Adaptive Fragment-Based Parallel State Recovery for Stream Processing Systems

Hailu Xu, Pinchao Liu, Sarker Tanzir Ahmed, Dilma Da Silva, and Liting Hu

***Abstract*—Today, large-scale cloud organizations are deploying datacenters and "edge" clusters globally to provide low-latency access to services. Running stream applications across geo-distributed sites are emerging as a daily requirement. However, existing efforts have dominantly centered around *stateless stream processing*, leaving another urgent trend-*stateful stream processing*-much less explored. A driving need is to store and update states during processing, and most importantly, successfully recover large distributed states when faults and failures happen. Existing studies exhibit major limitations including: (1) they mostly inherit MapReduce's "single master/many workers" architecture, where the central master can easily become ascalability bottleneck; (2) they offer state recovery mainly through three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery, which are either slow, resource-expensive or failing to handle multiple failures; and (3) they are not adaptive to heterogeneous hardware settings. We present A-FP4S, a novel adaptive fragments-based parallel state recovery mechanism for stream processing systems. A-FP4S organizes stream operators into a distributed hash table based peer-to-peer overlay and divides each node's local state into many fragments. These fragments are periodically stored in node's multiple neighbors, ensuring different sets of available fragments can reconstruct failed states in parallel. This mechanism is extremely scalable to the lost state, significantly reduces failure recovery time, and can tolerate multiple node failures. A-FP4S is adaptive to heterogeneous hardware settings by automatic parameter tuning over phases. Compared to Apache Storm, A-FP4S achieves 31.8% to 50.5% reduction in recovery latency. Large-scale experiments using real-world datasets demonstrate A-FP4S's attractive scalability and adaptivity properties.**

***Index Terms*—Distributed hash tables, state recovery, stream processing.**

Manuscript received 6 April 2021; revised 8 July 2022; accepted 24 February 2023. Date of publication 21 March 2023; date of current version 6 July 2023. This work was supported by the National Science Foundation under Grants NSF-SPX-1919126, NSF-SPX-1919181, NSF-CAREER-1943071, NSF-CCF-1934904, and NSF-OAC-2212256. A preliminary version of this paper appeared at the 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS' 20)[55]. Recommended for acceptance by X. Sun. *(Corresponding author: Liting Hu.)*

Hailu Xu is with the Department of Computer Engineering and Computer Science, California State University, Long Beach, CA 90840 USA (e-mail: hailu.xu@csulb.edu).

Pinchao Liu is with the School of Computing and Information Science, Florida International University, Miami, FL 33199 USA (e-mail: pliu002@fiu.edu).

Sarker Tanzir Ahmed and Dilma Da Silva are with the Texas A&M University, College Station, TX 77843 USA (e-mail: tanzir@tamu.edu; dilma@cse.tamu.edu).

Liting Hu is with the Department of Computer Science and Engineering, University of California, Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: liting@ucsc.edu).

Digital Object Identifier 10.1109/TPDS.2023.3251997

I. Introduction

TODAY, many computing applications that are critical to society are undergoing a profound transformation with the use of large-scale, diverse, and distributed data sets that allow for data-intensive analytics and decision-making at a level never before imagined. Stream processing is proposed and popularized as a "technology like Hadoop but can give you up-to-date results faster", which lets users query a continuous data stream and quickly get results within very short periods from receiving the data. For that reason, stream processing technology has become a critical building block of many applications, such as making business decisions from marketing streams, identifying spam campaigns from social network streams, predicting tornados and storms from radar streams, and analyzing genomes in different labs and countries to track the sources of a potential epidemic.

What is stateful stream processing? Over the last decade, a bloom of stream processing systems has been developed including Storm [8], Trident [10], Spark Streaming [24], Borealis [29], TimeStream [66], S4 [61], etc. Stream processing systems typically organize the distributed processing operators in the form of a directed acyclic graph (DAG), process the tuples of data streams instantly as they flow through the DAG, and execute the application logic to produce results in real-time. Traditionally, stream processing systems are *stateless*. An interesting trend is that more complex streaming pipelines generally need to keep some sort of task or operator state in order to execute the application logic, called *stateful* stream processing [25].

Examples of the state include (depending on the application): user profiles, email digests, aggregate counts, summary of the received elements, etc. State computations include aggregations over a window (a mini-batch), counts over a window, and joining a stream with a database. Fig. 1 illustrates a simple example of stateful stream processing that might be seen in the backend of a consumer website. Suppose the application is to count the number of page views for each user per hour. The input stream is partitioned by some key in the data and distributed over multiple task instances for parallel computation, each of which is responsible for some key range. In this case, the state is the stored key-value pairs consisting of user IDs and the corresponding counters. The state can be kept in memory (e.g. a hashtable in the task instances), at disk (e.g., using RocksDB [22]), or in a remote database management system shared among applications. When a new data event is processed, the history counter value from the stored state is retrieved and incremented by the new value (e.g.,$\langle x, 1\rangle + \langle x, 10\rangle \rightarrow \langle x, 11\rangle$). The aggregation is typically

1045-9219 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

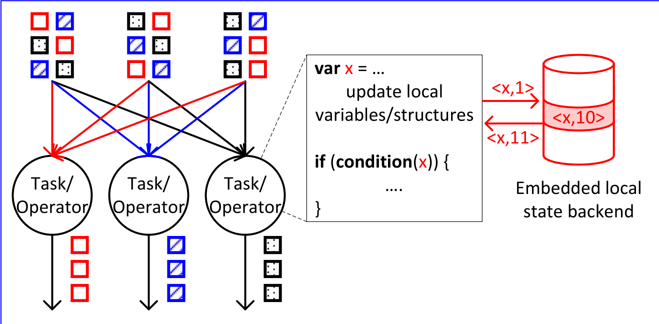


Fig. 1. Example of a stateful stream application.

limited to a time window (e.g. 1 minute, 1 h, 1 d) so that we can observe changes of activity over time. This kind of windowed processing is common for ranking and relevance, detecting "trending topics", as well as real-time reporting and monitoring

Why does state recovery matter? Stream computations are by nature long-running. They run in a highly dynamic distributed environment. Many stream operators may leave or fail at the same time, resulting in expensive time and space costs to recover them. When stream operators are deployed on a large number of nodes in the cloud, research shows that a non-negligible percentage (0.5%-1%) of computation nodes cannot come back to life after a power outage [76]. The failed operators take a long time to recover; for example, in August 2019, a failure in Amazon's AWS Tokyo region impaired the operations of many users for around 6 hours [26]. Stream operator failures can be easily caused by node replacements, service changes, device locations (i.e., in different geo-distributed datacenters), environment changes such as temperature, faulty interconnections, and human mistakes [78].

What makes it particularly challenging for stateful stream processing is that the state can grow extremely large depending on the nature of the task, the length of the sliding window(window means mini-batch of stream), or simply due to the input volume. The state of many real-world stream applications can easily expand to the order of hundreds of gigabytes [77]. For example, LinkedIn's stream applications create a large amount of distributed states at runtime, including user profiles, email digests, and aggregate counts [62].

This paper focuses on scalable and adaptive state recovery for modern stateful stream processing systems. It addresses the significant challenges in handling many simultaneous failures for a large number of concurrently running stream applications.

The first challenge is "*how to scale recovery with the state size, the number of simultaneous failures, and the number of concurrently running stream applications on a shared platform?*" Existing studies [2], [5], [6], [8], [10], [24], [32], [53], [54], [61] mostly inherit MapReduce's "single master/many workers" architecture, where the central master is responsible for all scheduling activities. As such, they are limited to a fixed computation model, e.g., asynchronous stream processing like Storm [8], synchronous mini-batch processing like Spark [7], etc. Note that the recovery operation is a critical consumer of time and space. It must quickly recover all failed operators' lost states on failover nodes (if any) without blocking the normal processing of stream applications. However, example shows that the failure recovery time takes nearly 60% of the running time when failures happen [82]. Experimental results in Apache Flink show that even with a very small failure rate, such as the failure rate of 0.0022 per hour, the overhead of state recovery dramatically increases to 68.8% for 1000 nodes and 226.83% for 2000 nodes [49]. As a result, it is difficult (or even unfeasible) for a centralized master to manage state recovery for a large number of concurrently running applications due to the inherent centralization bottlenecks.

The second challenge is "*how can we handle many simultaneous failures while achieving fast recovery with low hardware cost?*" State-of-the-art stream processing systems offer failure recovery mainly through three approaches: replication recovery [34], [71], checkpointing recovery [8], [10], [66] and DStream-based lineage recovery [35], [72], [81]. These solutions are either slow, resource-expensive or fail to handle many simultaneous failures. Replication recovery adds significant hardware cost because multiple copies must concurrently run on distinct nodes for failover. Checkpointing recovery is known to be prohibitively expensive, and users in many domains disable it as a result [33], [46], [59], [64], [65]. DStream-based lineage recovery is slow when the lineage graph is long and falls short in handling multiple simultaneously failures.

The third challenge is "*how to recover state adaptively based on runtime properties, but without manual interventions?*" Most of the existing studies provide fixed recovery mechanisms [1], [8], [10], [35], [38], [44]. Although applications have various Quality-of-Service (QoS) requirements, they assume unchanging state size and recovery paths. Existing mechanisms may result in slow recovery when the state size becomes extremely large, there are resource limitations, or network conditions vary dynamically.

We present A-FP4S, a novel adaptive fragment-based parallel state recovery mechanism to address the challenges listed above: to efficiently handle many simultaneous failures for a large number of concurrently running stream applications in a **fast**, **scalable**, and **adaptive** manner.

A-FP4S operates as follows: (1) we first organize all the application's operators into a distributed hash table (DHT) based consistent ring [69] to provide each operator with a unique set of neighbors; (2) afterward, we divide each operator's in-memory state into many fragments using erasure codes [67]. Erasure codes operate by converting a data object into a larger set of code blocks such that any qualified available subset of the generated code blocks can be used to reconstruct the original data object; (3) we periodically store each node's state in its neighbors, ensuring that different sets of available fragments can be used to reconstruct failed state in parallel, and (4) finally, we provide adaptive recovery mechanism by adjusting the size and number of fragments based on the hardware properties (e.g., network bandwidth, disk speed), the application characteristics (e.g., state size, number of nodes), and the directed acyclic graph (DAG) length characteristics. This failure recovery mechanism

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

TABLE I
OVERVIEW OF STATE MANAGEMENT AND RECOVERY IN STREAM PROCESSING SYSTEMS

System	State management	State storage			State recovery policy			State recovery traits		Processing traits
		In memory	Local	Remote	Static	Dynamic	Adaptive	Latency	Cost	
Mupppet [54]	Checkpointing	✓			✓			Slow		Stream
Trident [10]	Checkpointing	✓			✓			Slow		Micro-batch
Millwheel [31]	Checkpointing	✓		✓				Slow		Event-based
Dataflow [32]	Checkpointing			✓	✓			Slow		Hybrid stream
Samza [6]	Checkpointing		✓		✓			Slow		Stream & batch
Flux [71]	Replication	✓	✓		✓			Fast	High	Stream
Borealis [34]	Replication	✓	✓		✓			Fast	High	Stream
TimeStream [66]	Checkpointing	✓	✓		✓			Slow		Stream
Drizzle [75]	Checkpointing	✓			✓			Slow		Micro-batch
Spark [80]	DStream	✓			✓			Slow		Micro-batch
Flink [1]	Checkpointing	✓			✓			Slow		Event-based
Storm [8]	Checkpointing	✓			✓			Slow		Stream
SR3 [79]	Parallel fragments	✓	✓			✓		Fast	Low	Hybrid stream
A-FP4S	Parallel fragments	✓	✓			✓	✓	Fast	Low	Hybrid stream

is extremely scalable to the size of the lost state, significantly reduces the failure recovery time, and can tolerate many simultaneous operator failures.

We build A-FP4S on top of Apache Storm and evaluate it using large-scale experiments with real-world datasets. Experimental results demonstrate the scalability, adaptivity, and fast failure recovery of A-FP4S. When compared to a state-of-the-art solution (Apache Storm [8]), A-FP4S reduces in 37.8% the state recovery latency and reduces more than half of the hardware costs. It can scale to many simultaneous failures and successfully recover the states when up to 66.6% of nodes fail or get lost.

Contributions: We make the following technical contributions:

- We propose a decentralized architecture using a DHT-based consistent ring and erasure codes to recover the distributed states for numerous concurrently running stream applications. To the best of our knowledge, A-FP4S is the first work to use a fully decentralized architecture for state recovery (Section III).
- We implement the A-FP4S prototype on the state-of-the-art stream processing system Storm and demonstrate its portability to many other stream processing systems (Section III).
- We provide a theoretical analysis of A-FP4S's adaptive recovery mechanism by adjusting the size and number of fragments (Section IV).
- We make a comprehensive evaluation of the scalability, recovery time, and adaptivity of A-FP4S on a large cluster using real-world stream application's datasets (Section V).

The remainder of this paper is organized as follows. Section II discusses the related work. Section III describes the A-FP4S design and implementation. Section IV presents the A-FP4S adaptivity analysis. Sections V shows the experimental setup and performance evaluation. We conclude with directions for future work in Section VI.

II. RELATED WORK

Designing a state recovery mechanism for stateful stream processing systems is non-trivial, and existing failure recovery techniques for stream processing do not achieve the necessary scalability and adaptivity. In this section, we summarize existing stateful stream processing systems (see Table I) and examine why their failure recovery techniques are either slow, resource-expensive or fail to handle multiple failures.

A. Stateful Stream Processing Systems

Many industrial stream processing systems either do not support state (Heron [53], S4 [61], early version of Storm [8]), or rely on in-memory data structures such as hash tables and hash table variants to store state. For example, Muppet [54] and Trident [10] (an extension of Storm) store state via hash tables. Spark Streaming [24] enables state computation via Resilient Distributed Datasets (RDDs) [80] which are inherent hashmaps. Some other systems such as Millwheel [31] and Dataflow [32] choose to separate state from the application logic and have state centralized in a remote storage [30], [33], [37] (e.g., a database management system, HDFS [3] or GFS [43]) shared among applications, along with periodically checkpointing state for fault tolerance. A few other systems such as Kafka [5], Samza [6], [62], Spark Streaming [24], and Flink [1], [35] use a combination of "soft state" stored in in-memory data structures along with "hard state" persisted in on-disk data store (e.g., RocksDB [22], LevelDB [16]).

However, it is not easy for these systems to quickly recover large distributed states from the many concurrent failures. This is because when a single node fails due to power outage, system reboot, or environment changes (e.g., temperature), the large distributed states of all dependent nodes must be reset to the last checkpoint, and computation must resume from that point, costing a lot of extra time and space to accomplish recovery. Moreover, these systems rely on a single master for handling failures and stragglers, exhibiting significant overhead from centralization bottlenecks.

B. Failure Recovery in Stream Processing Systems

Existing stream processing systems offer failure recovery mainly through the use of three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery.

Replication Recovery: In the process of replication recovery, as shown in Fig. 2, there is a completely separate set of hot

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

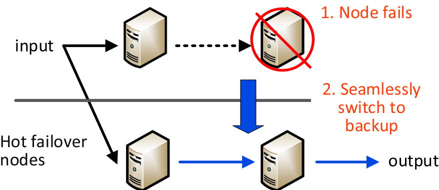


Fig. 2. The replication recovery workflow.

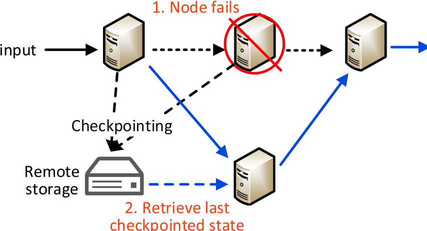


Fig. 3. The checkpointing recovery workflow.

failover nodes that processes the same stream in parallel with the primary set of nodes. Input records are sent to both. When there is a failure or multiple failures in the primary nodes, the system automatically switches over to the secondary set of nodes and the system can continue processing with very little or no disruption. The replication recovery has been widely used in systems such as Flux [71] and Borealis [34]. The failover is fast, and it can handle multiple concurrent failures. However, replication recovery has a linear increment in hardware cost. For example, if each node fails at most once, the hardware cost doubles.

Checkpointing Recovery: In the process of checkpointing recovery, as shown in Fig. 3, each of the nodes in the pipeline has a buffer in memory to retain a backup of the records that it has forwarded to the downstream nodes since the last checkpoint. All nodes periodically checkpoint their states to remote storage such as HDFS or GFS. A standby set of nodes is maintained in the system. If any of the primary nodes fails, a standby node will retrieve the latest checkpoint from the persistent storage, and its upstream node essentially replays the backup records serially to this failover node to recreate the lost state. The checkpointing recovery has been widely used in systems such as TimeStream [66] and Trident [10], Drizzle [75]. It avoids the replication hardware cost. However, the failover is much slower than the replication recovery because it has to retrieve the checkpointed state from the disk and replay the buffered data on the last state to recompute the new state. Multi-level Checkpointing [58] is widely used in high-performance computing (HPC) systems. For example, asynchronous multi-level checkpointing [57], [70] has been a common method for efficient checkpointing. However, the overhead of the checkpointing, especially for the enormous I/O traffic, can be a performance bottleneck without adjusting optimal intervals of checkpointing and checkpoint count configurations [41]. Drizzle [75] introduced group scheduling and pre-scheduling to reduce the centralized scheduling bottleneck. However, it uses a batch processing model and focuses on scheduling tasks for one application, while

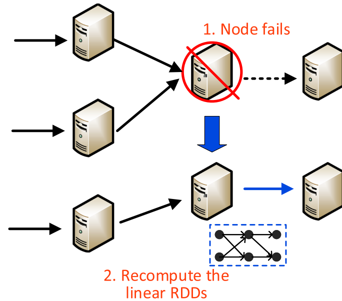


Fig. 4. The DStream-based lineage recovery workflow.

A-FP4S uses a record-at-a-time processing model and focuses on many concurrently running jobs.

DStream-Based Lineage Recovery: To achieve both fast recovery and small hardware overhead, the DStream-based lineage recovery was proposed, as shown in Fig. 4. It has been used in Apache Spark-based systems [1], [35], [72], [81]. The most recent state is stored in each node's memory using a data structure called Resilient Distributed Dataset (RDD) [80], together with the lineage graph, that is, the graph of deterministic operators used to build RDDs. When nodes fail in the system, instead of preparing nodes for failover, DStream will re-run the lost tasks in parallel on other reliable nodes in the cluster using the lineage graph. These tasks can be parallelized to recompute the lost states. However, the entire recovery processing is linear, that is, the lost tasks need to be executed or computed strictly in line with the original lineage graph on other nodes. As such, the recovery process may be slow when the lineage graph is long and incur multiple data uploads through the network.

To our best knowledge, the very few research projects that are broadly relevant to state management solutions are [4], [17], [48], [74]. These projects either point out the criticality of making state explicit [48], [74] or develop mechanisms for reprocessing state [4], [17], but propose no effective solutions for fast state recovery when concurrently running stream applications.

Static State Recovery: The replication-based recovery, checkpointing-based recovery, and DStream-based lineage recovery have a common characteristic: they are static at runtime. No matter how the computing environment changes, their configurations do not change while the application is running. For example, to reduce the checkpointing resources due to extreme large state, Flink [1], [35] provides a minimum duration between checkpoints that can determine the minimum time interval between the end of the latest checkpoint and the beginning of the next. But this minimum duration is a static value that does not change to adapt to runtime conditions. Other approaches such as Noria [44] and window-based recovery [38] either use selective rollback by recomputing data-flow state or assemble different window sizes in checkpointed state, but they do not dynamically adjust the windows or size of checkpointed state. Ozeer et al. [63] introduced a resilience fault tolerance approach in the Fog-IoT environment, where the state recovery takes into consideration uncoordinated checkpoints, message logs,

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

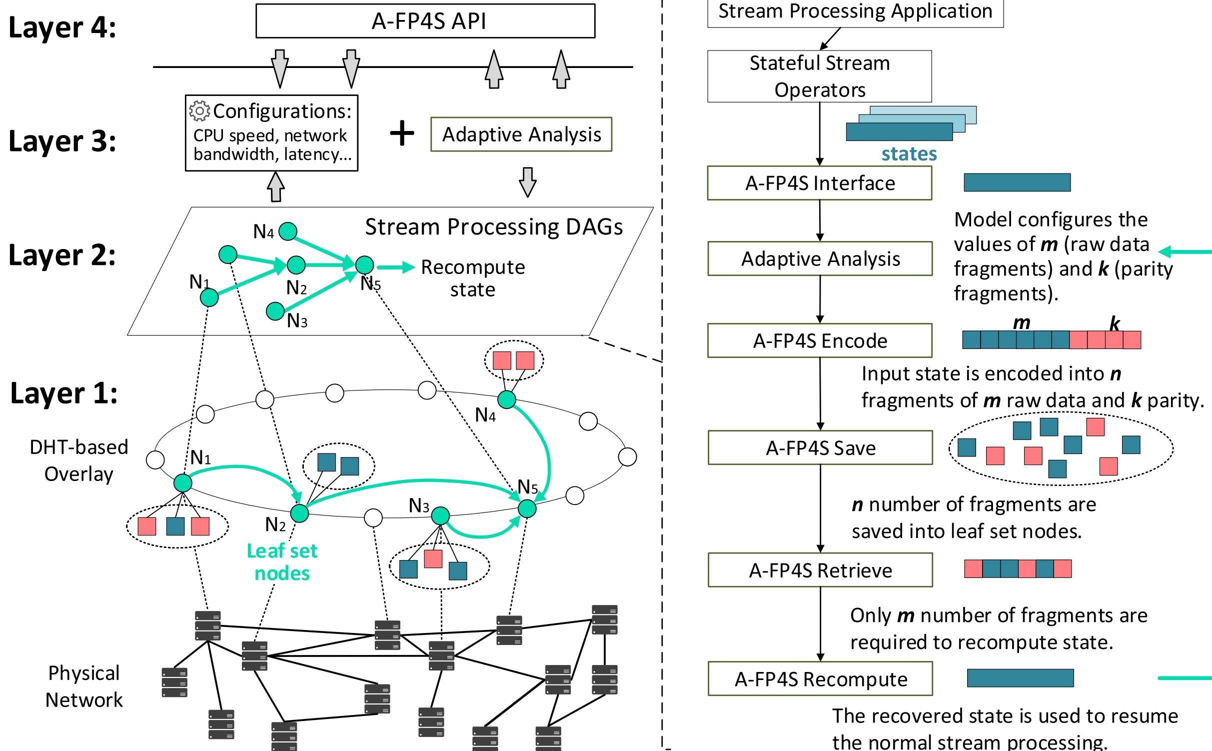


Fig. 5. A-FP4S system design.

and function call records; its state recovery differs in appliance, software element and server failures. Castro et al. [36] proposed an integrated approach for scale-out and recovery of stateful operators, where it periodically checkpoints the process state. However, they do not dynamically customize the configurations of parallel recovery for various applications. Instead, our work provides adaptive state recovery mechanism based on the state size and availability of computing resources.

In our previous work, we proposed SR3 [79], a customizable state recovery framework that offers three recovery mechanisms to cater to the needs of different stream processing computation models, state sizes, and network settings. Compared to SR3, A-FP4S integrates the adaptive run-time analysis of various applications and considers to be flexible to different kinds of stateful stream applications with run-time refinement by adjusting the number of fragments. SR3 can only choose one of three mechanisms even with hundreds of different applications. A-FP4S is more flexible and available for various kinds of stateful stream applications.

III. System Design and Implementation

In this section, we describe the basic workflow of A-FP4S, introduce each component, show how stream applications' distributed states are recovered by the A-FP4S-enabled stream processing system, and explain the performance, scalability and adaptivity benefits of using A-FP4S.

A. Overview

The A-FP4S design aims to achieve the following goals:

- *Resource efficiency:* Avoid the replication hardware overhead.
- *Fast recovery:* Avoid the slow recovery of retrieving state from disk and replaying the data input that hurts the service quality of stream applications.
- *Resilient to multiple failures:* The mechanism needs to handle multiple simultaneous failures due to the much higher node dynamics in large clusters.

As show in Fig. 5, the A-FP4S system consists of four layers: *The DHT-based consistent ring overlay*, *the fragmented parallel state recovery mechanism*, *the adaptive run-time analysis*, and *the high-level A-FP4S interfaces* that are exposed to the stream processing systems (e.g., Storm [8], Spark Streaming [24], Heron [53]) for implementing the state recovery for stream applications.

- *Layer 1: DHT-based ring overlay:* Each data center server is installed with one or many in-situ stream operators, also called "nodes" in this study. We organize these potentially hundreds of thousands of nodes into a distributed hash table (DHT) based ring overlay (e.g., Pastry [69], Chord [73]) which is commonly used in Bitcoin [60], BitTorrent [40], and FAROO [14]. This overlay is self-organizing and self-repairing. To do that, each node needs to maintain two data structures: a routing table and a leaf set, in which the routing table is used for looking for the state (within $log(N)$ hops) and the leaf set nodes are used for recovering the application state if one or more nodes fail.
- *Layer 2: fragmented parallel state recovery:* Periodically, the state in each node's memory is divided into m identically-sized blocks, which are encoded into n blocks,

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

TABLE II
A-FP4S API

List <Fragment>`Encode`(int rawDataNumber, int parityNumber, State inputState)
The function is invoked to encode a state into many fragments. The fragment number is decided based on the arguments `rawDataNumber` and `parityNumber`. The output is a list of encoded fragments with length `rawDataNumber` + `parityNumber`.
Boolean[] `Save`(List<>fragment, DHTNetwork dhtNetwork, int numberOfThreads)
The function is invoked to save state into the DHT's overlay. It generates multiple threads to concurrently save the fragments. The inputs are the fragments, the DHT overlay information and the number of threads. The output is a Boolean array that indicates the status of each fragment.
List<Fragment>`Retrieve`(String stateName, DHTNetwork dhtNetwork, int numberOfThreads)
The function is invoked when a state recovery request is issued.
String Recompute(List<>fragments)
The function is invoked to recover the state. It loops through all the retrieved fragments. If the number of fragments is equal or larger than the number of raw fragments, the function will perform further computation to recompute the retrieved fragments into the original state.

where $n > m$. The *n* blocks of the state are replicated to *n* nodes from the original node's leaf set nodes in parallel, guaranteeing that the original state can be reconstructed from any *m* blocks.

- *Layer 3: adaptive run-time:* For a variety of applications that have different resource and QoS requirements, the A-FP4S runtime uses a module to automatically adjust system parameters to meet their needs. For example, some applications may prefer faster recovery time while some other applications may prefer less storage overhead. The A-FP4S system automatically adapts to these user preferences without manual interventions.
- *Layer 4: high-level interfaces to stream processing systems:* The high-level A-FP4S programming API (Table II) is exposed to the stream processing systems and programmers for implementing the parallel state recovery policies for concurrently running stream applications based on frameworks such as Storm [8], Spark [24], and Flink [1].

B. DHT-Based Ring Overlay

A-FP4S leverages a DHT-based consistent overlay [69], [73] to support parallel recovery of distributed states for a large number of concurrently running stream applications. In this DHT-based consistent ring overlay (e.g., Pastry [69], Chord [73]), each node is equal to the other nodes, having the same rights and duties. The primary purpose of this model is to enable all nodes to work collaboratively to deliver a specific service. For example, in BitTorrent [40], if someone downloads some file, the file is downloaded to her computer in parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others that ask for it.

Similar to BitTorrent, where many machines work collaboratively to undertake the task of downloading files and uploading files, we enable distributed stream operators to work collaboratively to undertake the original centralized master's failure recovery task. First, each stream operator maintains an in-memory buffer to store the application state. Instead of storing states at a remote storage, these distributed stream operators store the states for each other. Second, these distributed stream operators (nodes) are self-organized into a DHT-based overlay. Each node is randomly assigned a unique `NodeId` (128 bits in length) in a large circular `NodeId` space. `NodeIds` are used to identify the nodes and route stream data. It is guaranteed that any data can be routed to a node whose `NodeId` is numerically closest to the destination node within $O(logN)$ hops. To do that, each node maintains two data structures: a routing table and a leaf set.

1) *Routing table:* The routing table consists of physical node characteristics (`NodeId`, `IP`) organized in rows by the length of common prefixes of `NodeId`. When routing a message, each node forwards it to the node in the routing table with the longest prefix in common with the destination `NodeId`. At each routing step, given a key, Pastry [69] routes messages to the node whose `NodeId` is numerically closest to the key. The node first checks if the key falls in the range of the `NodeIds`' leaf set. If so, the message is directly forwarded to that node. If not, the message is forwarded to another node in the routing table whose `NodeId` shares a common prefix with the key by at least one more digit. In some cases, there is no appropriate entry in the routing table or the associated node is not reachable. Then the message is forwarded to a node whose prefix is the same as the local node, but numerically closer.

2) *Leaf set:* The leaf set contains a fixed number of nodes whose `NodeIds` are numerically closest to each node. The `NodeIds` in the leaf set are half larger and half smaller than the current node's `NodeIds`. Leaf set nodes are maintained by piggybacking information about the leaf set membership in keep-alive messages within a configurable time period T (the default T is 30 seconds) [47]. Nodes in the leaf set are symmetric, so that each node can receive a keep-alive message from its leaf set. We use the keep-alive message to detect node failures. If a keep-alive message cannot be received within a specific time window, it can be assumed that the node has failed [56]. When one node fails, its neighbor node contacts the live node with the largest index of the failed `NodeId` in the current node's leaf set and this live node will replace the failed node. Nodes are highly unlikely to suffer correlated failure once the current node fails, so that they can assist in rebuilding routing tables and reconstructing application's state when any operator fails (see Section III-C, next, for more details).

C. Fragmented Parallel State Recovery

The parallel recovery mechanism of A-FP4S leverages a key idea from erasure code. Erasure code is a forward error correction code [13] by utilizing polynomial interpolation [20].

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

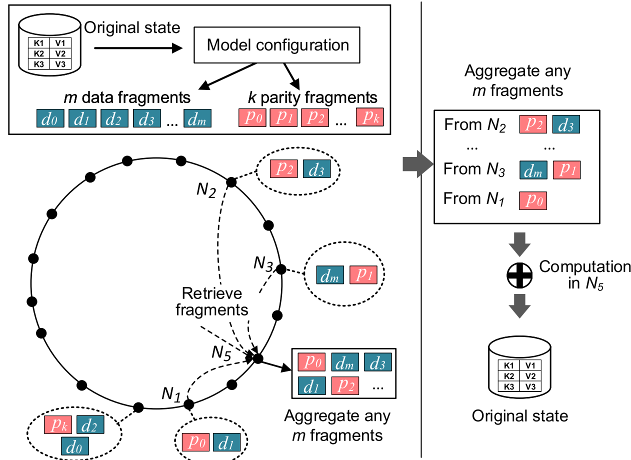


Fig. 6. The fragment-based parallel state recovery process.

It transforms a data object of k symbols into a longer data object with n $(n > k)$ symbols such that the original data object can be recovered from any k of the n symbols [12]. It chooses a finite field F with the default order of 2. It first splits the data symbols from 0 to $k-1$, then constructs a (Lagrange) polynomial $p(x)$ of order k such that $p(i)$ is equal to data symbol i. It then sends $p(k), \ldots, p(n-1)$ to others. Others can use polynomial interpolation to recover the lost packets by using any k symbols [12]. For example, (32, 16)-Reed-Solomon (RS) code [67] divides a data object into 16 blocks and transforms these blocks into 32 coded blocks, guaranteeing that any 16 out of the 32 coded blocks are sufficient to reconstruct the original data object. So that it can tolerate up to 16 errors. Erasure codes have been widely used in massive storage systems (e.g., OceanStore [52]), Bar codes (e.g., QR Code [51]), data transmission technologies (e.g., DSL [45]) and space transmission technologies (e.g., Galileo Probe). Fig. 6 shows the steps of the erasure-code-based parallel recovery algorithm.

Built upon Section III-B's DHT-based ring overlay, each node maintains a routing table and a leaf set. Each node periodically sends heartbeat messages to its neighboring nodes for maintenance. The state store frequency is determined by systems and applications. By default, we set it to be 30 seconds. When needed to save the state at each cycle of streaming processing, the state in each node's memory is encoded into n identically-sized fragments, which include m raw data fragments and k parity fragments, where $k >= 1, n = m + k$. Then these n fragments of the state are replicated to n nodes in the original node's leaf set in parallel. The error correction mechanism of erasure codes guarantees that any m out of the n fragments are sufficient to correctly recompute, even when some fragments are not available in the leaf set (denoted as e), to reconstruct the original state. Thus, as long as $n - -e >= m$, the original state is safe to be accurately recomputed from the node's leaf set nodes.

- *Step 1. Adaptive configuration:* A-FP4S selects the number of raw blocks m and the number of parity blocks k in an adaptive manner based on the resource availability, application's QoS requirements, and user's preferences. By providing the option to adjust m and k, A-FP4S allows users to weigh recovery reliability, speed/efficiency, and storage overhead during the failure recovery process. For example, in A-FP4S, choosing a larger k will produce higher reliability, but will result in higher storage overhead and slightly longer running time.
- *Step 2. Encoding state:* For each node, A-FP4S converts its current version of state in a sliding window into n fragments (configurable parameter) according to the RS-code algorithm [67]. These n fragments include m raw data fragments and k parity fragments.
- *Step 3. Saving state:* Each node sends these n fragments to any n of its leaf set nodes. We ensure that the size of the leaf set is larger than n. We assign the `NodeIds` to reflect the physical proximity in order to ensure that the leaf set nodes are also geographically close nodes that have abundant bandwidth.
- *Step 4. Retrieve state:* Once a failure happens, the retrieve routine is triggered. A request to obtain the lost state's fragments will be sent out. To recompute the lost state, A-FP4S only requires m out of n total fragments. These fragments are stored at the leaf set nodes that are quite easy to access.
- *Step 5: Recompute state:* Finally, the state recomputation routine is triggered, which reconstructs the lost state using erasure codes. After that, the recovered state will be used as input for the downstream operators and the system can resume the normal stream processing.

The benefits are the following: (1) it allows for tolerating a maximum of $(n - m)$ simultaneous failures; (2) the recovery process is fast. For multiple failures, different nodes from non-overlapping leaf set nodes can work in parallel to recompute the lost state, which is faster than DStream's line-structured recovery that executes strictly in line with the original lineage graph; and (3) we achieve data locality because the leaf set contains nodes that are geographically close to the original nodes (e.g., in the same rack or in the same site) that have abundant upload bandwidth.

D. A-FP4S API

A-FP4S is platform-agnostic and can be easily integrated with stream processing platforms such as Storm [8], Spark Streaming [24], Flink [1], Timely Dataflow [66], Heron [53], etc. In our design, using A-FP4S is essentially a configuration option. Depending on the usage scenario (e.g., stateful or stateless, latency requirement, reliability requirement), users can choose to configure whether and when they want A-FP4S support. Table II shows the A-FP4S API.

E. Instrumentation Requirements

Here we describe the instrumentation requirements A-FP4S imposes and discuss the issues we encountered when integrating it with the Apache Storm processing engine.

In Apache Storm [8], stream processing applications are deployed and executed as *topologies*. The *topologies* contain the business logic that is then transformed into a Directed Acyclic Graph (DAG) implemented using *spouts* and *bolts*. *Spouts* are

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

the data sources of the stream, which accept input data from raw data sources like the Twitter Streaming API [27] or the Apache Kafka queue [5]. *Bolts* are the logical processing units. *Spouts* pass data to bolts and *bolts* process and produce a new output stream. `IRichBolt` is the common interface for implementing *bolts*.

A-FP4S interacts with the `IRichBolt` interface in Storm [8]. If A-FP4S is enabled, A-FP4S periodically saves the states into the DHT-based ring overlay for all stateful operators (*bolts*). For record-at-a-time systems like Storm, saving every operator's state may incur a lot of overhead. Instead, we aggregate the states for all the operators except for sources (*spouts*) and *sinks*. The aggregated state size is configurable in order to satisfy different real-world stream applications' requirements. After the size reaches a certain threshold, the `Encode` function encodes the states into fragments and the `Save` function puts these fragments into the DHT-based overlay. If any node fails, the leaf set nodes call the routines to `Retrieve` and `Recompute` states on fail over nodes. Any qualified available subset of fragments will be sufficient to recover the lost states through the `Recompute` function.

F. Discussion

Why DHT? A-FP4S leverages a DHT-based ring overlay, and the main benefit of choosing a DHT-based ring overlay is that it can flexibly handle the nodes that suffer from high churn for stream applications in distributed systems. Stream processing, as a technique to implement real-time processing, has been widely used in many Big Data applications. Many of them need to be deployed in heterogeneous environments, such as edge devices or IoT systems. These environments naturally lead to high churn, in which workloads change unexpectedly, wide-area network bandwidth changes unexpectedly and Edge nodes leave or fail unexpectedly (e.g., due to signal attenuation, interference, and wireless channel contention). The flexibility and scalability of DHT can be a good solution for these environments.

Scalability and Robustness. A-FP4S is built upon the DHT-based ring overlay, which is self-organizing and self-repairing. Distributed nodes can easily join and leave the overlay, enabling flexible management for large-scale state recovery for stream applications. Besides, all nodes are equal. They have the same duties and responsibilities. The system can easily avoid the central bottleneck caused by the client/server architecture. Further, when one node fails, it's neighbor nodes in its leaf set can quickly detect the failure by using keep-alive messages. Neighboring nodes will immediately contact another active node to resume progress, which can support robust performance for a large number of concurrently running applications.

Intervals Refer to State. How to find an optimal interval strategy for state saving is a key issue in the stream processing systems. Without an optimal state saving interval, the performance of the application may degrade badly. Besides, due to the dynamic performance of various kinds of stream applications, the strategy of state relevant intervals should be more flexible. For example, different intervals should be applied when handling different stream applications with various workloads [50].

Dynamic Streams. The data streams in modern stream processing applications dynamically change with respect to volume, velocity, and variety [42]. Workloads are in general variant in the long-term and short-term. Long-term workload fluctuations have received considerable research efforts. However, workload fluctuations are mostly short-term and random in nature [42]. Therefore, the runtime overhead of failure recovery for short-term workloads is time-varying and non-trivial. In A-FP4S, the runtime adaptive analysis supports to a variety of short-term stream applications and can dynamically tune the parameter when dealing with different applications. A-FP4S provides flexibility for both long-term and short-term stream applications.

IV. Adaptivity Analysis

A. Adaptive Parameter Tuning

We provide a theoretical analysis of the A-FP4S model that dynamically determines the size and number of fragments, achieving system adaptability. A-FP4S collects the instrumentation data during each round of application, uses this data to train the model that will be exported next, and then configures system parameters for the next processing cycle. The input to the model includes hardware properties (e.g., network and CPU speed), application characteristics and also user preferences. Using this information, our models can accurately moderate the number of data fragments m and parity fragments k to match the requirements for the subsequent episodes.

A-FP4S can adjust the value of k so that it can accommodate multi-fragments failures during the recovery process. Such extra cushion of reliability is particularly desirable when the application nodes are known to be more failure-prone and unreliable. On the other hand, some applications (e.g., real-time network monitoring) may opt for faster recovery time over 100% reliability. A-FP4S can adjust m and thereby the default size of each fragment to reduce the recovery latency.

B. Analysis

In this subsection, we analyze the performance of A-FP4S with adaptive number of fragments and compare it with the checkpoint-based recovery (e.g., Apache Storm) in the cluster. We compare the different methods of failure recovery based on three aspects: (1) the hardware properties (e.g., network bandwidth, disk speed), (2) the application characteristics (e.g., state size, fragment size), and (3) the DAG length characteristics. Besides, evaluation results show that the model analysis performance is consistent with the experimental results.

Assume the volume of state saved by each operator is s bytes. In a DAG where the operator A sends its output to the operator B, A retains the s bytes of records that it has passed down to B since the last checkpoint. For simplicity, we only consider buffer state, ignoring processing state for the time being. When (and if) operator B fails, operator C takes over and receives s bytes from A. In a checkpoint-based recovery method, these s bytes must come from reading HDFS or some sort of network file system. Assuming HDFS bandwidth to be h-bytes/sec, a checkpoint-based recovery scheme such as used in Apache Storm will take:

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

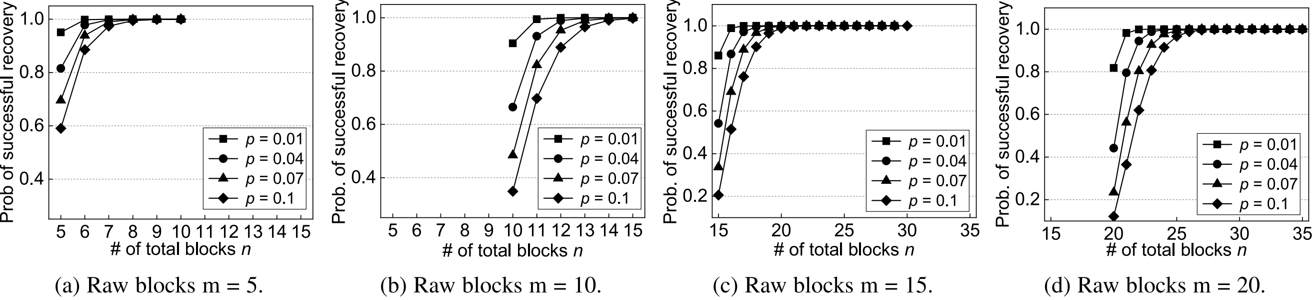


Fig. 7. Probability of successful recovery with varying m, n and p.

$$R_c = \frac{s}{h}. \tag{1}$$

When implementing A-FP4S recovery instead of checkpointing in the DAG, the buffer state of node A is periodically backed up in its leafset nodes. Note that the s bytes of buffer state is first split into m blocks (each with s/m bytes) that are then erasure-coded into n blocks stored in n leaf-set nodes, where $n > m$. Therefore, s bytes of buffer state requires sn/m bytes of storage in A-FP4S, leading to an overhead factor of $(n-m)/m$.

Note that although only s bytes of state are needed, C still issues requests for all n coded blocks in anticipation of any potential failures among the sending nodes. However, after m blocks are received correctly, C can ignore the remaining amount. Assuming a network bandwidth of η-bytes/sec, this retrieval takes s/η seconds. After that, C can recompute s bytes of state from these coded blocks, say, at a rate c-bytes/second, which takes s/c seconds. Therefore, the recovery time of A-FP4S, denoted by R_f, is:

$$R_f = \frac{s}{\eta} + \frac{s}{c}. \tag{2}$$

We next consider the reliability aspect of the derived models, which is an important metric to consider because of the random node failures that can lead to some non-determinism. Assume that p is the probability for a node failure at any time. We also assume that node failure is a Poisson process, which means that previous failures do not affect the current failure.

When B fails in the DAG, another operator C needs at least m out of n leafset nodes of A to recover fully. That means C will recover in a single hop of data transfer if $n-m$ or less nodes from A's leafset fail. Let the random variable X denote the event when this happens, i.e., C can recover using A's leafset nodes in a single hop, which requires at least m leafset nodes of A to be alive at that moment. Note that if fewer leafset nodes are available, the DHT overlay will reorganize itself and provide functioning leafset nodes for A's leafset. However, that would take more time and our model in (2) does not cover that. Therefore, our model reliability is given by:

$$P(X=1) = \sum_{i=0}^{n-m} \binom{n}{i} p^i (1-p)^{n-i}. \tag{3}$$

While there is no close-form solution to the above expression, we perform numerical evaluations with varying m, then varying n in range $[m, 2m]$, and also varying the the node failure probability p. Results are shown in Fig. 7.

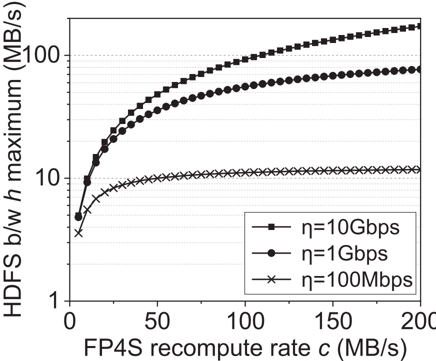


Fig. 8. Maximum limit for HDFS rate versus Recompute rate c in A-FP4S.

Note that another way to interpret our model reliability equation in (3) is that it also works as a measure of A-FP4S's efficiency: the failure recovery takes the minimum time when the right parameters m and k are chosen. Of course, A-FP4S will continue to provide reliability without such parameter tuning, although potentially sub-optimal in its use of resources.

Combining (1) and (2), we get:

$$R_c > R_f \Rightarrow \frac{s}{h} > \frac{s}{\eta} + \frac{s}{c} \Rightarrow h < \frac{c\eta}{c+\eta}. \tag{4}$$

We show the effect of (4) in Fig. 8, where we compare the maximum allowed HDFS bandwidth h against A-FP4S's recompute rate c. The goal is to see up to what HDFS speed it is still viable to use A-FP4S for a given network bandwidth η. It is clear that for the most realistic values of c, η and h, A-FP4S is the preferred choice in terms of performance.

V. Evaluation

We evaluate A-FP4S by using large scale real-world experiments, demonstrating its scalability, adaptivity, and fast failure recovery. Experimental evaluations answer the following questions:

- How does A-FP4S scale with the state size, the number of concurrently running applications and the number of simultaneously failed operators?

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

TABLE III
REAL-WORLD APPLICATION'S DATASET

Application	Dataset	Size
Trending Topics	Twitter Streaming API [27]	>1TB
Bargain Index	Google Finance [15]	>1TB
Word Count	Project Gutenberg [21]	8GB
	Wikimedia Dumps [28]	9GB
Traffic Monitoring	Dublin Bus Traces [11]	4GB

- How does the efficiency of the fragment-based parallel state recovery algorithm change with different parameters such as the number of the raw fragments (*m*), the number of the coded fragments (*n*)? How does A-FP4S balance the workload?
- What are the performance and functionality benefits of A-FP4S compared to state-of-the-art solutions?
- What is the runtime overhead of A-FP4S?

We organize our evaluation with the following key results.

- A-FP4S achieves 31.8% to 50.5% improvement in the total time (i.e., summation of state saving time and recovery time) compared to the checkpointing recovery approach in Apache Storm (Section V-B).
- A-FP4S achieves runtime efficiency by adaptively tuning the parameters with various conditions (Sections V-C and V-D).
- A-FP4S evenly distributes the huge volume of states across all nodes in the overlay, demonstrating A-FP4's attractive load balancing and scalability features (Section V-E).
- The CPU overhead of state recovery and saving in A-FP4S are on average 17.4% and 16.5% less than checkpointing recovery in Storm, and for memory overhead, it achieves 25.3% and 28.1% reduction, respectively (Section V-F).

A. Setup

Experiments are conducted on up to 4 machines, each with 16 Intel Xeon Gold 6130@2.10 GHz cores and 256 GB of RAM, running GNU/Linux 3.10.0. On top of these machines, we boot 50 virtual machines to host 650 stream operators in total, each with 4 cores and 8 GB of memory, running Linux Ubuntu 4.4.0. We use Apache Storm 2.0.0 [9] configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We use Pastry 2.1 [19] configured with leafset size of 24, max open sockets of 5000 and transport buffer size of 6 MB.

We deploy Yahoo streaming benchmarks [39] and real-world stream applications using A-FP4S (see Table III) to demonstrate its generality. These include various representative streaming operators such as stateless streaming transformations (e.g., `map`, `filter`), stateful operators (e.g., `incremental join`), and various window operators (e.g., sliding window, tumbling window and session window). We compare A-FP4S with a state-of-the-art check pointing recovery approach commonly used in TimeStream [66], Storm [8], and Trident [10]. We are not able to compare with Drizzle [75] because its source code is not publicly available. We choose the checkpointing recovery approach as the baseline because the alternatives either incur significant hardware cost (e.g., replication recovery requires twice the hardware) or are not generally applicable (e.g., DStream-based lineage recovery approach lacks programming transparency and must be used with Spark's RDDs [80]).

For our experiments, the base value of raw fragments m and the total coded fragments n are derived from production systems such as Pond [68] and Sia [23], which set $m = 16$, $n = 16$ and $m = 10$, $n = 20$ respectively. To fully evaluate the A-FP4S performance, we vary the values of m, n and the input state size.

B. A-FP4S versus Storm

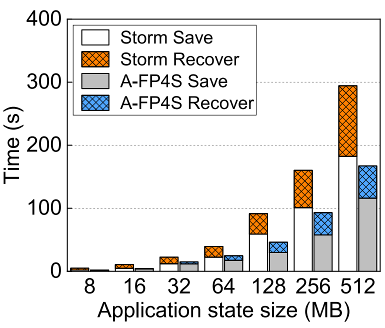
We evaluate the failure recovery time of A-FP4S by varying the state size and the number of concurrently running stream applications.

The fragment-based parallel recovery process in A-FP4S consists of two steps: (1) saving the state to leafset nodes in the DHT-based overlay, and (2) recomputing the state after any failure happens. Similarly, the checkpointing recovery process in Storm [8] also consists of two steps: (1) checkpointing the state to the HBase [4] or HDFS [3], and (2) retrieving the state from HBase or HDFS if failure happens. Note that, for both approaches, the first step can run asynchronously with the second step, so the first step may not impact the failure recovery time if executed in a pipeline.

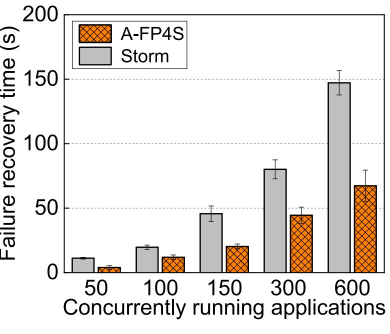
Fig. 9(a) shows the comparison of the state recovery time of A-FP4S and Storm by varying the input state sizes. In general, Fig. 9(a) shows that A-FP4S achieves 31.8% to 50.5% improvement in the total time (i.e., summation of state saving time and recovery time) compared to Storm's checkpointing recovery. Specifically, we see that A-FP4S achieves 36.2% to 50.8% less state saving time compared to Storm. A-FP4S's state saving time includes the time of fragmenting state into blocks, encoding them, and then uploading them into the leafset nodes. In order to have a fair comparison with Storm, we assume that the upload operation happens sequentially (i.e., one leafset node at a time). However, in a realistic scenario where the uploading node has higher network bandwidth compared to the leafset nodes, A-FP4S will deliver even faster state saving time by uploading the data asynchronously and in parallel. We also see that A-FP4S achieves 40.3% to 87.1% less recovery time compared to Storm. This is because A-FP4S's multiple leafset nodes can contribute by recalculating the state in parallel. In contrast, Storm relies on a single node to retrieve the state from HBase or NFS [18] whose speed is largely determined by network bandwidth, network interference, node placement and many other factors.

We next evaluate A-FP4S's total failure recovery time by varying the number of concurrently applications and compare it with Storm. The failure rate of stream operators is set to 1% for these experiments according to Zorro [65]. As shown in Fig. 9(b), A-FP4S achieves 43.8% to 54.4% less total recovery time compared to Storm. The reason behind this is that A-FP4S's recovery workload can be distributed evenly across all participating nodes in the DHT-based overlay. As a result, many operators

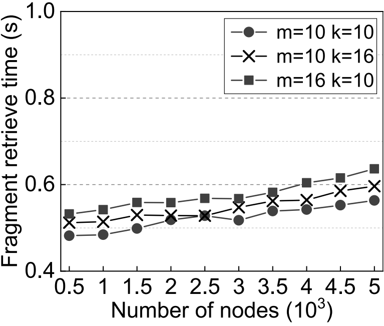
Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.



(a) State recovery time for different input state sizes.

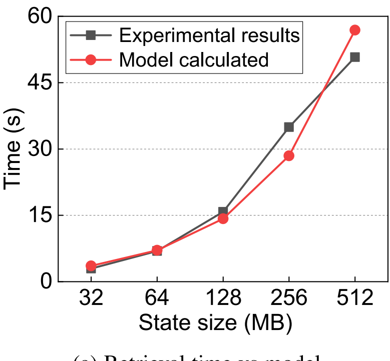


(b) Total failure recovery time by varying # concurrently running stream applications.

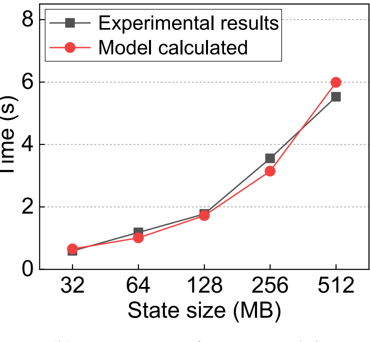


(c) Average retrieve time for state fragments in A-FP4S.

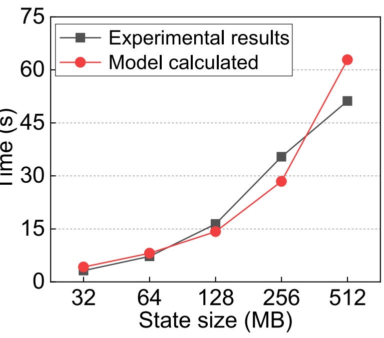
Fig. 9. Performance comparison A-FP4S performance versus checkpointing strategy for different input state size and applications number.



(a) Retrieval time vs model.



(b) Recompute time vs model.



(c) Total recovery time vs its model

Fig. 10. The retrieve (a) and recovery (b) time compared against the corresponding models. The total recovery time is also given in part (c).

can run the recovery process simultaneously, leading to much better and resilient performance under many failures.

Finally, we evaluate the average per-fragment retrieval time of A-FP4S by varying the number of nodes, raw fragments m and parity fragments k. As shown in Fig. 9(c), the average time does not vary much with the number of nodes. As the number of nodes increases from 500 to 5,000, the retrieval time only increases slightly.

C. A-FP4S versus Theoretical Model

We compare our model derived in (2) against observed recovery times in Fig. 10. Note that (2) has two parts: retrieval of the fragments over the network and recomputing state from them. We verify the accuracy of the model on these two parts separately in parts (a) and (b), respectively. In Fig. 10(a), we see that our model, after assuming network bandwidth of around $72 - 75$ Mbps, matches with retrieval time. Such variance from the true bandwidth of 100 Mbps is possible because of multiple operators being placed in the same physical machine and thus interfering with each other. However, this effective network bandwidth can be derived easily from observation even with very little instrumentation cost.

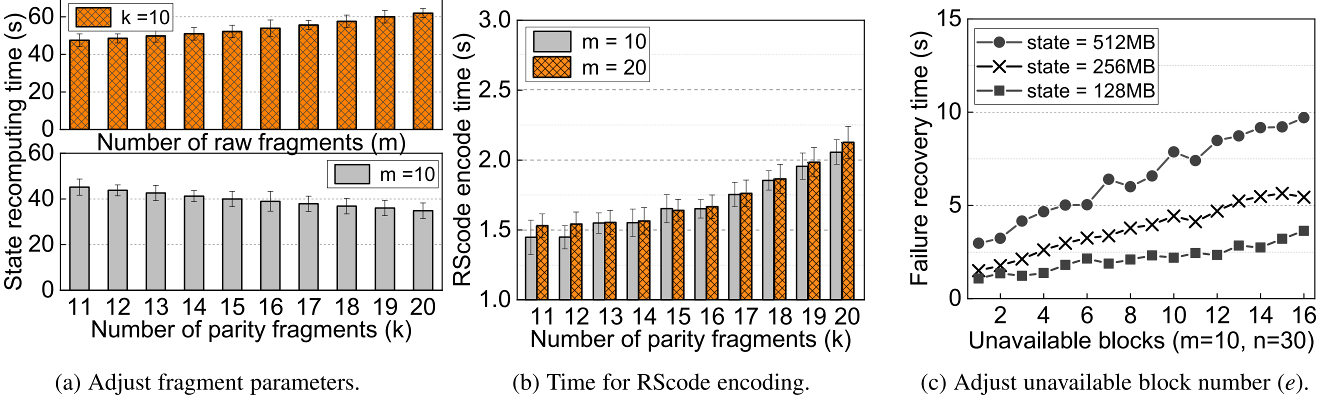
Then, in Fig. 10(b), we hypothesize that recomputing the state is linear on state size for a give computation rate c. Under this assumption, we derive the empirical or observed recomputation rate to be roughly 65 MBps. This leads us to fairly accurate recomputation time as a function of state size, as demonstrated in Fig. 10(b). Finally, we combine these two and present the final model of recovery time (i.e., retrieve time plus recomputation time) against the observed time for the same in Fig. 10(c), which again shows a good match, deeming our overall model accurate.

D. A-FP4S Parameters

We next evaluate the state recovery as a function of several factors: the number of raw fragments m, the number of the parity fragments k of a state, the number of unavailable or failed blocks e of a state, and the routing performance in the overlay.

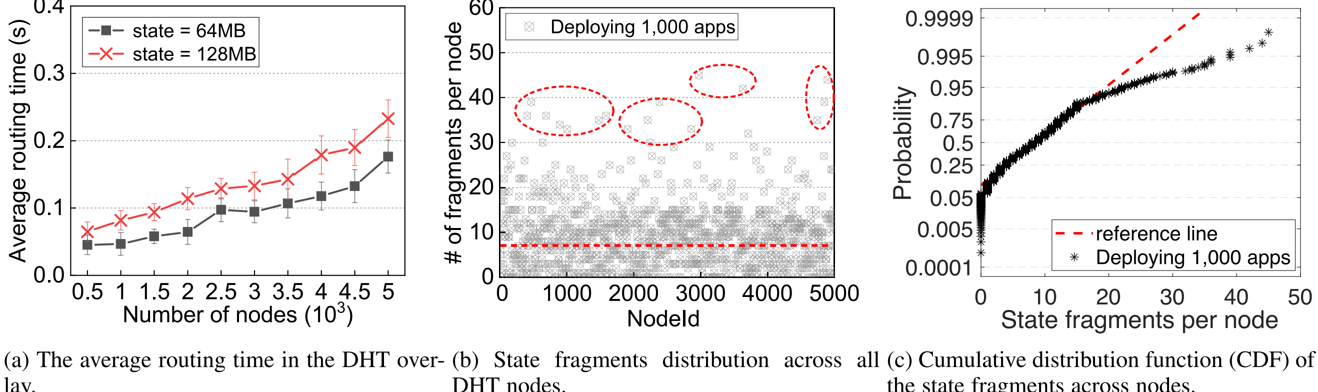
In Fig. 11(a), the up-side sub-figure shows the performance of state recomputing time when recovering from single failure by varying m. The number of the raw fragments m in a state on the recovery performance varies from 11 to 20, where k is set to be 10. We can observe that the state recomputing time increases as the number of raw fragments m increases. The reason lies in that the recovery time of A-FP4S is mainly determined by $mB/(m+k-1)$, where B is the amount of data that any providing peer uploads. $mB/(m+k-1)$ increases with the increases of m when the values of k and B are given. Thus, the performance of A-FP4S is more sensitive to m when k is smaller. The down-side

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.



(a) Adjust fragment parameters. (b) Time for RScode encoding. (c) Adjust unavailable block number (e).

Fig. 11. The recovery performance evaluation by adjusting number of raw fragments, number of parity fragments, and number of unavailable blocks.



(a) The average routing time in the DHT overlay. (b) State fragments distribution across all DHT nodes. (c) Cumulative distribution function (CDF) of the state fragments across nodes.

Fig. 12. The load balance evaluation of A-FP4S for a collection of concurrently running stream applications.

sub-figure of Fig. 11(a) shows that it achieves better recovery performance when k is increasing from 11 to 20. The reason is that the recovery time of A-FP4S is mainly determined by by $mB/(m+k-1)$, where B is the amount of data that any providing peer uploads. $mB/(m+k-1)$ decreases with the increases of k when the values of m and B are given.

Next, we evaluate the encoding overhead in the RS-code under varying parity fragments of k. Fig. 11(b) shows the average encoding time for a state size of 128 MB. We see that the encoding time increases slightly with the increase of k. Still, the encoding operation completes within 1.5 to 2 seconds.

Fig. 11(c) shows how the number of unavailable blocks e impacts the recovery performance as we vary e in the range of $[1, 16]$. A-FP4S's recovery time increases slowly but linearly with e in this range. A stand-by node, together with the failed node's leafset nodes form a star structure for receiving the backup state, causing the stand-by node to be the main I/O bottleneck. Therefore, the recovery performance of A-FP4S is inversely proportional to the amount of data uploaded by the peers.

Fig. 12(a) shows the average routing time in the A-FP4S overlay with the number of nodes in the range of $[500, 5000]$. We see a slight increase in routing time with the increasing number of nodes. This is because each routing decision requires only $O(logN)$ steps, where N is the number of nodes in the overlay.

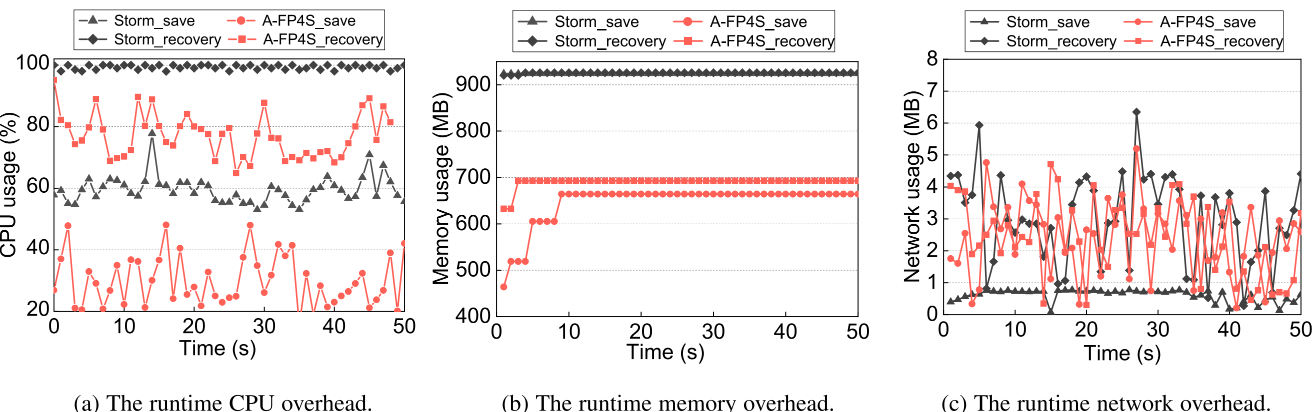
This evaluation confirms that we can improve the efficiency of failure recovery by tuning the A-FP4S parameters dynamically to adapt to runtime conditions.

E. Load Balancing

Since A-FP4S distributes the tasks of state saving and recovery across the whole overlay, it has load balancing as one of its differentiating features. As a demonstration, we deploy 5,000 nodes on a platform of 50 virtual servers running 1,000 stream applications. Each application is configured with $m = 10$ and $n = 30$. Therefore, there are 30,000 fragments in total that need to be saved in the 5,000 nodes.

Fig. 12(b) shows the distribution of the state fragments on all nodes. We can observe that the fragments are almost evenly distributed across all nodes in the overlay. This is because the DHT overlay can distribute all applications evenly in the whole ID space. In addition, only a few nodes contain more than 30 state fragments. These nodes are usually the root nodes or nodes that are close to the root nodes. Fig. 12(c) shows the cumulative distribution function of the number of fragments saved per node.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.



(a) The runtime CPU overhead.

(b) The runtime memory overhead.

(c) The runtime network overhead.

Fig. 13. The overhead analysis of the A-FP4S-enabled Storm at runtime.

We see that 95% of nodes store less than 25 fragments, demonstrating A-FP4S's advantageous load balancing and scalability features.

F. Overhead Analysis

Finally, we evaluate A-FP4S's runtime overhead in terms of CPU, main memory, disk and network.

CPU Overhead. Fig. 13(a) shows the per-node CPU runtime overhead comparison of A-FP4S versus checkpointing recovery. The CPU overhead of state recovery and saving in A-FP4S are on average 17.4% and 16.5% less than those in checkpointing recovery, respectively. While A-FP4S requires additional CPU bandwidth to compute the fragments, this cost accounts for only a small fraction ($<10\%$) of the total recovery time.

Memory Overhead. Fig. 13(b) shows the per-node memory run-time overhead comparison of A-FP4S versus checkpointing recovery. We see that A-FP4S takes on average 25.2% and 28.1% less than the checkpointing recovery, respectively. The large memory overhead of checkpointing recovery is mainly due to the adoption of a centralized daemon process such as Zookeeper for coordinating the operators. In contrast, nodes in A-FP4S work in a peer-to-peer (P2P) fashion that avoids any centralized daemons.

Network Overhead. Fig. 13(c) shows the per-node network run-time overhead comparison of A-FP4S versus checkpointing recovery. Checkpointing recovery incurs lower network usage, but results in longer latencies. In contrast, A-FP4S failure recovery can be completed in a fast fashion by utilizing the distributed nodes among the overlay.

VI. Conclusion

In this paper, we present A-FP4S, an adaptive fragment-based parallel state recovery mechanism that can handle many simultaneous failures for stateful stream applications. A-FP4S leverages DHTs and erasure codes to divide each operator's in-memory state into fragments that are periodically saved in the corresponding leaf set nodes. Since the recovery operation stays local within a small cluster of nodes, it can proceed in parallel for simultaneously failed nodes over different parts of the network. Besides, based on its performance models, A-FP4S's adaptive component can dynamically adjust several system parameters (e.g., fragment numbers m, parity blocks k, state size) at runtime with minimal instrumentation cost. Therefore, unlike the replication, checkpointing or DStream-based methods, A-FP4S is resilient against simultaneous failures, achieves low-latency and is less resource (CPU, memory, disk space, network traffic) intensive.

A-FP4S is framework-agnostic and thus broadly applicable to a large collection of streaming systems. We have implemented A-FP4S atop the state-of-the-art stream processing engine Apache Storm, and demonstrated its scalability, efficiency, and fast failure recovery features that incur negligible instrumentation overheads. In the future, we envision exploring A-FP4S's performance on heterogeneous compute clusters (e.g., nodes with varying network bandwidth, CPU speed, and storage capacity) that have the potential to take further advantage of A-FP4S's decentralized architecture, resilience, reliability and efficiency.

References

[1] "Apache flink," 2023. [Online]. Available: http://flink.apache.org/
[2] "Apache flume," 2023. [Online]. Available: http://flume.apache.org/
[3] "Apache hadoop HDFS," 2023. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/
[4] "Apache HBase," 2023. [Online]. Available: http://hbase.apache.org/
[5] "Apache kafka," 2023. [Online]. Available: http://kafka.apache.org/
[6] "Apache samza," 2023. [Online]. Available: http://samza.apache.org/
[7] "Apache spark," 2023. [Online]. Available: http://spark.apache.org/
[8] "Apache storm," 2023. [Online]. Available: http://storm.apache.org/
[9] "Apache storm 2.0.0," 2023. [Online]. Available: https://storm.apache.org/2019/05/30/storm200-released.html
[10] "Apache trident," 2023. [Online]. Available: http://storm.apache.org/releases/current/Trident-tutorial.html
[11] "Dublin bus GPS sample data from dublin city council," 2023. [Online]. Available: https://data.gov.ie/dataset/
[12] "Erasure code," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Erasure_code
[13] "Error correction code," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Error_correction_code#Forward_error_correction
[14] "Faroo," 2023. [Online]. Available: https://en.wikipedia.org/wiki/FAROO
[15] "Google finance data API," 2023. [Online]. Available: http://finance.google.com/finance/feeds/
[16] "Leveldb," 2023. [Online]. Available: https://github.com/google/leveldb/
[17] "Mongodb," 2023. [Online]. Available: http://www.mongodb.com/

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

[18] "Network file system," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Network_File_System
[19] "Pastry," 2023. [Online]. Available: https://www.freepastry.org/FreePastry/
[20] "Polynomial interpolation," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Polynomial_interpolation
[21] "Project gutenberg," 2023. [Online]. Available: http://www.gutenberg.com/
[22] "Rocksdb," 2023. [Online]. Available: http://rocksdb.org/
[23] "Sia: A decentralized storage platform secured by blockchain technology," 2023. [Online]. Available: http://sia.tech/
[24] "Spark streaming," 2023. [Online]. Available: https://spark.apache.org/streaming/
[25] "Stateful stream processing," 2023. [Online]. Available: https://www.oreilly.com/library/view/stream-processing-with/9781491974285/ch01.html
[26] "Summary of the amazon EC2 and amazon ebs service event in the tokyo (ap-northeast-1)," 2023. [Online]. Available: https://aws.amazon.com/message/56489/
[27] "Twitter streaming apis," 2023. [Online]. Available: https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data
[28] "Wikimedia dumps," 2023. [Online]. Available: https://dumps.wikimedia.org/
[29] D. J. Abadi et al., "The design of the borealis stream processing engine," in *Proc. Conf. Innov. Data Syst. Res.*, 2005, pp. 277–289,.
[30] D. J. Abadi et al., "Aurora: A new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.
[31] T. Akidau et al., "MillWheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.
[32] T. Akidau et al., "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.
[33] A. Arasu et al., "STREAM: The stanford data stream management system," in *Data Stream Management*, Berlin, Germany: Springer, 2016, pp. 317–336.
[34] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 13–24.
[35] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *Proce. VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.
[36] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.
[37] S. Chandrasekaran et al., "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, Art. no. 4.
[38] Q. Chen and M. Hsu, "Recovering a failure in a data processing system," US Patent App. 13/857, Oct. 9, 2014.
[39] S. Chintapalli et al., "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1789–1792.
[40] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. Workshop Econ. Peer-to-Peer Syst.* 2003, pp. 68–72.
[41] T. Dey et al., "Optimizing asynchronous multi-level checkpoint/restart configurations with machine learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2020, pp. 1036–1043.
[42] J. Fang et al., "Parallel stream processing against workload skewness and variance," in *Proc. 26th Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2017, pp. 15–26.
[43] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[44] J. Gjengset et al., "Noria: Dynamic, partially-stateful data-flow for high-performance web applications," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 213–231.
[45] P. Golden, H. Dedieu, and K. S. Jacobsen, *Implementation and Applications of DSL Technology*. Boca Raton, FL, USA: CRC Press, 2007.
[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[47] A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel, "Consistent key mapping in structured overlays," Rice Univ., Houston, TX, Tech. Rep. TR05-456, 2005.
[48] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[49] S. Jayasekara, A. Harwood, and S. Karunasekera, "A utilization model for optimization of checkpoint intervals in distributed stream processing systems," *Future Gener. Comput. Syst.*, vol. 110, pp. 68–79, 2020.
[50] S. Jayasekara, S. Karunasekera, and A. Harwood, "Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice," *Softw.: Pract. Experience*, vol. 52, no. 1, pp. 296–315, 2022.
[51] P. Kieseberg et al., "QR code security," in *Proc. 8th Int. Conf. Adv. Mobile Comput. Multimedia*, 2010, pp. 430–435.
[52] J. Kubiatowicz et al., "Oceanstore: An architecture for global-scale persistent storage," *ACM SIGARCH Comput. Architecture News*, vol. 28, pp. 190–201, 2000.
[53] S. Kulkarni et al., "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 239–250.
[54] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: Mapreduce-style processing of fast data," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, 2012.
[55] P. Liu, H. Xu, D. D. Silva, Q. Wang, S.T. Ahmed, and L. Hu, "FP4S: Fragment-based parallel state recovery for stateful stream applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 1102–1111.
[56] R. Mahajan, M. Castro, and A. Rowstron, "Controlling the cost of reliability in peer-to-peer overlays," in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2003, pp. 21–32.
[57] A. Maurya, B. Nicolae, M. M. Rafique, T. Tonellot, and F. Cappello, "Towards efficient I/O scheduling for collaborative multi-level checkpointing," in *Proc. 29th Int. Symp. Modeling, Anal. Simul. Comput. Telecommun. Syst.*, 2021, pp. 1–8.
[58] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.
[59] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 439–455.
[60] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008, Art. no. 21260.
[61] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, 2010, pp. 170–177.
[62] A. Shadi et al., "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.
[63] U. Ozeer, X. Etchevers, L. Letondeur, F. Ottogalli, G. Salaün, and J.-M. Vincent, "Resilience of stateful IoT applications in a dynamic fog environment," in *Proc. 15th EAI Int. Conf. Mobile Ubiquitous Systems: Comput. Netw. Serv.*, 2018, pp. 332–341.
[64] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 251–264.
[65] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, "Zorro: Zero-cost reactive failure recovery in distributed graph processing," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 195–208.
[66] Z. Qian et al., "TimeStream: Reliable stream computation in the cloud," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 1–14.
[67] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[68] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The oceanstore prototype," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2003, pp. 1–14.
[69] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2001, pp. 329–350.
[70] K. Sato et al., "Design and modeling of a non-blocking checkpointing system," in *Proc. Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–10.
[71] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 827–838.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.

[72] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, “Flint: Batch-interactive data-intensive processing on transient servers,” in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 6.

[73] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001.

[74] R. Tudoran, G. Antoniu, and L. Bouge, “SAGE: Geo-distributed streaming data analysis in clouds,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2013, pp. 2278–2281.

[75] S. Venkataraman et al., “Drizzle: Fast and adaptable stream processing at scale,” in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 374–389.

[76] H. Wang, H. Shen, and Z. Li, “Approaches for resilience against cascading failures in cloud datacenters,” in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 706–717.

[77] Y. Wu and K.-L. Tan, “ChronoStream: Elastic stateful stream computation in the cloud,” in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 723–734.

[78] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, “Lessons and actions: What we learned from 10K SSD-related storage system failures,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 961–976.

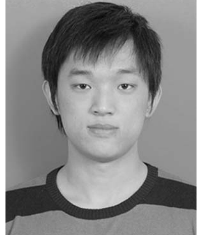
[79] H. Xu, P. Liu, S. Cruz-Diaz, D. Da Silva, and L. Hu, “SR3: Customizable recovery for stateful stream processing systems,” in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 251–264.

[80] M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. 9th USENIX Conf. Networked Syst. Des. Implementation*, 2012, pp. 2–2.

[81] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.

[82] Y. Zhuang, X. Wei, H. Li, Y. Wang, and X. He, “An optimal checkpointing model with online OCI adjustment for stream processing applications,” in *Proc. 27th Int. Conf. Comput. Commun. Netw.*, 2018, pp. 1–9.

Hailu Xu received the BS degree in computer science from North China Electric Power University, in 2014, the MS degree in computer science from the University of Toledo, USA, in 2016, and the PhD degree in computer science from Florida International University, in 2020. His current research interests include cloud computing, Big Data system, and operating systems. Currently he is an assistant professor with the Department of Computer Engineering & Computer Science, California State University, Long Beach.



Pinchao Liu received the BSc (Hons) degree from the Tianjin University of Science and Technology (TUST), China, and the MSc degree from the Tianjin University of Science and Technology, China. He is currently working toward the PhD degree with the School of Computing and Information Science, Florida International University, Miami, USA. His research interests include systems virtualization, cloud computing, and operating systems.

Sarker Tanzir Ahmed received the BS degree in computer science from the Bangladesh University of Engineering and Technology, Bangladesh, and the PhD degree in computer science from Texas A&M University, College Station. Currently, he works as an instructional assistant professor with the Department of Computer Science and Engineering, Texas A&M University. His research interests include large-scale information processing, streaming frameworks with state-management, web crawling, and high-performance computing.

Dilma Da Silva received the PhD degree in computer science from Georgia Tech, in 1997. She is a professor and holder of the Ford Motor Company Design Professorship II with the Department of Computer Science and Engineering, Texas A&M University, USA. She is an ACM distinguished scientist. Her research interests include operating systems addresses the need for scalable and customizable system software. She is a member of the board of CRA-WP (Computer Research Association’s Committee on Widening the Participation in Computing) and a co-founder of the Latinas in Computing group.

Liting Hu received the graduate degree in computer science from the Huazhong University of Science and Technology, China, 2007, and the PhD degree in computer science from the Georgia Institute of Technology, USA, 2016. She conducts experimental computer systems research in the areas of stream processing systems, cloud and edge computing, distributed systems, and systems virtualization. Currently she is an assistant professor with the University of California Santa Cruz.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 08,2023 at 14:16:19 UTC from IEEE Xplore. Restrictions apply.