Crail: A High-Performance I/O Architecture for Distributed Data Processing

Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle,
Radu Stoica, Bernard Metzler, Nikolas Ioannou, Ioannis Koltsidas
IBM Research Zurich

Abstract

Effectively leveraging fast networking and storage hardware for distributed data processing remains challenging. Often the hardware integration takes place too low in the stack, and as a result performance advantages are overshadowed by higher layer software overheads. Moreover, new opportunities for fundamental architectural changes within the data processing layer are not being explored. Crail is a user-level I/O architecture for the Apache data processing ecosystem, designed from the ground up for high-performance networking and storage hardware. With Crail, hardware performance advantages become visible at the application level and translate into workload runtime improvements. In this paper, we discuss the basic concepts of Crail and show how Crail impacts workloads in Spark, like sorting or SQL.

1 Introduction

In recent years, we have seen major performance improvements in both networking and storage hardware. Network interconnects and fabrics have evolved from 1-10Gbps bandwidth ten years ago to 100Gbps today. Storage hardware has undergone similar improvements while moving from spinning disks to non-volatile memories such as flash or Phase Change Memory (PCM). These hardware changes have also brought up the need for new I/O interfaces. Traditional operating system abstractions like sockets for networking or the block device for storage were shown to be insufficient to make the rich semantics and high performance of the hardware available to applications. Instead, new interfaces such as RDMA[1] or NVMe have emerged. These interfaces enable user-level hardware access and asynchronous I/O and allow applications to take full advantage of the high-performance hardware. Table 1 compares the performance of some of the new I/O technologies with the performance of legacy software/hardware interfaces. As can be observed, the performance improvements are in the range of 100-1000x.

Meanwhile, systems for large scale data processing are confronted with increasing demands for better performance while applications are becoming more complex and data sizes keep growing. Here, the new I/O technologies present opportunities for data processing systems to further reduce the response times of analytics queries on large data sets. In particular, performance critical I/O operations, such as data shuffling or sharing of data between jobs or tasks, should benefit from faster I/O. Today, examples of applications that are capable

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]Commodity RDMA network interconnects have been around since early 2000 and have their roots in user-level I/O from the 90s. Only recently they have gained acceptance in the broader data center community.

	RTT (us)	Throughput (MBit/s)
1GbE / sockets	82	942
100GbE / verbs	2.4	92,560

(a)

	Latency (us)		Throughput (MB/s)	
	read	write	read	write
Disk / Block device	5,978	5,442	136.8	135.3
NVMe Flash / SPDK	64.2	9.18	2,657	1,078

(b)

Table 1: I/O performance legacy vs. state-of-the-art for (a) networking (b) storage.

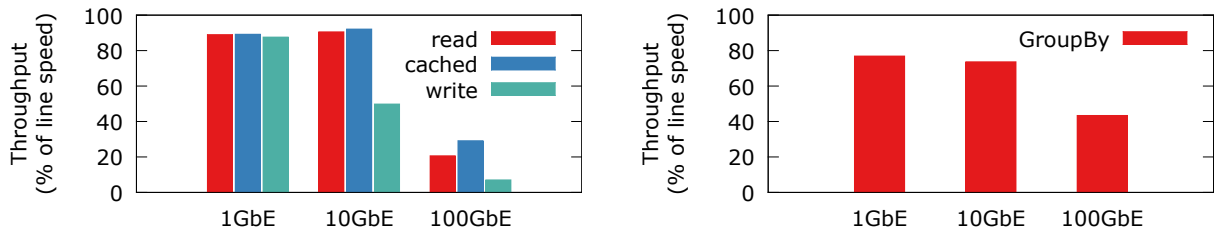

Figure 1: HDFS remote read performance (left) and Spark groupBy() performance (right). HDFS deployment is on NVMe SSDs, Spark deployment is completely in memory.

of leveraging fast I/O hardware can be found in supercomputing, but also more recently in several academic efforts [11, 3, 12]. These applications, however, are written from scratch to serve one particular workload. Unfortunately, properly leveraging high-performance networking and storage hardware in general purpose data processing frameworks, such as Spark, remains challenging for a number of reasons:

Deep Layering: Often hardware integration takes place too low in the stack, and as a result performance advantages are overshadowed by higher layer software overheads. This problem gets amplified with today's prominent frameworks such as Spark, HDFS, Flink, etc., due to the heavy layering these systems employ. Besides involving the host operating system for each I/O operation, data also needs to cross the JVM boundaries and the specific I/O subsystems of the frameworks. The long code path from the application down to the hardware leads to unnecessary data copies, context switches, cache pollution, etc., and prevents applications from taking full advantage of the hardware capabilities. For instance, writing out an object to the network during a Spark shuffle operation requires no less than 5 memory copies. The effect of those overheads at the application level is illustrated in Figure 1, where we show examples of the I/O performance in HDFS and Spark for different network technologies. As can be observed, while both systems make good use of the 1 and 10Gbps network, the network is underutilized in the 100Gbps configuration as the software becomes the bottleneck. A more detailed analysis of software related I/O bottlenecks in data processing workloads can be found in [26].

Data locality: The improved performance of modern networking and storage hardware opens the door to rethinking the interplay of I/O and compute in a distributed data processing system. For instance, low latency remote data access enables schedulers to relax the requirements on data locality and in turn make better use of compute resources. At the extreme, storage resources can be completely disaggregated which is more cost effective and simplifies maintenance. The challenge is in the first place to spot what opportunities the hardware provides, and then to exploit the opportunities in a way that is non-intrusive and still compatible with the general architecture of a given data processing framework.

Legacy interfaces: With modern networking and storage hardware, I/O operations are becoming more complex. Not only are there more options to store data (disk, flash, memory, disaggregated storage, etc.) but also

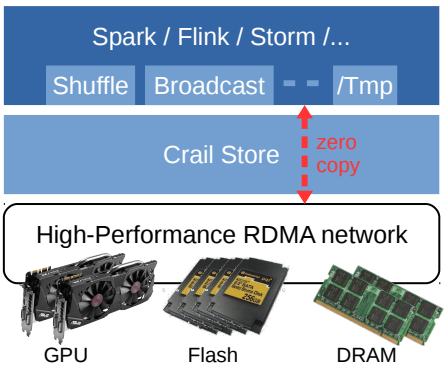

Figure 2: Crail I/O architecture overview.

it is getting increasingly important to use the storage resources efficiently. For instance, some newer technologies such as PCM permit data to be accessed at a byte granularity. Mediating storage access through a block device interface is a bad fit in such a case. Moreover, as the traditional compute layer is extended with GPUs and FPGAs, new interfaces to integrate accelerator memories into the distributed storage hierarchy are needed. Integrating such a diverse set of hardware technologies in the most efficient way, while still keeping the data processing system general enough, is challenging.

2 Crail I/O

In this paper, we present Crail, an open source user-level I/O architecture for distributed data processing. Crail is designed from the ground up for high-performance storage and networking hardware and provides a comprehensive solution to the above challenges. The design of Crail was driven by five main goals:

1. Bare-metal performance: enable I/O operations in distributed systems to take full advantage of modern networking and storage hardware.
2. Practical: integrate with popular data processing systems (e.g., Spark, Flink, etc.) in a form that is non-intrusive and easy to consume.
3. Extensible: allow future hardware technologies to be integrated at a later point in time in a modular fashion.
4. Storage tiering: provide efficient storage tiering that builds upon the new data localities of high-performance clusters.
5. Disaggregation: provide explicit support for resource disaggregation.

Crail is specifically targeting I/O operations on temporary data that require the highest possible performance. Examples of such operations in data processing systems are data shuffling, broadcasting of data within jobs, or general data sharing across jobs. Crail is not designed to provide the high availability and fault tolerance required by durable storage systems.

2.1 Overview

The backbone of the Crail I/O architecture is the Crail store, a high-performance multi-tiered data store for temporary data in analytics workloads. Data processing frameworks and applications may directly interact with

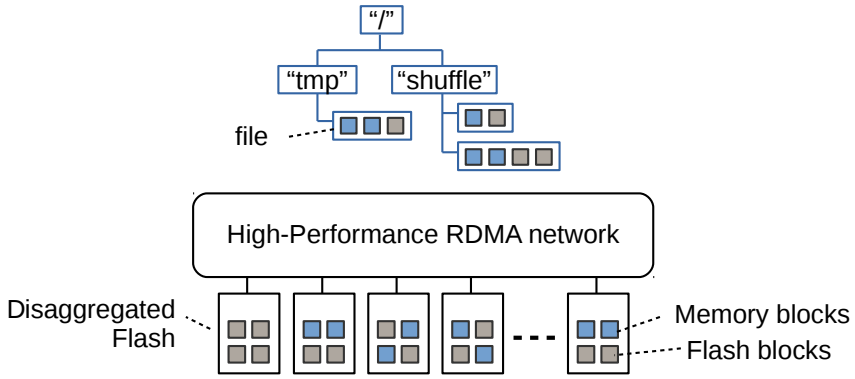

Figure 3: Crail file system namespace

Crail store for fast storage of working-set data, but more commonly the interaction takes place through one of the Crail modules (see Figure 1).

Crail modules implement high-level I/O operations, typically tailored to a particular data processing framework. For instance, the Crail Spark module implements various I/O intensive Spark operations such as shuffle, broadcast, etc. The Crail HDFS module provides a standard HDFS interface that can be used by different frameworks or applications to store performance-critical temporary data. Both modules can be used transparently with no need to recompile either the application or the data processing framework.

Crail modules are thin layers on top of Crail store. Consequently, the modules inherit the full benefits of Crail in terms of user-level I/O, performance and storage tiering. Due to the modules being so lightweight, implementing new modules for a particular data processing framework or a specific I/O operation requires only a moderate amount of work.

2.2 Crail Store

For the rest of the paper, if the context permits and to simplify the reading, we refer to Crail store simply as Crail. Crail implements a file system namespace across a cluster of RDMA interconnected storage resources such as DRAM or flash. Storage resources may be co-located with the compute nodes of the cluster, or disagreggated inside the data center, or a mix of both. Files in the Crail namespace consist of arrays of blocks distributed across storage resources in the cluster as shown in Figure 3. Crail groups storage resources into different tiers. A storage tier is defined by the storage media used to store the data, and the network protocol used to access the data over the network. In the current implementation, Crail offers three storage tiers:

1. **DRAM**: The DRAM tier uses one-sided RDMA operations (read/write) to access remote DRAM. With one-sided operations, the storage nodes remain completely passive and, thus, are not consuming any CPU cycles for I/O. Using one-sided operations is also effective for reading or writing subranges of storage blocks, as only the relevant data is shipped over the network rather than shipping the entire block. The DRAM tier provides the lowest latency and highest throughput among all the storage tiers. It is implemented using DiSNI [2], an open-source user-level networking and storage stack for the JVM [25].

2. **Shared Volume:** The shared volume tier supports flash access through block storage protocols (e.g., SCSI RDMA protocol, iSCSI, etc.). This tier is best used to integrate disaggregated flash.

3. **NVMeF:** The NVMeF storage tier exports NVMe flash accessed over RDMA fabrics. This storage tier provides ultra-low latency access to both direct attached and disaggregated flash.

[2] http://github.com/zrlio/disni

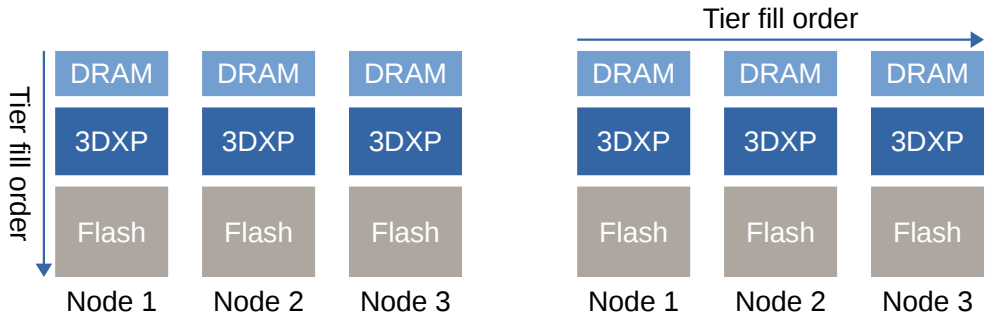

Figure 4: Vertical tiering (left) vs horizontal tiering (right).

Crail provides an extendable storage interface where new storage tiers can be plugged in transparently without requiring changes to the file system core. Apart from supporting traditional storage tiers, we are working on integrating accelerator memory into the Crail storage hierarchy. This will allow, for instance, GPU memory to become accessible via Crail from anywhere in the cluster.

2.3 Crail Store Interface

Crail exposes a hierarchical file system API comprising functions to create, remove, read and write files and directories. Besides those standard operations, much of the Crail API is designed to empower the higher level Crail modules with the right semantics to best leverage the networking and storage hardware for their data processing operations (e.g., shuffle, broadcast, etc.). We discuss a few specific examples below:

Asynchronous I/O: All file read/write APIs in Crail are fully asynchronous which facilitates interleaving of computation and I/O during data processing. Also, the asynchronous APIs perfectly match with the asynchronous hardware interfaces in RDMA or NVMeF, therefore allowing for a very efficient implementation.

Storage tiering: Crail provides fine-grained control over how storage blocks are allocated during file writing. By specifying location affinities, applications can indicate a preference as to which physical storage node or set of nodes should be used to hold the data of a file. Similarly, by specifying storage affinities, applications can indicate which storage tier should preferably be used for a particular file. In the absence of a dedicated storage affinity request, Crail uses horizontal tiering where higher performing storage resources are filled up across the cluster prior to lower performing tiers being used. Horizontal tiering stands in contrast to traditional vertical tiering where local resources are always preferred over remote ones. With low latency RDMA networks, remote data access has become very effective, and as a result, horizontal tiering often leads to a better usage of the storage resources. Figure 4 illustrates both approaches in a configuration with DRAM, 3DXP and flash.

File types: Crail supports three different file types: regular data files, directories and multifiles. Regular data files are append-and-overwrite with only a single-writer permitted per file at a given time. Append-and-overwrite means that – aside from appending data to the file – overwriting existing content of a file is also permitted. Directories in Crail are just regular files containing fixed length directory records. The advantage is that directory enumeration becomes just a standard file read operation which makes enumeration fast and scalable with regard to the number of directory entries. Multifiles are files that can be written concurrently. Internally, a multifile very much resembles a flat directory. Multiple concurrent substreams on a multifile are backed with separate files inside the directory. The *multistream* API allows applications to read the collective data as if it were a single file. Later in the paper, we show how multistreams are used during mapreduce shuffle operations.

I/O buffers: To allow zero-copy data placement from the network interface to the application buffer (as implemented in RDMA), the corresponding memory has to be pinned and registered with the network interface. Crail exports functions to allocate dedicated I/O buffers from a reusable pool – memory that is pinned and

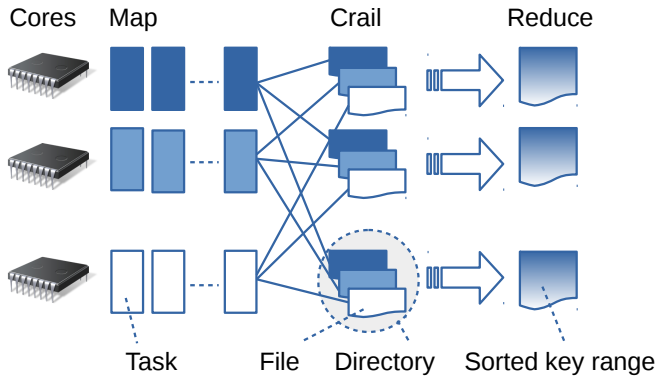

Figure 5: The Crail shuffle module.

registered with the hardware to enable bypassing of both the OS and the JVM during network transfers.

2.4 Crail Modules

Crail offers two independent modules that interface directly with the data processing layer: (a) a Spark module that contains Spark specific implementations for shuffle and broadcast operations, and (b) an HDFS adaptor that allows many of the prominent data processing frameworks to use Crail seamlessly for storing performance critical temporary data. We discuss the different components briefly as follows.

Shuffle: The shuffle engine maps key ranges to directories in Crail. Each map task, while partitioning the data, appends key/value pairs to individual files in the corresponding directories. Tasks running on the same core within the cluster append to the same files, which reduces storage fragmentation (see Figure 5). Individual shuffle files are served using horizontal tiering. In most cases that means the files fill up the memory tier as long as there is some DRAM available in the cluster, after which they extend to the flash tier. The shuffle engine ensures such a policy using the Crail location affinity API. Note that the shuffle engine is completely zero-copy, as it transfers data directly from the I/O memory of the mappers to the I/O memory of the reducers.

Broadcast: The Crail-based broadcast plugin for Spark stores broadcast variables in Crail files. In contrast to the shuffle engine, broadcast is implemented without location affinity, which makes sure the underlying blocks of the Crail files are distributed across the cluster, leading to a better load balancing when reading broadcast variables.

HDFS adaptor: The HDFS adaptor for Crail implements a straightforward mapping between the synchronous HDFS function calls and the corresponding asynchronous operations in Crail. Functionality in Crail that is not exposed in the HDFS API, such as storage affinities, can be configured through admin tools on a per-directory basis.

3 Evaluation

A detailed evaluation of the individual components in the Crail I/O architecture (Crail store, individual modules) is beyond the scope of this paper. Instead, we demonstrate the benefits of Crail on two specific workloads, sorting and SQL, both evaluated using Spark. We used a cluster with the following specification: 2xPower8 10-core @2.9Ghz, 512 GB DDR4 DRAM, 4x Huawei ES3600P V3 1.2TB NVMe SSD, with 100Gbps Ethernet Mellanox ConnectX-4 EN (RoCE). Nodes in the cluster run Ubuntu 16.04 with Linux kernel version 4.4.0-31.

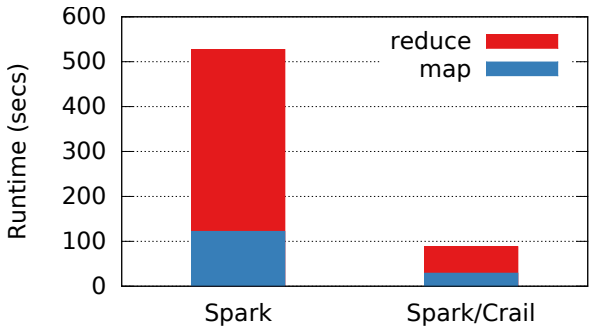

Figure 6: Performance comparison for Spark sorting, vanilla vs. Crail

3.1 Sorting

Sorting large data sets efficiently on a cluster is particularly interesting from a network perspective, as most of the input data will have to cross the network at least once. Here, we explore the sorting performance of Spark/Crail on a 128-node slice of the cluster. We compare vanilla Spark with Spark/Crail on a 12.8 TB dataset (Spark version 2.0.2). In both cases, plain HDFS on NVMe SSDs is used for input and output, and the shuffle engine (including networking, serialization and sorting) is exchanged in the Spark/Crail configuration.

Anatomy of Spark Sorting: A Spark sorting job consists of two phases. The first phase is a mapping or classification phase - where individual workers read their part of the key-value (KV) input data and classify the KV pairs based on their keys. This phase involves only very little networking as most of the workers run locally on the nodes that host the HDFS data blocks. During the second so called reduce phase, each worker collects all KV pairs for a particular key range from all the workers, de-serializes the data and sorts the resulting objects. This pipeline runs on all cores in multiple waves of tasks on all the compute nodes in the cluster.

The main difference between the Crail shuffler and the Spark built-in shuffler lies in the way data from the network is processed in a reduce task. The Spark shuffler is based on TCP sockets; thus, many CPU instructions are necessary to bring the data from the networking interface to the buffer inside Spark. In contrast, the Crail shuffler shares shuffle data through the Crail file system, and therefore data is transferred directly via DMA from the network interface to the Spark shuffle buffer within the JVM.

Performance: Figure 6 shows the overall performance of Spark/Crail vs vanilla Spark. With a cluster size of 128 nodes, each node sorts 100 GB of data - if the data distribution is uniform. As can be seen, using the Crail shuffler, the total job runtime is reduced by a factor of 6. Most of the gains come from the reduce side, which is where the networking takes place. The Crail shuffler – due to its zero-copy data ingestion – avoids data copies, system calls, kernel context switches and generally reduces the CPU involvement during the network phase. As a result, data can be fetched and processed much faster using the Crail shuffler than with the built-in shuffler. In Figure 7 we show the data rate at which the different reduce tasks fetch data from the network. Each point in the figure corresponds to one reduce task. In our configuration, we run 3 Spark executors per node and 5 Spark cores per executor. Thus, 1920 reduce tasks are running concurrently (out of 6400 reduce tasks in total), generating a cluster-wide all-to-all traffic of about 70Gbps per node during that phase. In contrast, the peak network usage we observed for vanilla Spark during the run was around 10Gbps. Also note that Spark/Crail is sorting 12.8 TB of data in 98 seconds, which calculates to a sorting rate of 3.13 GB/min/core. This is only about 28% slower than the current 2016 winner of the sorting benchmark – a sorting benchmark running native code optimized specifically for sorting[3].

[3]http://www.crail.io/blog/2017/01/sorting.html

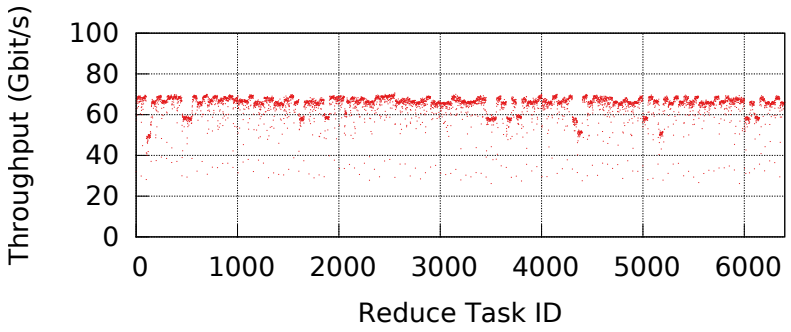

Figure 7: Network usage in a Spark/Crail sorting benchmark.

3.2 SQL

SQL is one of the most important and popular workloads supported by Spark. In a recent development cycle (between `1.6` and `2.0`), Spark/SQL has gained significant amounts of development and optimization efforts, most notably in the unification of DataSet and DataFrame APIs, and the whole-stage code generation logic. Put together, Spark/SQL has evolved into a fast and powerful SQL system. In the following we show how Crail can lift the performance of a Spark/SQL join operation through efficient network and storage I/O. The experiments are executed on a 9-node (1 driver + 8 workers) slice of the cluster.

Anatomy of a Spark/SQL Join: A join operation in Spark can be thought of as a cartesian product of two RDDs. The operation re-arranges the RDD partitions and executes a predicate on the desired columns. Here, we focus on EquiJoin where the predicate for matching rows is based on the equality of two values (referred to as a key). An EquiJoin between two tables in Spark is implemented as a map reduce job.

During the map phase, the tables are read from distributed storage and materialized as rows. The rows are then shuffled in the cluster based on a hash computed from the key column. This ensures that (a) the work will be equally distributed between worker nodes; and (b) all rows within a certain key range will be sent to the same worker node. Spark schedules a stage (comprising a number of tasks) for each of the two input tables. The key I/O operations that dictate the performance of the map phase are the reading of the input data and the writing of the shuffle data.

In the reduce phase, each worker collects two sets of shuffled rows from two input files over the network, sorts the rows in both sets based on the keys, and then scans both sets for matches. These steps are analogous to a sort-merge join operation on a local dataset. Finally, the matching rows are again written out to the distributed storage. The three key I/O operations in the reduce phase are fetching row data from the network (network bounded), sorting (CPU bounded), and joining and writing-out of rows (mixed CPU-I/O).

Performance: Spark/SQL supports multiple data sources and formats (e.g., HDFS, Hive, Parquet, JSON, etc). Our input tables consist of two parquet datasets each with a `<int,long,float,double,String>` schema. Each table contains 128 million rows, with randomly populated column entries. The `<String>` column is a 1024 character long random string. Hence, on average, each row contains a little more than 1kB of data. The join is made on the `<int>` column.

We explore two configurations: in the first configuration, vanilla Spark is used and the parquet files are stored on HDFS (`v2.6.5`). In the second configuration, we store the parquet files on Crail (using the HDFS adaptor) and also enable the Crail shuffler for Spark. In both configurations, the input tables are evenly distributed over the 8 machines with a total size of 256GB (128GBx2). The parquet files are generated using the open-sourced

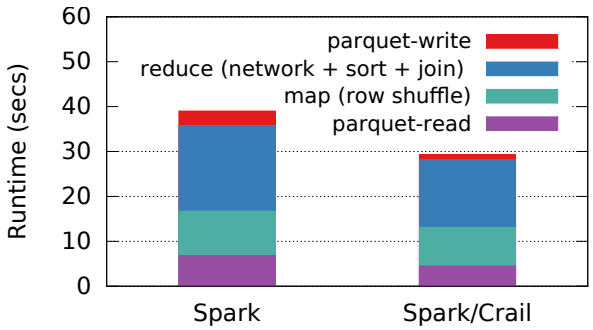

Figure 8: Performance of a SQL join in Sprak vs Spark/Crail.

`ParquetGenerator`[4] tool.

Figure 8 illustrates the performance of the join operation in both configurations. As can be observed, the use of the Crail architecture results in around 24.8% performance gains for simple SQL join workloads. These gains come from all four phases where I/O operations are involved, including parquet reading/writing and data shuffling. For instance, it takes 1.9 μs to fetch a single row from HDFS during parquet reading, but only 1.1 μs to fetch a row from Crail (using the HDFS adaptor). The net gains are about 33.8% (6.8 vs 4.5 seconds) for reading the parquet data. After reading, data is classified and written out as shuffle data. Here as well, the use of the Crail shuffler results in about 13.1% (9.9 vs 8.6 seconds) gains. The gains are limited as for both vanilla Spark and Crail, shuffle-writeouts are local operations. The shuffle map phase is followed by the reduce phase where the shuffle data is read over the network. Here, Crail delivers significant gains in raw networking performance. The shuffle data (32 (16x2) GB/node) is read in about 3.2 seconds, which calculates to a bandwidth utilization of a little more than 80Gbps. For vanilla Spark, the peak bandwidth utilization does not go above the 50Gbps mark. These gains from the raw network performance result in a 20.9% (19.1 vs 15.1 seconds) runtime improvement in the reduce phase. However, in comparison to sorting (Section 3.1), gains from the network do not proportionally translate into performance improvements of the reduce phase. This is because fetching data from the network is followed by de-serialization, sorting, and join operations, whose collective time eclipses the raw networking time. We are exploring various serializer designs with Crail to improve this situation.

As the final step, the joined rows are written out (roughly 2 GB/node), and here as well, Crail's simple write path delivers significant gains over HDFS (3.2 vs 1.1 seconds). Overall, the use of the Crail architecture results in 24.8% performance gains.

3.3 Flash Disaggregation

Traditionally, data processing platforms such as Spark or Hadoop have been designed around the concept of data locality where computation is scheduled to execute on the node that stores the data to avoid expensive network operations. Recently, the idea of storage disaggregation has become more popular [9, 16]. Disaggregation decouples storage and compute, which simplifies the deployment and management of the hardware resources, and also drives down cost. The hope is that due to the improved network speed of high-performance interconnects the overhead of crossing the network during I/O can be reduced to a minimum. However, as we have seen in Figure 1, the actual overheads in Spark and Hadoop network operations are caused mainly by the software stack despite the fast networking hardware. Consequently, running Spark or Hadoop in a disaggregated environment is costly from a performance point of view. With Crail, the software overheads during I/O are reduced which opens the door for storage disaggregation. This is illustrated in Figure 9 where we compare the performance of the distributed sorting workload in two configurations. In the Crail configuration, data is read, shuffled and

[4]https://github.com/zrlio/parquet-generator

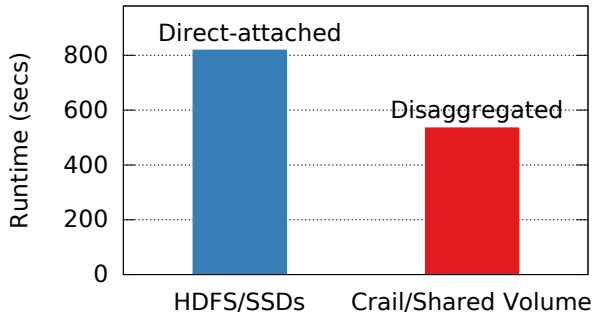

Figure 9: Using Crail for flash disaggregation in Spark.

written out using Crail's disaggregated flash tier. We use Crail's shared volume storage plugin in combination with an industry-standard flash enclosure connected via SRP (SCSi over RDMA) over a 56Gbps InfiniBand link. The second configuration is a vanilla Spark deployment, non-disaggregated, with both HDFS and Spark shuffler configured to use local memory and SSDs. We made sure that both configurations have the same aggregated flash bandwidth (10 GB/s) and about the same total flash capacity. As can be observed, the disaggregated Crail configuration performs better than the non-disaggregated Spark configuration, despite all the I/O operations being remote. This shows that with Crail, remote I/O operations can be accelerated to a level that outperforms local I/O operations in Spark, and as a result makes disaggregation a viable solution.

4 Related Work

Lately, various efforts have focused on making better use of fast network or storage hardware for data processing. Most of those works fall into one of two categories. The first category consists of systems developed from scratch for the new hardware. Examples of this type are flash optimized key/value stores [6, 18], RDMA-based key/value stores [12, 8, 22], RDMA-based distributed memory systems [3, 21], RDMA-based distributed transaction processing systems [8, 15], distributed join engines leveraging RDMA [1, 24], or even completely new database architectures designed for high-speed networks [4]. All these efforts manage to achieve very good performance due to the hardware software co-design, but they are often not integrated into a publicly available data processing platform and/or only target very specific applications. In contrast, Crail is an I/O architecture that can be applied to different open source data processing frameworks supporting a wide range of applications and workloads.

The second category of work aims to integrate fast I/O hardware into existing systems and frameworks. For instance, in one body of work the network stack of the HDFS file system or the Spark shuffle manager have been re-written for high-speed networks [11, 3, 12]. Similarly, efforts have been put in place to better integrate fast networks into Tachyon [19], a popular distributed caching system. However, all these works are retrofitting an InfiniBand-based messaging service in an existing message-based networking code, resulting in only marginal improvements in performance. Crail, on the other hand, implements a tight semantic integration of fast I/O hardware and high-level data processing operations such as shuffle, broadcast etc., and as a result manages to improve workload performance substantially.

With the availability of high-performance networks, resource disaggregation has become a viable option for data processing systems [10]. A recent workload-driven assessment has concluded that current high-performance network technologies are now sufficient to maintain application performance in a disaggregated datacenter – assuming the necessary end-host optimizations (e.g., RDMA) are being leveraged [9]. Specifically for storage, ReFlex – a recently proposed system for flash disaggregation – enables applications to maintain their performance while accessing remote flash devices [17]. Erfan Zamanian et. al further propose NAM-DB, a novel

distributed database architecture that uses RDMA and disaggregated memory for implementing snapshot isolation and scalable transactions. Crail's disaggregated architecture builds on similar concerns of providing fast access to remote resources. However, in contrast to those existing works, Crail provides a comprehensive solution to accessing memory and storage across a cluster, both local and disggregated. For instance, Crail naturally exploits any mix of local and remote resources at a fine granular manner through horizontal tiering.

Finally, there has been a body of work on improving the I/O efficiency within a single machine, most recently [23, 2, 5, 13]. These efforts are orthogonal to Crail as Crail permits different user-level I/O techniques to be integrated into its distributed I/O architecture through the storage tier interface.

5 Conclusion

We presented Crail, an I/O architecture tailored to best integrate fast networking and storage hardware into distributed data processing platforms. Crail is based on a distributed data store acting as a fast I/O bus for platform specific I/O modules. Crail not only takes full advantage of the capabilities of modern hardware in terms of zero-copy or asynchronous I/O, but it also changes the way local and remote storage resources are used in a high-speed network deployment. Crail is a recently started open source project (`www.crail.io`) and there are many opportunities to contribute.

References

[1] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1463–1475, 2015.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, 2014.

[3] Adithya Bhat, Nusrat Islam, Xiaoyi Lu, Wasi Rahman, Dipti Shankar, and Dhabaleswar Panda. A plugin-based approach to exploit rdma benefits for apache and enterprise hdfs. In *The Sixth workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, 2015.

[4] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[5] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, 2012.

[6] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, 2010.

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

[8] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, 2015.

[9] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, 2016.

[10] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

[11] Nusrat Islam, Xiaoyi Lu, and Dhabaleswar Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *24th International Conference on Supercomputing (ICS '16)*, 2016.

[12] Nusrat Islam, Wasi Rahman, and Dhabaleswar Panda. In-memory i/o and replication for hdfs with memcached: Early experiences. In *IEEE International Conference on Big Data*, pages 213–218, 2014.

[13] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[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. 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.

[16] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash local flash. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[18] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 207–219, 2015.

[19] Mellanox. RDMA Improves Tachyon Remote Read Bandwidth and CPU Utilization by up to 50%, https://community.mellanox.com/docs/DOC-2128.

[20] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, 2013.

[21] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, 2015.

[22] 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.

[23] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014.

[24] Wolf Rodiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205, 2016.

[25] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. jverbs: Ultra-low latency for data center applications. In *The Proceedings of the Fourth ACM Symposium on Cloud Computing*, SoCC '13, pages 10:1–10:14, 2013.

[26] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir]relevance of network performance for data processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016.