Unification of Temporary Storage in the NodeKernel Architecture

Patrick Stuedi, *IBM Research;* Animesh Trivedi, *Vrije Universiteit;*
Jonas Pfefferle, *IBM Research;* Ana Klimovic, *Stanford University;*
Adrian Schuepbach and Bernard Metzler, *IBM Research*

https://www.usenix.org/conference/atc19/presentation/stuedi

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Unification of Temporary Storage in the NodeKernel Architecture

Patrick Stuedi[†] Animesh Trivedi[‡] Jonas Pfefferle[†] Ana Klimovic[§]
Adrian Schuepbach[†] Bernard Metzler[†]
[†]*IBM Research* [‡]*Vrije Universiteit* [§]*Stanford University*

Abstract

Efficiently exchanging temporary data between tasks is critical to the end-to-end performance of many data processing frameworks and applications. Unfortunately, the diverse nature of temporary data creates storage demands that often fall between the sweet spots of traditional storage platforms, such as file systems or key-value stores.

We present NodeKernel, a novel distributed storage architecture that offers a convenient new point in the design space by fusing file system and key-value semantics in a common storage kernel while leveraging modern networking and storage hardware to achieve high performance and cost-efficiency. NodeKernel provides hierarchical naming, high scalability, and close to bare-metal performance for a wide range of data sizes and access patterns that are characteristic of temporary data. We show that storing temporary data in Crail, our concrete implementation of the NodeKernel architecture which uses RDMA networking with tiered DRAM/NVMe-Flash storage, improves NoSQL workload performance by up to 4.8× and Spark application performance by up to 3.4×. Furthermore, by storing data across NVMe Flash and DRAM storage tiers, Crail reduces storage cost by up to 8× compared to DRAM-only storage systems.

1 Introduction

Managing temporary data efficiently is key to the performance of cluster computing workloads. For example, application frameworks often cache input data or share intermediate data, both both within a job (e.g., shuffle data in a map-reduce job) and between jobs (e.g., pre-processed images in a machine learning training workflow). Temporary data storage is also increasingly important in serverless computing for exchanging data between different stages of tasks [17].

Storing temporary data efficiently is challenging as its characteristics typically lie between the design points of existing storage platforms, such as distributed file systems and key-value stores. For instance, shuffle data in a map-reduce job may consist of a large number of files which are organized hierarchically, vary widely in size, are written randomly, and read sequentially. While file systems (e.g., HDFS) offer a convenient hierarchical namespace and efficiently store large datasets for sequential access, distributed key-value stores are optimized for scalable access to a large number of small objects. Similarly, DRAM-based key-value stores (e.g., Redis) offer the required low latency, but persistent storage platforms (e.g, S3) are more suitable for high capacity at low cost. Overall, we find that existing storage platforms are not able to satisfy all the diverse requirements for temporary data storage and sharing in distributed data processing workloads.

In this paper we present *NodeKernel*, a new distributed storage architecture designed from the ground up to support fast and efficient storage of temporary data. As its most distinguishing property, the NodeKernel architecture fuses file system and key-value semantics while leveraging modern networking and storage hardware to achieve high performance. NodeKernel is based on two key observations. First, many features offered by long-term storage platforms, such as durability and fault-tolerance, are not critical when storing temporary data. We observe that under such circumstances, the software architectures of file systems and key-value stores begin to look surprisingly similar. The fundamental difference is that file systems require an extra level of indirection to map offsets in file streams to distributed storage resources, while key-value stores map entire key-value pairs to storage resources. The second observation is that low-latency networking hardware and multi-CPU many-core servers dramatically reduce the cost of this indirection in a distributed setting by enabling scalable RPC communication at latencies of a few microseconds.

Based on these insights we develop the NodeKernel architecture by implementing file system and key-value semantics as thin layers on top of a common storage kernel. The storage kernel operates on opaque data objects called “nodes” in a unified namespace. Applications can store arbitrary size data in nodes, arrange nodes in a hierarchical namespace, and obtain file system or key-value semantics through specialized

node types if needed. For instance, key-value and table nodes permit concurrent creation of nodes with the same name, offering last-put-wins semantics. On the other hand, file and directory nodes permit efficient enumeration of data sets at a given level in the storage hierarchy. By splitting functionality in a common storage kernel and custom node types, NodeKernel enables applications to use a single platform to store data that may require different semantics, while generally offering good performance for a wide range of data sizes and access patterns.

NodeKernel is designed explicitly with modern hardware in mind. Following strict separation of concerns, the storage kernel is composed of a lightweight and scalable metadata plane tailored to low-latency networking hardware and an efficient data plane that provides access to multiple tiers of network-attached storage resources. The metadata plane is trimmed down to offer only the most critical functionality, with low overhead. The data plane runs a lightweight software stack and leverages modern networking and storage hardware to achieve fast access to arbitrary size data sets while also optimizing cost efficiency. For instance, data attached to "nodes" may either be pinned to a particular storage technology tier or may spill from one storage tier to another depending on performance and cost requirements.

Crail is our concrete implementation of the NodeKernel architecture using RDMA networking and two storage tiers based on DRAM and NVMe SSDs respectively. We evaluated Crail on a 100Gb/s RoCE cluster equipped with Intel Optane NVMe SSDs using raw storage microbenchmarks as well as using the NoSQL YCSB benchmark and different Spark workloads. Our results show that Crail matches the performance of current state-of-the-art file systems and key-value stores when operated in their sweet spot, and outperforms existing systems up to 3.4× for data accesses outside the sweet spot of file systems and key-value stores. Moreover, Crail creates new opportunities to reduce cost and gain flexibility with almost no performance penalties by using NVMe Flash in addition to DRAM. For instance, using Crail to store shuffle data in Spark allows us to adjust the ratio between DRAM and Flash with only a minimal increase in job runtimes.

In summary, this paper makes the following contributions:

- We propose NodeKernel, a new storage architecture fusing file system and key-value semantics to best meet the needs of temporary data storage in data processing workloads.
- We present Crail, a concrete implementation of the NodeKernel architecture using RDMA, DRAM, and NVMe Flash.
- We show that storing temporary data in Crail reduces the runtime and cost of data processing workloads. For instance, Crail improves performance up to 4.8× for NoSQL workloads. When integrated in Spark's shuffle and broadcast services, Crail improves application performance up to 3.4× and reduces cost up to 8×.

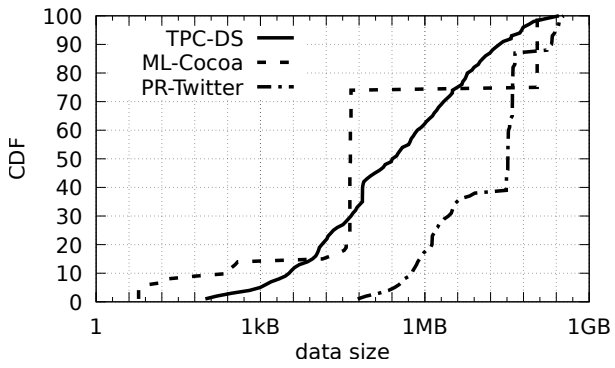


Figure 1: CDF of the size of intermediate data written or read per compute task in Spark for different workloads.

Crail is an open source Apache project [2, 3] with the code available for download from the project website as well as directly from GitHub at https://github.com/apache/incubator-crail. Further, all benchmarks used in this paper are open source.

2 Background and Motivation

Temporary data represent a large and important class of in-processing data in analytics frameworks. For example, Zhang et al. report that over 50% Spark jobs executed at Facebook contain at least one shuffle operation, generating significant amounts of temporary data [37].

We define temporary data as the multitude of all application data being created, handled, or consumed during processing, excluding the original input and final output data. Specifically, we identify three distinct classes of temporary data: intra-job, inter-job, and cached input/output datasets. Intra-job temporary data is generated within a framework when executing a single job like page-rank, or a SQL query. Common examples are datasets generated during shuffle or broadcast operations in frameworks like Spark, Hadoop or Flink. Such data is typically generated and consumed by the same job, which puts a bound on the lifetime of the data. Inter-job temporary data are intermediate results in multi-job pipelines. For example, there are many pre-processing and post-processing jobs in a typical machine learning pipeline [35], where the output of one job becomes the input of another job. Lastly, examples of cached input/output data are mostly read-only datasets that are pulled into a cache for fast repetitive processing. For instance, users may run many SQL queries on the same table (or a view) over a short period of time. In this case, a copy of the input table might be cached on a fast storage media.

Technology	Local Latency	Remote Latency	Bandwidth	Price
DRAM	80ns	2us	10s GB/s	5$/GB
3D XPoint	5us	10us	2-3 GB/s	1.25$/GB
NAND Flash	50us	55us	2-3 GB/s	0.63$/GB

Table 1: Price and performance of DRAM (DDR4), 3D XPoint (NVMe) and NAND Flash (NVMe). Remote DRAM latency for RDMA, remote 3D XPoint and NAND Flash latency for NVMf.

Building an efficient storage platform for the different types of temporary data requires careful consideration of application demands, data characteristics and hardware opportunities. In the following section we discuss several requirements for a temporary storage platform and provide an overview of current state-of-the-art solutions.

2.1 Requirements and Challenges

Size, API, and Abstractions Diversity: Temporary data in data processing workloads can vary substantially with regard to the data size. In Figure 1 we show the size distribution (CDF) of temporary data generated per task during the execution of (a) PageRank on the Twitter graph; (b) SQL queries on a TPC-DS dataset; and (c) Cocoa machine learning on a sparse matrix dataset [24]. As shown, the per-task data sizes are ranging from a few bytes (for machine learning) to a GB (for TPC-DS). Historically, different storage systems are used to handle the two ends of this spectrum. Distributed key-value stores (e.g., RAMCloud, memcached, etc.) have an object API and are optimized to store small values efficiently for fast random lookups [20, 33]. In contrast, file systems like HDFS or Ceph, can store large datasets (GBs) efficiently by partitioning the dataset and maintaining indexes for lookups. Moreover, filesystem abstractions of appendable files, enumerable hierarchical namespace, and a streaming-byte interface for I/O provide additional support for an easy mapping of temporary datasets, such as all-to-all shuffle, to the underlying storage.

A temporary storage platform should be able to store small and large values efficiently, with the unified benefit of file and key-value abstractions in a single system.

Performance: Temporary data often lies in the critical path of data processing, hence it is imperative that access to the temporary data is fast. As with the size, the access pattern also varies widely. For example, as there is no global order, shuffle data is often written randomly [11], whereas SQL tables are read in large sequential scans [32]. Hence, one requirement a storage platform for temporary data has to fulfill is that it should be capable of performing well on the entire spectrum of data sizes for any access pattern.

Fortunately, over the last decade, I/O devices have evolved rapidly to support high-bandwidth (100s of Gbps), ultra low-latencies (less than 10 usec), with millions of IOPS. In order to meet data processing demands, these devices are now being deployed in the cloud (AWS, Azure), and used inside data processing frameworks. Consequently, an ideal temporary storage system should be able to run efficiently on modern networking and storage hardware while delivering close to bare-metal performance.

Beyond In-Memory Storage: The total amount of temporary data that is generated or consumed by data processing workloads can be large. For instance, between the workloads whose temporary data object size CDFs are shown in Figure 1, the collective total volume of data differs by 100s of GBs (10-100% of the input dataset size, not shown in the figure). Efficiently storing large volumes of data while offering good data access performance is difficult. For instance, storing all the data in DRAM is preferred from a performance standpoint but doing so typically is too costly. Thankfully, over the past years different media types such as NAND Flash, and PCM storage, have emerged to store data at a different cost, performance, and energy price point. Hence, an efficient storage platform for temporary data should integrate multiple storage technologies that offer different performance cost trade-offs, and allow applications to choose between different points in the trade-off space. A comparison of different storage technologies with respect to price and performance is given in Table 1.

Non-Requirements: We observe that in the specific case of temporary data storage, many traditional storage features such as durability and fault-tolerance are not a priority. Durability, for instance, is of low importance due to the short lifetime of temporary data. While fault-tolerance is generally useful for short-lived data, it still is not a high priority for temporary data storage. Today, fault tolerance is often implemented at the level of the compute framework, in a coarse grained manner. For instance, Spark [36] and Ray [25] use lineage tracking to re-compute data in case of data loss by relaunching tasks.

2.2 Limitations of Existing Approaches

We review current state-of-the-art storage systems with regard to the design goals listed in the previous section. We classify the systems discussed into the following three categories.

Key-Value Stores: Memcached [4] and Redis [5] are two of the most popular key-value stores designed to store data in DRAM. Network-optimized KVs like MICA [21], Herd [14], FaRM [12], KVDirect [19] and RAMCloud [26] use RDMA operations to provide high-performance data accesses but cannot easily integrate data storage to different tiers beyond DRAM. Redis has an extension to spill data to Flash, however keys are still stored in DRAM, and hence are limited by the DRAM capacity. Storage-optimized KVs such as Aerospike [30] or BlueCache [34], use NAND Flash for storage. Hence, their performance is bounded by the performance of Flash, and they are not optimized for the next-generation

of NVM storage devices like Optane (see Section 5.1). Other systems, like HiKV [33], have hybrid DRAM-Flash indexes but only target a single node deployment, hence limiting their applications to a wider class of operations such as shuffling. Furthermore, the design of these KV stores is tailored to very small data sets of a few hundred bytes up to a few MB maximum, thus, limiting their operational window to these object sizes.

Distributed Data Stores: Storage systems such as Ceph-over-Accelio [8] and Octopus [23] are high-performance distributed file systems used for fast network and NVM devices. However, due to the focus on providing fault-tolerant, durable storage their performance for small objects is poor (see Section 5). The recently proposed Regions system provides a file abstraction to remote memory [6]. As with other in-memory storage systems, however, Regions is not a cost-effective solution for storing large data sets. Systems like Alluxio [1] and Pocket [17] provide support for multiple storage technology types. But Alluxio does not deliver the performance of high-end hardware, and is targeted towards building local caches. Pocket shares our aim of a dedicated storage system for temporary data and its design has similarities with the NodeKernel architecture. However, the focus of Pocket is on providing efficient and elastic temporary storage on commodity hardware in the cloud whereas NodeKernel is designed for low-latency high-bandwidth network and storage hardware. Moreover, Pocket has only an object based I/O interface which is well suited for data sharing in serverless workloads. In contrast, NodeKernel's unified API provides semantics like "append" and "bag" to support storing a wide range of temporary data in different workloads.

Temporary-data specific operations: A number of works accelerate specific storage operations in data processing workloads. For instance, Riffle [37] is an optimized shuffle server that aims to reduce overheads associated with large fanouts. Sailfish [27] is a framework that introduced I-files which are shuffle optimized data containers. ThemisMR [28] also aims to optimize shuffle and target small rack-scale deployments. In general, the aim of these systems is to optimize disk-based, file-oriented shuffle data management for map-reduce type workloads. It is not clear how their design can support other communication patterns, such as broadcast and multicast, or integrate different storage types to optimize for different access patterns. Parallel databases [7, 22] use RDMA-optimized shuffling operations for database operators. These works, however, are highly database specific and do not extend naturally to other data processing workloads or other forms of temporary data.

3 The NodeKernel Architecture

We present the NodeKernel, a new storage architecture designed to match the diverse and complex demands of temporary data storage in data processing workloads. The NodeKernel tackles this challenge by fusing storage semantics that are otherwise available separately in file systems and key-value stores, such as hierarchical naming, scalability, multiple storage tiers, fast enumeration of datasets, and support for both tiny and large data sizes (Figure 2). The NodeKernel architecture is guided by three design principles:

1. Distill higher-level storage semantics into thin layers on top of a common storage kernel.
2. Separate data management concerns into a lightweight metadata plane and a "dumb" data plane optimized for modern networking and storage hardware.
3. Leverage multiple storage technologies for efficient storage of large datasets.

We discuss each of these design principles in more detail below before describing Crail, a concrete implementation of the NodeKernel architecture, in Section 4.

3.1 Storage Kernel and Node Types

In the NodeKernel architecture, higher-level storage semantics are implemented as thin layers – or more precisely, as specialized data types – on top of a shared storage kernel exporting a hierarchical namespace of opaque data "nodes". Nodes are objects of an abstract type `Node` as shown in the following code snippet.

```
abstract class Node {
protected:
  /*implemented by derived types*/
  abstract bool addChild(Node child);
  abstract bool removeChild(Node child);
  /*implemented by the storage kernel*/
  future<int> read(byte[] buf);
  future<int> append(byte[] buf);
  future<int> update(byte[] buf, int off);
  string getPath();
  int size();
  ...
}
```

The kernel is responsible for allocating storage resources on behalf of nodes, manipulating the hierarchical namespace, and implementing basic data access operations such as `read`, `append` and `update`. Applications interface with the storage kernel to create data nodes at a given location in the hierarchy, attach data of arbitrary size to a node, look up nodes, and fetch the associated dataset from a node. Nodes are identified using path names encoding the location in the storage hierarchy, similar to files and directories in a file system.

Applications do not create raw `Node` objects directly, instead they create objects of derived types offering specialized functionality. These so-called custom types implement higher-level storage semantics by extending `Node` in two ways. First,

custom types provide implementations for the abstract operations `addChild` and `removeChild`. These operations are called by the kernel whenever a new node is inserted or removed to/from the storage hierarchy. Second, custom data types provide specialized data access operations implemented using `read` and `append` available in `Node`.

NodeKernel defines five custom node types, each offering slightly different semantics and operations:

- `File` and `KeyValue`: Both node types provide `read` and `append` interfaces by exposing the corresponding operations in `Node`. The two types, however, provide different semantics during the creation and insertion of new nodes, controlled via the implementation of `addChild` and `removeChild`. For `File` nodes, the first create operation for a given path name succeeds and subsequent create operations on the same path name fail. For `KeyValue` nodes, subsequent create operations on a path name representing an existing node will succeed, replacing the existing node. As we will see in Section 5, `KeyValue` nodes are useful to cache input datasets in NoSQL workloads, permitting concurrent updates of the data, whereas `File` nodes are a better match to cache read-only input data in Spark workloads.

- `Directory` and `Table`: Those node types are containers for `File` and `KeyValue` nodes respectively. `Directory` and `Table` nodes store the name components of all of their children as part of their data (implemented using `append` and `update` operations available in `Node`). For instance, the data segment of a `Directory` with path name "/a/b" storing two files with path names "/a/b/c1" and "/a/b/c2" consists of the two name components "c1" and "c2". Both `Directory` and `Table` nodes offer operations to enumerate the names of all of their children (implemented using `read` available in `Node`). `Tables` can optionally be configured as "non-enumerable" in which case no name components are stored and enumeration returns the empty set. Creating `KeyValue` nodes in a non-enumerable table is typically faster because it eliminates the step of updating the name component (as described in Section 4.1).

- `Bag`: The `Bag` node type is designed to support efficient sequential reading of data spread across many data nodes. A `Bag` behaves like a directory such that it acts as a container for `File` nodes. Applications create and write files in a `Bag` just like they create and write files in a `Directory`. When reading a `Bag`, however, the `Bag` appears to the application like a single file. Using a `Bag`'s `read` operation allows applications to sequentially read through the full set of files in the bag. Generally, the `read` operation of a `Bag` offers better performance than reading each `File` node separately, due to more efficient metadata access at file boundaries. As we will see in Section 5, Spark applications can use `Bags` to store shuffle data, allowing reduce tasks to efficiently fetch data written by map tasks. The `Bag` type in NodeKernel is similar in spirit than bags in Hurricane, a recent work on taming skew in data processing workloads [9].

NodeKernel restricts the way node types can be stacked. For instance, `KeyValue` nodes can only be attached to `Table` nodes, whereas `File` nodes can only be attached to `Bag` and `Directory` nodes. Moreover, directories can be arbitrarily nested, whereas bags and tables implement a flat namespace. The permitted combinations of node types match the specific use cases of temporary data storage. At the same time, preventing arbitrary combinations of node types ensures the architecture is not over-designed and simple to implement. For instance, `Bags` are mainly used to store shuffle data which is organized in flat per-reducer buckets, thus, enabling arbitrarily nested bags seemed unnecessary. At the time, a `read` operation on flat `Bags` is easier to implement than a `read` operation on an arbitrarily nested `Bag`.

Why provide a single unified storage namespace? By splitting functionality in a common storage kernel and a set of custom data types, the NodeKernel achieves two things. First, it permits different types of temporary data requiring different semantics to be managed by a single storage platform. For instance, as we will see later, Spark applications can use Crail for storing both broadcast and shuffle data, as well as for caching RDDs. Second, decoupling core data access from storage semantics allows applications to choose a particular node type based on the semantics they need (key/value vs file) rather than based on the size of the data or the access pattern. As discussed in Section 2.1, temporary data often varies in terms data size and access pattern even within a single workload, making it difficult to store the data efficiently in a storage platform like a filesystem or a key-value store. By contrast, node types in NodeKernel have no size limitation and provide efficient data access for different access patterns as we will see later in Section 5.

3.2 System Architecture

Figure 2 illustrates the NodeKernel system architecture. At the data management level, NodeKernel's architecture resembles the architecture of distributed file systems like HDFS or GFS, consisting of a set of metadata and storage servers deployed across a cluster. Data attached to a "node" in the storage hierarchy appears to clients as a stream, but internally the data is composed of a sequence of blocks. A block refers to a fixed sequence of bytes stored in one of the storage servers. Metadata servers maintain the hierarchical storage namespace as well as block metadata, i.e., a mapping between storage blocks and storage servers. Storage servers allocate a large set of storage blocks at startup and register them with

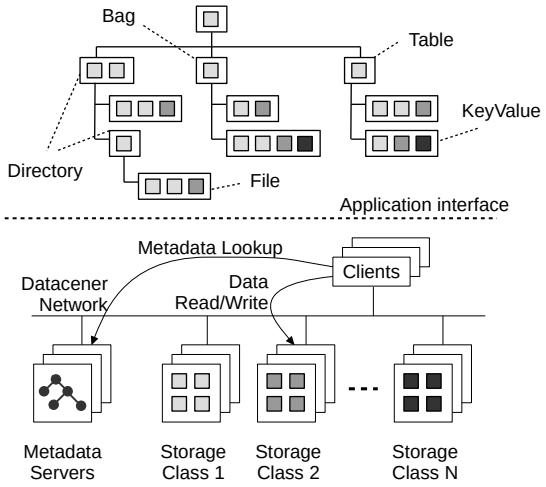


Figure 2: The NodeKernel storage architecture.

one of the metadata servers. Metadata servers maintain a free list with blocks that are not assigned to a particular node, and move blocks from the free list to a per node list during data writes and appends. When accessing data, clients first contact one of the metadata servers and request the metadata for the corresponding block. Based on this information clients then contact the given storage server to read or write the data.

The `Node` abstract data type exports two abstract operations, `addChild` and `removeChild`, to be implemented by the derived types described in Section 3.1. Those operations are executed at the metadata server each time a new node is created or removed. The `Node` further exports functions to manipulate data such as `read`, `append` or `update`. Those functions are implemented as part of the client library and require interactions with both metadata and storage servers. Finally, the basic metadata operations such as `getPath` or `size` are also implemented by the client library returning cached values if possible.

Target deployment: NodeKernel targets temporary data which is short-lived and simple to regenerate. Furthermore, NodeKernel targets small to medium size deployments consisting of few compute racks. Considering the target deployments and the nature of the data to be stored, NodeKernel prioritizes performance over fault tolerance. However, if deemed necessary, additional fault-tolerance mechanism such as replication, erasure-coding, etc., can be added. The lack of these features is not fundamental to the design.

Performance challenges: The main challenge in NodeKernel was to come up with a system architecture that can serve the full spectrum of demands discussed in Section 2.1. In particular, the architecture should be scalable like a key-value store despite offering a convenient hierarchical namespace. Furthermore, the architecture should support both low-latency key-value style access, as well as high-bandwidth file system like data access. In the following we discuss in how we accommodate these requirements in the NodeKernel architecture.

3.2.1 Low-latency metadata operations

Fusing file system and key-value semantics into a single storage kernel primarily requires a fast metadata access. In Section 2.1, we observed that software architectures of distributed file systems fundamentally differ from key-value stores only by an extra metadata operation required to map offsets in file streams to storage resources (e.g., blocks on storage servers). Thus, keeping the overheads of metadata operations low will make NodeKernel's architecture amenable to key-value style operations on small datasets, while also improving the efficiency of large data accesses.

Today, modern low-latency networking hardware enables RPC communication at latencies of a few microseconds [13, 16]. The NodeKernel features a lightweight metadata plane that matches well with low-latency networking hardware. For one, the metadata plane is trimmed down to offer only the most critical functionality consisting of six key RPC operations: `create` to create a new node, `lookup` to retrieve the metadata of a node, `remove` to delete an existing node, `move` to move a node to a different location in the storage hierarchy, `map` to map a logical offset in a node's data stream to a storage resource, and `register` used by storage servers to register storage resources with the metadata server. All these operations have low compute and I/O intensity and can be implemented at the latency of a few microseconds on a high-performance network fabric. We deliberately move data intensive metadata operations like `enumeration` to the data plane to avoid interference (see Section 3.1 and 4).

3.2.2 Metadata partitioning

The scalability of NodeKernel heavily depends on the throughput and scalability of its metadata RPC subsystem. Recent work has shown that RPC systems can be scaled to millions of operations per second on a single server using high-performance networking hardware on the one hand [15], but also using efficient software stacks on commodity hardware [13]. The lightweight RPC interface in NodeKernel is designed for high throughput and can drive up to 10 million metadata operations per second using a single metadata server, as shown later in Section 5. Up to now in all of our deployments we have never had a situation where a single metadata server would reach its limit. In fact, in one of our largest deployments on 128 nodes the metadata throughout reached 4.5 million operations per second, thus, roughly half of what a single metadata server can support.

Nevertheless, for the case where a single metadata server

is not sufficient, NodeKernel permits partitioning the metadata space over multiple servers. Thereby, the top-level root namespace is hash-partitioned among an ordered list of metadata servers. Using metadata partitioning, one can scale the metadata plane horizontally assuming a sufficiently large top level fan out.

One drawback of static of partitioning based on top-level namespaces is that load may be unevenly distributed among the metadata servers depending on the size of the subtree and the activity within the subtree. One alternative approach would be dynamic partitioning at a more fine grained level. For instance, past work has proposed a partitioning scheme for file systems where metadata partitioning is implemented at the directory level, with an option to split large directories on-demand as they grow too big and distribute the splits over multiple metadata servers [29]. Even though such an approach creates a more even load distribution, it comes at a significant performance cost as it requires multiple RPC invocations during path lookup and traversal. Given the performance target of 5-10 μs in NodeKernel we ultimately decided to adopt the simpler partitioning scheme where node paths are always local to a metadata server.

3.2.3 Hardware-accelerated storage

NodeKernel's data plane is designed to work well with modern networking and storage hardware. The goal is to keep the storage interface simple to avoid excessive software overheads and permit as much of the data access functionality to be implemented in hardware. Clients in NodeKernel interact with local and remote storage servers via two interfaces: `read(blockid, offset, length, buffer)` to fetch a certain number of bytes from a block, and `write(blockid, offset, buffer, length)` to write a data stored in a buffer to a block. Later in Section 4 we show that `read` and `write` operations in Crail can almost completely be offloaded to networking and storage hardware for both DRAM and Flash. Also note that the storage interface explicitly supports byte addressable storage hardware by defining the access granularity at the byte level as opposed to block level.

3.2.4 Tiered storage

NodeKernel employs a simple tiered storage design to accommodate large datasets that cannot be stored in DRAM in a cost-effective manner. Storage servers are grouped into different classes (see Figure 2), typically a class per storage technology (DRAM, NVMe SSDs, HDD, etc.). A storage server is a logical entity, i.e., one physical or virtual machine may host multiple storage servers of different types. For instance, a common deployment is to run two storage servers per host, one exporting some amount of local DRAM and one exporting storage space on the local NVMe SSD. In principle, storage classes are user-defined sets of storage servers.

Object type	Methods
Crail	**create**<T extends Node>(*path*, *sc*) → future(*T*) creates a data node of type T
	lookup<T extends Node>(*path*) → future(*T*) lookup of an existing node
	delete(*path*) → future(*boolean*) deletes an existing node at the given *path*
	move(*src*, *dst*) → future(*boolean*) moves node *src* to new location *dst*
File & KeyValue	**read**(*offset*, *buffer*, *length*) → future(*Int*) reads data at given offset
	append(*buffer*, *length*) → future(*Int*) appends application buffer to data
Bag	**read**(buffer, length) → future(*Int*) sequentially reads through all subfiles
Directory & Table	**enumerate**() → iterator(*Node*) enumerates all nodes in a given directory

Table 2: Application programming interface of Crail.

A storage server always belongs to exactly one storage class. In our evaluation we configure two storage classes, one for DRAM servers and one of NVMe SSD servers.

The traditional approach to storage tiering is to migrate data to more cost effective storage classes as the faster storage classes fill up. We found this strategy to be ineffective for temporary data due to the short lifetime of the data. Instead, we opted for a simpler approach where storage classes are filled up according to a user-defined order – typically DRAM first followed by Flash and hard disk – without ever migrating data between the tiers. Specifically during data write operations, metadata servers try to allocate DRAM blocks first, turning to lower priority storage tiers only once no higher priority storage blocks are available across the entire cluster.

4 Crail

Crail is a concrete implementation of the NodeKernel architecture. We implemented Crail in about 10K lines of Java and C++ code. Table 2 shows the application interface of Crail. The top level data type in Crail is `CrailStore`. Applications use `CrailStore` to `create`, `lookup` and `delete` data nodes, or to `move` data nodes to a different location in the storage hierarchy. Nodes are identified using path names similar to file systems. When creating a new node using `create`, applications may choose a preferred storage class for the data to be stored (parameter "sc" in Table 2). We also refer to the storage class preference as storage affinity because it allows users to specify affinity for a particular set of data to a particular set of storage servers or storage media.

Crail implements the full set of node types discussed in Section 3.1. Note that all operations in Crail are non-blocking and asynchronous (returning a `future` object). Crail's asyn-

chronous API matches well with asynchronous software interfaces for modern networking and storage hardware. In fact, almost all of Crail's high-level operations can be mapped directly to a set of non-blocking and asynchronous network and storage operations. Failures of Crail API calls are communicated either via invalid `futures` or exceptions (not shown in Table 2). For instance, an attempt to `lookup` a node that does not exist will result in an invalid `future` (`nullpointer`), while an attempt to `read` data from a node beyond the node's capacity will result in an exception.

Crail can be operated either as a shared storage service or in the form of per-user or per-application deployments. The current implementation of Crail, however, does not provide any tools to virtualize, protect and isolate multiple tenants from each other.

4.1 Metadata plane

Crail metadata servers maintain an in-memory representation of (a) the storage hierarchy, (b) the set of free-blocks, and (c) the assignments of blocks to "nodes". Specifically, each metadata server maintains a per storage class list of free blocks. Storage classes are ordered according to a user-defined preference. As discussed in Section 3.2, if during a write operation the current write position does not yet point to an allocated block, a client requests a fresh new block by calling the metadata `map` RPC operation. The metadata server selects a free block based on the selected storage affinity (or "sc" in Table 2). If there are no free blocks in the selected storage class, the metadata server attempts to allocate a block from the next storage class in the priority list. When selecting a block in a storage class, the metadata server uses round-robin over all storage servers in the given storage class to make sure data is distributed uniformly in the cluster. If no free block is available in any of the storage classes the write operation at the client will fail.

Crail partitions metadata across an array of metadata servers as discussed in Section 3.2.1, meaning, each metadata server is responsible for a partition of the storage hierarchy. Each individual server is implemented as a lightweight RPC service using DaRPC [31], an asynchronous low-latency RPC library based on RDMA send/recv. To achieve high throughput, client connections are partitioned across the different CPU cores. Each core manages a subset of the client connections in-place within a single process context to avoid context-switching overheads. All the memory for RPC buffers is allocated local to the NUMA node associated with the given CPU core that is responsible for the particular connection. Figure 3 illustrates the different aspects of the metadata processing in Crail.

Enumeration: In Section 3.1 we discussed how container nodes (`Table`, `Directory`, `Bag`) maintain a list of the names of all child nodes as part of their data. The rational behind this

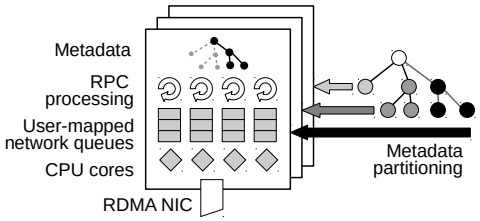


Figure 3: Lightweight metadata plane in Crail.

design is that it allows us to implement container enumeration efficiently in the data plane. We have seen use cases where storing Spark shuffle data in Crail generated close to hundred thousand `File` nodes in a `Bag`. Implementing enumeration at the metadata server would lead to substantial data transfers between clients and metadata servers and in many cases would require multiple rounds of RPC to enumerate all of the nodes.

Crail's file format for container nodes is structured as an array of fixed-size records consisting of the children's name component along with a valid flag. Upon creating a new node, the metadata server assigns a unique offset within the array based on which the client writes the corresponding record. During a delete operation, after the metadata server has removed the node entry, a client clears the valid flag of the corresponding record (by zeroing the valid bit in the record). Note that the node record inside a container's data is only considered supplementary information. The metadata server always serves as the authority for validating the existence of a node. Thus, an enumerate operation running concurrently with a delete operation may lead to a situation where a node's record in the directory file is still valid, but the node's metadata state at the metadata server has already been deleted. In that case, the node is considered deleted and no read or write operations will be permitted.

4.2 Data plane

Crail implements two storage classes, one for DRAM and one for NVMe-based SSDs. An implementation of a storage class consists of a server part exporting a storage resource (see Section 3.2.3), and a client part implementing efficient data access.

RDMA storage class: A storage server in the RDMA storage class exports large regions of RDMA registered memory to the metadata server. Metadata for RDMA based storage blocks contains the necessary RDMA credentials such as address, length and stag and allows clients to directly read or write a storage block using RDMA one-sided read/writes.

NVMe-over-Fabrics storage class: NVMe-over-Fabrics (NVMf) is a recent extension to the NVMe standard that enables access to remote NVMe devices over RDMA-capable

networks. It eliminates unnecessary protocol translations along the I/O path to a remote device, exposing the multiple paired queue design of NVMe directly to clients. As with RDMA, the queue pairs can directly be mapped into the application context to avoid kernel overheads.

A Crail NVMf storage server acts as a control plane for a NVMf controller by connecting to the controller and reporting its credentials like NVMe qualified name, size, etc., to the metadata server. With the credentials provided by the metadata server, the clients can directly connect to the NVMf controller and perform block read and write operations.

4.3 Failure semantics and persistence

Crail does not currently implement mechanisms for fault-tolerance (see Section 3.2) and therefore does not protect against machine or hardware failures. On a crash of a storage server the corresponding data blocks are lost. On a crash of a metadata server the correponding metadata partition is lost. Metadata servers remove inaccessible storage servers from the list of active servers based on keep-alive messages and make sure only active servers are considered during block allocation.

Crail provides optional mechanisms to persist data stored in DRAM, shut down a Crail deployment and to start a Crail deployment from a previously persisted state. Persistence is implemented via operation logging at the metadata servers and the use of memory mapped persistent storage at storage servers.

4.4 Anatomy of data access

Figure 4 illustrates how a Crail client interacts with storage and metadata servers on behalf of an application reading data from a `File` or `KeyValue` node. The application first calls `lookup` to retrieve a node handle, causing Crail to fetch the necessary metadata via RPC from the metadata server. The metadata contains information about the node such as the size of the data and the location of the first block. Following a successful `lookup` call, the application issues a `read` operation to read a certain number of bytes from the node. The requested number of bytes may be less than a block. In that case a single RDMA or NVMf operation will be sufficient to complete the request. If the requested number of bytes spawns multiple blocks, as it is case in the example, Crail immediately issues the data transfer for the first block, while in parallel requesting the metadata for the next block. Under normal circumstances – due to the low latency RDMA-based protocol between the client and the metadata server – the metadata request will complete ahead of the current block transfer and guarantee a continued data transfer without the client ever having to wait for missing metadata information.

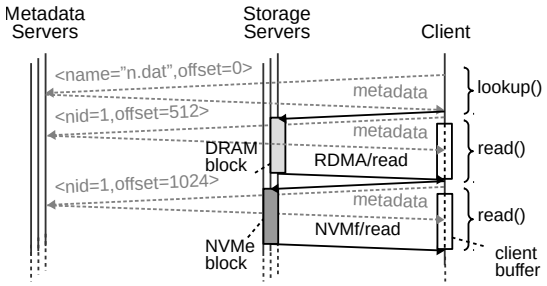


Figure 4: Anatomy of file read/write operations in Crail.

5 Evaluation

In our evaluation we assess if Crail meets the requirements for a temporary storage platform discussed in Section 2.1. Specifically we answer the following questions:

1. Does the unified abstraction of Crail, with its extra indirection layer, perform well for a wide spectrum of data sizes on high-performance devices? (Section 5.1)
2. How simple is it to map higher-level workloads (with their temporary data accesses) to Crail? (Section 5.2)
3. How big are the performance and cost benefits of a mixed-media storage system for data-processing frameworks? (Section 5.3)

Cluster configuration: We use a cluster of eight x64 nodes with two Intel(R) Xeon(R) CPU E5-2690 v1 @ 2.90GHz CPUs, 96GB DDR3 DRAM and a 100 Gbit/s Mellanox ConnectX-5 RoCE RDMA network card. For the client server microbenchmarks, the server is configured with 4 Intel Optane 900P SSDs, except for the IOPS experiments where we only use 2 Optane drives per server. For the larger cluster experiments, all 8 nodes are equipped with 4 Samsung 960 Pro SSDs. The nodes run Ubuntu 16.04.3 LTS (Xenial Xerus) with Linux kernel version 4.10.0-33-generic and Java 8.

5.1 Microbenmarks

Small and medium-size values: We first start by evaluating Crail's performance for storing small to medium size values, a use case typically well served by key-value stores. Consequently, we compare Crail's performance (latency and IOPS) with two state-of-the-art open-source key-value stores, namely, RAMCloud (for DRAM storage) and Aerospike (for NVM Optane). Figure 5 shows the performance for `get` and `put` operations for different data sizes. In Crail, a put operation is implemented by creating a `KeyValue` node using the `create` API call (see Table 2), followed by an `append` operation. The get operation is implemented using a `lookup` call followed by a `read` on the `KeyValue` node – similar to

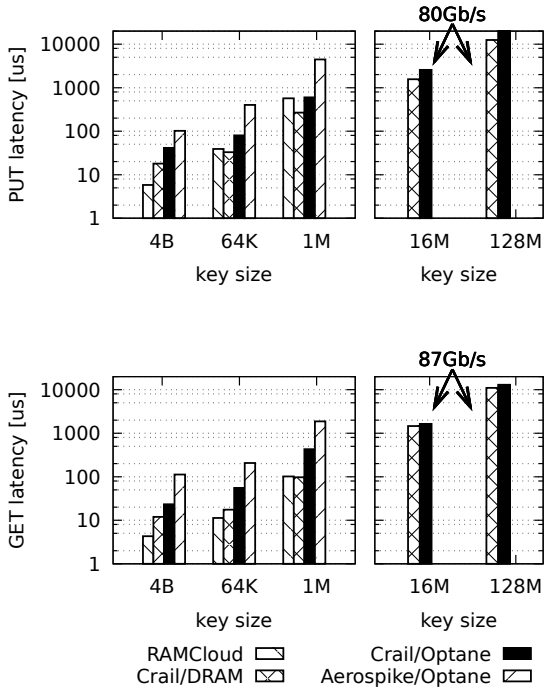


Figure 5: Put/Get latencies in Crail, RAMCloud and Aerospike for datasets of different sizes.

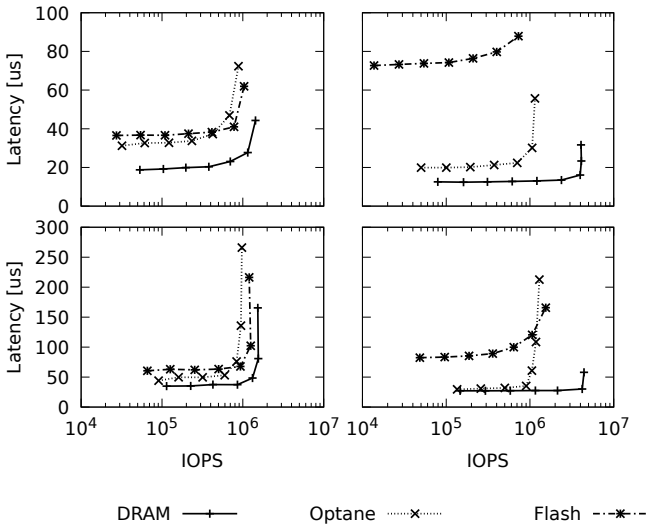


Figure 6: Media-specific loaded latency profile. Top left: Put, queue depth 1. Top right: Get, queue depth 1. Bottom left: Put, queue depth 4. Bottom right: Get, queue depth 4.

the scenario shown in Figure 4. For small datasets (4 bytes), Crail performs slightly worse than RAMCloud for DRAM storage (12 μs vs 6 μs), but outperforms Aerospike on Optane NVM by a margin of $2-4\times$ (23-40 μs for Crail vs. 100 μs for Aerospike). The difference between Aerospike and Crail comes from differences in their I/O execution. Aerospike uses synchronous I/O and multiple I/O threads, which cause contention and spend a significant amount of execution time in synchronization functions [18]. Crail uses asynchronous I/O and executes I/O requests in one context, avoiding context switching and synchronization completely. The latency difference between Crail and RAMCloud is acceptable considering that RAMCloud is a system optimized for small values. For medium sized values (64KB-1MB), Crail outperforms RAMCloud and Aerospike by a margin of $2-6.8\times$. For instance, a 1MB Put on Crail takes around 590 μs, versus 4 *ms* in Aerospike. These performance gains come from the efficient use of RDMA one-sided operations (for both DRAM and NVM), which eliminates data copies at both client and server ends and generally reduces the code path that is executed during put/get operations. While Crail natively supports arbitrary size datasets – by distributing the blocks over multiple storage servers – storing such large values in systems like RAMCloud or Aerospike is either difficult or prohibited. For instance, Aerospike limits individual key/value pairs to 1MB. RAMCloud does not have a strict size limitation but failed to store values larger than 4MB.

For sake of completeness, Figure 5 also shows put/get latencies for extra large values of 16MB and 128MB. As we can see, it takes around to 12 *ms* to store a 128MB value in Crail's DRAM tier, and 20 *ms* to store the same dataset in Crail's Optane tier. Storing such large values in Crail is entirely a matter of throughput. Consequently, the remote DRAM latency is limited by the 100 Gb/s network bandwidth, while the NVM latency is the determined by the bandwidth of the storage device. The aggregated bandwidth of the 4 Optane drives is around 10-12 GB/s. Hence, the resulting data access bandwidth for large datasets stored in Crail's NVM tier is 80-87 Gb/s.

IOPS scaling: So far we have discussed unloaded latencies. Figure 6 shows the latency profile for 256 bytes values for a loaded Crail system for different media types. In this setup, we increase the number of clients from 1 to 64. The clients are running on 16 physical machines, issuing put/get operations in a tight loop. We use only one storage server and one metadata server in this setup, configured either to serve DRAM, Optane NVM or Flash. The top row in Figure 6 show the case for a queue depth of 1, meaning, each client always has only one operation in flight. As shown in the figure, Crail delivers stable latencies up to a reasonably high throughput. For DRAM, the get latencies (top right in Figure 6) stay at 12-15 μs up to 4M IOPS, at which point the metadata server became the bottleneck. We ran the same experiment with multiple metadata servers and verified that the system throughput was scaling linearly (shown later in Figure 7 on top). For the Optane NVM configuration, latencies stay at 20 μs up until almost 1M IOPS, which is very close to the device limit. The Flash latencies are higher but the Samsung drives also have a higher throughput limit. In fact, 64 clients with queue depth 1 cannot saturate the Samsung devices. In order to generate

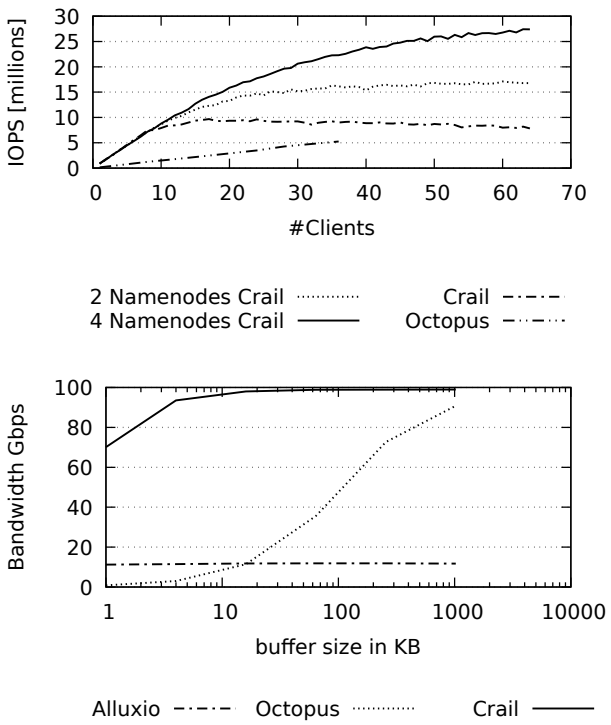


Figure 7: Top: metadata performance in Crail compared to Octopus (we could not run Octopus reliably after 36 clients). Bottom: sequential read performance in Crail and Octopus for large datasets and different buffer sizes.

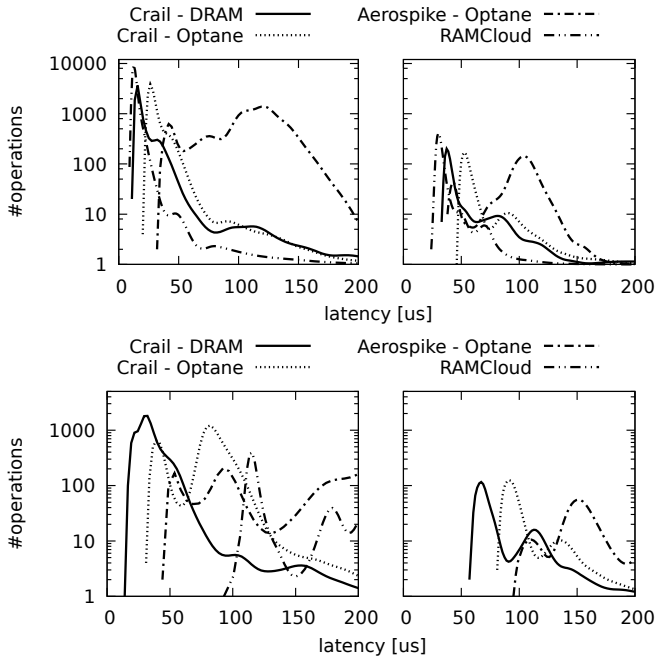


Figure 8: YCSB benchmark performance. Top: small value (1K per KV pair) read (left) and update (right) latencies. Bottom: large value (100K per KV pair) read and update latencies.

a higher load, we measured throughput and latencies for the case where each client always has four operations in flight (queue depth 4, bottom row in Figure 6). As shown, queue depth 4 generally achieves a higher throughput up to a point where the hardware limit is reached, the device queues are overloaded (e.g., for NVM Optane) and latencies sky rock. For instance, at the point before the exponential increase in the latencies, Crail delivers get latencies (Figure 6 bottom right) of 30.1 μs at 4.2M IOPS (DRAM), 60.7 μs for 1.1M IOPS (Optane), and 99.86 μs for 640.3K IOPS (Flash). The situation for put is similar, though generally with lower performance.

Metadata performance: In Figure 7 (top), we benchmark the performance of a simple `lookup` metadata operation which is used to retrieve node metadata in Crail, and compare it with the performance of a similar metadata operation `getattr` in Octopus [23] (an RDMA-optimized NVM file system running in DRAM). There are two main observations here. First, for the single namenode case, Crail outperforms Octopus by $1.7-5.9\times$. A single namenode in Crail peaks around 9.3M lookups/sec. Second, Crail can very efficiently scale the single namenode performance to multi-namenode setups. The system can deliver up to 16.7M and 27.4M lookups/sec for 2 and 4 namenode configurations.

Accessing large datasets: Figure 7 (bottom) shows the bandwidth (y-axis) measured when reading large datasets of `File` nodes in Crail, in comparison to file read operations in Octopus and Alluxio. The x-axis in Figure 7 refers to the size of the application buffer the client is using during read operations. Crail with its efficient data and metadata plane, and overlapping of lookup RPCs and data fetching quickly reaches the network bandwidth limit even for relatively small buffer sizes (just over 1kB). Alluxio performance is bottlenecked by the CPU due to data copies and inefficiencies in the network stack implementation. Octopus performs better than Alluxio, and gradually for large buffers (close to 1MB) reaches the line speed of 98 Gb/s. Note that Crail's peak bandwidth (98 Gb/s) in Figure 7 is better than the peak bandwidth (87 Gb/s) in Figure 5 because for the KV experiments each `KeyValue` node is opened, read, and closed whereas for the file experiments those accesses are amortized.

Summary: In this section we have shown that Crail can store a large spectrum of values effectively while offering comparable or superior performance than the other state-of-the-art systems that are optimized for a particular data range.

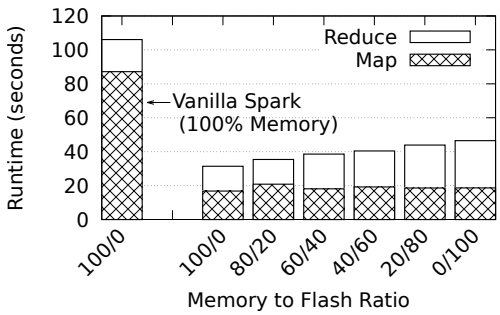


Figure 9: Spark Terasort with mixed DRAM-NVM.

5.2 Systems-level Benchmarks

5.2.1 NoSQL workloads

The Yahoo! Cloud Serving Benchmark (YCSB) is an open standard designed to compare the performance of NoSQL databases [10]. It comes with five workloads that stress different properties, e.g. workload A is update heavy whereas workload C is read heavy. We choose workload B to compare Crail to RAMCloud and Aerospike. Workload B has 95% read and 5% update operations and the records are selected with a Zipfian distribution. All systems run in a single namenode/datanode configuration. The purpose of this experiment is to evaluate the latency profile of Crail for a realistic workload beyond microbenchmarks presented in the last section.

Figure 8 (top part) shows the read and update latency distributions for workload B using a single client for the default setup of 10 fields of 100 bytes per record (1K per KV pair). The left part of the figure shows the read performance, and the right part shows the update performance. As shown for this default setup, the largest number of read operations observe a latency of 14 μs (95% and 99% percentile are at 37 and 84 μs) and 26 μs (95% and 99% percentile are at 47 and 81 μs) for DRAM and Optane respectively. Crail on Optane has an average latency of 38 μs and thus is only 15 μs slower than Crail on DRAM. On the other hand, Aerospike with Optane delivers an average latency of 108.7 μs, which is 2.84$\times$ worse than the average Crail latency (38.03 μs). Comparing Crail's DRAM performance to RAMCloud shows that RAMCloud is slightly faster than Crail. However, as we move to larger values of 10 fields of 10KB each (100KB per KV pair) in Figure 8 (bottom) Crail is almost $2.6-4.8\times$ better than Aerospike and RAMCloud, respectively.

Summary: Our experiments using the YCSB benchmark have demonstrated that (a) Crail can successfully translate the raw DRAM/NVMe performance advantages into workload level gains, and (b) Crail effectively deals with both small and large datasets while RAMCloud and Aerospike perform their best in a specific operating range.

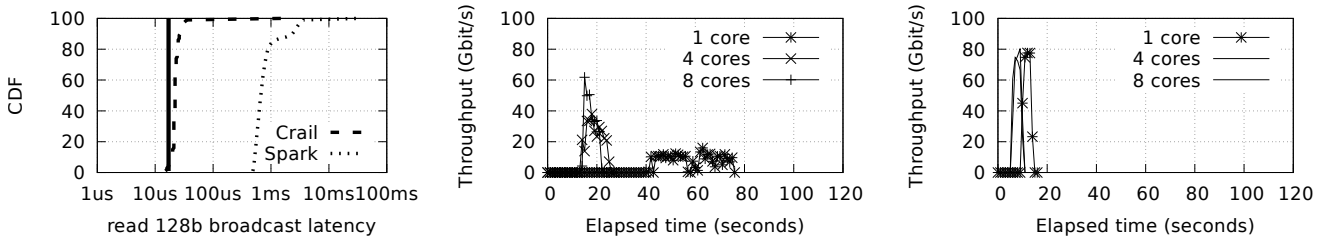
5.2.2 Spark Integration

We present the evaluation of Crail with Spark, one of the most popular data processing engines. Spark executes workloads as a series of map-reduce steps while sharing performance critical data in each step. There are multiple points in the Spark data processing pipeline where temporary datasets are generated. In this section, we show performance measurements specifically for two: shuffle and broadcast. Both subsystems can easily be implemented as plugin modules for Spark.

Broadcast: We implemented broadcast using Crail by storing broadcast data as `KeyValue` nodes in a non-enumerable `Table`. A broadcast writer creates a new `KeyValue` node, appends the broadcast data to the node, and passes the node "name" to the readers. Readers, which are distributed over multiple machines inside Spark executors, do a lookup on the "name" and read the data from Crail. Figure 10a shows the result. The x-axis shows the latency as observed by different broadcast readers in the Spark job, while the y-axis shows the percentage of readers. The solid vertical line represent the baseline latency of 12 μs, which we demonstrated in our microbenchmarks. As shown, most of the Crail broadcast readers observe latency very close to the minimum possible. A few observe a latency lower than 12 μs because some of these readers are co-located on the same physical machine where the values are stored. For those nodes, even though they still read the data using the local network interface, there is no actual network transfer happening, hence, their read performance is not limited by the network. In summary, Crail broadcast performance is 1-2 orders of magnitude better than the default Spark implementation.

Shuffle: In Spark, a shuffle writer continuously generates shuffle data during the map phase as it processes the input dataset and classifies data into different buckets that are later read by reducers. Due to the large fan-in and fan-out access pattern, we implemented shuffle using Crail `Bag` nodes. There is one `Bag` node per reducer and each shuffle writer appends data to an array of privately owned `File` nodes, one `File` node per writer per bag. After the map phase, each reducer reads its associated bag using the optimized read interface available in the `Bag` node type (see Section 3.1). We generated a large amount of data (512 GB) and triggered the shuffle operation using the GroupBy benchmark available in the Spark source code. Figures 10b and 10c show the performance (runtime on the x-axis) and the observed network throughput (y-axis) for various configurations. The values 1, 4, or 8 represent the number of cores given to each Spark executor. A quick comparison of the two figures shows that the Crail-accelerated Spark observes higher network throughput (for a corresponding core count) and, thus, as a result better runtimes (1 core 5x, 4 cores 2.5x and 8 cores 2x).

Summary: In this section we have demonstrated that Crail is able to successfully accelerate temporary data access in Spark for small values (e.g., broadcast) as well as large values



(a) Broadcast performance

(b) Vanilla Spark GroupBy performance

(c) Crail accelerated GroupBy performance

Figure 10: Spark broadcast and GroupBy performance, using Vanilla Spark vs. Crail temporary storage.

(e.g., shuffle) by taking advantage of the different node types (`KeyValue` and `Bag`) in Crail.

5.3 Efficiency of hybrid DRAM/NVM setup

As the last part of our evaluation, we quantify how Crail's tiered data plane helps with regard to performance and cost objectives. We consider the Terasort workload, which is one of the most I/O intensive applications on Spark. We implement Terasort as an external range-partition sort algorithm in two stages. The first stage maps the incoming key-value pairs (10 bytes keys and 90 bytes value) into external buckets. These buckets are then shuffled and sorted by individual reduce tasks. For this evaluation we use the accelerated shuffle and broadcast plugins that we previously developed.

In Figure 9, we explore the performance/cost trade-off of using NVM instead of DRAM to store shuffle data in a 200 GB Spark sorting workload. For this we configure Crail with different storage limits for the DRAM and the Flash storage tiers. The x-axis indicates what fraction of the total shuffle data is stored in DRAM versus Flash. Note that in this experiment we are using the Samsung Flash-based SSDs rather than the Optane devices. A configuration of 10/90 means that 10% of the data is held in DRAM, while 90% is held in Flash. The figure also shows the performance of vanilla Spark (first bar of the figure) that runs on its default shuffle engine completely in DRAM (using tmpfs as a storage backend). There are two key observations here. First, in comparison to vanilla Spark, the use of Crail for the shuffle backend already reduces the runtime by a factor of 3.4. This performance gain can be attributed to the efficient use of high-performance networking and storage hardware in Crail. For instance, during the reduce phase we measured an all-to-all network throughput of 70 Gb/s/machine. Second, as we decrease the fraction of DRAM in Crail in favor of Flash, Spark graciously and automatically spills shuffle data into the Flash tier. In the extreme configuration, where all shuffle data is stored in Flash, the performance degrades to 46.49 second (48% increase), while to total cost for storage is reduced by 8× from 1,000$ to 126$ (to store 200 GB of data based on the numbers in Table 1). The gradual spilling of data from DRAM to Flash happens transparently. Even in the all-Flash configuration, the performance of the Crail-integrated Spark Terasort is half of the completely-in-DRAM vanilla Spark performance. These results validate the design choices we made in Crail that permit trading performance for storage cost.

Summary: In this section, we demonstrated that the use of Crail in Spark (i) leads to better performance due to its efficient I/O path; (ii) reduces the cost of storage, and increases the performance due the hybrid DRAM-NVMe architecture.

6 Conclusion

Storing and accessing temporary data efficiently in data processing workloads is critical for performance, yet challenging due to complex storage demands that fall between the lines of existing storage systems like file systems or key-value stores. We presented NodeKernel, a novel storage architecture offering a new point in the storage design space by combining hierarchical naming with scalability and excellent performance for a wide range of data sizes and access patterns that are typical for temporary data. The NodeKernel architecture is driven by opportunities of modern networking and storage hardware that enabled us to reduce overheads that made such a design impractical in the past. We showed that storing temporary data in Crail, our concrete implementation of the NodeKernel architecture leveraging RDMA networking and NVMe storage, can improve NoSQL workloads by up to 4.8× and Spark application performance by up to 3.4×. Crail's use of NVMe Flash further reduces storage cost by up to 8× compared to storage systems that only use DRAM.

Acknowledgments

We thank our shepherd, Michael Swift, and the anonymous Usenix ATC reviewers for their helpful feedback.

References

[1] Alluxio: Open source memory speed virtual distributed storage. https://www.alluxio.org/.

[2] Apache Crail (Incubating). http://crail.apache.org/.

[3] Apache Crail (Incubating) Source Code. https://github.com/apache/incubator-crail.

[4] Memcached. http://memcached.org.

[5] Redis. https://redis.io/.

[6] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, 2018.

[7] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, January 2017.

[8] Matt Benjamin. Xiomessenger: Ceph transport abstraction based on accelio, a high-performance message-passing framework by mellanox, at https://www.cohortfs.com/ceph-over-accelio.

[9] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 20:1–20:15, New York, NY, USA, 2018. ACM.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010.

[11] Aaron Davidson and Andrew Or. Optimizing shuffle performance in spark. In *https://pdfs.semanticscholar.org/d746/505bad055c357fa50d394d15eb380a3f1ad3.pdf*, 2013.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014.

[13] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, 2014.

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016.

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, GA, 2016.

[17] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, Berkeley, CA, USA, 2018.

[18] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, 2019. USENIX Association.

[19] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017.

[20] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011.

[21] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014.

[22] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 48–63, New York, NY, USA, 2017.

[23] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 773–785, Berkeley, CA, USA, 2017.

[24] Chenxin Ma, Virginia Smith, Martin Jaggi, Michael I. Jordan, Peter Richtárik, and Martin Takáč. Adding vs. averaging in distributed primal-dual optimization. In *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ICML'15, pages 1973–1982, 2015.

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, 2018.

[26] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[27] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012.

[28] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012.

[29] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press.

[30] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[31] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 15:1–15:13, New York, NY, USA, 2014.

[32] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 615–630, Boston, MA, 2018.

[33] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017.

[34] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.*, 10(4):301–312, November 2016.

[35] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. In *IEEE Bulletin of the Technical Committee on Data Engineering*, pages 39–45, 2018.

[36] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012.

[37] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018.