Design of a Resilient, High-Throughput, Persistent Storage System for the ATLAS Phase-II DAQ System

Adam Abed Abud[1,2,*], *Matias* Bonaventura[2,3,**], *Edoardo* Farina[2,4], and *Fabrice* Le Goff[2]

[1]University of Liverpool, The Oliver Lodge, Oxford St, Liverpool L69 7ZE, United Kingdom
[2]European Laboratory for Particle Physics (CERN), Geneva 23, CH-1211, Switzerland
[3]Instituto de Ciencias de la Computación UBA-CONICET, Ciudad Universitaria, CABA, Argentina
[4]Università degli Studi di Pavia and INFN, Via Agostino Bassi, 6, Pavia, Italy

Abstract. The ATLAS experiment will undergo a major upgrade to take advantage of the new conditions provided by the upgraded High-Luminosity LHC. The Trigger and Data Acquisition system (TDAQ) will record data at unprecedented rates: the detectors will be read out at 1 MHz generating around 5 TB/s of data. The Dataflow system (DF), component of TDAQ, introduces a novel design: readout data are buffered on persistent storage while the event filtering system analyses them to select 10000 events per second for a total recorded throughput of around 60 GB/s. This approach allows for decoupling the detector activity from the event selection process. New challenges then arise for DF: design and implement a distributed, reliable, persistent storage system supporting several TB/s of aggregated throughput while providing tens of PB of capacity. In this paper we first describe some of the challenges that DF is facing: data safety with persistent storage limitations, indexing of data at high-granularity in a highly-distributed system, and high-performance management of storage capacity. Then the ongoing R&D to address each of the them is presented and the performance achieved with a working prototype is shown.

1 Introduction

Over the next few years the ATLAS experiment[1] at CERN will undergo a major upgrade to adapt and take advantage of the new conditions provided by the High-Luminosity LHC[2] (Phase-II upgrade). The upgrade involves all detectors as well as the Trigger and Data Acquisition system[3] (TDAQ) which is responsible for collecting data from the detectors, selecting the interesting fraction of them, and recording the selected data. The Phase-II TDAQ system plans to read out detectors at 1 MHz for a total throughput of approximately 5 TB/s of data, and to select $\approx$1 % of them for a total recording budget of 60 GB/s.

As shown in Figure 1 the Dataflow system (DF) receives data from the Readout system. DF has three main functionalities in TDAQ: 1) collecting all detectors' data corresponding to a single trigger signal into an entity (an event), 2) buffering the readout data before and while they are analyzed by the Event Filter (EF), and 3) aggregating and transferring selected events to permanent storage. For Phase-II DF introduces a novel design as compared to the

[*]e-mail: adam.abed.abud@cern.ch
[**]e-mail: matias.alejandro.bonaventura@cern.ch;mbonaventura@dc.uba.ar

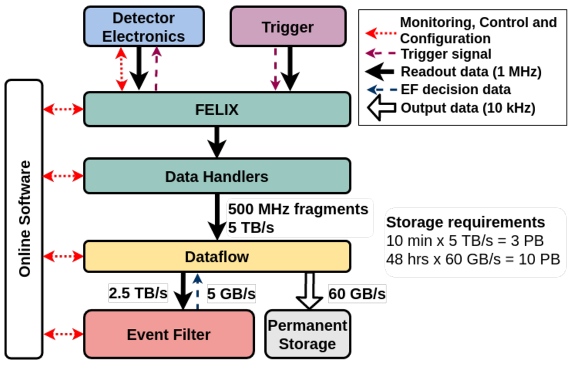

Figure 1. Phase-II TDAQ overview focused on DAQ components: the Readout system (green) forwards trigger signals from the Trigger (purple) to detectors' electronics (blue) and forwards readout data to the Dataflow system (yellow). The data are buffered on persistent storage and provided to the Event Filter (red) upon request for analysis. When selected, data are aggregated and eventually sent to permanent storage. All these components are operated and monitored via the Online Software system (white).

first three runs of the LHC: data are buffered on persistent storage to decouple the detectors' activity from the event selection process. DF has to support fragment insertion at 500 MHz, deletion at 500 MHz, and request at 230 MHz. These rates convert into a throughput of ≈5 TB/s of input, and ≈2.5 TB/s of output to EF. DF storage capacity is estimated according to Event Filter analysis maximum duration (ten minutes) and transient storage requirements for selected events (up to 48 hours), for a total of 13 PB. Finally, as any TDAQ component, DF has a deadtime and downtime budget of 0.02 % [4].

Overall DF is a large-capacity high-throughput distributed storage system indexing every detector data fragment, that needs to provide high service availability and data safety. Performance and efficiency have to be built from the start.

The following sections present some of the challenges for building such a system. Section 2 presents distributed indexing at high data rates. Section 3 discusses fault tolerance and redundancy policies. Section 4 introduces techniques to manage local storage. Finally, section 5 shows the performance of the current prototype system.

2 Indexing

Each 1-MHz event consists of 500 fragments. Each fragment can be requested individually and must therefore be software-addressable. In normal conditions, on average, DF should contain around 250 million fragments distributed across a few hundreds servers. Generic solutions exist to index such quantities of data: distributed file systems or object stores like Lustre[5] or Ceph[6]. These solutions implement data indexing with different baseline mechanisms: 1) a server in the system is used to maintain a global index; this can suffer bottleneck, scalability, overhead, and fault-tolerance issues. 2) an algorithm computes a location in the system based on properties of the data to index; this can suffer flexibility and fault-tolerance issues. A previous evaluation of the performance of distributed file systems for ATLAS DAQ[7] showed that these specific solutions do not provide all the requirements needed for DF. Another approach for a distributed data indexing system can be found in DAOS[8]. It provides a more complex technique based on a consistent hashing ring and is optimized for NVM technologies and Intel Optane hardware. An initial evaluation of DAOS on a small cluster interconnected via an Omnipath network showed poor performance. Also, DAOS requires the usage of Intel Optane persistent memory modules ; with a cost per capacity 50 % higher than the most expensive NVMe SSD this technology has significant financial implication. Technically, DAOS remains a very interesting project to implement DF but its performance and financial viability remains to be further studied.

Taking advantage of the static nature of DF a solution based on a combination of a simple static algorithm and indexing data structures was designed. The system is quite static: the topology never changes during a data-taking session and a specific fragment always comes from the same subset of Readout computers. The fragment identifiers are used for global placement and local indexing; they consist of three numbers: data-taking session, event, and fragment. At the global level, with simple modulo operations and static rules each fragment is associated to an ordered subset of storage servers. Given a fragment identifier, the static algorithm produces always the same subset of storage servers. A subset rather than a single server is returned to handle server failures: if the first server fails to respond the other servers in the set are used consecutively. The impact of this mechanism on data availability upon a server failure is explained in the following section. Both writers (Readout servers) and readers (Event Filter servers) share this algorithm to determine the same location for each fragment. When a server needs to write new data, it provides the fragment identifier to the algorithm to determine to which storage server the fragment will be sent. Similarly on the readers side, it uses the fragment identifier to determine to which server the read request will be sent. This static global placement scheme has the advantages that no central resource is involved and minimal communication between nodes is needed.

At the local level, once a fragment arrives to a DF server, the data's physical location on storage media is indexed by in-memory structures. Two technologies were tested: adaptive radix tree[9] (ART) and hash tables, in two different setups, single or nested. The fragment identifiers are used as the index key. The index can be implemented with a single structure or a nested set of the structures, one for each component of the key. As shown in Figure 2 synthetic performance benchmark showed that our use case is best handled by using nested Google dense hash maps. The benefit from nesting hash tables likely comes for higher density of data in memory and depends, therefore, on the sparseness of the input keys and their components'.

3 Fault Tolerance

The hardware implementation of DF consists of computing servers hosting storage devices interconnected via network switches and routers. The network will come with its own fault-tolerance mechanisms and is out of the scope of this paper. Both servers and storage devices are sensitive to hardware failure against which DF must protect itself to comply with its requirements of downtime and data loss. To provide the required throughput the majority of the storage devices will be high-throughput large-capacity NVME SSDs. They are particularly sensitive to failures: their MTBF[1] is similar to that of conventional HDDs: around a few million hours; in addition solid-state technology suffers significant degradation as a direct effect of program/erase cycles. The parameter used by SSD manufacturers to specify the longevity of the devices on this aspect is called *endurance*. A typical value for NAND-based SSDs is 3 DWPD (Disk Write Per Day): over its warranty lifetime the disk can be written entirely 3 times each day. As a first approximation, considering a DF implementation with 2000 SSDs of 10 TB each, constantly writing at 5 TB/s, each SSD will encounter failure after 230 days.

[1]Mean Time Between Failure: a common metric used by hardware manufacturers that corresponds to the average time between a first repaired failure and the occurrence of the next one

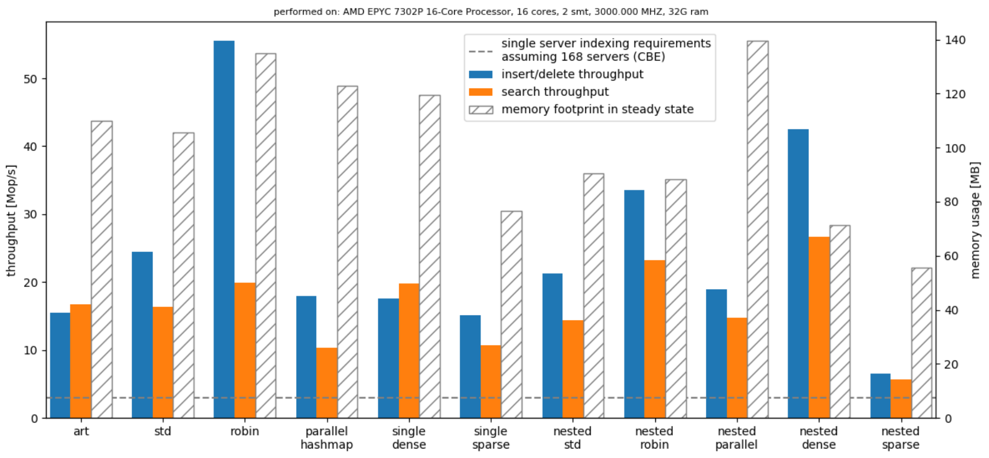

Figure 2. Indexing structures performance: throughput (left axis) and memory footprint for 1.5 M entries (right axis) for different types and implementations of indexing data structures. art: adaptive radix tree[10], std: C++ standard library hash map[11], robin: Robin implementation of hash map [12], parallel hashmap[13], sparse: Google sparse hashmap[14], dense: Google dense hash map[14]

In these conditions the system would simply not be exploitable. Two solutions can mitigate this: 1) larger SSDs; endurance is a linear function of the SSD capacity, so larger SSDs directly translates into lower failure rate; 2) in 2015 Intel and Micron Technology Inc. introduced a new technology, called 3D XPoint, for persistent storage that provides endurance up to 100 DWPD[2]. Regardless of the chosen technology, mechanisms to protect data and service availability are mandatory and can have impact on performance. These mechanisms were designed at two levels: at the global level to protect from server failures and guarantee availability; and at the server level preventing storage device failures to translate into data loss.

At the global level, the system is designed such that no server has a unique role. Upon failure new data is written and read from a different server as discussed in the previous section. If the failure is related to a storage device, affected data can be recovered thanks to the local redundancy discussed next. Later the server is manually fixed or replaced and put back online making data available again. The failure of a server is considered a rare event and only results in: 1) no data loss, but the data owned by the server is temporarily unavailable , 2) possible temporary performance degradation (throughput to this server is lost). With this mechanism a local-only data redundancy policy becomes possible at the server level. Compared to distributed redundancy no extra network traffic is generated to distribute redundant data across the system, and impact on CPU and storage overhead is kept to a minimum.

To implement data redundancy, RAID technology lacks flexibility in possible arrangement of data and parity drives. For example due to PCIe limitation a single hardware RAID controller can only make use of four devices, in a 3+1 configuration for 33 % space overhead. Each server is foreseen to host at least 6 devices so with a more configurable redundancy mechanism the space overhead can be limited to 20 % maximum.

Different free implementations of Reed Solomon[15] erasure coding were tested. Figure 3 shows that the isa-l library performs best and encodes almost 25 GB/s of data. An

[2]Example of such device's characteristics, from Intel: Optane DC P4800X

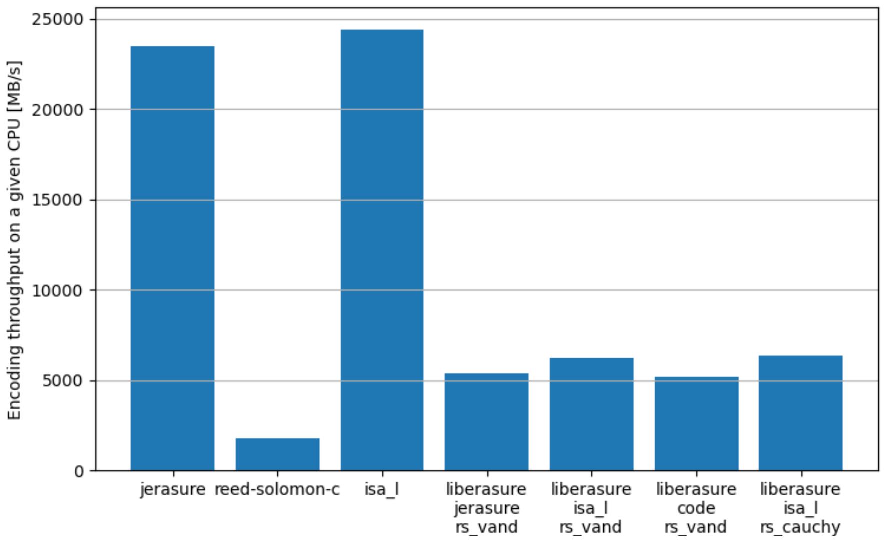

Figure 3. Encoding performance for different Reed Solomon algorithms and libraries: Jerasure [16], reedsolomon-c [17], ISA-L [18], liberasure [19]

implementation with the needed configuration parameters was made and integrated into the prototype: N being the number of storage devices per server, the input fragment is split into N-1 data shards, the Reed Solomon algorithm is used to produce redundant data which is stored along with the original shards on the N devices. This solution provides the necessary performance but has implication on performance addressed in section 4.

4 Local Storage Management

Local storage management groups all features allowing to allocate space in the storage capacity of a server and to manage the allocated space. Such features usually belong to a file systems but they come with computational, I/O, and space overhead that are incompatible with DF performance requirements (single drive performance requirement depend on the implementation but the current best estimate is 3.5 GB/s for each drive). On the other hand not all features provided by usual file systems are necessary for DF: only space allocation and resilience to power cycles are needed.

RocksDB[20] was tested as a possible technology for the local storage management. RocksDB is a high performance key-value store that is widely used in industry as a backend for storage systems[21]. The objective of the tests was to measure the throughput performance and compare it to the nominal bandwidth of the storage drive. The results show that it is possible to achieve 50 % of the nominal bandwidth [22], corresponding to a write-amplification (WA) factor of two. The WA factor represents the ratio between the amount of data that is written to the storage drive and the amount of data that is internally processed by a storage technology. The RocksDB performance tests were optimized to reduce the WA factor when writing objects with an averaged size of 10 KB. In order to achieve the highest throughput, *direct I/O* was enabled when flushing to disk. This allows to bypass the operating system cache and, therefore, write directly to the storage media. In addition, write-ahead logging was disabled and a memory buffer of 64 MB was introduced to hold the data before writing to disk.

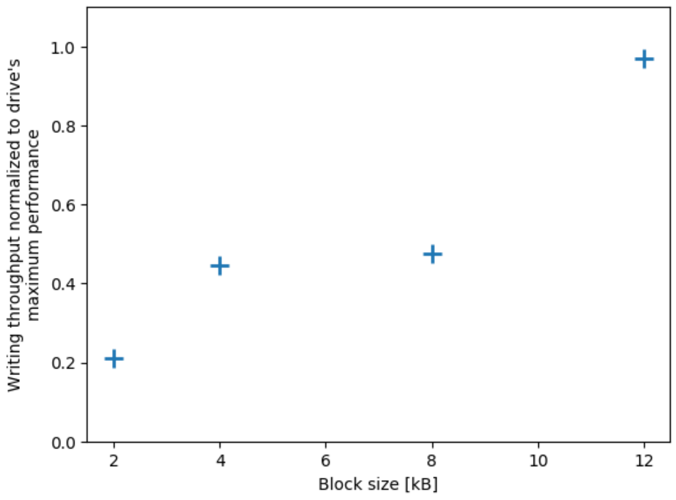

Figure 4. Single-drive writing throughput using the local storage management implementation as a function of the fragment size, normalized to the drive's maximum performance. A memory buffer of 12 kB is used to aggregate fragments before writing to disk.

Given the performance results obtained for available off-the-shelf local storage systems and inspired by conventional file systems an in-house implementation was then made. Two in-memory data structures manage the available storage capacity: a binary tree keeps track of the unallocated space, together with a log of recently-deleted blocks. The latter provides an interesting optimization as the vast majority of the written data is deleted shortly after (order of seconds or minutes): it can then be used to find free blocks without accessing the tree which is a longer computational operation. The tree itself is kept as small as possible by allocating same size blocks sequentially.

Part of this information, in addition to a header for each allocated block, is written to the drive itself to allow reconstructing the structures when in-memory version is lost (reboot, power cycle, failure). This process involves scanning the whole drive and is longer than for a usual file system. But this use case is considered a rare event and the necessary time is affordable. On the other hand, some operations such as deleting fragments do not, in general, induce any physical operation on the drive.

In addition, when performing writing operations single fragments are buffered in larger physically-allocated blocks. The size of the blocks can be adjusted to benefit from the maximum throughput offered by storage devices. Figure 4 shows the achieved writing throughput as a function of the fragment size in the case of a single storage device. Increasing the data fragment size produces a higher throughput. The lower value at 8 kB is influenced by the 12 kB buffer chosen for this test.

The whole software is multi-threaded and allows to simultaneously write, read, and delete data. A thread is dedicated to maintaining the in-memory fragment index, while space management and I/O is handled by a configurable number of threads. Threads communicate with each other via lock-free queues. Taking advantage of the particular data flow of the system, there is no need to handle concurrent accesses to the same fragment, making the design and implementation simpler than general purpose object stores. For example, there is never 2 clients trying to write, delete the same fragment at the same time. Local storage management, redundancy mechanism, and local indexing of data are strongly interleaved. For example, enabling redundancy means splitting the input fragment into smaller chunks of data, thus potentially affecting the effective throughput. These features must be assembled efficiently not to lose the performance provided by each individually. Section 5 discusses the impact of these on performance.

5 Evaluation of storage and network performance

This section presents the performance achieved with a prototype implementation of the previously discussed techniques. To benchmark DF two auxiliary applications are used to act as the Readout and the Event Filter systems. AMD EPYC 7302P servers with 32 GB RAM are used for DF, equipped with six Intel Optane P4800X NVMe SSDs and Mellanox Connect-X5 cards connected over a 100 GbE Ethernet network.

DF software is organized in layers, each implementing one of the techniques previously mentioned. Additionally, for the network communication RDMA is used relying on the event-driven NetIO library [23], and storage devices are accessed as block devices using the asynchronous libaio library [24]. Although RDMA provides better performance, TCP is being evaluated for the communication with the Event Filter where requirements are less demanding and might provide benefits for the network infrastructure (cost, operation, maintenance). For accessing storage devices SPDK [25] was also evaluated but didn't provide clear performance advantages and requires more complex programming patterns.

Figure 5 shows the achieved throughput of a first prototype implementation of DF for the different software layers as well as the raw network and storage performance when receiving write requests of increasing sizes. For the communication layer, fragments are sent and received through the network, but there is no access to the storage devices. When compared with the raw network performance, the plot shows a small overhead of the communication layer for bigger fragments (less than 1 GB/s) which increases considerably (up to 5 GB/s) for small fragments. This could be overcome in the future by aggregating small fragments at the Readout server.

As already mentioned, data redundancy is a pivotal aspect of the whole system and it has been integrated in the local storage management system. In this case, the space on the storage devices is managed by a single data indexing structure: the empty and used sectors are the same on all drives. When a new fragment needs to be written to disk, several equal-size data chunks (as many as drives) are obtained by means of a Reed-Solomon algorithm and they are written to the same position on all drives. Figure 5 shows a maximum achieved throughput of 6 GB/s when enabling the local storage management, indexing, and a redundancy scheme with 5 data drives plus 1 for redundancy with shards of 2 kB. Compared with the raw performance of randomly writing on 5 storage devices, the plot shows a considerable overhead of approximately 4 GB/s. Although in the previous sections benchmarks of individual features showed acceptable performance, this overhead reveals that the assembling and integration of all features still requires optimization. Possible optimization points include minimizing memory copies, improving threading model, general code refactoring, and reducing write amplification.

6 Conclusion

The future Dataflow system of ATLAS TDAQ is a large-capacity distributed storage system. This paper presented some of the key challenges studied so far, including indexing data at

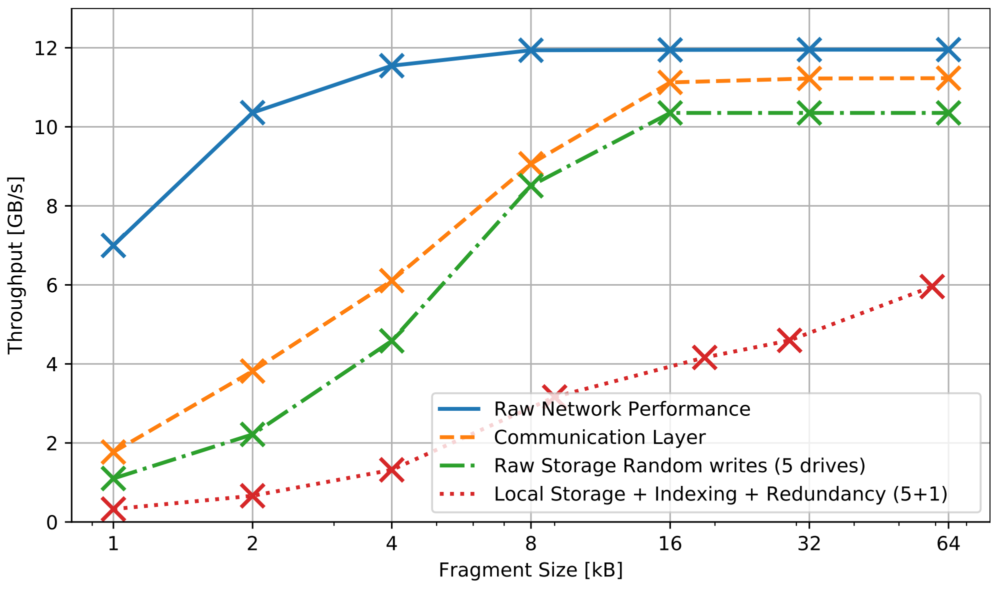

Figure 5. DF writing performance for the different software layers including raw hardware throughput. Raw network performance is measured using *ib_send_bw* and raw storage performance is measured using the fio tool [26] with the libaio engine [24].

high rates, sustaining TB/s of writing and reading throughput, ensuring data safety and efficiently managing the space on the storage devices.

By taking advantage of the static nature of the system a solution to index fragments based on a combination of a static algorithm and a simple data structure was developed. A flexible fault tolerance mechanism was implemented with the Reed-Solomon algorithm after a careful investigation of different libraries. The performance of the resulting prototype was tested with multiple drives including both indexing and redundancy mechanism. Testing of the individual components shows good performance. As expected at this stage the integrated system needs further software optimization in order to reduce the efficiency loss of the assembled features but these results from an early prototype are encouraging. The project is on track as a phase of benchmarking and optimization is planned through to the next project milestone in Fall 2021 when the prototyping phase will end. After this the implementation phase will start and will take advantage of the soon-to-be-released next generation of NVMe SSDs based on PCIe Gen4, promising to provide two to three times more throughput.

In parallel to the work of building an in-house implementation, the effort to monitor, evaluate, understand, and contribute to COTS solutions like DAOS will continue. COTS products could eventually provide alternatives either to the full system or to some of its components.

References

[1] ATLAS Collaboration, JINST **3**, S08003 1 (2008)

[2] ATLAS Collaboration, PoS **EPS-HEP2019**, 184 (2020)

[3] ATLAS Collaboration, Journal of Instrumentation **11**, P06008 (2016)

[4] ATLAS Collaboration, *Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System* (2017), `https://cds.cern.ch/record/2285584`

[5] P. Braam, *The Lustre storage architecture*, in *arXiv preprint arXiv:1903.01955* (2019)

[6] S.A. Weil, S.A. Brandt, E.L. Miller, C. Maltzahn, *CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data*, in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), p. 122–es, ISBN 0769527000

[7] A. Abed Abud, F. Le Goff, G. Avolio, *Performance evaluation of distributed file systems for the phase-II upgrade of the ATLAS experiment at CERN*, in *Journal of Physics: Conference Series* (IOP Publishing, 2020), Vol. 1525, p. 012028

[8] Z. Liang, J. Lombardi, M. Chaarawi, M. Hennecke, *DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory*, in *Supercomputing Frontiers* (Springer International Publishing, 2020), pp. 40–54, ISBN 978-3-030-48842-0

[9] T.N. Viktor Leis, Alfons Kemper, *The adaptive radix tree: ARTful indexing for main-memory databases*, in *ICDE '13: Proceedings of the 2013 IEEE International Conference on Data Engineering* (2013), pp. 38–49, `https://doi.org/10.1109/ICDE.2013.6544812`

[10] A. Dadgar, *libart*, `https://github.com/armon/libart`

[11] C++, *C++ standard library unordered map*, `https://en.cppreference.com/w/cpp/container/unordered_map`

[12] T. Goetghebuer-Planchon, *Robin map*, `https://github.com/Tessil/robin-map`

[13] G. Popovitch, *The parallel hashmap*, `https://github.com/greg7mdp/parallel-hashmap`

[14] Google, *Google sparse hash*, `https://github.com/sparsehash/sparsehash`

[15] G.S. Irving S. Reed, *Polynomial Codes over Certain Finite Fields*, in *Journal of the Society for Industrial and Applied Mathematics* (1960)

[16] *Jerasure: Erasure coding library*, `https://github.com/tsuraan/Jerasure` (2020)

[17] *reedsolomon-c*, `https://github.com/jannson/reedsolomon-c` (2020)

[18] *Isa-l*, `https://github.com/intel/isa-l` (2020), accessed: 2020-05-31

[19] *liberasure*, `https://github.com/atomgalaxy/liberasure/` (2020)

[20] Y. Jia, F. Chen, *From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution*, in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2020), pp. 192–201

[21] Facebook, *Users of rocksdb and their use cases*, `https://github.com/facebook/rocksdb/blob/master/USERS.md`

[22] A. Abed Abud, M. Bonaventura, E.M. Farina, F. Le Goff, *R&D Studies on a Persistent Storage Buffer for the ATLAS Phase-II DAQ System*, in *Technology and Instrumentation in Particle Physics 2021* (2021)

[23] J. Schumacher (ATLAS Collaboration), *Event-driven RDMA network communication in the ATLAS DAQ system with NetIO*, in *Proceedings of the 24th International Conference on Computing in High Energy and Nuclear Physics* (2019)

[24] S. Bhattacharya, S. Pratt, B. Pulavarty, J. Morgan, *Asynchronous I/O Support in Linux 2.5*, in *Proceedings of the Linux Symposium* (2010)

[25] Z. Yang, J.R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, L.E. Paul, *SPDK: A development kit to build high performance storage applications*, in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (IEEE, 2017), pp. 154–161

[26] J. Axboe, URL http://freecode. com/projects/fio (2014)