Enabling Low-Redundancy Proactive Fault Tolerance for Stream Machine Learning via Erasure Coding

Zhinan Cheng[1], Lu Tang[1], Qun Huang[2], Patrick P. C. Lee[1]
[1]*The Chinese University of Hong Kong* [2] *Peking University*
{zncheng, ltang}@cse.cuhk.edu.hk, huangqun@pku.edu.cn, pclee@cse.cuhk.edu.hk

***Abstract*—Machine learning for continuous data streams, or *stream machine learning* in short, is increasingly adopted in real-time big data applications. Fault tolerance is a critical requirement for stream machine learning applications in large-scale distributed deployment. However, existing reactive fault tolerance mechanisms, which trigger failure recovery upon the detection of failures, inevitably incur high recovery overhead and compromise the low-latency requirement of stream machine learning. We design StreamLEC, a stream machine learning system that leverages erasure coding to provide low-redundancy proactive fault tolerance for immediate failure recovery. StreamLEC supports general stream machine learning applications, and incorporates different techniques to mitigate erasure coding overhead. Evaluation on a local cluster and Amazon EC2 shows that StreamLEC achieves much higher throughput than both reactive fault tolerance and replication-based proactive fault tolerance, with negligible failure recovery overhead.**

I. Introduction

Stream machine learning refers to the use of machine learning (i.e., model training and prediction) for continuous streams of data items, and can be viewed as a special use case of stream processing. The demands for stream machine learning are significant in various domains, such as online advertising [24], real-time recommendation [8], anomaly detection [15], and network security [27]. To handle massive amounts of data, stream machine learning should be deployed in a distributed manner as in current distributed stream processing systems [7], [29], [38], [40], in which data streams are processed in parallel across different tasks (called *operators*) that are executed by multiple processes (called *workers*).

Failures are prevalent in distributed environments, as any worker unexpectedly stops working and its operators lose all their states (e.g., model parameters) and the items being processed. Thus, fault tolerance is a critical requirement for stream machine learning. Unlike offline batch processing on a complete dataset (e.g., MapReduce [11]), providing fault tolerance for stream machine learning, or stream processing in general, must deal with the following unique aspects: (i) the system needs to process numerous items that arrive as a continuous data stream, so it is infeasible to track and replay all dependent items for failure recovery; (ii) operators often keep states in main memory for fast processing, but main memory is volatile and subject to data loss in failures; and (iii) fast failure recovery is critical for real-time responses.

Most current stream processing systems supporting stream machine learning adopt *reactive* fault tolerance, by triggering failure recovery upon the detection of failures at both operator and worker levels. A common approach is to periodically issue backups for both states (e.g., via full-state [7], [38], [40] or partial-state [29] checkpointing) and items (e.g., via write-ahead logging [41]). Such backups are persisted in external shared storage (e.g., Hadoop Distributed File System (HDFS) [35]) that can be accessed by a different worker for failure recovery. If a worker fails, a new worker can restore the latest backup state of any operator of the failed worker and replay the items since the latest backup state. However, issuing backups too frequently not only incurs significant disk I/O that disturbs normal performance [18], but also incurs a non-zero recovery latency for retrieving backups from external storage.

Some stream processing systems (e.g., [20], [37]) adopt *replication*, which provides *proactive* fault tolerance by issuing multiple replicas of each item to different workers for concurrent execution. Replication allows trivial failure recovery, as the processing of items remains uninterrupted provided that at least one replica is successfully processed. However, replication is prohibitively expensive, as it multiplies the resource consumption by the number of replicas.

We explore *erasure coding* as a low-redundancy proactive fault tolerance alternative for stream machine learning. Erasure coding can tolerate the same number of failures with significantly less redundancy overhead than replication (see §II-B for details). It has been traditionally used to provide fault tolerance in the areas of communication [34] and distributed storage [14], [16]. Recent studies also explore erasure coding for protecting data analytics against failures, by incorporating erasure coding into caching [32], data shuffling [22], [25], and coded computation [13], [21], [22], [39]. However, most studies (e.g., [13], [22], [25], [39]) only focus on the theoretical performance guarantees with erasure coding. While erasure coding has been empirically evaluated in data analytics platforms (e.g., EC-cache [32] and ParM [21]), such systems are not designed for stream machine learning (see §VI for details). In particular, applying erasure coding to address the unique fault tolerance aspects of stream machine learning (as described above) remains unexplored.

Making erasure coding effective for stream machine learning is non-trivial. First, practical erasure coding constructions (e.g., Reed-Solomon codes [33]) mainly build on the linear operations on data units. However, non-linear operations are common in stream machine learning, implying that any erasure coding solution for stream machine learning must support both linear and non-linear operations for practical concerns. Second,

while the computational overhead of erasure coding is of less concern in storage deployment compared to the more dominant bandwidth and I/O constraints [16], the continuous real-time nature of stream machine learning requires highly efficient coding operations for low-latency responses.

We design StreamLEC, a stream machine learning system with erasure coding to provide low-redundancy proactive fault tolerance for *immediate failure recovery*, such that the processing remains undisturbed even under failures. StreamLEC aims for two primary goals: (i) *generality*, in which it supports general stream machine learning applications comprising both linear and non-linear operations, and (ii) *coding efficiency*, in which it mitigates the computational and communication costs in coding operations. To summarize, this paper makes the following contributions.

- We design an extensible programming model and a streaming workflow for StreamLEC, so as to integrate erasure coding into general stream machine learning applications.
- We propose two techniques for StreamLEC to mitigate erasure coding overhead: (i) *incremental encoding*, which incrementally performs per-item encoding on-the-fly given the streaming items; and (ii) *hybrid coded computation*, which performs coded computation on linear components and normal (uncoded) computation on the non-linear components to mitigate the communication overhead.
- We prototype and evaluate StreamLEC on both a local cluster and Amazon EC2. StreamLEC significantly outperforms both reactive and replication-based fault tolerance approaches (e.g., with throughput gains of 5.17× and 1.55× under double-fault tolerance, respectively), with negligible failure recovery overhead. Our Amazon EC2 experiments show that StreamLEC scales with an increasing number of EC2 instances, while replication cannot scale well due to its high redundancy.

The source code of our StreamLEC prototype is now available at **http://adslab.cse.cuhk.edu.hk/software/streamlec**.

II. Background and Motivation

We justify the limitations of existing reactive fault tolerance approaches in the context of stream machine learning (§II-A). We also present the basics of erasure coding, which we use to provide low-redundancy proactive fault tolerance (§II-B).

A. Limitations in Reactive Fault Tolerance

Existing stream processing systems mostly adopt reactive fault tolerance (§I). We argue that failure recovery can incur substantial performance overhead to stream machine learning. To motivate, we evaluate the recovery overhead of two widely deployed systems: Spark Streaming [40] and Flink [7].

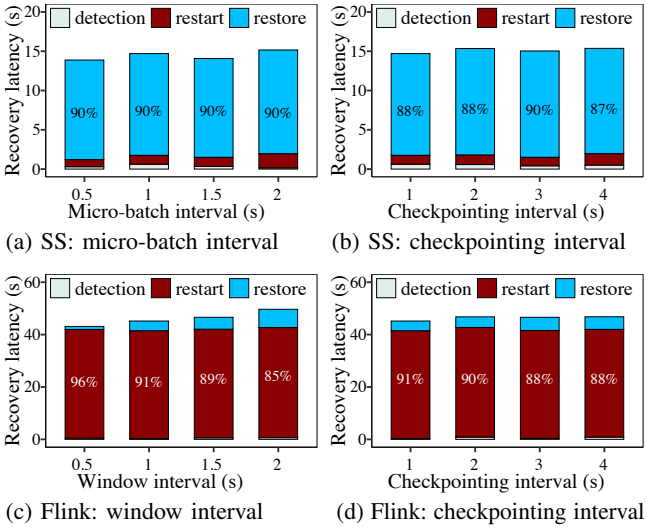
Spark Streaming processes items in units of *micro-batches*, where each micro-batch can be configured to include either a fixed number of items or all items over fixed-length time intervals. To provide fault tolerance, it issues periodic checkpointing for operators' states as Resilient Distributed Dataset (RDDs), and saves processed items in write-ahead logs [41]. Both the state checkpoints and write-ahead logs are maintained in external fault-tolerant storage (e.g., HDFS [35]). To recover from a failed worker, Spark Streaming re-allocates the tasks of the failed worker to other non-failed workers, each of which reconstructs the lost state from the latest checkpoint and replays the logged unprocessed items since the latest checkpoint.

Flink processes items based on different windowing semantics of Google's DataFlow model [7]; in particular, it supports micro-batch processing by splitting a stream of items into non-overlapping time windows (called tumbling windows), each of which resembles a micro-batch. For fault tolerance, Flink realizes state checkpointing by issuing snapshots for the states of all operators at regular time intervals, while the items are assumed to be stored in a persistent and replayable data source. It recovers from a failed worker by reverting the states of all operators from the latest snapshot and replays the unprocessed items from the data source since the latest snapshot.

Frequent checkpointing incurs high I/O overhead and degrades the processing throughput [18] (see also Exp#1 in §V-B). Here, we empirically show that the reactive fault tolerance approaches of Spark Streaming and Flink also incur high recovery overhead when recovering from failures. Recall that Spark Streaming and Flink recover from failures by retrieving the latest backup states of operators and the unprocessed items from external storage. This incurs high recovery latency due to the time needed to restart the processing and the substantial disk I/O introduced by state and item retrievals.

To justify our claim, we evaluate the recovery latencies of Spark Streaming (v2.5.4) and Flink (v1.7.2) under failures by varying their micro-batch intervals and checkpointing intervals, respectively. We conduct evaluation on a cluster of multiple nodes (see §V-A for the cluster setup and datasets). We train a logistic regression model on a data stream generated by the KDD12 dataset. We choose HDFS (v2.6.5) as the external storage system for backups (note that HDFS is also suggested in the official documentations in Spark Streaming [2] and Flink [1]). By default, we set both the micro-batch interval and the checkpointing interval as 1 s. In the midst of stream processing, we terminate a worker node (which mimics a node failure) to trigger recovery operations.

Figure 1 shows the recovery latencies of both systems under failures. For Spark Streaming (Figures 1(a) and 1(b)), the recovery latency is above 13 s in all settings. We further break down the latency into three parts: (i) detection time (i.e., the time to detect a failure), (ii) restart time (i.e., the time to restart failed tasks), and (iii) restore time (i.e., the time to restore the states and unprocessed data items). We find that the high recovery latency is mainly attributed to the high restore time, as Spark Streaming writes streaming items to HDFS and reads them back after restarting the failed tasks. For Flink (Figures 1(c) and 1(d)), the recovery latency is larger than 42 s in all settings. The breakdown of the recovery latency shows that the high recovery overhead is due to the high restart time. The reason is that when a worker fails, Flink needs to stop and restart the processing of all operators to roll back to a consistent condition.



(a) SS: micro-batch interval (b) SS: checkpointing interval

(c) Flink: window interval (d) Flink: checkpointing interval

Figure 1: Recovery latencies in Spark Streaming (SS) and Flink for varying micro-batch intervals and checkpointing intervals.

B. Erasure Coding

Basics. We elaborate the concept of erasure coding in detail. In this work, we construct the erasure code based on the classical Reed-Solomon (RS) codes [33], which have been widely used in production storage systems [14], [16] and are suitable for stream machine learning due to their optimal redundancy and low encoding/decoding time overhead [30]. RS codes are associated with two configurable integer parameters k and r. In the context of stream machine learning, the units for encoding/decoding in RS codes are streaming items. Specifically, for every k uncoded fixed-size streaming items (called the *data items*), RS codes encode them into r coded redundant items (called the *parity items*) of the same size, such that any k out of the $k+r$ data/parity items can reconstruct (or decode) the original k data items. This implies that RS codes provide fault tolerance against the failures of any r items. The collection of the $k+r$ data/parity items is called a *stripe*. The stream machine learning typically processes multiple stripes of items in a data stream, in which the stripes are independently encoded/decoded.

Mathematically, each parity item in RS codes can be computed as a linear combination of the k data items with the encoding coefficients being powers of two. A key property of RS codes is that the redundancy overhead (i.e., $\frac{k+r}{r}\times$ the original data size) is provably minimum for tolerating any r failed items among any erasure code construction (i.e., redundancy-optimal); note that replication incurs a redundancy of $(r+1)\times$ to tolerate r failures. Figure 2 depicts an example of an RS code with $k=2$ and $r=2$, in which the redundancy overhead is $2\times$. In contrast, the redundancy overhead of replication for tolerating $r=2$ failures is $3\times$.

RS codes in stream machine learning. RS codes are traditionally designed for operating in finite field arithmetic [33], as also in storage systems [14], [16]. However, the values operated by stream machine learning are often composed of real numbers [21], [22]. To apply RS codes in stream machine learning, we perform encoding/decoding operations directly on real numbers based on linear algebra. Specifically, each data item is composed of a vector of scalar values, so each parity item is also computed as a vector of scalar values, in which each scalar value is a linear combination of the corresponding scalar values of k data items at the same vector position. The vector of values of the k data items can then be reconstructed by solving a system of k independent linear equations, obtained from any k out of $k+r$ available data/parity items.

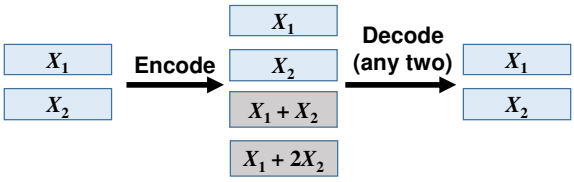


Figure 2: The RS code with $k=2$ and $r=2$, where X_1 and X_2 are data items, and X_1+X_2 and X_1+2X_2 are parity items.

Coded computation. Recent studies [13], [21], [22], [39] explore *coded computation* as a special case of applying erasure coding for fault tolerance in distributed computation. Coded computation applies to linear operations, such that the operation outputs of the k data items can be reconstructed from any k out of the $k+r$ operation outputs of the data/parity items. Take Figure 2 as an example. Let $f(.)$ be a linear function, such that $f(aX+b)=af(X)+b$ for some item X and scalars a and b. Then the outputs of $f(X_1)$ and $f(X_2)$ can be reconstructed from any two of the outputs $f(X_1)$, $f(X_2)$, $f(X_1+X_2)$, and $f(X_1+2X_2)$. However, coded computation cannot directly support non-linear operations, which are commonly found in stream machine learning algorithms. Nevertheless, we explore coded computation as an optimization technique to reduce the communication overhead (§III-D).

III. StreamLEC Design

We present StreamLEC, an erasure-coding-based stream machine learning system that provides low-redundancy proactive fault tolerance for immediate failure recovery. StreamLEC aims for (i) *generality* for various stream machine learning applications that comprise both linear and non-linear operations, and (ii) *coding efficiency* with limited communication and computational overhead incurred to normal processing. In this section, we present the architecture of StreamLEC (§III-A). We design an extensible programming model for general stream machine learning applications (§III-B), and a streaming workflow that performs incremental encoding on continuous streaming items and supports both linear and non-linear operations (§III-C). We further design hybrid coded computation (§III-D) to mitigate erasure coding overhead.

A. Architectural Overview

Types of workers. StreamLEC's architecture (Figure 3) comprises different types of workers, each of which is a long-running process deployed in a physical node. StreamLEC is composed of three stages of workers, namely *sources*, *processors*, and *sinks*:

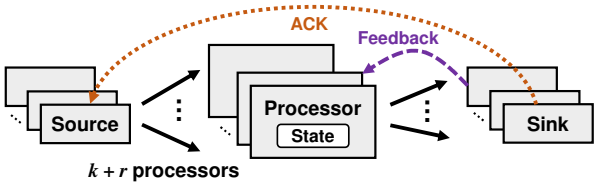


Figure 3: StreamLEC architecture. Each source encodes and emits items to downstream processors, each of which processes the items, updates its state, and emits output items to downstream sinks.

- Each source encodes and distributes a stream of data items to multiple processors. It encodes every group of k data items into r parity items using RS codes (§II-B) and forms a stripe. It distributes the $k+r$ data/parity items of each stripe across $k+r$ processors.
- Each processor receives data or parity items from one or multiple sources. It executes user-defined operators on each received data item. It also keeps an in-memory operator state (e.g., model parameters) for tracking the processing results. It emits outputs to one or multiple sinks.
- Each sink aggregates the outputs from any k out of $k+r$ processors and reconstructs the processing results of the k data items. The aggregated results may be later stored in persistent storage. To support iterative processing (e.g., model training), a sink can send a *feedback* to each of the processors for updating the in-memory states.

Deployment of workers. We deploy each source, processor, and sink as a distinct worker process associated with a unique identifier. We pair up each source (with identifier *sourceID*) and sink (with identifier *sinkID*) for the encoding and decoding of each stripe, respectively. Note that if the source and the sink are colocated in the same server, the topology follows the parameter server paradigm for general distributed machine learning applications [10].

Micro-batches. StreamLEC processes items in units of micro-batches as in Spark Streaming [40] (§II-A) and other stream processing systems [38]. The micro-batch setting allows StreamLEC to readily support model training, which often requires synchronization among processors due to the iterative-convergent feature of machine learning [6]. Specifically, each micro-batch typically comprises multiple groups of k data items and hence forms different stripes of $k+r$ data/parity items. Without loss of generality, we assume that the micro-batch size is a multiple of k. After a source emits all data/parity items for a micro-batch, it waits for an *acknowledgment (ACK)* from the sink, which returns the ACK after reconstructing all processing results for the micro-batch.

Item representation. Each item is represented as a *key-value* pair. A key corresponds to a unique identifier for an item. We associate each item's key with four fields: (i) the source identifier *sourceID*, (ii) the sink identifier *sinkID*, (iii) the stripe identifier *stripeID*, which identifies the stripe to which the item belongs, and (iv) the stripe index *sIndex*, which specifies the index of an item in a stripe (from 0 to $k+r-1$). All $k+r$ data/parity items of the same stripe have the same *stripeID*. Without loss of generality, we assume that each of the k data items has the *sIndex* from 0 to $k-1$, and each of the r parity items has the *sIndex* from k to $k+r-1$. Each item's value is a vector of multiple numerical values (called *attributes*). StreamLEC performs encoding/decoding operations on the values of items (i.e., the vectors of attributes) (§II-B).

State recovery. StreamLEC recovers the in-memory state of a failed processor from the feedback of the sink. After restarting a failed processor, the processor restores its state using the feedback of the sink at the end of processing a micro-batch. Note that the sink can always provide the latest state (e.g., the newest model parameters) for the failed processor, as it continues the processing with the results from other non-failed processors.

Design assumptions. Each source provides persistent and replayable storage for the items as in Flink [7] (§II-A). If any source or sink fails, or if there exist more than r processor failures (i.e., beyond the protection of erasure coding), we can repair the failed workers and resume the processing of the current micro-batch using the persisted data items. Thus, our current analysis assumes that each source and sink is reliable and there are at most r processor failures. Under this assumption, StreamLEC provides proactive fault tolerance via erasure coding without accessing the persistent storage in general cases.

B. Programming Model

We design a programming model for StreamLEC to support general stream machine learning applications based on different programming interfaces, as shown in Listing 1.

StreamLEC manages a number of objects of different data types: (i) a data/parity item (of type `Item`), (ii) a processing result of a processor (of type `Result`), (iii) a feedback from a sink (of type `Feedback`), and (iv) an ACK from a sink (of type `ACK`). All objects inherit from the base data type `Message`, which defines a message to be exchanged among the workers.

StreamLEC defines two *communication interfaces* to construct the message workflow among the workers: (i) `Emit`, which allows a worker to emit a message to another worker identified by the *workerID*; and (ii) `Recv`, which allows a worker to receive a message from another worker.

StreamLEC further defines a set of *user-defined interfaces*, which are extensible and allow programmers to add implementation details for specific machine learning applications. We summarize the user-defined interfaces as follows: (i) `Encode` and `Decode`, which realize encoding and decoding operations for an erasure code, respectively; (ii) `ProcessData`, `ProcessFeedback`, and `ProcessACK`, which process the input data item, feedback, and ACK, respectively; (iii) `ProcessLinear` and `ProcessNonLinear`, which process the linear and non-linear operations for hybrid coded computation, respectively; and (iv) `Aggregate` and `Recompute`, which are called by a sink to commit a processing result and recompute the processing result of a decoded data item, respectively. The following subsections elaborate the use of the user-defined interfaces in detail.

```
/***** Communication interfaces *****/
void Emit(int workerID, Message& msg);
Message Recv(int workerID);

/***** User-defined interfaces *****/
/* Called by a source */
void Encode(int sIndex, Item& data, vector<Item> parity);
void ProcessACK(ACK& ack);

/* Called by a processor */
Result ProcessData(Item& data);
Feedback ProcessFeedback(Feedback& feedback);
Result ProcessLinear(Item& item);
Result ProcessNonLinear(Result& linear, Item& data);

/* Called by a sink */
void Decode(vector<Item> received, vector<Item> decoded);
void Aggregate(Result& result);
Result Recompute(Item& item);
```

Listing 1: Programming interfaces of StreamLEC.

C. Streaming Workflow

We describe the streaming workflow of StreamLEC, including: (i) the encoding workflow executed by a source, (ii) the processing workflow executed by a processor, and (iii) the decoding workflow executed by a sink.

Encoding workflow. Recall that StreamLEC processes items on a per-micro-batch basis. However, encoding an entire micro-batch can incur substantial computational overhead and hence high processing latency. To achieve low-latency encoding, StreamLEC performs *incremental encoding*, which computes a parity item on a per-data-item basis. Our insight is that each parity item is a linear combination of k data items (§II-B), in which addition operations are associative. Specifically, consider a parity item $Y = a_0X_0 + a_1X_1 + a_2X_2$, where $k = 3$, X_0, X_1, and X_2 are the data items generated in order by the source, and a_0, a_1, and a_2 are the corresponding encoding coefficients. When X_0 is available, the source first updates the parity item as $Y' = a_0X_0$; when X_1 is available, the source updates the parity item as $Y'' = Y' + a_1X_1$; finally, when X_2 is available, the source computes the final parity item $Y = Y'' + a_2X_2$. After the source updates the parity item with a data item, it can send the data item to one of the processors. Thus, it pipelines the encoding operations and the transmissions of data items.

Algorithm 1 shows the encoding workflow of a source. The source defines a counter *Count* for the *stripeID* and *sIndex* assignments and a set of r parity items $\{Y_0, Y_1, \cdots Y_{r-1}\}$ for encoding operations (both variables are initialized from zeros). For each data item X, the source first specifies its *stripeID* and *sIndex* using *Count*, which is incremented by one after the assignments (Lines 1-3). It specifies X's key and emits X to a processor (Lines 4-5). While X is in-transit, the source calls the user-defined `Encode` function and updates the parity items on the per-data-item basis (Line 6). If the r parity items are ready (i.e., all k data items of a stripe are encoded), the source sends them to the remaining r processors (Lines 7-13). Finally, if the source has processed a micro-batch, it waits for an ACK from the sink and processes the ACK accordingly via the user-defined `ProcessACK` interface (e.g., preparing for processing the next micro-batch) (Lines 14-17).

Algorithm 1 Encoding workflow of a source

Variable: Counter *Count* (initialized from zero)
Variable: Parity items $Y_0, Y_1, \cdots, Y_{r-1}$ (initialized from zero)
Input: data item X
1: *stripeID* $\leftarrow \lfloor Count/k \rfloor$
2: *sIndex* $\leftarrow$ *Count* mod k
3: *Count*++
4: Add {*sourceID*, *sinkID*, *stripeID*, *sIndex*} to X's key
5: `Emit`(*pid[sIndex]*, X) ▷ Emit X to worker *pid[sIndex]*
6: `Encode`(*sIndex*, X, $\{Y_0, Y_1, \cdots, Y_{r-1}\}$)
7: **if** *sIndex* $= k-1$ **then** ▷ k data items are encoded
8: **for** $i = 0$ to $r-1$ **do**
9: Add {*sourceID*, *sinkID*, *stripeID*, $k+i$} to Y_i's key ▷ $k+i$ is the stripe index of Y_i
10: `Emit`(*pid[k+i]*, Y_i) ▷ Emit Y_i to worker *pid[k+i]*
11: **end for**
12: Reset $\{Y_0, Y_1, \cdots, Y_{r-1}\}$ to zero
13: **end if**
14: **if** X is the last item of a micro-batch **then**
15: $A \leftarrow$ `Recv`(`sinkID`) ▷ A is the received ACK from a sink
16: `ProcessACK`(A)
17: **end if**

Algorithm 2 Processing workflow of a processor

Input: item X (which is either a data item or a parity item)
1: Identify the *sinkID* from X's key
2: **if** X is a data item **then**
3: $R \leftarrow$ `ProcessData`(X) ▷ R is the returned result
4: `Emit`(*sinkID*, $\langle R, X \rangle$) ▷ Emit $\langle R, X \rangle$ to worker *sinkID*
5: **else** ▷ X is a parity item
6: `Emit`(*sinkID*, X) ▷ Emit X to worker *sinkID*
7: **end if**
8: **if** X is from the last stripe of a micro-batch **then**
9: $F \leftarrow$ `Recv`(*sinkID*) ▷ F is the received feedback from a sink
10: `ProcessFeedback`(F)
11: **end if**

Processing workflow. Each processor may receive a data or parity item from a source. If the processor receives a data item, it generates the processing result on the data item. In addition to emitting the processing result to a sink, the processor also attaches the input data item into the emitted outputs. If the processor receives a parity item, it directly emits the parity item to the sink. Thus, the sink now receives not only the processing result, but also the data/parity items of each stripe for reconstructing any unavailable data items.

The benefits of the processing workflow are two-fold. First, it allows immediate failure recovery, as the sink can recover any lost data items as long as it receives at least k data/parity items of a stripe. Second, it is applicable to general stream machine learning applications composed of both linear and non-linear operations. On the other hand, as each processor now attaches a data item in its emitted outputs, the communication overhead increases. In §III-D, we show how we can mitigate the overhead via hybrid coded computation.

Algorithm 2 shows the general workflow of a processor. Upon receiving an item X from a source, the processor first identifies the worker ID (denoted by *sinkID*) of the sink to which the output is emitted (Line 1). If X is a data item, the processor calls the user-defined interface `ProcessData` on item X to produce the processing result (denoted by R), and emits both R and X to the sink (Lines 2-4); otherwise,

Algorithm 3 Decoding workflow of a sink

```
Input: Received message M from a processor
 1: if M = ⟨R,X⟩ then                          ▷ M has result R and data item X
 2:     Buffer the data item X
 3:     Aggregate(R)
 4: else if M = ⟨X⟩ then                       ▷ M has a parity item X
 5:     Buffer the parity item from M
 6: end if
 7: if k items are received for a stripe then
 8:     if some data items are unavailable then
 9:         b ← set of k received items of the stripe
10:         d ← set of decoded data items
11:         Decode(b, d)
12:         for each decoded data item X' ∈ d do
13:             R ← Recompute(X')
14:             Aggregate(R)
15:         end for
16:     end if
17: end if
18: if all results are aggregated for a micro-batch then
19:     for each upstream processor with worker ID pid do
20:         Emit(pid, F)                      ▷ F is a feedback sent to each processor
21:     end for
22:     Emit(sourceID, A)                     ▷ A is an ACK sent to a source
23: end if
```

if X is a parity item, the processor directly emits X to the sink (Lines 5-6). Finally, if X is from the last stripe of a micro-batch, it implies that the processor receives the last item of the micro-batch. In this case, the processor waits for the feedback F from the sink. It then processes F via the user-defined interface `ProcessFeedback` (e.g., updates the internal in-memory state) (Lines 8-11). Note that the time to wait for F is negligible even under failures, due to the proactive fault tolerance of StreamLEC.

Decoding workflow. Algorithm 3 shows the decoding workflow of a sink. Given the input message from a processor, the sink buffers the received data/parity item, and commits the processing result (if a data item is received) via the user-defined `Aggregate` function (Lines 1-6). If the sink receives k items of a stripe, it checks whether some data items remain unavailable; if so, it decodes the unavailable data items via the user-defined `Decode` function and re-processes the decoded data item via the user-defined `Recompute` function (Lines 7-17). Note that the sink performs a decoding operation only when some data items of a stripe are unavailable, so the decoding overhead over the entire streaming workflow is generally limited. Finally, after a sink has aggregated all results of a micro-batch, it sends a feedback to each upstream processor and sends an ACK to the source (Lines 18-23).

Discussion. In the processing workflow, we require the source to send the parity items to the sink through the processors, even though the processors do not perform any computation on the parity items. The reasons are three-fold. First, it reduces the synchronization complexity between the source and sink. Second, it enables us to flexibly control the emitting time of a parity item to avoid the situation where the parity items always arrive before the normal results and trigger the re-computation procedure at the sink. Finally, it provides viable opportunities for applying coded computation in processors to mitigate the communication overhead of workers (§III-D).

By incorporating erasure coding into the streaming workflow, StreamLEC provides proactive fault tolerance with low redundancy and fast failure recovery. Compared to replication, StreamLEC can tolerate the same number of failures with significantly less redundancy overhead. Also, StreamLEC prevents expensive disk I/O of accessing the states and items in the streaming workflow, thereby improving the failure recovery performance over reactive fault tolerance. In the following subsection, we introduce hybrid coded computation to mitigate the communication overhead of erasure coding to make StreamLEC more efficient.

D. Hybrid Coded Computation

Instead of always attaching a data item in a processor's emitted outputs (§III-C), StreamLEC can incorporate coded computation [22] into linear operations, such that each processor sends only its processing result on a data/parity item to a sink without attaching the data item, while the processing results of k data items can still be reconstructed from any k out of the $k+r$ processing results of data/parity items (§II-B). A challenge is that stream machine learning applications may comprise both linear and non-linear operations, making the direct use of coded computation infeasible. Thus, StreamLEC adopts *hybrid coded computation*, which performs coded computation on linear operations, while keeping the normal computation for non-linear operations as in §III-C.

Algorithm details. Suppose that a stream machine learning application can be decomposed into the linear and non-linear components, such that the linear component runs before the non-linear component and both components can be realized by programmers; the assumption is well satisfied by state-of-the-art machine learning algorithms, such as logistic regression and neural networks [5]. Under this assumption, Algorithm 4 shows the pseudo-code of hybrid coded computation performed by a processor, by extending the processing workflow in Algorithm 2. It builds on two user-defined interfaces, namely `ProcessLinear` and `ProcessNonLinear`, which correspond to the linear and non-linear operations, respectively. After identifying the *sinkID* from X's key (Line 1), the processor first computes the result R_L of linear operations via `ProcessLinear` (Line 2). If X is a data item, it further computes the result R_N of non-linear operations via `ProcessNonLinear` using the inputs of both R_L and X (Lines 3-5). It then emits both R_L and R_N (Line 6). Finally, at the end of processing a micro-batch, the processor waits for the feedback from the sink (Lines 7-10), as in the original processing workflow (Algorithm 2).

When a failure happens, a sink now decodes the k linear results from any k out of $k+r$ processors via coded computation, and recomputes the unavailable non-linear results from the decoded linear results. We extend Algorithm 3 for the decoding workflow for hybrid coded computation by substituting the result R and the data/parity item X with R_N and R_L, respectively.

Note that Algorithm 4 is a generalized version of the processing workflow in §III-C. It reduces to Algorithm 2 (detailed in §III-C) if `ProcessLinear` returns the input item X and `ProcessNonLinear` returns the same result as `ProcessData`.

Algorithm 4 Hybrid coded computation

```
Input: item X (which is either a data item or a parity item)
Identify the sinkID from X's key
R_L ← ProcessLinear(X)                          ▷ R_L is linear result
if X is a data item then
    R_N ← ProcessNonLinear(R_L, X)              ▷ R_N is non-linear result
end if
Emit(sinkID, ⟨R_N, R_L⟩)                        ▷ R_N is empty if X is a parity item
if X is from the last stripe of a micro-batch then
    F ← Recv(sinkID)                            ▷ F is the received feedback from a sink
    ProcessFeedback(F)
end if
```

```
Variable: Vector model
function ProcessLinear(Item X)
    Extract value V from X                      ▷ V is a vector of attributes
    Compute the dot product R_L ← V · model
    return R_L
end function
function ProcessNonLinear(Result R_L, Item X)
    Compute prediction R_N ← 1.0/(1.0 + exp(−R))
    return R_N
end function
```

Figure 4: Logistic regression prediction with hybrid coded computation.

StreamLEC currently requires users to reason about the linear and non-linear components in their computations. Our future work is to extend StreamLEC to support automated splitting of user computations.

Figure 4 shows an example of the hybrid coded computation for logistic regression prediction. We implement the linear operator of logistic regression prediction in `ProcessLinear`, which computes the dot product of the input item's value and the vector of model parameters (i.e., *model*). We also implement the non-linear operator of logistic regression in `ProcessNonLinear`, which computes the final prediction result using the dot product as input. This example also shows how the non-linear operator can build on the results of the linear operator to simplify the implementation.

Hybrid coded computation now emits the results of linear and non-linear operations (i.e., R_L and R_N, respectively) instead of attaching a data item. It thus reduces the communication traffic to the sink as R_L and R_N often represent scalar values (e.g., Figure 4), while a data item comprises a vector of attributes. In the extreme case where an application comprises purely linear operations, each processor only sends the result R_L of linear operations to a sink, without including the result R_N of non-linear operations. This further reduces the communication traffic to the sink.

IV. Implementation

We have implemented a prototype of StreamLEC in C++ on Linux, with around 19,000 lines of code.

Worker communication. Our prototype uses the ZeroMQ messaging library [3] for worker communication. Each worker contains a main thread for computation, as well as two communication threads for managing the TCP connections with the upstream and downstream workers based on ZeroMQ. The threads within each worker exchange data via lock-free ring buffers [23]. Each source or sink also supports non-blocking communication to bypass any failed processor.

SIMD optimization. Recall that each item's value comprises a vector of attributes. Instead of performing encoding/decoding operations on attributes individually, we leverage Single Instruction Multiple Data (SIMD) instructions to parallelize the encoding/decoding operations. Specifically, each of the attributes in an item's value is a 64-bit floating-point number. We use the AVX2 256-bit instruction set to perform encoding/decoding operations on every four 64-bit attributes in parallel. The SIMD optimization further mitigates the computational overhead due to erasure coding.

V. Evaluation

We evaluate StreamLEC on both a local cluster (§V-B) and Amazon EC2 (§V-C). Our major findings on StreamLEC include: (i) it achieves much higher throughput than both reactive fault tolerance and replication; (ii) it incurs negligible recovery overhead; (iii) it mitigates the computation and communication costs of erasure coding; and (iv) it achieves scalable performance on Amazon EC2.

A. Methodology

Datasets. We consider two real-world datasets, namely (i) *KDD12* [28] and (ii) *HIGGS* [4]. The KDD12 dataset contains items from KDD Cup 2012 and models click-through rate prediction in production. Each item has 11 attributes. The HIGGS dataset contains items for classification in high-energy physics experiments. Each item has 24 attributes. For each of the datasets, we select the first 10 M items for evaluation.

Algorithms. We consider four stream machine learning algorithms: (i) *linear regression*, (ii) *logistic regression*, (iii) *support vector machine (SVM)*, and (iv) *K-means*. We adapt their implementations in MLlib [26] into our prototype. Take logistic regression as an example. Each processor computes the gradient for each received item via stochastic gradient descent [6]. It then sends the gradient result to a sink. The sink aggregates the gradient results of a micro-batch from its upstream processors and updates the model parameters by minimizing the logistic loss function with L2-regularization. It also feedbacks the model parameters to its upstream processors. Other algorithms have similar workflows. All algorithms involve non-linear operations in the processors and sinks.

Schemes. We compare StreamLEC's erasure coding with reactive fault tolerance and replication. For fair comparisons, we implement all schemes under the StreamLEC prototype and allocate them with the same resources in our evaluation. For reactive fault tolerance and replication, we also assume that the sources and sinks are reliable and focus on the fault tolerance for the processors.

For reactive fault tolerance (denoted by *Reactive*), we re-implement Spark Streaming's approach [40] (§II-A). Each processor checkpoints any received item and its state to HDFS (v2.6.5) [35]. If a processor fails, we restore the state and reprocess the lost items from HDFS in a new processor.

For replication, we partition all processors into multiple groups of w processors, where w denotes the number of replicas being configured (assuming that the total number of processors is divisible by w). For each data item, a source randomly selects a group of processors, and issues w replicas to the w processors in the group. We consider replication with $w = 2$ for single-fault tolerance and $w = 3$ for double-fault tolerance (denoted by *Rep-2x* and *Rep-3x*, respectively).

For StreamLEC, we evaluate different erasure coding configurations by varying k and r (denoted by *EC(k,r)*). EC(k,r) distributes data/parity items to $k+r$ processors and tolerates r processor failures; $r = 0$ implies no fault tolerance.

Default setting. For each dataset, we set the micro-batch size as 100 K items (i.e., 100 micro-batches in total). Note that StreamLEC's performance remains similar for different micro-batch sizes. To exclude the I/O overhead on performance, we load all datasets into memory prior to all experiments. We plot the average results over 10 runs, including the error bars showing the 95% confidence interval based on the student's t-distribution.

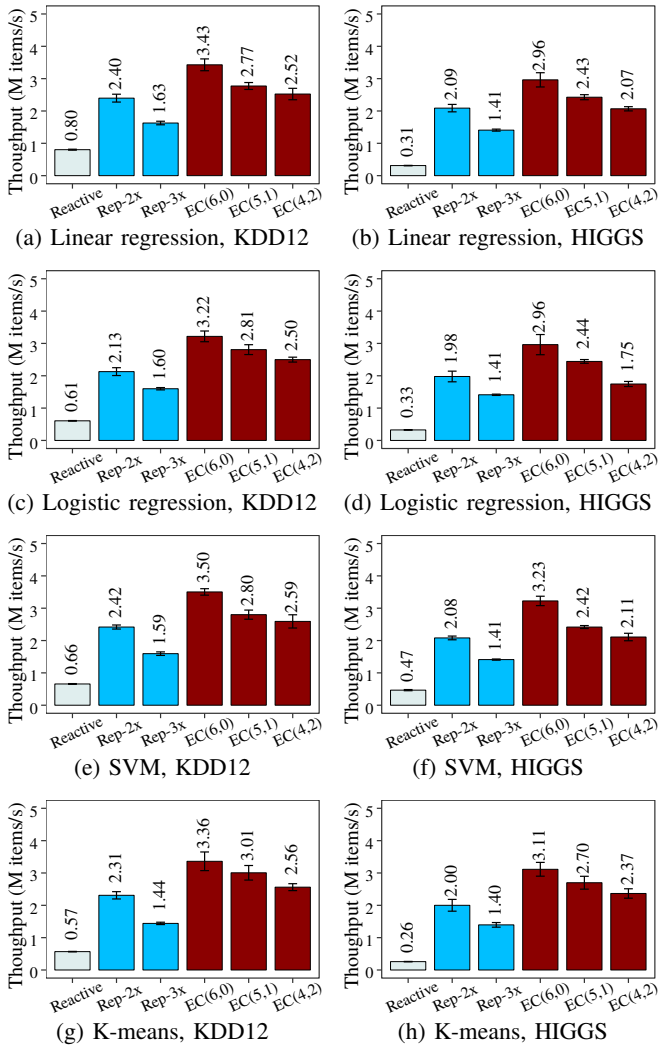
B. Local Cluster Experiments

Our local cluster comprises eight machines, equipped with an Intel Core i5-7500 3.40 GHz quad-core CPU, 32 GB RAM, and a TOSHIBA DT01ACA100 7200 RPM 1 TB SATA disk. All machines run Ubuntu 16.04 LTS and are connected via a 10 Gb/s Ethernet switch. We deploy one source, six processors, and one sink in distinct machines. For StreamLEC, we consider EC(6,0) (no fault tolerance), EC(5,1) (single-fault tolerance), and EC(4,2) (double-fault tolerance).

Exp#1 (Throughput in normal mode). Figure 5 shows the throughput of different algorithms and fault tolerance schemes in normal mode (i.e., no failure). We observe a similar performance pattern in all cases. Reactive has the lowest throughput, as it incurs heavy I/Os for issuing state and item backups to HDFS. Compared to the no-redundancy case (i.e., EC(6,0)), replication incurs higher throughput drops than erasure coding due to its higher redundancy. Considering the average over the eight cases in Figure 5, for single-fault tolerance, EC(5,1) achieves 6.07× and 1.23× throughput compared to Reactive and Rep-2x, respectively; for double-fault tolerance, EC(4,2) achieves 5.17× and 1.55× throughput compared to Reactive and Rep-3x, respectively.

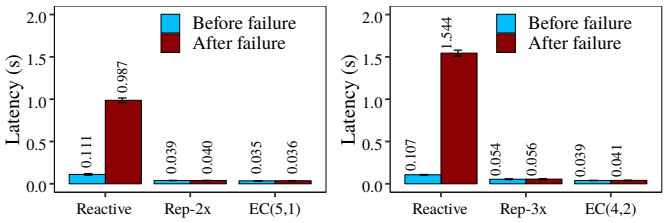
Exp#2 (Failure recovery). We study the performance impact of both single-fault and double-fault recovery cases. We stop one or two of the processors in the midst of processing a micro-batch, and restart the failed processors in the same machines. We measure the processing latencies of the two micro-batches right before and after failure recovery. We focus on logistic regression on KDD12.

Figure 6(a) shows the latencies of EC(5,1), Rep-2x, and Reactive in single-fault recovery, while Figure 6(b) shows the latencies of EC(4,2), Rep-3x, and Reactive in double-fault recovery. Both StreamLEC (i.e., EC(5,1) and EC(4,2)) and replication (i.e., Rep-2x and Rep-3x) have negligible latency differences before and after failure recovery. However,



(a) Linear regression, KDD12 (b) Linear regression, HIGGS
(c) Logistic regression, KDD12 (d) Logistic regression, HIGGS
(e) SVM, KDD12 (f) SVM, HIGGS
(g) K-means, KDD12 (h) K-means, HIGGS

Figure 5: Exp#1: Throughput on normal mode.

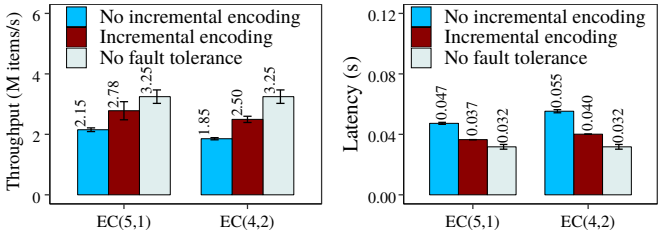


(a) Single failure (b) Two failures

Figure 6: Exp#2: Failure recovery.

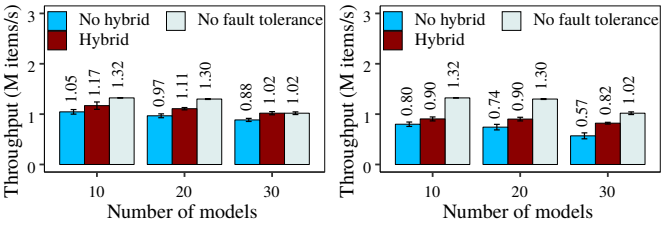
the processing latency of Reactive incurs 8.9× and 14× increases after single-fault recovery and double-fault recovery, respectively. We break down the recovery latency of Reactive and observe that the latency is mainly contributed by the processor restarting and data/state restoring (e.g., 27.4% and 61.8% in a single failure, respectively).

Exp#3 (Incremental encoding). We study incremental encoding in StreamLEC. For comparison, we consider a baseline that disables incremental encoding and forces StreamLEC to perform encoding until all data items of a micro-batch are



(a) Throughput in normal mode (b) Processing latency

Figure 7: Exp#3: Incremental encoding.



(a) EC(5,1) (b) EC(4,2)

Figure 8: Exp#4: Hybrid coded computation.

available. We focus on logistic regression on KDD12.

Figure 7 shows the results. Compared to no fault tolerance, enabling incremental encoding reduces the throughput drops of EC(5,1) and EC(4,2) from 33.9% to 14.5% and from 43.4% to 23.1% (Figure 7(a)), respectively, since it allows a source to emit items and perform encoding in parallel. It also reduces the processing latency of each micro-batch by 22.8% and 27.5% in EC(5,1) and EC(4,2), respectively (Figure 7(b)).

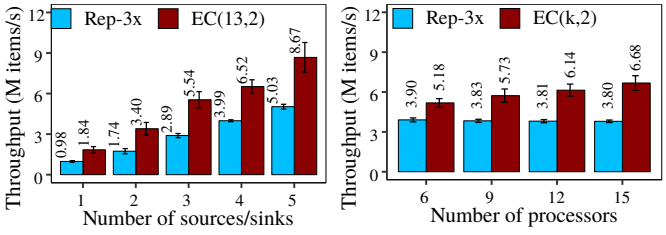
Exp#4 (Hybrid coded computation). We study hybrid coded computation. We focus on logistic regression prediction (Figure 4) on a synthetic dataset, in which each data item has 100 attributes. Note that we generate a synthetic dataset via the `make_classification` function in the `sklearn` toolkit [36]. Here, we consider a scenario where the communication from all processors to the sink is the bottleneck, and vary the number of logistic regression models used for prediction. For each model, each processor outputs a linear result instead of attaching the input item under hybrid coded computation.

Figure 8 shows the results. Hybrid coded computation shows an increasing throughput gain as the number of models increases, for example, from 1.13× for 10 models to 1.44× for 30 models in EC(4,2). The reason is that hybrid coded computation reduces the processor-to-sink communication overhead, which becomes more significant as the number of models increases.

C. Amazon EC2 Experiments

We evaluate StreamLEC on Amazon EC2. We deploy up to 25 EC2 instances of type `m5.2xlarge` in the `us-west-1a` zone. Each instance has eight vCPUs and 32 GB RAM. All instances are connected via a 10 Gb/s network.

Exp#5 (Scalability on Amazon EC2). We study StreamLEC's scalability in two aspects: (i) varying the number of source/sink pairs from one to five and fixing $k = 13$ and $r = 2$; and (ii) varying k from 4 to 13, while fixing $r = 2$ and four source/sink pairs. We compare StreamLEC with Rep-3x and focus on



(a) Varying # of sources/sinks (b) Varying # of processors

Figure 9: Exp#5: Scalability on Amazon EC2.

logistic regression on KDD12. For both cases, each worker runs in a distinct EC2 instance. Figure 9 shows the throughput results in normal mode. Both StreamLEC and Rep-3x scale linearly with the number of source/sink pairs (Figure 9(a)), while StreamLEC achieves an average throughput gain of 1.82× over Rep-3x. However, Rep-3x cannot scale its throughput even if the number of processors increases (Figure 9(b)), since its high redundancy incurs high communication overhead in the source/sink pairs. In contrast, StreamLEC's throughput increases with the number of processors.

VI. Related Work

Erasure coding in data analytics. Some studies exploit erasure coding to improve data analytics performance. Coded data shuffling [22] and coded MapReduce [25] broadcast coded data to multiple compute nodes, each of which then decodes its own required data for processing. This reduces the communication cost compared to uncoded broadcast. EC-cache [32] caches coded data in memory to achieve low-latency and load-balanced data analytics.

Coded computation can be traced back to algorithm-based fault tolerance (ABFT) [17], which uses erasure coding for tolerating hardware faults in matrix operations. Several theoretical studies analyze how coded computation addresses failures in linear computation and matrix multiplication [12], [13], [22], [39]. Lee *et al.* [22] formally prove the performance speedups of coded computation over uncoded computation. Such analysis is also applied for high-dimensional vectors [13] and heterogeneous clusters [39]. Some studies [12], [39] also propose new erasure codes for coded computation.

ParM [21] implements coded computation in prediction serving systems and supports non-linear operations. It uses machine learning to train the proper erasure codes that minimize the errors of prediction outputs obtained from decoding. Note that ParM is designed for machine learning inference and cannot support stream-based model training, while StreamLEC targets both training and inference of stream machine learning and particularly mitigates erasure coding overhead.

Fault tolerance in stream processing. Current stream processing systems [7], [31], [40] achieve fault tolerance via state checkpointing and upstream backup [19]. To mitigate backup overhead, AF-Stream [18] issues backups only when the errors upon failures exceed pre-defined thresholds, and is shown to preserve the model convergence of stream machine learning [9]. Drizzle [38] decouples the time intervals for

normal processing and fault tolerance coordination to reduce the overhead of fault-tolerance maintenance. Samza [29] issues partial-state checkpointing to reduce the overhead of full-state checkpointing. All the above approaches are based on reactive fault tolerance and incur non-negligible recovery delays.

Some stream processing systems (e.g., Borealis [20] and PPA [37]) use replication for proactive fault tolerance, yet replication multiplies the resource usage and is non-scalable (§V-C). In contrast, StreamLEC leverages erasure coding to achieve low-redundancy proactive fault tolerance.

VII. Conclusion

Erasure coding is traditionally used in communication and storage. We make a case of applying erasure coding into stream machine learning via StreamLEC, so as to provide low-redundancy proactive fault tolerance and allow immediate failure recovery. StreamLEC supports general stream machine learning algorithms and achieves efficient coding. Experiments demonstrate that StreamLEC achieves high throughput in normal mode and incurs negligible failure recovery overhead.

Acknowledgments. This work was support in part by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Research Grants Council of Hong Kong (AoE/P-404/18), Joint Funds of the National Natural Science Foundation of China (U20A20179), and National Natural Science Foundation of China (61802365). The corresponding author is Lu Tang (ltang@cse.cuhk.edu.hk).

References

[1] Apache flink documentation - state & fault tolerance. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/state/.

[2] Spark Streaming Programming Guide. https://spark.apache.org/docs/2.4.5/streaming-programming-guide.html.

[3] ZeroMQ. http://zeromq.org.

[4] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308, 2014.

[5] G. Bebis and M. Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4), 1994.

[6] L. Bottou. Online algorithms and stochastic approximations. In *Online Learning and Neural Networks*. 1998.

[7] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4), 2015.

[8] S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang. Streaming Recommender Systems. In *Proc. of ACM WWW*, 2017.

[9] Z. Cheng, Q. Huang, and P. P. C. Lee. On the performance and convergence of distributed stream processing via approximate fault tolerance. *The VLDB Journal*, 28(5):821–846, Oct 2019.

[10] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *Proc. of AAAI*, 2015.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data procesing on large clusters. In *Proc. of USENIX OSDI*, 2004.

[12] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover. A unified coded deep neural network training strategy based on generalized polydot codes. In *Proc. of IEEE ISIT*, 2018.

[13] S. Dutta, V. Cadambe, and P. Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Proc. of NeurIPS*, 2016.

[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[15] S. Guha, N. Mishra, G. Roy, and O. Schrijvers. Robust random cut forest based anomaly detection on streams. In *Proc. of ICML*, 2016.

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[17] K.-H. Huang and A. Jacob. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. on Computers*, 100(6), 1984.

[18] Q. Huang and P. P. C. Lee. Toward high-performance distributed stream processing via approximate fault tolerance. *Proc. of the VLDB Endowment*, 10(3), 2016.

[19] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of IEEE ICDE*, 2005.

[20] J.-H. Hwang, S. Cha, U. Cetintemel, and S. Zdonik. Borealis-r: A replication-transparent stream processing system for wide-area monitoring applications. In *Proc. of ACM SIGMOD*, 2008.

[21] J. Kosaian, K. Rashmi, and S. Venkataraman. Parity models: A general framework for coding-based resilience in ml inference. In *Proc. of ACM SOSP*, 2019.

[22] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. Speeding up distributed machine learning using codes. *IEEE Trans. on Information Theory*, 64(3), 2018.

[23] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Proc. of IEEE IPDPS*, 2010.

[24] C. Li, Y. Lu, Q. Mei, D. Wang, and S. Pandey. Click-through prediction for advertising in twitter timeline. In *Proc. of ACM SIGKDD*, 2015.

[25] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr. Coded mapreduce. In *Proc. of IEEE Allerton*, 2015.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(1), 2016.

[27] P. Mulinka and P. Casas. Stream-based machine learning for network security and anomaly detection. In *Proc. of Big Data Analytics and Machine Learning for Data Communication Networks*, 2018.

[28] Y. Niu, Y. Wang, G. Sun, A. Yue, B. Dalessandro, C. Perlich, and B. Hamner. The tencent dataset and kdd-cup'12, 2012.

[29] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. of the VLDB Endowment*, 10(12), 2017.

[30] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[31] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proc. of ACM EuroSys*, 2013.

[32] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, 2016.

[33] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of Society for Industrial and Applied Mathematics*, 8(2), 1960.

[34] T. Richardson and R. Urbanke. *Modern coding theory*. Cambridge university press, 2008.

[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of IEEE MSST*, 2010.

[36] Sklearn datasets. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html.

[37] L. Su and Y. Zhou. Passive and partially active fault tolerance for massively parallel stream processing engines. *IEEE Trans. on Knowledge and Data Engineering*, 31(1), 2019.

[38] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proc. of ACM SOSP*, 2017.

[39] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr. Lagrange coded computing: Optimal design for resiliency, security, and privacy. In *Proc. AISTAT*, 2019.

[40] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of ACM SOSP*, 2013.

[41] Zero Data Loss in Spark Streaming. http://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html.