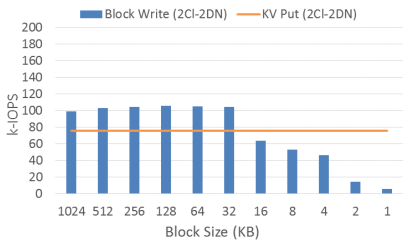
performance. With Crail-KV, a mixed-IO-size workload has less impact because Crail-KV can satisfy both file sizes with a single RPC, by offloading the physical space management of both sizes to the KV-SSD itself, enabling it to choose the right location(s) to place data.

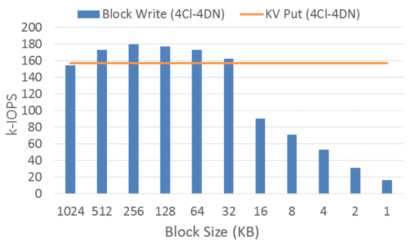
5) Scalability wtih Multi-Nodes: Crail-KV is advantageous for its ability to provide close to linear scalability with respect to node count. Figure 10 shows our evaluation of scalability of Crail-KV over multiple nodes and a comparison against that of Crail-Block. Specifically, we scale the number of clients in the same way as the DataNodes: from 1, 2, to 4 nodes, and show the aggregated throughput across all clients. As we can see, Crail-KV's aggregated throughput (lines) scales linearly from 39, 76, to 156 k-IOPS for KV put corresponding to 1, 2, and 4 nodes. The KV get throughput presents a similar linear growth trend. On the other hand, for Crail-Block, the performance growth (bars) is also similar and close to linearity, but is dependent on the block size chosen. The performance gap between Crail-Block and Crail-KV gets bigger for small block sizes with four nodes.

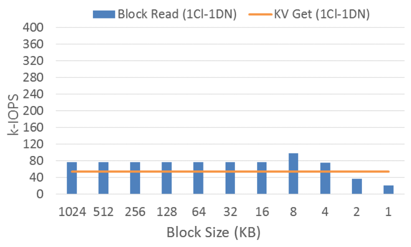
To reiterate, we've chosen a file size of 28KB for both Crail-Block and Crail-KV in order to show Crail-KV's maximum I/O performance with respect to the current KV-SSD prototype. This explains the knee in the Crail-Block plots, which is roughly between 16KB and 32KB. With a larger maximum KV-SSD value size, the knee should also shift to the left.

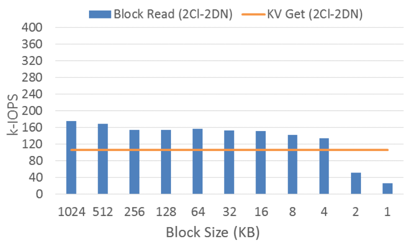
As mentioned earlier, Crail-KV deploys a light-weight client with a thin IO stack and has eliminated the block abstraction. Correspondingly, there are low software overheads at the client side, which result in better scalability over multiple DataNodes. Figure 11 shows the multi-node performance evaluation with a slightly different setup: the DataNode count increases


(a) 1-client, 1-DN, block write vs. KV put.


(b) 2-client, 2-DN, block write vs. KV put.


(c) 4-client, 4-DN, block write vs. KV put.


(d) 1-client, 1-DN, block read vs. KV get.


(e) 2-client, 2-DN, block read vs. KV get.

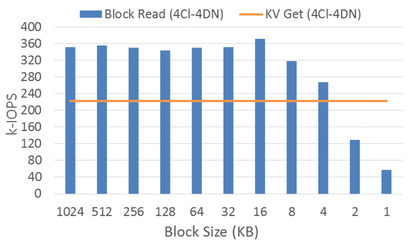
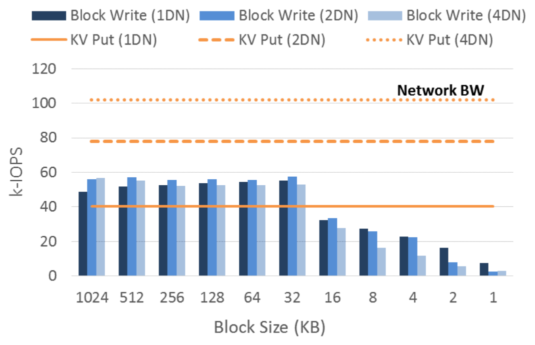

(f) 4-client, 4-DN, block read vs. KV get.

Fig. 10: Scalability comparison between Crail-Block (bars) and Crail-KV (line) over aggregated client-side operations/sec. We increase node count from 1, 2, to 4, and DataNode is denoted as “DN”.


(a) Single-client, multi-DataNode, block write vs. KV put.

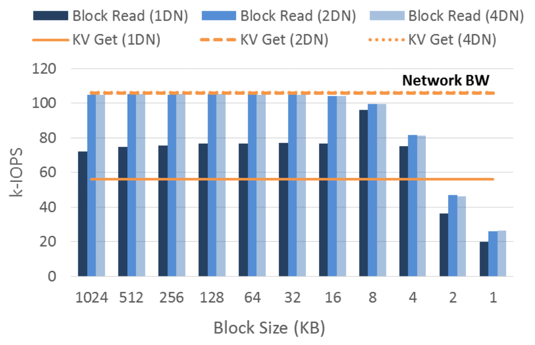

(b) Single-client, multi-DataNode, block read vs. KV get.

Fig. 11: Performance comparison between Crail-Block and Crail-KV with single client and multiple DataNodes.

from 1, 2, to 4, while we keep a single client. As shown in Figure 11a, Crail-KV still demonstrates a linear growth in KV put throughput, and eventually reaches the limitation of network bandwidth at 25Gb/s for a single client in the 4-DataNode case. However, Crail-Block's write performance does not scale with more DataNodes and cannot reach network bandwidth, and consistently decreases from 2 to 4 DataNodes. For the block-read/KV-get case, Figure 11b shows that at two clients, both Crail-Block and Crail-KV saturates the network bandwidth over large block sizes. And for small block sizes, the conclusion is the same that Crail-KV outperforms Crail-Block because of better scalability.

V. Related Work

Key-value devices have previously been explored in both academia and industry. The Kinetic key-value HDD was a commercial attempt to provide an object storage device [13], which presented a key-value interface over ethernet, enabling clustered storage services to be built from multiple kinetic drives. The KV-SSD also provides a key-value interface, but its goals are more incremental – provide a more flexible interface for flash-based storage, resulting in high performance. Similar to KV-SSD, recent research has been done to bridge the semantic gap between key-value caches and the SSDs with block interfaces. Specifically, the DIDACAche [14] provides a key-value interface using an open-channel SSD, mapping values directly to flash blocks. Our work leverages physical SSDs that expose a key-value interface and could potentially run on a DIDACache-based SSD.

There is plethora of persistent key-value stores designed for local systems such as RocksDB [15], WiscKey [16], and LevelDB [17], which use an LSM tree-based design [18] to leverage the sequential write performance that today's commercial SSDs offer. These single node systems are what physical key-value SSDs aim to replace. Our work builds upon the functions that local key-value stores provide. To that end, any local key-value store that have high throughput and low latency access could be used in a Crail-KV DataNode.

Many distributed key-value stores implement a key-value interface by modifying or extending the file interface to support key-value operations. For example, Hadoop's Ozone provides an object storage interface on top of HDFS by re-implementing the NameNode's metadata for an object, which is still built on the block abstraction [19]. Crail-KV differs from Ozone in that the metadata tracks key-value locations, offloading the management of physical space to the individual KV-SSDs, thus removing the block-abstraction. Additionally, the Cloud hyperscalers all provide object based storage, including Amazon S3 [20], Microsoft Blob Storage [21], and Google Cloud Storage [22], and provide extreme scalability. We believe that implementing the Crail-KV architecture in these hyper-scale storage system will offer similar efficiency gains.

Comet is an active distributed key-value system. It it similar to Crail-KV in that it leverages key-value storage engines as the primary building block. However, it differs from Crail-KV in that it has active handlers associated with the value [23]. These handlers are routines which travel with data operations, enabling data transformation, such as replication. FlashStore is a high-throughput key-value storage system [24]. Its intent is to maximize the throughput of flash. They achieve this by adding in-memory data structures to minimize the number of flash-based metadata operations. With Crail-KV all of the metadata has been off-loaded to the KV-SSD itself.

Finally, distributed NoSQL stores, which are built on-top of pluggable storage engines that implement a local key-value store are similar in spirit to Crail-KV. For example, project Voldermort [25] from LinkedIn and Cassandra [26] provide additional database operations, such as transactions. Such operations are orthogonal to Crail-KV's goals, which is distributed file system replacement, but with with key-value semantics.

VI. Conclusion

In this paper we have introduced KV-SSDs, which provide an alternative interface to the physical storage device (i.e., get()/put()/delete()), and remove the block interface alltogether. We have leveraged these KV-SSDs to improve the system efficiency of the Crail distributed storage system by removing the block translation from Crail and allowing clients to issue native key-value operations over NVMe-oF directly to KV-SSDs. Removing the block abstraction from Crail eliminates two of the biggest limitations in a master-slave distributed storage systems: memory-based block management, and multiple RPC block requests per file.

Our performance results show that when the file size is aligned to the block size, the performance of Crail is on-par with that of Crail-KV. However, Crail-KV shows its advantage with mixed size workloads: Crail's single block size becomes the limiting factor for large files, and a large block size introduces internal fragmentation for smaller files. Conversely, Crail-KV offloads key-value storage mapping to the KV-SSD itself, thus enabling Crail-KV to provide minimum metadata overhead regardless of file size.

References

[1] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance i/o architecture for distributed data processing," *IEEE Data Eng. Bull.*, vol. 40, pp. 38–49, 2017.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[3] J. Hilland, P. Cully, J. Pinkerton, and R. Recio, "Rdma protocla verbs specification (version 1.0)," http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf.

[4] Samsung, "Samsung key vaue ssd enables high performance scaling," https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Key_Value_Technology_Brief_v7.pdf.

[5] P. Stuedi, B. Metzler, and A. Trivedi, "jverbs: Ultra-low latency for data center applications," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:14.

[6] K. Chodorow, *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[7] K. Bostic, "The wiredtiger btree implementation (wiredtiger)," in *International Workshop on High Performance Transaction Systems*, 2013.

[8] I. Canadi, "Mongorocks," https://github.com/mongodb-partners/mongo-rocks/wiki.

[9] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009, pp. 10:1–10:9.

[10] SPDK, "Nvme over fabrics target," http://www.spdk.io/doc/nvmf.html.

[11] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 29:1–29:15.

[12] A. S. Tanenbaum, J. N. Herder, and H. Bos, "File size distribution on unix systems: Then and now," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 100–104, Jan. 2006.

[13] Seagate, "Seagate kinetic hdd," https://www.seagate.com/www-content/product-content/hdd-fam/kinetic-hdd/en-us/docs/100764174b.pdf.

[14] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "Didacache: A deep integration of device and application for flash based key-value caching," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 391–405.

[15] Facebook, "Rocksdb," https://rocksdb.

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 133–148.

[17] Google, "Leveld - a fast and lightweight key/value database library," http://code.google.com/p/leveldb.

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[19] J. Pandy, "Ozone: An object store in hdfs," https://hortonworks.com/blog/ozone-object-store-hdfs/.

[20] Amazon Web Services, "Amazon s3," https://aws.amazon.com/s3/.

[21] Microsoft Azure, "Blob storage," https://https://azure.microsoft.com/en-us/services/storage/blobs/.

[22] Google, "Google Cloud Platform," https://cloud.google.com/storage/.

[23] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 323–336.

[24] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1414–1425, Sep. 2010.

[25] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 18–18.

[26] D. Gu, "Cassandra on rocksdb," https://www.percona.com/live/18/sessions/cassandra-on-rocksdbHortonworkshttps://hortonworks.com/blog/ozone-object-store-hdfs/.