Leveraging Apache Arrow for Zero-copy, Zero-serialization Cluster Shared Memory

Philip Groet
Delft University of Technology
Delft, The Netherlands
philip.groet@gmail.com

Joost Hoozemans
Voltron Data
United States/Worldwide
joost@voltrondata.com

Andreas Grapentin
Hasso Plattner Institute, University of Potsdam, Germany
andreas.grapentin@hpi.de

Felix Eberhardt
IBM/Hasso Plattner Institute
Germany
felix.eberhardt@ibm.com

Zaid Al-Ars
Delft University of Technology
Delft, The Netherlands
Z.Al-Ars@tudelft.nl

H. Peter Hofstee
IBM/TU Delft
United States
hofstee@us.ibm.com

arXiv:2404.03030v1 [cs.ET] 3 Apr 2024

ABSTRACT

This paper describes a distributed implementation of Apache Arrow that can leverage cluster-shared load-store addressable memory that is hardware-coherent only within each node. The implementation is built on the *ThymesisFlow* prototype that leverages the OpenCAPI interface to create a shared address space across a cluster. While Apache Arrow structures are immutable, simplifying their use in a cluster shared memory, this paper creates distributed Apache Arrow tables and makes them accessible in each node.

KEYWORDS

Apache Arrow, ThymesisFlow, Cluster Shared Memory, Memory disaggregation, OpenCAPI, PowerPC, Parallel computing, Power9

1 INTRODUCTION

Unlike compute resources, that can be flexibly traded off against performance, main memory in each node in a cluster must be provisioned for a worst case to achieve prevent severe performance degradation. In addition, to prevent jobs from being inadvertently killed because they ran out of memory users often over-estimate their memory requirements. The result is internal fragmentation and significant memory under-utilization. A study by Google into its Borg clusters reveals that only around 40% of memory is used [17], and a study by Microsoft shows 50% of VMs never touch 50% of their memory [9]. Considering that memory is one of the largest and growing contributors to the total cost of a server [4], it is clear that systems capable of sharing memory nodes across a cluster could be a major cost saver.

The ThymesisFlow [13] prototype enables byte-addressable shared memory regions between nodes in a cluster, completely transparent to the application. In other software RDMA implementations such as FastSwap [5], memory is copied/swapped to local memory, while ThymesisFlow memory stays in one place and transactions are sent to that memory. All data can be load-store accessed and cached locally, speeding up repeated accesses to remote memory locations. While built leveraging the OpenCAPI interfaces, ThymesisFlow does break with the OpenCAPI goal of being fully cache coherent as cached instances of accessed remote memory locations will not be updated when the remote node invalidates the cache.

We propose to use the in-memory format of Apache Arrow [1]. Arrow allows for various applications to access the same data, without copying and without serializing any data [3, 12]. The key aspect we utilize is that most Arrow objects, once instantiated, are immutable and thus the missing cache coherency is not a problem with read-only access. This work extends Arrow's ability to be readable by multiple *applications* on the same machine, to multiple *machines* connected with ThymesisFlow. Accessing the shared memory through the Arrow API allows ensuring memory consistency, while allowing applications to leverage the shared memory without worrying about cache coherence. Our implementation extending Apache Arrow is open-source and available on github [7].

2 BACKGROUND

Scaling workloads can fundamentally be done in a *vertical* or *horizontal* fashion. The former is done by adding resources in a single large SMP system with multiple CPU-sockets coherently interconnected. However, this type of system has its limitations: maintaining coherence of the caches leads to performance problems due to latency and bandwidth overheads of the coherency protocol. These limitations are addressed with *horizontal scaling* where resources are available in the form of networked clusters of servers with e.g. Ethernet as the interconnect. However, in those clusters the cost of communication is significantly higher than in SMP systems [10]. Consequently, workloads with tightly coupled communication are better suited for *vertical scaling*, whereas the opposite is true for *horizontal scaling* [8]. By combining ThymesisFlow with Apache Arrow, this paper minimizes data copy bottlenecks which currently hinder efficient communication between server nodes.

2.1 Zero-copy, zero-serialization

In a coordinated compute cluster, the classical approach to data transfer involves a serialization step, broadcasting the data to every node through the network, and then de-serializing it into a format known to the local machine. These might be expensive operations, e.g. if pointers need to be resolved and converted to relative offsets. To enhance traditional data transfer methods, a logical progression is to establish a "common language" for systems to communicate

This work is licensed under a Creative Commons Attribution 4.0 International License.
Proceedings of the 3rd Workshop on Heterogeneous Composable and Disaggregated Systems, San Diego, California, USA (HCDS '24), 2024.

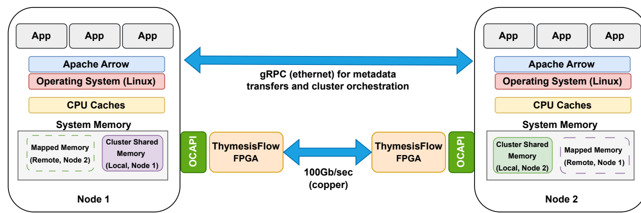

Figure 1: High-level system components. Arrow as a user library, ThymesisFlow connecting the two OpenCAPI busses together allowing for remote memory accesses.

in. The goal is to have a uniform in-memory data format. Apache Arrow aims to achieve this standardization.

Previous work on this topic [2] succeeded in integrating Apache Arrow with ThymesisFlow using the Plasma object store, thereby enabling transparent data communication across multiple compute nodes. However, the use of Plasma requires extra copy operations, which reduces efficiency.

This work eliminates the need for serialization and reduces the cost of copy operations. As shown in Figure 1, this is done by leveraging the load-store based sharing of node memory across systems offered by ThymesisFlow. A concept we call *cluster shared memory* can be utilized to eliminate copy operations, introducing a *zero-copy* paradigm underpinning the *zero-serialization* paradigm formulated by Arrow.

2.2 Cluster shared memory

This paper introduces the new term *cluster shared memory* (CSM) to describe a system that shares memory addresses across a cluster. Depending on how cache coherence is enforced, we can define three levels of CSM:

- Non-coherent CSM (NC-CSM): this refers to memory sharing without any coherence guarantees.
- Locally-coherent CSM (LC-CSM): this refers to enforcing cache coherence on individual nodes only. *Our work targets LC-CSM clusters.*
- Globally-coherent CSM (GC-CSM): this refers to enforcing cache coherence at the cluster level.

2.3 ThymesisFlow

ThymesisFlow is an open-source HW/SW co-designed memory disaggregation prototype. A ThymesisFlow system supports memory *lending* and *borrowing* across nodes. A *lender* may transparently map parts of a *borrower's* main memory into its local physical address space, as if additional memory modules were plugged into the *lender*. It does this using an FPGA-based NICs attached to the Power9 OpenCAPI bus. In the prototype, the connection between the two machines is a 100Gib/s link, with an effective bandwidth up to ~10GiB/s [13] and a RTT latency of ~650ns [16].

Using the OpenCAPI connection allows for communication between connected devices in a partially cache coherent fashion. This paper focuses on a scenario where both the *lender* and *borrower* access the shared memory with concurrent read and write accesses. When a borrower reads from a lender's memory, first the borrower CPU cache is accessed, then the read is sent through the ThymesisFlow link, then the lender cache is snooped through the OpenCAPI bus, and finally if both caches miss, the actual memory is read. Because this setup shares architecture with local memory accesses, CPU features such as memory pre-fetching, out-of-order processing and pipelining are still active for remote memory [18]. However, while memory is coherent within the node, it is not coherent across nodes. In the methodology section of this paper, we outline solutions to these cache coherency issues.

In Power10 the concept of *Memory Inception* was announced, which also enables disaggregated memory sharing in a cluster. With Memory Inception, the memory latency is expected to be significantly lower than ThymesisFlow, with only 50-to-100 nanoseconds of additional latency incurred through the link [14]. Additionally, with the OpenCAPI standard integrated into CXL [6, 15] and CXL 3 offering memory pooling functionality among several systems, our approach promises to inspire comparable configurations on other platforms.

2.4 Apache Arrow

Apache Arrow is an in-memory data format specification for tabular data organized in columnar structures, with corresponding libraries for various popular programming languages. It supports many different data types, including complex ones such as dictionaries, nested and variable-length data. The power of the Arrow format lies in its in-memory data layout, which allows for interoperability of objects between different applications. It enables a Python program to create a dataset, hand the metadata to another application implemented in a different programming language, which is then able to read the data without marshaling or serialization, or even copying the data itself. Finally, Apache Arrow tables use offsets rather than pointers.

Apache Arrow objects, once instantiated, are immutable. Modifying data requires creating a new object. The structure in which Arrow encodes arrays is called a *RecordBatch*. The structure of a RecordBatch is encapsulated in the library supporting each language, and not part of the location agnostic data format. For example, in the Arrow C++ library, the RecordBatch is implemented as a class instance, and Arrow Arrays are stored with the C++ `std::vector` type. Every node reading the data will need to instantiate its platforms Arrow library with this data. In Section 3.1, we describe how we serialize the structure and references of columnar data.

3 METHODOLOGY

3.1 Serializing Arrow columnar structure

Apache Arrow already contains methods for serializing a full table, including the data itself, into a buffer, this is done using the RecordBatchWriter and RecordBatchReader IPC methods, useful for copying objects between nodes. However, these IPC API calls are not zero-copy and will write data into a new buffer, and move it to a shareable location. To be able to serialize only the table descriptor of an Arrow object, we modify the IPC API of Arrow to not include the data, only the table descriptor and a reference to the data.

The data itself does not need to be copied, as it is placed in a globally accessible shared memory When another node wants to

access the table, it only needs to serialize, send, and de-serialize the structure and reference information. Because ThymesisFlow allows for creating a shared address space across nodes where every address uniquely designates memory, all the pointers to the data remain valid. This way, no extensive redesign of Arrow is necessary, as the general structure of the code base to instantiate objects is kept intact.

3.2 Flushing CPU caches before invalidity

We need to take care when creating new Arrow objects that are accessible by other nodes. When CPU caches are populated with data of a certain memory region, and another CPU writes to that region, only the writer's CPU cache will be updated. This poses a problem when the CPUs with incorrect cache entries try to read data, their request will hit only their local cache, not reading the newly updated remote memory. Thus, we need to first invalidate the caches of all CPUs, so that the caches become up to date when they query the newly updated memory. Power9 processors, however, do not support cache invalidation and data cache flush instructions must be used instead. We use the `dcbi` instruction to flush single 128-byte cachelines.

Flushing, however, may change the backing memory. For the purpose of emptying cache-lines we use flushing operations on all CPUs in a cluster to empty the relevant caches of memory regions. We execute the flush operations before we write the actual wanted data, allowing for the new data to then be repopulated in all the CPU caches on reads.

Power9 has an out-of-order instruction execution unit. To ensure no calculations are done before all memory blocks have been flushed, we add memory barriers before and after all flushing operations. We do not need to place the barriers between every flush operation, as we do not care if individual flush instructions are swapped.

After the initialization of an object has finished, we do no longer need to worry about cache coherency. The Apache Arrow format guarantees that created objects are immutable. Thus, when the data is written to the memory, and all CPU caches are made to be coherent once, any and all reads afterwards will update the CPU caches with up-to-date data. Arrow will guarantee no writes happen to the region, and we thus do not need to make the system coherent again. The expensive invalidate operations only happen during initialization of the data, during reading of the data we have no overhead except refetching data into local CPU caches.

A typical flow for creating an Arrow object on shared memory looks like this:

- Allocation: memory owning node allocates buffer and passes address to requesting node
- Clear cache-lines: all processors will flush their caches of the requested memory region
- Write into memory: any processor in the cluster writes the data to the shared memory region.
- Flush when not local: if the processor has written to memory it does not own, it must flush its CPU cache to ensure the data is actually written to the remote memory.
- Coherent reading: any processors which read this newly created memory region will not have local caches of this region. Subsequent reads will thus populate the CPU cache with up-to-date values.

3.3 Preventing Address Translation by mapping regions to the same address on every node

Even though the Arrow data itself does not contain any absolute pointers, the metadata does. This includes metadata containing absolute pointers to memory where the data buffers are stored. Therefore to transfer information where an Arrow object is stored, all pointers to data buffers would have to be updated with the new location where data is mapped. To ease this process, we map every memory region to the exact same location in every node. We do this using the Linux mmap flag `MAP_FIXED`. This flag tells the kernel that the address passed is not a suggestion, but an exact requirement where the region is mapped to. There are caveats to this, since we need to make sure on each participating process that the requested region in virtual memory is actually free and available to map, otherwise the mmap call will fail.

3.4 Allocating in custom memory regions

In a standard application, malloc and its family of functions only give the application limited control where memory is mapped. When a user calls malloc, the c library decides if the currently available heap space is sufficient to satisfy the new request. If the current heap is not big enough, more pages may be requested from the kernel using the `brk` and `sbrk` syscalls, or the c library may decide to map pages directly using mmap for large allocations. We cannot however instruct malloc to allocate in a certain memory region.

For this reason, we extended Apache Arrow to include a memory manager. The manager allows for defining custom memory regions that are made available to malloc to satisfy allocation requests from.

3.5 Remote memory allocations

We choose an architecture where the CPU that owns the backing memory is responsible for handling allocations and cache behaviors. We chose this to prevent race conditions, and to have only one CPU be responsible for managing the malloc data structures. Concretely, this means that when a remote node wants to write to local memory, it will have to first request memory from the owning node. The owning node will then call local malloc methods and return the allocated address to the remote node.

This architecture allows for any node in a cluster to write to any other node. The result is an architecture where a single node writes data to all other nodes, not only nodes writing to their own local memory. Some applications are:

- One node writes a dataset to all other nodes.
- In a big data pipeline, we can have the result of one stage be immediately written to the memory of the next stage.

3.6 Spanning Tables across nodes

Arrow allows for creating not only contiguous columnar data such as a RecordBatch, but also columnar data which has non-contiguous columns called Tables. Unlike RecordBatches which contain Arrays which guarantee contiguous memory buffers, Tables contain

Table 1: Time to initialize table in remote memory (1GiB of data, uint64 elements). For reference a simple gRPC call takes on average 3.3ms.

Component	Time avg [ms]
Malloc request (gRPC)	4.99
Remote pre-write flush (gRPC call + flush)	51.84
Write to remote memory	180
Flush local write cache to remote	60.32
Serialize table descriptor	0.058
Send table descriptor to other nodes	3.23
Total	300.44

ChunkedArrays. ChunkedArrays may contain multiple contiguous Arrays, making the non-contiguous Chunked Array. Chunked Arrays are not part of the Arrow memory format, but rather is a library abstraction on top of the Array memory format spec.

With Arrow data being accessible to every other node in a cluster we can use the ChunkedArray abstraction to split one big array across multiple nodes. Every node will contain an Array with a contiguous memory buffer belonging to it. After which we create a single ChunkedArray which contains the Arrays of all the nodes. From a user application perspective, we can now use any Arrow supported compute function, and index any data in the array as if it is local. Arrow will resolve an index to a pointer location, which ThymesisFlow will then transparently hand off to the relevant node.

The power here is that we can have columnar data bigger than a single node, without having to share data in between nodes. Data is stored only once, and every node can access the data using its own memory instructions making use of CPU caches.

4 RESULTS

Introducing Apache Arrow promises to be a suitable candidate for managing the pitfalls of LC-CSM. However, managing the shared memory introduces some overhead during the allocation and management of objects. In particular:

- Communication between nodes to exchange metadata
- Cache flushing before and after object initialization
- Serialization and de-serialization of table descriptors
- Communication during remote memory allocations

However, we believe these effects are outweighed by the scalability benefits gained by eliminating transfers of large datasets over the network. Specifically we measured the initialization times as seen in Table 1. The total time it takes to create a 1GiB table in remote memory takes 300.44*ms* on average, of which 118*ms* is overhead. Flushing the cachelines of every node takes the longest, as the flushing is done on a per 128-byte cacheline basis, and every flushed line potentially needs to be written to remote memory.

In our LC-CSM testbed built upon a ThymesisFlow installation, we compared our implementation to a standard ethernet transfer of shared data in a cluster. Figure 2 shows that the time spent copying a dataset over ethernet is eliminated by utilizing the disaggregated memory. Only the metadata needs to be transferred over the ethernet link, which allows orders of magnitude faster sharing between nodes.

After the transfer of the metadata, the data is fully and transparently available to the application with a cache-line granularity over

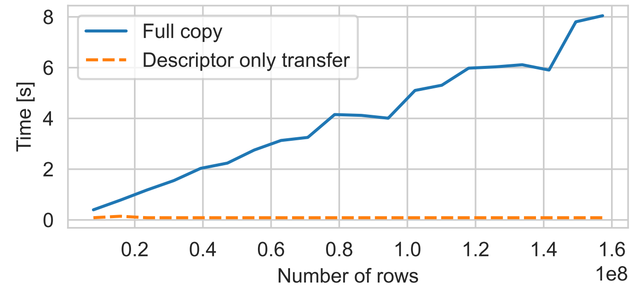

Figure 2: Throughput of data transfers over ethernet compared to sharing metadata zero-copy through the extended Apache Arrow interface.

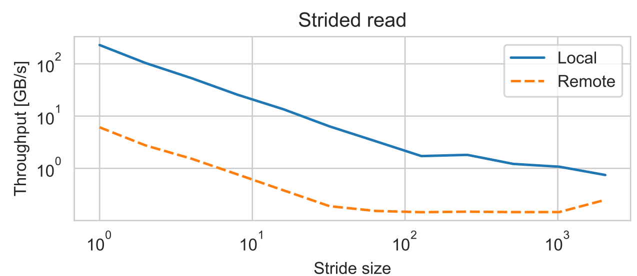

Figure 3: Apache Arrow application throughput for strided read accesses to local and remote memory.

the ThymesisFlow link, albeit with a penalty to memory latency and throughput, which becomes the limiting factor. To quantify the impact of LC-CSM on performance, we ran a series of experiments with strided read and write accesses to the remote data, shown in Figure 3. These strided access patterns are common for data analytics pipelines and are a good indicator for the performance penalty incurred by accessing remote memory [11]. An expected decrease in throughput is measured for an increase in stride size. We theorize this is because more 128-byte transactions need to be done, saturating the ThymesisFlow bus quicker, but also the memory pre-fetching units in the processor may be less able to predict future memory accesses. Comparing the remote to local memory accesses show that remote memory is considerable more limited in maximum throughput, although for a very large stride the difference is considerably less.

In conclusion, we have shown that it is feasible to utilize Apache Arrow for bridging the gap between non-cache-coherent nodes in a LC-CSM setup, maintaining memory consistency as well as flexibility of implementation. While the Arrow interface as well as the ThymesisFlow link do pose some restrictions regarding efficiency, overhead and throughput, we have shown that Arrow is well equipped to become a stepping stone towards facilitating the sharing of large data sets in a shared memory cluster.

ACKNOWLEDGMENT

This research was performed with the support of the Eureka Xecs project TASTI (grant no. 2022005).

REFERENCES

[1] The Apache Software Foundation [n. d.]. *Apache Arrow*. The Apache Software Foundation. https://arrow.apache.org/

[2] R. Abrahamse, A. Hadnagy, and Z. Al-Ars. 2022. Memory-Disaggregated In-Memory Object Store Framework for Big Data Applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–7. https://doi.org/10.1109/IPDPSW55747.2022.00211

[3] Tanveer Ahmad, Nauman Ahmed, Johan Peltenburg, and Zaid Al-Ars. 2020. ArrowSAM: In-Memory Genomics Data Processing Using Apache Arrow. In *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*. 1–6. https://doi.org/10.1109/ICCAIS48893.2020.9096725

[4] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory disaggregation: advances and open challenges. *ACM SIGOPS Operating Systems Review* 57, 1 (2023), 29–37.

[5] Wenqi Cao and Ling Liu. 2018. Dynamic and Transparent Memory Sharing for Accelerating Big Data Analytics Workloads in Virtualized Cloud. In *2018 IEEE International Conference on Big Data (Big Data)*. 191–200. https://doi.org/10.1109/BigData.2018.8621991

[6] Compute Express Link Consortium. 2023. CXL Consortium and OpenCAPI Consortium Sign Letter of Intent to Transfer OpenCAPI Specifications to CXL. https://computeexpresslink.org/wp-content/uploads/2024/01/OCC_CXL-Announcement_FINAL.pdf.

[7] ABS group. 2023. Zero-Copy, Zero-Serialization Memory Disaggregation using Apache Arrow and ThymesisFlow. https://github.com/abs-tudelft/memory-disaggregation-ThymesisFlow-Arrow.

[8] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H. Peter Hofstee. 2021. FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 30–47. https://doi.org/10.1109/MCAS.2021.3071608

[9] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Monish Shah, Scott Lee, Ishwar Agarwal, Mark D Hill, Marcus Fontoura, Stone Co, Ricardo Bianchini, Stanko Novakovic, and Samir Rajadnya. [n. d.]. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. 2 ([n. d.]). Issue 23. https://doi.org/10.1145/3575693.3578835

[10] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[11] Johan Peltenburg, Ahmad Hesam, and Zaid Al-Ars. 2017. Pushing big data into accelerators: Can the JVM saturate our hardware?. In *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^ 3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers 32*. Springer, 220–236.

[12] Johan Peltenburg, Lars T.J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. 2020. Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 281–286. https://doi.org/10.1109/ICFPT51103.2020.00048

[13] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. Thymesisflow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* 2020-October, 868–880. https://doi.org/10.1109/MICRO50266.2020.00075 Explanation of the ThymesisFlow framework. OpenCAPI with a software stack allowing for pooling of remote system memory..

[14] John Russell. 2020. IBM debuts Power10; touts new memory scheme, security, and inferencing. https://www.enterpriseai.news/2020/08/18/ibm-debuts-power10-touts-new-memory-scheme-security-and-inferencing/

[15] Debendra Das Sharma. 2019. Compute express link. *CXL Consortium White Paper* (2019).

[16] Dimitris Syrivelis. [n. d.]. *OpenPOWER Summit NA 2019: Thymesis-P: An Approach to Rack-scale Disaggregation Over OpenCAPI*. OpenPower Foundation. https://www.youtube.com/watch?v=XcjRL3Lh8Ig

[17] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[18] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. 2022. Performance Evaluation on CXL-enabled Hybrid Memory Pool. *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 1–5. https://doi.org/10.1109/NAS55553.2022.9925356