HStream: A hierarchical data streaming engine for high-throughput scientific applications

Jaime Cernuda
Illinois Institute of Technology
Chicago, Illinois, USA
jcernudagarcia@hawk.iit.edu

Jie Ye
Illinois Institute of Technology
Chicago, Illinois, USA
jye20@hawk.iit.edu

Anthony Kougkas
Illinois Institute of Technology
Chicago, Illinois, USA
akougkas@iit.edu

Xian-he Sun
Illinois Institute of Technology
Chicago, Illinois, USA
sun@iit.edu

ABSTRACT

Data streaming is gaining traction in high-performance computing (HPC) as a mechanism for continuous data transfer, but remains underutilized as a processing paradigm due to the inadequacy of existing technologies, which are primarily designed for cloud architectures and ill-equipped to tackle HPC-specific challenges. This work introduces HStream, a novel data management design for out-of-core data streaming engines. Central to the HStream design is the separation of data and computing planes at the task level. By managing them independently, issues such as memory thrashing and back-pressure, caused by the high volume, velocity, and burstiness of I/O in HPC environments, can be effectively addressed at runtime. Specifically, HStream utilizes adaptive parallelism and hierarchical memory management, enabled by this design paradigm, to alleviate memory pressure and enhance system performance. These improvements enable HStream to match the performance of state-of-the-art HPC streaming engines and achieve up to a 1.5x reduction in latency under high data loads.

CCS CONCEPTS

• **Information systems** → **Stream management**; **Main memory engines**; *Distributed storage*; **Hierarchical storage management**; • **Computer systems organization** → **Real-time system architecture**.

KEYWORDS

Data Streaming, HPC, hierarchical storage, elastic system, in-transit

ACM Reference Format:
Jaime Cernuda, Jie Ye, Anthony Kougkas, and Xian-he Sun. 2024. HStream: A hierarchical data streaming engine for high-throughput scientific applications. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673150

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673150

1 INTRODUCTION

High-Performance Computing (HPC) has seen a great growth in the volume, velocity and variety of data. From scientific instruments (e.g., large-scale physics experiments [23], telescopes [4], and ptychography [28]), simulations and workflows [17], or system monitoring [24]. Traditional mechanisms to manage this data have focused on leveraging batch-based I/O, periodical processing, and a globally available parallel file system (PFS). However, the ever-growing disparity between the production rate and the ingestion capability of the I/O system has led to a bottleneck known as the I/O gap. Attempts to reduce or solve this gap have seen the introduction of new I/O layers (remote or local) [13], systems (e.g., Hermes [15]), and new data paradigms, of which data streaming is one.

Data streaming is a data paradigm that focuses on the management of unbounded flows of data. It moves away from batch-based solutions found in other systems like MapReduce and has been primarily adopted in HPC as a mechanism for continuous memory-to-memory data transfer. Systems such as ADIOS2 [11], Scistream [7], or cloud-based message queues [2] provide scientists with the ability to continuously transmit data between applications (e.g., workflow tasks, simulation/analytics pairs, etc.). Yet, data streaming remains underutilized as a processing paradigm due to the inadequacy of existing systems. This is despite a growing interest in the community for high-performance, in-transit computation in fields like data reduction [5], continuous learning [3], or application steering [21]. Data streaming engines, computational systems that leverage the data streaming paradigm for continuous data processing, have been primarily designed for cloud architectures and are ill-equipped to tackle HPC-specific challenges. Most of the approaches to use them on HPC have focused on a direct transference of cloud systems, either by facilitating their deployment [6]; enabling pre-existing platforms to make use of HPC technologies [12]; or, by co-locating Cloud and HPC clusters [10]. HPC-first streaming engines do exist with explorations of using MPI as a data streaming communication library with capabilities for processing data within the application user space [22] and Neon [20] which presents a c++, RDMA-based, out-of-core data streaming engine for HPC.

While these engines have been designed as HPC-first engines, they adhere to traditional cloud-based architectural designs. This work argues that the unique demands of HPC, particularly data demands of scientific applications, require specific architectural

modifications not present in current systems. Data streaming engines rely heavily on in-memory operations. The high volume and velocity of data generation and the decreasing memory-per-core [27] in HPC clusters can put extreme pressure on memory, leading to memory thrashing [8]. HPC systems benefit from the presence of node-local and remote burst buffer technologies [14]. These technologies provide a support structure for data placement, which can allow HPC streaming engines to ease memory demands. Additionally, data streaming platforms lack the ability to reconfigure the resources allotted to computational tasks at runtime. The bursty I/O patterns of scientific applications [25] and the increasing multi-tenancy environments [5] where they are deployed can lead to severe competition for resources, causing starvation and back-propagation. A mechanism to adaptively shuffle resources between tasks can help alleviate this issue, with the high-speed networks in HPC being more capable of supporting such a mechanism when compared with the slower TCP/IP networks used in cloud platforms.

To address these issues, we present HStream. HStream is a data streaming engine with a new architectural design aimed at embracing the high data throughput of scientific applications. HStream's core novelty is the separation of the data and compute planes, allowing engines to control each plane at a lower granularity. To alleviate memory pressure and avoid thrashing, HStream proposes a data plane composed of a hierarchical memory manager, a globally accessible namespace controlling node-local memory while embracing HPC local and remote burst buffers. Algorithms for data management are provided to ensure low latency access in a data streaming context. To aid in managing the non-uniform data generation of HPC applications and clusters, HStream introduces the concept of ephemeral tasks which can be killed, spawned or migrated at will. Ephemeral tasks maintain access to their data through the global namespace of the hierarchical manager. Building the compute plane on top of ephemeral tasks allows HStream to implement an adaptive parallelism controller, where resources can be shifted between jobs and tasks to respond to the demand imposed by the applications. Finally, to drive these two mechanisms intelligently, HStream introduces a stream observer to collect and understand both the characteristics of the streams flowing through the systems as well as the hardware utilization metrics of the platform.The contributions of this work are as follows:

(1) An **adaptive parallelism controller** allowing the engine to adapt the parallelism of computation to the variable I/O requirments of scientific applications.
(2) A **hierarchical memory manager** leveraging HPC local and remote storage devices to reduce memory pressure.
(3) An **holistic stream observer** collecting per-task and per-job software and hardware metrics to drive the decision-making.

2 BACKGROUND AND MOTIVATION

2.1 Data Streaming Engines

Data streaming engines are distributed, out-of-core systems responsible for the intake, processing, and analysis of continuous data streams. These streams are generated by various sources and fed into the engine. Within the engine, streams are processed by a series of linked operators that define a Directed Acyclic Graph (DAG) known as a job. This job dictates the data flow and encapsulates the application's logic. It is the responsibility of the engine to support the deployment and management of these jobs. Examples of such platforms include Apache Flink, Google Cloud DataFlow, Apache Storm, and Kafka Streams.

2.2 The Data Streaming Engine Model

Tasks: In data streaming engines, the smallest unit of computation is a task. The collection of all tasks comprises the compute plane. Each task executes a specific operator, such as filtering, but an operator can be instantiated into multiple tasks depending on its parallelism requirements.

State Management: While simple tasks may be stateless (e.g., filtering or mathematical operations), more complex tasks (e.g., joins) require historical data to function. Each task typically manages its state in-memory and interfaces with a key-value store API. The collective state of all tasks represents one half of the data plane.

Windowing: Windowing is a common and state-intensive operator in data streaming used to temporarily collect data within a task. The window's length may be based on time intervals, the number of events collected, or session length. Windowing is essential for operations that calculate rolling or totals over a specified period.

Inter-task Communication: Communication between tasks is facilitated through queues and per-task network buffers. These buffers accommodate variations in processing speeds from preceding tasks, allowing tasks to operate independently and in parallel. Advanced algorithms control the timing of data transmission and reception. The collection of queue buffers assigned to each task forms the other half of the data plane.

Keyed vs. Non-keyed Streams: Keyed streams organize data by a specific attribute (e.g., a file name), using the key to divide the stream into multiple logical streams. This allows stream processing systems to apply operations like windowing to all data associated with a file. While keyed streams are limited in parallelization by the number of unique keys, they often host stateful operators. Non-keyed streams, lacking specific keys, can be parallelized across an unlimited number of tasks.

2.3 Data Streaming: Challenges and Limitations

Memory Thrashing: Memory thrashing occurs when node processes exceed available memory, leading to system crashes or significant performance degradation from excessive OS paging. Some engines, like Apache Spark, do not impose memory limits, potentially causing abrupt failures upon memory exhaustion. Others, like Apache Storm, impose strict memory limits on a per-task basis and preemptively terminate tasks that exceed these thresholds, necessitating application tuning to establish optimal limits. Alternatively, some engines integrate with persistent, node-local, general-purpose key-value stores, such as RocksDB, to alleviate memory load and ensure balanced resource utilization across multiple tasks.

Back-pressure: A significant challenge in data streaming is back-pressure, which occurs when the rate of incoming data exceeds a task's processing capabilities, causing data to accumulate in the input queue. This mismatch, often resulting from varying computational complexities among sequential operators, can severely degrade job throughput and affect other jobs in the system due to resource hogging. To mitigate back-pressure, engines may

limit the flow of data using algorithms such as credit-based flow control, adjust the parallelism of affected tasks (which may require a costly restart), or shed messages that have reached their TTL.

3 DESIGN AND IMPLEMENTATION

This work presents HStream, a novel data streaming engine. HStream was designed and implemented with the following new architectural components:

(1) A holistic observer: that can track and identify the characteristics of each data stream (e.g., velocity, average size) and collect cluster-wide hardware utilization.
(2) An adaptive job manager: to adapt to the bursty I/O of scientific applications and HPC multi-tenancy, the job manager of the data streaming engine should be able to adjust the parallelism of operators during runtime.
(3) A hierarchy-aware data manager: to process the high volumes and velocity of data produced by scientific applications without overwhelming the memory, HStream must be able to leverage the entire storage hierarchy.

Overall, these three design goals aim to present an HPC-aligned data streaming engine capable of handling the unique data requirements of scientific applications.

3.1 HStream Architecture

Data streaming is a flexible programming model designed to operate on real-time data. A variety of HPC applications (e.g., simulation, IoT, AI, analytics) and storage targets (e.g., PFS, inference server) can act as either producers or consumers of data. Further, a streaming platform can provide extensive in-transit transformations on the data without the need for changing the application. This section presents the design and architecture of HStream. HStream can be used, for example, to build an online prediction pipeline, where it can process the generated data from a telescope and provide online detection of cosmic events for the applications as soon as possible. Users can define the logic and flow of the needed transformations through the submission of a job, HStream can then perform various operations on the stream of data flowing through the job all completely transparent to the application.

Figure 1 presents the high-level architecture of HStream, its major components, and showcases the data path through the platform. HStream adopts a master-worker architecture, in which multiple masters (the Job Managers) are responsible for a number of workers (the Task Managers). This architecture, and the distributed nature of each component, provides a flexible, scalable, and efficient platform for distributed processing of data streams.

3.1.1 HStream's data path. The first step towards interacting with HStream is to leverage HStream's client library. HStream's client library is composed of three core components:

The API: The HStream API allows applications to interact with the system for job submission (administration layer) and publication/subscription for data stream events (data layer).

The Job constructor and operator library: Before sending data to HStream for processing, the application first needs to create a job. The job describes the transformations that users desire to happen on the data stream. It is defined as a directed acyclic graph (DAG)

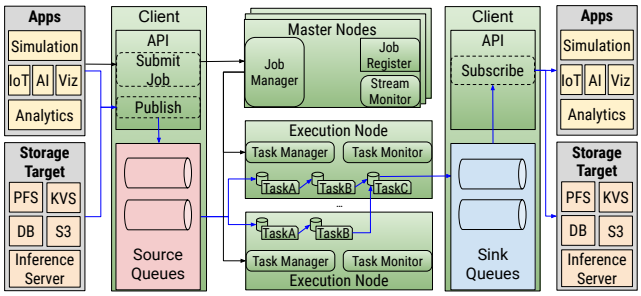

Figure 1: HStream architecture, showcasing in blue the data path from client to source queues, across the job, and into the sink queues of the clients. Black arrows show a simplified depiction of the control pathway for a job submission.

of operators, where each *operator* is the implementation of a transformation to be performed on the data stream being processed. A *task*, a concept that will be used later in this section, is an instance of an operator executed by a thread. One operator can be parallelized into multiple tasks. HStream's client library includes several pre-defined operators that can be used by any user, as well as the utilities required to define, instantiate, and compile the job into a shared library compliant with HStream's requirements.

The client queues: HStream's client provides distributed queues for the producers (source queues) and consumers (sink queues). The implementation of these queues leverages Boost interprocess communication for node-local data accesses by the application cores, and an RPC layer for remote data accesses from the HStream cluster.

When an application submits a job to the cluster, it communicates with the elected Job Manager's leader. Job Managers are the only entities deployed on HStream deployment and are launched on a set of configurable master nodes. Each *Job Manager* is designed to be mapped to a single master node. These Job Managers have many responsibilities. During job submission, they are responsible for receiving the request, storing the job metadata in the Job Register (e.g., user, path to the shared library, timestamp, job ID, required operators), and initializing the job. This initialization process is depicted by the black lines in Figure 1. For this initialization, the Job Manager links to the shared library of the job and loads the DAG into memory. Once it understands the job requirements, the Job Manager will instantiate the first operator of the job. This operator is usually referred to as the collector and is responsible for gathering data from the source queues. To do so, The Job Manager will initialize a set of Task Managers (if none exist) and instruct them to instantiate a collector operator. The collector operator task can have multiple copies spawned across different nodes, with their quantity determined based on the parallelism of the first operator defined by the user within the job (and a default from the platform configuration file). After this initialization, the application or the storage server can publish/subscribe to data stream events to/from HStream.

Living alongside each of the Job Managers on the Master nodes are two additional components: The Stream Monitor, a key component in HStream's intelligent observer, is responsible for monitoring and collecting statistical information on the data stream. This information helps estimate workloads, detect bottlenecks, and adapt task parallelism; and, the Job Register which contains several in-memory

distributed data structures for storing metadata and statistical information related to submitted jobs. These data structures store the job information, a map of the tasks assigned to the job, resource utilization of the cluster, and more. In general, all the information needed for the Job Manager to perform its adaptive scheduling of tasks is stored within these distributed data structures of the Job Register.

Once the collectors are ready to handle data for the job, the application will start publishing the data stream into the source queues. The blue lines in Figure 1 present the data streams processing path from producer to consumer. During the running phase, each collector proactively pulls stream events from the client queues for processing. This pull-based design enables the Job Manager to also affect the parallelism of the collectors without having side effects on the clients. These collectors, like any other task, executed on the Task Manager, a component of HStream running on the execution nodes that manages tasks (spawning and killing them). It is the core component establishing the separation between the data and operation planes. The Task Manager is also responsible for creating the input queue and state for any task, controlling the task's movement between memory and permanent storage devices (e.g., NVMe, HDD) during runtime, and presenting a global namespace to the data, allowing for the querying of data from different tasks. Upon reception of an event, if no tasks exist to handle it, the Task Manager will spawn a task and assign an input queue to it. Finally, the event will be placed into the corresponding input queue for processing.

Tasks within the job communicate and transfer events between queues, but rely on the Job Manager to tell them their destination. As such, every so often when a task finalizes processing an event, it will contact the Job Manager to ask for directions. If the cluster has maintained the load, no changes will be made. Otherwise, the Job Manager can reconfigure the parallelism of the operator by redirecting traffic: to new tasks (to increase) or away from tasks (to reduce). This scheduling is the final responsibility of the Job Manager. Upon reaching the final task (the sink), the task outputs the processed data stream events to the sink queues running together with the consumer. These final sink tasks can be replaced by the network queues of the nodes hosting a storage target.

Task Managers are co-located with a Task Monitor. Task Monitors, paired with the Stream Monitor, implement the observation layer of HStream. While an execution node can (but rarely will) contain more than one Task Manager (e.g., for data isolation purposes between two applications), HStream only requires a single Task Monitor per node. Its primary responsibility is tracking and collecting hardware resource utilization on a per-task basis at an adaptive time interval. All this information is stored in an in-memory data structure and is used by the Hierarchical Manager to instruct data movement between memory and storage.

3.2 Intelligent Stream Observer

For the proper execution of HStream's algorithms, it is essential to have a comprehensive understanding of the characteristics of the data stream and the current hardware utilization status. To collect them HStream makes use of a holistic stream observer to collect statistics a per-task and per-stream level. HStream avoids static monitoring pooling rates as they are inefficient and variable per system. Instead HStream leverages an adaptive and dynamic time interval strategy inspired by existing works [24]. This dynamic time interval modifies the polling rate to measure more often on highly variable resources and reduces it on more statically used resources. This dynamic interval allows monitoring a higher number of resources. The initial value is defined empirically by the user, and the system is responsible to search for a pooling rate that maintains CPU overhead below 10% of the node, adjustable by the user. The stream observer is composed of two distinct layers: *the data observation layer* and *the hardware observation layer*.

3.2.1 Data Observation Layer. The data observation layer is a sublayer of the stream observer, implemented by the *Stream Monitor* component, as illustrated in Figure 1. Each master node runs a single Stream Monitor component. The main responsibility of the Stream Monitor is to collect statistical information about jobs and data streams. This includes metadata of live jobs within the cluster, the resources used by each job, the available resources in the cluster, the data production and consumption rate of each task (defined as the number of events per second received or sent), and the load (the length and volume of the queue) of each cluster queue, including source and sink queues. These stream stats are stored in the Job Register after collection.

To help with the Job Manager's scheduling, the Job Register stores the data in two data structures: a *loadmap (unordered_map)* and a *reversed_loadmap (multimap)*. In the *loadmap*, the *taskmanager_id* is used as a key, while in the *reversed_loadmap*, the combined load of a Task Manager is used as the key and sorted in ascending order. This allows both searching for the specific load of a Task Manager and quickly finding the least loaded Task Manager. During runtime, the Job Manager uses these maps to detect bottlenecks (e.g., back-pressure) or idleness of tasks and find appropriate locations to spawn or kill tasks.

3.2.2 Hardware Observation Layer. The hardware observation layer is the other sublayer of the stream observer, implemented by the *Task Monitor* component shown in Figure 1. Each execution node runs a single Task Monitor. The Task Monitor is responsible for collecting the hardware resource utilization per node and per task, including both CPU usage and memory usage. Here, CPU and memory utilization refer to the average percentage of CPU or memory in use over a user-configured time interval. Individual task CPU metrics are obtained by naming each task thread and extracting its information from the *proc* file system. The Task Monitor has shared access to the data structures and files that hold the queues and task state, and can leverage this access to track their size and memory utilization. All these hardware statistics are stored in a hardware statistics database, which is an in-memory hashmap. All this information will be used by a Task Classifier, a component of the Hierarchical Manager, to compute a score for each task, helping the Hierarchical Memory Manager determine which tasks to move to or from storage.

3.3 Reducing Network Pressure Via Adaptive Parallelism

Data streaming jobs are statically deployed with a defined parallelism per operator, determined at startup by the user submitting the job. In multi-tenant environments, over-provisioning resources can make one application perform at maximum capacity, but may

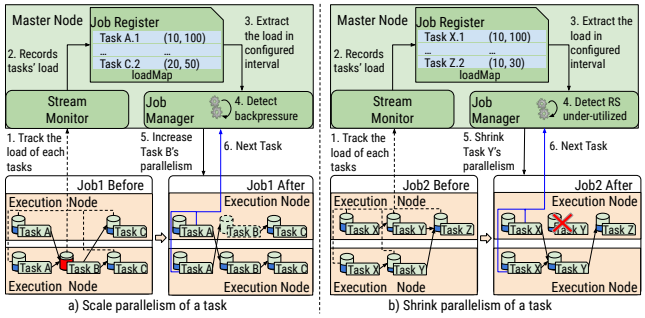

Figure 2: HStream's adaptive parallelism, showcasing on the left the detection of back-pressure on Task B and its increase in parallelism when any of the previous tasks (Task A) request the next tasks. The right side shows the reverse.

lead to reduced availability of resources for other applications. In environments where data is generated unevenly over time, such as the bursty I/O patterns of scientific applications, under-provisioning resources can cause events to clog the network queues due to a mismatch between production and consumption rates.

To solve this problem, HStream provides an adaptive parallelism controller to address both the under- and over-provisioning of resources, making it better suited for the multi-tenant and bursty I/O patterns in HPC clusters. It allows for adaptive adjustment of an operator's parallelism during runtime by scaling in and scaling out the number of tasks executing the given operator. This decision is made by considering the current load on a given task, measured by the length of its input queues and the current inflow of data into the job, metrics tracked by the Stream Monitor. The adaptive parallelism controller is implemented through two core mechanisms: a *next task scheduling* algorithm implemented in the Job Manager and *ephemeral tasks*, which are a result of the decoupling of the data (both state and queue) from the task, allowing their spawning and termination at will. Ephemeral tasks are crucial, as they allow the Job Manager to instruct new tasks to be spawned or, if a task does not receive any events for a configurable duration, it marks itself as terminated for the corresponding operator, freeing it to be selected later for a different task. These two mechanisms allow the Job Manager to dynamically adjust the parallelism of tasks by redirecting the flow of events.

HStream's adaptiveness is illustrated in Figure 2. When a task completes processing an event, it sends a message to the Job Manager with its *job_id*, *task_id*, and the *event_key* to request the location of its subsequent task. Upon receipt of a *next_task* request, the Job Manager checks the current status of the queues for the previously selected, if any, next task. If the task has no issues, the current one is asked to **maintain** the current connection. If the task is detected to be overloaded, the Job Manager will trigger an **scale out** of the operator parallelism by finding the Task Manager with minimal load on the cluster. To efficiently execute this process, we make use of the Job Register *reversed_loadmap(std::multimap)* that maintains a record of the load of each Task Manager, with the load as the key, sorted in ascending order. The Job Manager then returns to the task the corresponding *taskmanager_id*, the ID of the next operator, and the *task_list* containing all available task slots (free cores) on the Task Manager. When received, the task

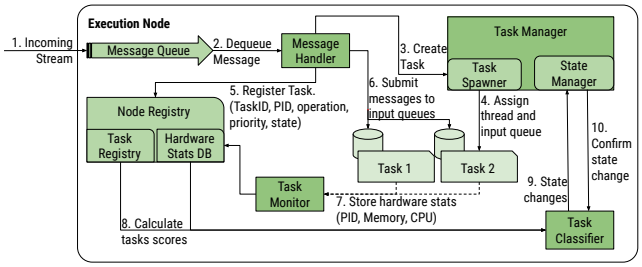

Figure 3: HStream control flow on a single execution node, depicting the control path for creating a new task (steps 1–6) and the communication required for the Hierarchical Stream Manager (steps 7–10).

establishes a connection with the new Task Manager. The spawning of a new task is depicted in Figure 3. When a message arrives containing a *task_list*, it will be sent to the Task Manager, which chooses among the available task slots in the list to place the new tasks, requesting a new state and queue to be created for it, and placing the new event on the recently created queue. Finally, if the queue load of the next task is below a threshold, the Job Manager initiates a **scaling in**. To do so, it searches for the current load of the other tasks from the same operator. If any of the tasks are detected to be underloaded, the Job Manager will redirect the current task to the found task. Note that the *next_task* call can happen every *n* events or seconds, configurable by the user, establishing a trade-off between reactiveness of the system and network utilization.

The need for the *task_list* is important to manage keyed streams where all events with the same key must be processed by the same tasks (e.g., all events belonging to a unique topic). For these cases, the *task_list* allows the Job Manager to make specific requests for where to place the tasks. Previous non-processed elements of this key can be extracted, at the cost of some increased latency, from the global namespace managed by the Hierarchical Manager. HStream's design does support the ability to migrate task state, but it lies beyond the current scope of this work. An important unique case for the adaptive control is the Collector tasks, which are the first task on any job and interface with the client queues. Under a normal push-based model, where clients send the data to the engine, it is very hard to manage their parallelism, requiring updating information on the client. Instead, HStream uses a pull-based model, where users push data to user queues that are then pulled by the collectors. This design, combined with the distributed nature of the queues, allows HStream to alter the number of collectors without any side effects on the user processes. The collectors' load is measured by the length of the user queues, monitored by the Stream Observer.

3.4 Alleviating Memory Pressure with Storage Hierarchies

Existing data streaming engines manage state as a holistic resource across the job, where the state is tied to the tasks, and they either operate in memory, memory-mapped files, or a KV store tied to the task. This decision presents a trade-off between memory utilization and performance that must be made at the job level rather than by individual tasks. To overcome this challenge, HStream presents an intelligent hierarchical memory manager that can manage, store,

and access individual task states and input queues on the storage hierarchy when memory pressure is high. However, implementing such a hierarchical memory manager requires understanding the I/O behavior and resource usage of each task (Observational Layer) and deciding when to execute data movement, as randomly moving a task state may degrade overall performance.

Algorithm 1: Calculating Task Score Algorithm

```
Data: total_CPU, total_Mem, config, Task Statistics (TS)
Result: Top Task based on score
1  Procedure CalculateTaskScore():
2      TaskManagerStats ← (total_CPU, total_Mem);
3      MaxMemTHR ← config.get(maxMemThreshold);
4      if TaskManagerStats.total_Mem > MaxMemTHR then
5          foreach ID ⊂ TR do
6              params ← TS.{CPU, Mem, input_rate, Priority,
                 Retention};
7              score ← Score(params);
8              ScoreMap.append(ID, score);
9          sortedScoreMap ← sortByScore(ScoreMap);
10         return ReturnTopTask(sortedScoreMap);
11 Function Score(PRI, IN_RATE, RT, CPU, Mem):
12     return (w1·Mem + w3·RT) / (w2·CPU^w6 + w4·IN_RATE) + w5 · PRI;
```

HStream presents an algorithm to select which task needs to be moved based on different policies. Note that these movements occur on a per-task basis (not on an operator or job basis) and can move both the task state and its input queue, if necessary. Algorithm 1 showcases how HStream grades tasks to move to storage. The *Task Classifier* is responsible for executing the algorithm. This score is based on five factors: priority, input event rate, retention, CPU usage, and memory usage. Priority and retention are extracted from job definition and are static during the runtime of the application, the rest are monitor by our observer. These parameters are normalized through a min-max normalization method. As depicted in Figure 3, the classifier is triggered by the *Task Monitor* when the memory usage of the current node exceeds the user-defined maximum memory threshold. The algorithm will then score each task running on the node, storing their ID and score as a key-value pair into a shared map. The top task is picked to be moved, and the process repeats until the memory usage of the current node is below the maximum memory threshold. Similarly, the algorithm also supports moving tasks from storage to memory by reversing the sorting order. The *Task Classifier* will notify the *State Manager* of the need to move tasks, and the information will be gathered from the shared map.

Algorithm 1 used by HStream to grade tasks can be tuned (shown on line 21) for different jobs and depending on the HPC site's applications, by adjusting the weights ($w1$ to $w6$) and changing the relative importance of each of the five factors. Through this tuning, HStream allows the implementation of different policies. We propose several policies to cover some common application types, leaving system administrators and users with a flexible system that enables them to enhance HStream with more:

3.4.1 Frequency-based policy. This policy aims to optimize memory utilization by retaining hot tasks in memory and moving cold tasks to storage. The hotness of a task is influenced by the input event rate. In this policy, the input event rate weight ($w4$) is given the highest value.

3.4.2 Computing-intensive policy. Typically, a computing-intensive task is expected to have more memory accesses, and therefore a lower score. However, computing-intensive tasks, such as AI training or inference, may have high CPU utilization but fewer memory accesses compared to other tasks. For these use cases, one can set a high value for the CPU usage weight ($w2$) and set its exponent ($w6$) to negative one.

3.4.3 Retention-based policy. This policy targets tasks with long retention, such as long-term windowing tasks. It allows finer control of how to manage the relationship between input rates and retention lengths. For example, a daily window that averages many events over a day compared to a minute window with fewer events. The default policy will put the daily window on disk, which might not be feasible due to its high input rate. Instead, in this policy, the retention weight ($w3$) should be set to a low value, while the input rate weight ($w4$) can be set to a higher value.

4 DESIGN CONSIDERATION

The adaptive parallelism in HStream has limitations. For key streams, the maximum parallelism of a task depends on the number of unique keys used. For non-key streams, the maximum parallelism of a task is limited by the maximum available hardware resources (usually cores). When hitting these adaptive limitations, the system cannot adjust the degree of parallelism as desired. In this situation, we expect an increase the back-pressure. In these situations, the hierarchical manager can reduce this pressure by placing cold, large-sized, or low-priority tasks into storage.

The current implementation of HStream leverages high-speed local burst buffers. However, these clusters often present a much deeper hierarchy in the form of remote SSD-backed Burst Buffers and remote PFS. The expected latency of these storage systems for state placement is too high to be directly used in data streaming engines. Yet, the global namespace enabled by the Task Manager can allow for a much deeper state storage pool for the tasks, opening avenues for new functionality, such as shared state between tasks or the migration of tasks between nodes, as the state would remain accessible. We aim to explore this mechanism in future work. Additionally, HStream uses a lazy deserialization method to handle the events, events are stored in the queues and task states in a serialized format and are only deserialized when used. This allows movements of the task state between the hierarchy layers more performant.

Extensive work exists in the literature and on production systems to support exactly-once semantics and fault tolerance in streaming engines. Currently fault tolerance lies beyond the scope of the paper, but the design of HStream is compatible with mechanism like watermarking and checkpoints. In fact, we expect that the ability of the Hierarchical Manager to interface with remote storage might present interesting opportunities to optimize these processes.

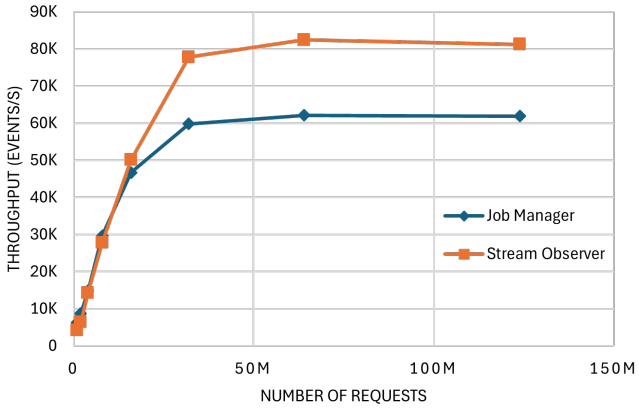

Figure 4: Scalability of Job Manager and Stream Monitor

5 EVALUATIONS

5.1 Methodology

Hardware: All experiments were conducted on our local research cluster. The cluster consists of a compute rack with 32 nodes, connected via two isolated Ethernet networks (40 Gb/s and 10 Gb/s) with RoCE enabled. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114, 48 GB RAM, a Samsung 250GB NVMe SSD, and a Seagate 1TB SATA HDD.

Software: The implementation of *HStream* is written in C++ with over 9,000 lines of code, publicly available on GitHub[1]. The OS is Ubuntu 22.04. For the evaluations, we used HCL 0.9.3 [9], a high-performance library that provides distributed data structures over RPCs, used for all metadata structures and for the source and sink queues. HStream uses Mochi Thallium [26] for RPC communication, supporting libfabric and verbs. The evaluations compare HStream against the Neon streaming engine, a state-of-the-art HPC streaming engine that shows significant performance improvements over cloud engines that rely on Java and the TCP stack.

5.2 Evaluations

This section aims to answer several questions. Firstly, it examines HStream's scalability by measuring the response rate of the Job Manager and Stream Collector (responsible for the adaptive parallelism of HStream). In addition, it measures the overhead of the task collector. Secondly, it explores how an adaptive streaming engine boosts performance in a multi-tenant and bursty environment by deploying three I/O kernels simultaneously, each with varying I/O generation on three identical streaming jobs. Thirdly, it demonstrates how a hierarchical streaming engine improves performance when handling high data volumes. This is shown through a ChronoLog-inspired [14] streaming job that collects data for different topics and buffers the data over time before sorting and writing to disk. Lastly, it investigates how the combination of these technologies can enhance holistic systems, using an AI inference workload to demonstrate how adaptable and hierarchical engines can reduce latency and improve overall system performance.

[1] https://github.com/scs-lab/HStream

5.3 HStream Scalability

This first set of evaluations consists of three experiments. For the first two, we evaluate the scalability of the Job Manager. Both experiments are executed using a single Job Manager process with 4 RPC threads. In the first experiment, the clients behave as tasks, continuously requesting new task allocations from the Job Manager. In the second experiment, the clients act as task managers, updating the Stream Monitor with synthetic loads. Each experiment deploys 8 processes per node across 31 nodes, with the 32nd node reserved for the Job Manager/Stream Monitor. Each process generates 500k request for a total of 124M at the largest scale.

Figures 4 show the results, revealing similar behaviors with slow performance increases as requests grow, peaking at 60K events per second for the Job Manager and 80K events per second for the Stream Monitor. Performance was not affected by scaling the RPC server threads from 4 to 16, with 16 threads even reducing performance due to shared resource locking. When collocating the clients with the Job Manager and Stream Monitor on the same node, they achieved 800k and 2.5M requests. This helps to conclude that the algorithms for adaptive management are not the bottleneck of the system and are limited by the networking stack on the node. Note that these experiments simulate maximum load, where clients are continuously sending requests, unlike real deployments, where requests are spaced out. Subsequent evaluations will the same configuration, with the 40Gb/s network and 4 RPC threads for both services.

# Tasks	CPU Use (%)
1	0.4
2	1.2
4	2.2
8	4.3
16	8.4
32	18.3

Table 1: Overhead of the Task Monitor

The final experiment, shown in Table 1, measures the CPU utilization of the Task Monitor. A super observer, identical to the Task Monitor, oversees the task monitor as we deploy between 1 to 16 tasks on the node. CPU utilization correlates with the number of monitored processes, with an average observational interval of 2 seconds, adjustable to balance reactiveness and CPU usage. We choose 2 seconds empirically as a good trade-off between reactiveness and overhead. On a 20-core machine, we observe a 4-8% CPU utilization with 8-16 tasks, with the utilization jumping to 16% when we started oversubscribing the node.

5.4 Improve performance through adaptive data streaming

This evaluation showcases HStream's improvements in a multi-tenant environment running multiple application kernels performing bursty I/O. The kernels include VPIC, HACC, and K-Means, chosen for their diverse I/O behaviors: VPIC is a particle simulation that generates significant I/O (32 MB per rank), HACC is a simulation that executes checkpoints and provides a balanced compute-to-I/O ratio (8 MB per rank), and K-Means is a clustering algorithm that stores most of its data in memory and is compute intensive (2 MB per rank). The messages generated by all 3 applications are small, less than 1kB. All three applications run I/O phases between compute phases of varying lengths.

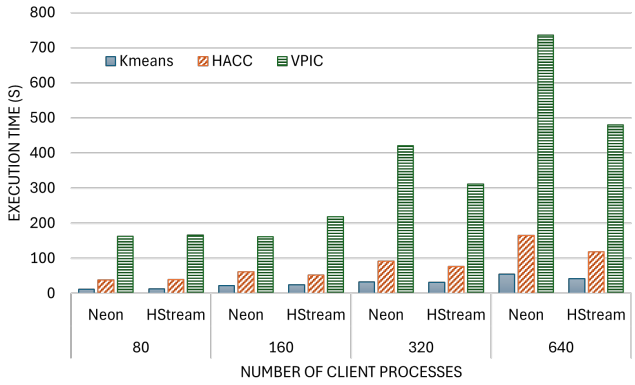

Figure 5: Execution time of workloads based on the number of clients.

Figure 6: Acquisition and delivery time of a stateful job.

The evaluation job consists of a collector that pulls data from the applications, a key-by task that transforms the streams into keyed streams, and partitions them into windowing tasks. The windowing tasks collect up to 10 events, sort them, and send them to sink tasks that write them to disk. The key is set as the filename, and we use 4 filenames per application to ensure parallelism and prevent bottlenecks from the computing tasks. Thus, the only adaptable tasks are the data collectors, which remain unchanged for Neon but can be modified during runtime by HStream based on the applications' demands. Both engines are initially deployed across 16 nodes, with a single collector task limit on each node. Each application receives 5 collectors, with VPIC receiving an additional one. The 3 applications are then deployed weakly scaling from 80 to 640 nodes, with the aim of an even split of processes between the applications, forcing the ranks of different applications to coexist on the same node.

In Figure 5, it can be observed that HStream and Neon perform similarly on a low scale, as HStream has limited capabilities to adapt to I/O. In fact, a small performance loss is observed due to the network overhead HStream suffers from the communication between the Job Manager and Stream Monitor. As the scale increases, a greater disparity in the I/O generated by the applications becomes apparent. At maximum scale, VPIC generates 10 GB, while K-Means generates only 0.5 GB. At this same scale, Neon maintains the same 5 collectors per application, while HStream's adaptive task management allows it to shift to an 11:3:1 ratio of collectors for VPIC, HACC, and K-Means, respectively. This shift in collector pull rate towards high I / O intake jobs helps to improve performance by up to 1.5× on VPIC, while showing little performance change on KMeans.

5.5 Improved latency through the hierarchy

To showcase HStream's hierarchical capabilities, a workload based on the ChronoLog paper [14] is used. ChronoLog proposes a hierarchical log-store using event time as the primary key for log-order. As a hierarchical log, ChronoLog initially submits all events to a fast local NVMe-backed memory journal, from which the data are continuously streamed out and formalized in the lower layers of the hierarchy. To implement this part of the system, ChronoLog proposes the use of a data streaming engine.

The evaluation job maintains the same four steps as the previous evaluation: collection, key-by, aggregation, and sinking to file. For this evaluation, we make greater use of the task state by making the aggregation task a 5-minute tumbling window and using a single unique key. Each processes generates 40 messages of a medium size of 4MB, we can ensure nearly full memory utilization when using 160 processes, and surpassing it in subsequent tests. A 5-minute window is unnecessary, but it ensures all data land in the window regardless of startup delays or network lag spikes, a window of 30 seconds would have been generally enough having little effect on the results when tested. To avoid any memory interactions, 8 compute nodes are allocated to the streaming engines, while 24 are left available for the clients. We maintain the 20 client processes per node. HStream's Hierarchical Manager has access to both memory and the node-local burst buffers supported by NVMe SSDs.

Because of the delaying effects of the windowing, measuring pure latency is complicated, instead the experiment introduces two new metrics: *acquisition time*, described as the average of time between event collection and inclusion on the task state; and *delivery time*, described as the average time between the window closing time and issuing their write to disk. We refer to the sum of both of those metrics as the *adjusted latency*, as it removes the otherwise dominant window time, even on a smaller number, and due to the uneven arrival of events through the window, subtracting its length from the latency measure did not result in correct values.

For the results, in terms of acquisition time, both HStream and Neon behave similarly at low scales when maintaining the state in memory. As HStream starts to transition state into NVMe (160 processes), the collision of the movement operations results in an overall increase in latency for the events. As data volume further increases, our hierarchical management algorithm ensures that events can always be accepted into memory, thus outperforming OS-managed memory. For delivery time, results improve from the outset, showcasing the benefits of lazy serialization, allowing for faster message sending at window closing. As the streaming engines start using swap space or the hierarchy, the gap closes as the RDMA operations can be performed faster when data is in memory.

5.6 A complete case study: An AI inference platform

The final evaluation tests an AI inference service. The model used is ResNet-50, whose weights are publicly available. The job

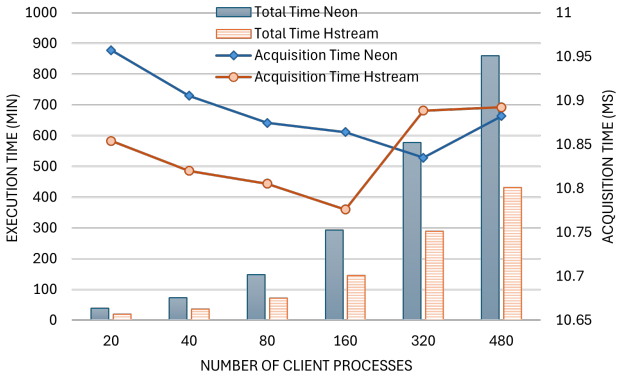

Figure 7: Acquisition and run time of an AI inference service.

for this evaluation consists of a collector that pulls images from the client queues, a map that transforms the high-resolution images to the 224x224 images required by ResNet, a map that runs the ResNet model, and a file sink to a client memory queue with the result.

The evaluation aims to showcase a workload that leverages both the adaptive and hierarchical nature of HStream. The job for this workload is non-keyed, giving HStream more opportunities to increase and decrease the parallelism of both the collectors and inference operators. For the hierarchy, the disparity of CPU complexity between the inference and transformation operators will cause the communication queues of the operators to fill up. For Neon, this will result in back-propagation, while HStream will manage the input queue to the inference tasks hierarchically.

To set up the experiment, the ResNet-50 model weights are stored in a file accessible to all nodes on the cluster. The C++ TensorFlow library is used for inference by the tasks. As the testing cluster lacks GPUs, version 2.11.0 of the CPU-optimized library for x86_64 architecture is employed. HStream includes a series of utility classes to handle both models and tensors within C++ for use inside any of the operators. Similar to the previous evaluation, 8 compute nodes are allocated for the streaming engines, and 24 to the clients. Each client node executes 20 processes. Each client reads a subsample of the ImageNet dataset and places the images on the queue. Each client sends 1,000 images of 200 MB per image, a large size for a streaming application. Both Neon and HStream begin with an initial parallelism of 8 on the collectors and 4 on the other operators. We measure the overall execution time and continue to use acquisition time, defined as the time it takes for an event to reach the inference task queue, as measuring overall latency yields incorrect results due to the dominant effects of wait time on the queue.

In terms of results, while acquisition time shows that HStream slightly outperforms Neon at low scales, at 16 processes, we start to see the effects of the queues reaching close to full memory utilization, at which point HStream starts to make use of the hierarchy with an increase in acquisition time compared to Neon. However, as an inflection point in memory capacity is reached, HStream's state management maintains its performance and catches up to the increased latency of Neon's OS-managed memory. In terms of overall execution time, the adaptive parallelism of HStream makes the biggest difference in terms of throughput. HStream's adaptiveness allows it to increase the parallelism of the inference tasks at runtime, showcasing a performance improvement of almost 2× compared to Neon at the highest scale.

6 RELATED WORK

6.1 Streaming as a Transfer Model in HPC

The increasing volume and velocity of data in HPC have driven the need for more efficient data movement strategies than reliance on PFS as a shared storage system. Streaming emerges as a memory-to-memory connection directly between the user spaces of two applications or systems. Adios2 [11] introduced the SST engine, which leverages memory queues and incorporates HPC technologies such as RDMA. SciStream [7], based on the Globus infrastructure, offers a streaming-based transfer system connecting scientific instruments and compute clusters across facilities. Cloud-based systems have also been used in HPC. MQTT message queues are used by Beneventi et al. [2] to enable model transfers in continuous learning. STREAM [1] presents a Kafka-based system for telemetry collection for the Frontier supercomputer.

6.2 Streaming as a Transformation Model

Attempts to bring data streaming to HPC have focus on 2 strategies:

Some have focused on the direct deployment of pre-existing cloud systems, either by facilitating the processes [6] or by altering engine to be able to use HPC hardware, such as Infiniband and Omni-Path [12]. Despite this, cloud engines, even those adapted to use HPC hardware, suffer from performance bottlenecks due to their reliance on cumbersome software stacks, predominantly based on Java runtime [16]. Another avenue has seen the proposals for the co-location of Cloud and HPC clusters [10]. However, these approaches demand a significant financial investment in hardware resources.

A second approach has seen the development of new data streaming engines directly for HPC with HPC software at their core. A first wave of this engines, utilized MPI as their communication framework [19, 22]. Depending on MPI comes with drawbacks, as MPI operates under a very performant but rigid network model where all ranks must be known at start up and any failure on a task leads to the entire engine failing [18]. Recent work by Matri et al. introduced Neon [20]. Neon utilizes a full HPC stack built on C++, and supporting Infiniband, RDMA, and intra-node shared memory communication. Neon maintains the core architecture of other cloud engines and demonstrates significant performance improvements over them. While Neon represents a significant step forward, HStream showcases how changes to the existing architectures can be made to better address the demands of scientific applications.

7 CONCLUSION

This work introduced HStream, a new architecture for data streamign in HPC. HStream aims to solve performance issues present in state-of-the-art streaming when subjected to the high volumes, velocities and burstiness of I/O generated by modern scientific applications. Its core innovation lies in the separation of the data and compute planes allowing finer control of the data through the system. This is achieved through an adaptive parallelism controller, adapting compute parallelism to adapt to the bursty and multi-tenant environments, and a hierarchical data management system, which leverages high speed non-volatile storage systems present in HPC

clusters to alleviate memory pressure and avoid thrashing under the high data loads of HPC. By leveraging HStream's adaptiveness, we show up to a 1.5x decrease in the overall cluster-wide execution time when serving under a multi-tenant deployment of applications. Similarly, HStream's hierarchical management of task state provides up to a 75% reduction in latency under high volume of data by alleviating memory thrashing on a per-task basis. Lastly, both mechanisms allow HStream to present up to a 2× increase in overall throughput when serving as a holistic AI inference service when compared to state-of-the-art HPC data streaming engines.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF), Office of Advanced Cyberinfrastructure, under Grants CSSI-2104013 and Core-2313154. Additionally, this work is partially supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contracts DE-SC0023263 and DE-SC0024593. We would like to thank the Chameleon testbed, supported by the NSF, for providing an environment for development and debugging.

REFERENCES

[1] Ryan Adamson, Tim Osborne, Corwin Lester, and Rachel Palumbo. 2023. STREAM: A Scalable Federated HPC Telemetry Platform. (5 2023). https://www.osti.gov/biblio/1995656

[2] Francesco Beneventi, Andrea Bartolini, Carlo Cavazzoni, and Luca Benini. 2017. Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools. In *Proceedings of the Conference on Design, Automation & Test in Europe* (Lausanne, Switzerland) *(DATE '17)*. European Design and Automation Association, Leuven, BEL, 1038–1043.

[3] Francesco Beneventi, Andrea Bartolini, Carlo Cavazzoni, and Luca Benini. 2017. Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 1038–1043. https://doi.org/10.23919/DATE.2017.7927143

[4] P. Chris Broekema, Rob V. van Nieuwpoort, and Henri E. Bal. 2012. ExaScale High Performance Computing in the Square Kilometer Array. In *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date* (Delft, The Netherlands) *(Astro-HPC '12)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/2286976.2286982

[5] Jaime Cernuda, Hariharan Devarajan, Luke Logan, Keith Bateman, Neeraj Rajesh, Jie Ye, Anthony Kougkas, and Xian-He Sun. 2021. HFlow: A Dynamic and Elastic Multi-Layered I/O Forwarder. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 114–124. https://doi.org/10.1109/Cluster48925.2021.00064

[6] Georgios Chantzialexiou, Andre Luckow, and Shantenu Jha. 2018. Pilot-Streaming: A Stream Processing Framework for High-Performance Computing. In *2018 IEEE 14th International Conference on e-Science (e-Science)*. 177–188. https://doi.org/10.1109/eScience.2018.00033

[7] Joaquin Chung, Wojciech Zacherek, AJ Wisniewski, Zhengchun Liu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. 2022. SciStream: Architecture and Toolkit for Data Streaming between Federated Science Instruments. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 185–198. https://doi.org/10.1145/3502181.3531475

[8] Peter J. Denning. 1968. Thrashing: Its Causes and Prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (San Francisco, California) *(AFIPS '68 (Fall, part I))*. Association for Computing Machinery, New York, NY, USA, 915–922. https://doi.org/10.1145/1476589.1476705

[9] Hariharan Devarajan, Anthony Kougkas, Keith Bateman, and Xian-He Sun. 2020. Hcl: Distributing parallel data structures in extreme scales. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 248–258.

[10] Nicolas Dube, Duncan Roweth, Paolo Faraboschi, and Dejan Milojicic. 2021. Future of HPC: The Internet of Workflows. *IEEE Internet Computing* 25, 5 (2021), 26–34. https://doi.org/10.1109/MIC.2021.3103236

[11] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561. https://doi.org/10.1016/j.softx.2020.100561

[12] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox. 2017. Low Latency Stream Processing: Apache Heron with Infiniband & Intel Omni-Path. In *Proceedings of The10th International Conference on Utility and Cloud Computing* (Austin, Texas, USA) *(UCC '17)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/3147213.3147232

[13] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S. Vazhkudai, and Misbah Mubarak. 2019. Evaluating Burst Buffer Placement in HPC Systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. https://doi.org/10.1109/CLUSTER.2019.8891051

[14] Anthony Kougkas, Hariharan Devarajan, Keith Bateman, Jaime Cernuda, Neeraj Rajesh, and Xian-He Sun. 2021. ChronoLog: A Distributed Shared Tiered Log Store with Time-based Data Ordering. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST 2020)*.

[15] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 219–230. https://doi.org/10.1145/3208040.3208059

[16] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. 2022. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 835–852. https://www.usenix.org/conference/atc22/presentation/lion

[17] GK Lockwood, D Hazen, Q Koziol, RS Canon, K Antypas, and et al. Balewski, J. 2017. *Storage 2020: A Vision for the Future of HPC Storage*. Technical Report LBNL-2001072. Lawrence Berkeley National Laboratory. Retrieved from https://escholarship.org/uc/item/744479dp.

[18] Nuria Losada, Patricia González, María J. Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems* 106 (2020), 467–481. https://doi.org/10.1016/j.future.2020.01.026

[19] Emilio P. Mancini, Gregory Marsh, and Dhabaleswar K. Panda. 2010. An MPI-Stream Hybrid Programming Model for Computational Clusters. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 323–330. https://doi.org/10.1109/CCGRID.2010.33

[20] Pierre Matri and Robert Ross. 2021. Neon: Low-Latency Streaming Pipelines for HPC. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 698–707. https://doi.org/10.1109/CLOUD53861.2021.00089

[21] Marta Mattoso, Jonas Dias, Kary A.C.S. Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vítor Silva, and Daniel de Oliveira. 2015. Dynamic steering of HPC scientific workflows: A survey. *Future Generation Computer Systems* 46 (2015), 100–113. https://doi.org/10.1016/j.future.2014.11.017

[22] Ivy Bo Peng, Stefano Markidis, Erwin Laure, Daniel Holmes, and Mark Bull. 2015. A data streaming model in MPI. In *Proceedings of the 3rd Workshop on Exascale MPI*. 1–10.

[23] Andreas Peters and Lukasz Janyst. 2011. Exabyte Scale Storage at CERN. *Journal of Physics: Conference Series* 331 (12 2011). https://doi.org/10.1088/1742-6596/331/5/052015

[24] Neeraj Rajesh, Hariharan Devarajan, Jaime Cernuda Garcia, Keith Bateman, Luke Logan, Jie Ye, Anthony Kougkas, and Xian-He Sun. 2021. Apollo: An ML-Assisted Real-Time Storage Resource Observer. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) *(HPDC '21)*. Association for Computing Machinery, New York, NY, USA, 147–159. https://doi.org/10.1145/3431379.3460640

[25] Kenneth J Roche. 2022. *Introduction to HPC IO*. Technical Report. Pacific Northwest National Laboratory.

[26] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Gregory R. Ganger, Garth A. Gibson, Samuel Keith Gutierrez, Robert Latham, Robert W. Robey, Dana Robinson, Bradley W. Settlemyer, Galen M. Shipman, Shane Snyder, Jérome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Computer Science and Technology* 35 (2020), 121–144.

[27] Galen M. Shipman, Jered Dominguez-Trujillo, Kevin Sheridan, and Sriram Swaminarayan. 2022. Assessing the Memory Wall in Complex Codes. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 30–35. https://doi.org/10.1109/MCHPC56545.2022.00009

[28] Xiaodong Yu, Viktor Nikitin, Daniel J. Ching, Selin Aslan, Doğa Gürsoy, and Tekin Biçer. 2022. Scalable and accurate multi-GPU-based image reconstruction of large-scale ptychography data. *Scientific Reports* 12, 1 (29 Mar 2022), 5334. https://doi.org/10.1038/s41598-022-09430-3