Tolerating Transient Late-timing Faults in Cloud-based Real-time Stream Processing

Peter Garraghan, Stuart Perks, Xue Ouyang, David McKee, Ismael Solis Moreno*

School of Computing
University of Leeds
Leeds, UK
{p.m.garraghan, sc14sp, scxo, scdwm}@leeds.ac.uk

*Advanced Research Center
CIATEQ
Santiago de Querétaro, Mexico
ismael.solis@ciateq.mx

Abstract — **Real-time stream processing is a frequently deployed application within Cloud datacenters that is required to provision high levels of performance and reliability. Numerous fault-tolerant approaches have been proposed to effectively achieve this objective in the presence of crash failures. However, such systems struggle with transient late-timing faults – a fault classification challenging to effectively tolerate – that manifests increasingly within large-scale distributed systems. Such faults represent a significant threat towards minimizing soft real-time execution of streaming applications in the presence of failures. This work proposes a fault-tolerant approach for QoS-aware data prediction to tolerate transient late-timing faults. The approach is capable of determining the most effective data prediction algorithm for imposed QoS constraints on a failed stream processor at run-time. We integrated our approach into Apache Storm with experiment results showing its ability to minimize stream processor end-to-end execution time by 61% compared to other fault-tolerant approaches. The approach incurs 12% additional CPU utilization while reducing network usage by 44%.**

Keywords- Fault-tolerance, Stream Processing, Data Prediction, Cloud computing.

I. INTRODUCTION

Huge surges in data generation and consumption globally has resulted in a rapid increase of both data volume and velocity – two key characteristics of Big Data – exploited to address societal, technologic, and scientific needs. A popular application that exploits these characteristics are real-time data stream processing systems. These systems are capable of processing single or multiple data sources in motion through multiple data processing nodes to fulfill timing requirements of users. Such systems perform filtration or aggregation of large volumes of data in real-time, and are used in a plethora of application domains including social media [1], Cloud datacenter monitoring [2], security [3] and the Internet of Things (IoT).

Fault-tolerance – a means to attain dependability – is an important approach towards achieving reliable streaming [4]. However, its effectiveness is directly threatened by the system's requirement for massive scalability in response to Big Data. Specifically, the increased demand of voluminous and high-velocity data results in increased complexity, and subsequent failure manifestation [5]. Such failures directly translate into violation of user Quality of Service (QoS) as well as economic costs to application providers.

There have been numerous works that have proposed novel fault-tolerant approaches through replication [6][7], perform active/passive standby [8], upstream backup [25], and speculative execution [9] of data processing nodes. In addition, works in [10][11] have proposed state rollback recovery and micro-batch processing across remaining nodes. While these approaches are successful in improving system reliability, they face challenges in overhead; ranging from high resource usage to complex coordination protocols that slows down replication. This is particularly true for tolerating late-timing transient faults that are increasingly commonplace in massive-scale distributed systems caused by stragglers [23]. This emergent system phenomena at scale manifest from numerous root-causes transient in nature (i.e. high server utilization, daemon processes) [24]. These faults result in poor performance of stream processing systems, and cascade their impact upon the entire system due to data dependencies. As a result, applying current fault-tolerant approaches still results in delays to real-time processing and violation to deadlines imposed by user QoS.

One promising approach for tolerating transient (and short-lived) faults is data prediction, where speculated data values are determined based off historical data patterns. This approach has been recently applied successfully to the Cloud gaming domain [12], however incurs substantial overhead and applies an identical prediction algorithm across the entire system. It has been demonstrated in [13] that a combination of data prediction techniques can achieve greater performance. As a result, different algorithms for data prediction are more effective within certain scenarios (i.e. overhead, data value, QoS) in response to the failure characteristics of a stream processing system.

In this paper we propose a novel fault-tolerant approach that uses QoS-aware data prediction to tolerate transient late-timing faults in real-time stream processing. The approach uses distributed agents to detect late-timing faults within each processing node. Upon detection the system applies the most appropriate data prediction algorithm to the scenario that is capable of achieving the highest accuracy whilst fulfilling timing deadlines specified by QoS. We implemented our approach within Storm [22], an open source real-time distributed computation framework and demonstrate through experiments the approach's effectiveness under numerous operational scenarios as well as contrasted against other fault-tolerant approaches.

The paper is structured as follows: Section 2 describes the research background; Section 3 discusses related work; Section 4 presents the system design and architecture; System 5 details the experiment setup; Section 6 presents the evaluation; Section 7 discusses the conclusions and future work.

II. BACKGROUD

Stream processing (also known as complex event processing systems [14] and continuous query processing systems [15]) are systems developed to process single or multiple sources of data in motion [7][16]. These systems are typically represented as Directed Acyclic Graphs [9] and can be deployed as sequential (processing node performs a small subset of operation which is outputted to the next node), parallel (i.e. no dependency between tasks) or a combination of both. A common usage of such systems is the ability to perform query processing in real-time - where the physical timings of the result is equally important as result correctness [17].

Stream processing frameworks such as Storm [18], Spark, and Kinesis have enabled a transition from capturing data for online transaction processing to real-time analytics processing [19], with numerous applications across multiple domains such as IoT, fraud detection, social media and video. Real-time stream processing effectiveness is measured by four criteria; *high availability* to handle demand at all times [4], *low latency* for high volume processing to avoid bottlenecks in dataflow [9][18], *scalability* to operate across potentially thousands of nodes, and *fault-tolerance* for reliable system operation in the presence of failures. An important consideration for the latter is minimal overhead in terms of resource utilization, as well as rapid failure recovery that does not impact other processing nodes within the system [20].

Late-timing transient faults are classified as intermittent faults that produce late timing failures (i.e. late service delivered). Such faults are difficult to reproduce and occur due to system conditions affecting the hardware or software by high workload intensity and synchronization issues [28]. Such faults are increasingly common in large-scale Cloud datacenters due to complexity and larger system scale resulting in an increased number of faults resulting in subsequent failure manifestation [29]. These faults result in debilitated effectiveness for stream processing systems which are required to process data in a timely manner and provision applications effectively in soft real-time.

III. RELATED WORK

There have been numerous fault-tolerant approaches proposed for streaming processing systems broadly categorized as replication, upstream and data prediction.

Hwang et al. [8] present three approaches to achieve high availability stream processing. These approaches consist of replication techniques deployed as (1) passive standby, (2) upstream backup (i.e. data sent to a faulty node is rerouted to other processing nodes), and (3) active standby. Each of these approaches are extended using *K*-safety where nodes are replicated *K* times to enable recovery from multiple node failures. Approaches are compared though simulation demonstrating that approaches result in low recovery time and that trade-offs exist between recovery time, overhead, and network distribution.

Shah et al. [6] present Flux; a technique for replica coordination for parallel data flows using traditional query processing for partitioned parallelism combined with process-pairs. This allows Flux to provide automated recovery of lost state rapidly with minimal interference to other partitions. The approach was implemented in a four node cluster demonstrating its ability to tolerate injected faults with incurred overhead costs due to synchronization between two processes resulting in task slowdown.

Balazinska et al. [7] present DPC (Delay, Process, and Correct) – a replication protocol to tolerate process nodes and network failures. Their approach uses a combination of upstream backup to neighboring process nodes and replicas in order to guarantee eventual consistency, however results in overhead when buffering tuples during failure occurrence. Their results demonstrate that different failure duration characteristics require different fault-tolerant techniques to be applied for maximum effectiveness.

Koldehofe et al. [11] propose a method for rollback recovery without requiring persistent memory for state storage to recovery from multiple failures in streaming architectures. State is saved when its execution is dependent on the state of incoming event streams and where the state has minimal non-reproducible state. They prove the proposed algorithm correctness and evaluate its behavior in different parameter configuration for active replications.

Zaharia et al. [10] develop a new processing model termed discretized streams (D-Streams). The main objective is a scalable means to tolerate both faults and task stragglers. This is achieved through using micro-batch processing which simplifies synchronization between distributed nodes. They provide comprehensive detail of the processing model system architecture and fault recovery. D-Stream is implemented within Spark, demonstrating its ability to process over 60 million records/second on 100 nodes and recover from faults and stragglers in sub-second time.

In terms of data prediction, Wang et al. [13] present a hybrid approach for prediction in order to tolerate data dependences in instruction level parallelism. These methods for prediction comprise last value outcome and time stamp distance. They demonstrate that their method achieves accuracy between 25-49% separately.

Zhou et al. [21] proposed two approaches on how prediction can be applied in the presence of failures. They claim that using a confidence measure within the context of a given scenario is essential, that includes determining the impact of incorrect predictions prior to execution. They propose two models to address this – confidence saturating counter and confidence history counter.

Lee et al. [12] present the speculative execution engine Outatime. Their main objective is to overcome network latency in Cloud based mobile gaming systems. Outatime produces predictive future frames based on recent input behavior, state space sub-sampling, incorrect prediction compensation and bandwidth compression. Results demonstrate that their approach is capable of masking 250ms network latency in Cloud based mobile gaming.

It is observable that these fault-tolerant techniques applied in the context of real-time stream processing were not developed in order to tolerate transient slow processing tasks. Discretized stream approach is capable of tolerating slow tasks however uses micro batch processing which is not applicable to stream processing. Outatime is the most relevant piece of work, however is restricted to a single prediction algorithm within the gaming domain, and is unable to handle heterogeneity of transient faults which may exist within a large processing application (i.e. processing

nodes experiencing heterogeneous failure scenarios in terms of data volume, data velocity and slowdown).

IV. System Architecture

In this section we describe how data prediction can be used as a means to develop QoS-aware fault-tolerance for transient late-timing faults in Stream Processors.

Figure 1 provides a high-level description of the approach, with Table 1 detailing all leveraged parameters. A *Data Source* (an external system) sends numerical data d_{in} to a *Stream Processor* to perform data aggregation or filtration. The Stream Processor sends the resultant data d_{out} to *Output* as a final result, intermediate results to another stream processor, or another external system.

The *Prediction Agent* monitors the data flow of the Stream Processor, specifically time stamps for d_{in} and d_{out} occurrence. When a late-timing fault is detected in the Stream Processor by the Prediction Agent, a data prediction algorithm executes and sends predicted data d_{pred} to Output. The philosophy of this approach is that the data prediction is capable of tolerating transient late-timing faults for a finite period of time until an appropriate recovery technique is deployed (i.e. hot-standby replica, checkpointing), and that within the context of real-time systems it is advantageous to send potentially inaccurate results rather than omitted results. A practical example of such practice can be found in Twitter where approximate answers for meeting time requirements are desired [18].

A. Data Prediction Heterogeneity

An important consideration for the Prediction Agent is determining which algorithm is most effective within the context of tolerating late-timing faults in real-time systems. All data prediction algorithms have the potential to output incorrect results or possess the inability to produce a prediction due to insufficient training data. Furthermore, forming a prediction model takes time and requires computing power on stored data. Studies within [13] have demonstrated that a combination of prediction techniques and different algorithms are effective for improved performance. As a result, it is necessary to demonstrate the heterogeneity of prediction algorithms with respect to accuracy and timing constraints for different types of data attribute characteristics.

We applied five data prediction algorithms to publicly available datasets for weather sensor data [26] and Cloud datacenter server monitoring [27]. We implemented a Java program within a single node that sends n rows at a regular time interval (n = 1000), and applies model training on previously recorded data. The algorithms selected include Last Outcome and Stride based (detailed in [13]), Long Stride Based, Simple Markov, and k-Nearest Neighbor (KNN) where k = 4.

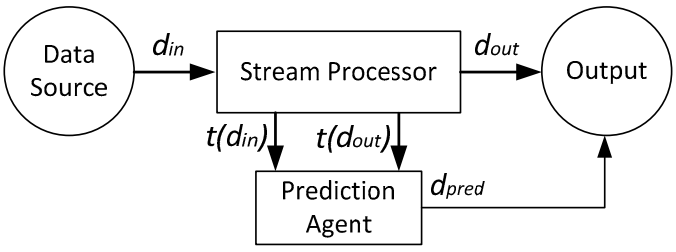

Figure 1. Data prediction fault-tolerant approach.

Table 1. Fault-tolerant Approach Parameters.

Symbol	Description
S	Stream Processor
tn	Current time
f	Late-timing fault
D	Deadline
din	Data input
$dout$	Data output
$dpred$	Predicted data
c	Prediction Algorithm
C	Set of Prediction Algorithms i.e. $c \in C$

Table 2 and Figure 2 depict algorithm accuracy when applied to different data attributes for weather sensors and Cloud datacenter monitoring, respectively. It is observable that while prediction for rainfall and battery are highly accurate across all algorithms, it is apparent that certain algorithms are better suited for different data attributes. This is demonstrated by accuracy ranging between 1.26 – 99% for atmospheric pressure and 4 – 100% for surface temperature. This is due to the nature of the prediction algorithm as well as the data characteristics; for example KNN performs well on small spatial data while attributes such as rainfall and battery exhibit minor deviation from historical data, resulting in stable model creation. In contrast wind direction exhibits larger deviation resulting in difficulties for constructing an accurate prediction model.

While the algorithm Last Outcome produced the highest accuracy on average at 87.95%, it is observable that there is no single prediction algorithm that outperforms all others for each data attribute. However as shown in Figure 3, each algorithm requires different execution time to complete ranging between 0.02 – 0.8 seconds. This is also true when configuring algorithm parameters such as KNN, with algorithm execution time ranging between 0.2 – 1.1 seconds when varying k between 10 to 10,000. This deviation in time is worth highlighting, as the model must be repeatedly trained after k new values enter into the system, or data is trained periodically. This has direct impact on additional resource usage as well as increased data volume within the Stream Processor.

These results demonstrate that data prediction accuracy varies dependent on the data heterogeneity and attribute modeled each requiring different processing times for model training. Both aspects must be considered in the context of the proposed real-time fault-tolerant streaming system; algorithms must achieve the highest prediction accuracy whilst abiding to a specified time deadline.

Table 2. Prediction Accuracy for Weather Sensor Data.

	Atmos. pressure	Rainfall	Wind Dir.	Surface Temp.	Humid.	Battery
Simple Markov	54%	100%	8%	68.2%	77.9%	100%
Stride Based	1.38%	99.2%	0.4%	6.1%	5.1%	100%
Stride Multiple	1.26%	98.4%	0.6%	4%	3.26%	100%
Last Outcome	97.3%	98.3%	20%	100%	97.3%	100%
KNN	99%	98.6%	9%	94.6%	38.8%	100%

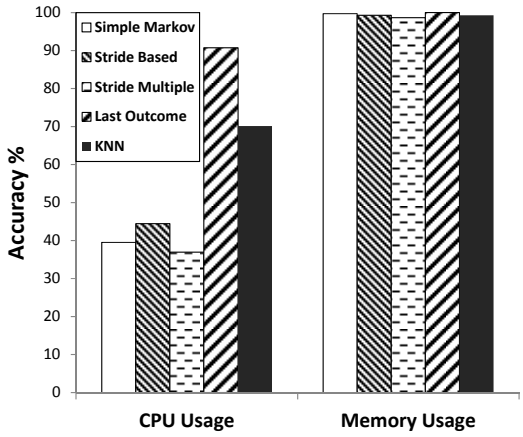

Figure 2. Prediction Accuracy for Cloud monitoring.

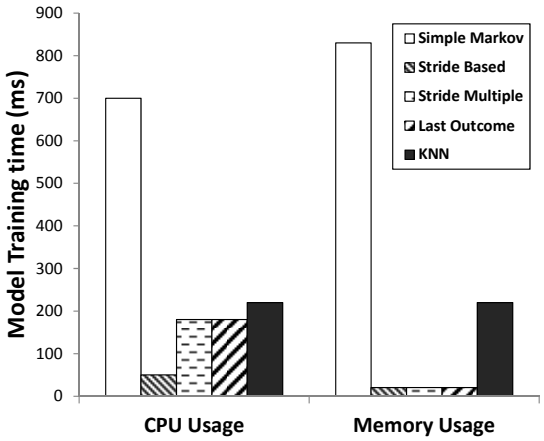

Figure 3. Algorithm overhead time for Cloud monitoring.

B. Prediction Agent

The Prediction Agent is responsible for monitoring the Stream Processor, fault detection, as well as determining and executing the most appropriate data prediction algorithm for tolerating failure until recovery. This functionality comprises multiple sub-components depicted in Figure 4.

Processor Monitor: Responsible for monitoring data flow through the Stream Processor. The monitor records timestamps for both *din* and *dout* within the Stream Processor; if *dout* is not detected before a specified timing constraint imposed by the QoS, a late-timing fault is detected within the Stream Processor.

Prediction Model Builder: Periodically executes data prediction algorithms by training data previously completed by the Stream Processor prior to failure. Data prediction is performed by training a limited subset of historical data stored within a data cache that is periodically updated. An agent can hold multiple prediction algorithms at a given time which data is trained for individually.

Algorithm Rank: Models trained within the Prediction Model Builder are periodically evaluated asynchronous to the Stream Processor. This is to ensure that models are kept as accurate as possible and reflect recent data characteristics. The QoS calculator records several Key Performance Indicators (KPIs) for accuracy (mean squared error, root mean squared error, mean absolute error) and execution time (time to train, execution). This is used to construct a ranking order for algorithms.

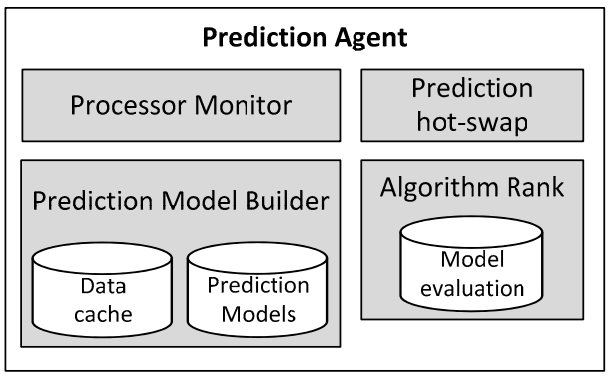

Figure 4. Prediction Agent components.

Prediction Hot-swap: This component acts as the decision maker for selecting the optimal prediction algorithm upon fault detection. The component will execute the prediction algorithm with the highest recorded accuracy and execution time (defined in Algorithm Rank) while adhering to time deadlines specified by the QoS. *Dpred* will continue to output data until the Stream Processor recovers or another mitigation technique is applied (i.e. upstream backup). This decision making can be expressed formally shown in Equation (1) and Algorithm 1.

$$\forall t_n \leq \mathcal{D}, d_{out} \equiv \emptyset : f = \max_{\forall c \in C} P(c) : d_p = f(d_{in}) \Longrightarrow t_n > \mathcal{D}, d_{out} = d_p \quad (1)$$

C. System Model

The concept of the Prediction Agent monitoring a Stream Processor can be applied to the entire streaming system as shown in Figure 5. One advantage of this approach is that Prediction Agents are loosely coupled from a specific stream processor (i.e. they can be deployed on separate physical machines).

The system design is decentralized in nature; the decision to not include a central Prediction Agent is due to increase data traffic transferring data for model training, as

```
Inputs:
d_in = data source input
d_out = data output
QoS = Quality of Service with timing
deadline
Algorithm:
1.  Start d_out = STREAM_PROCESS (d_in, QoS)
2.  PREDICT ( t(d_in), QoS )
     1.  Deadline = t(d_in) + QoS
     2.  SLEEP UNTIL Deadline
     3.  IF d_out == NULL
        1.  GET Manager.Predictions
        2.  GET Manager.Evaluations
        3.  SORT Predictions BY
            Evaluations.Accuracy THEN BY
            Evaluations.Overhead
        4.  d_out = Prediction[0].Run
        5.  STREAM_PROCESS.Clear
     4.  END IF
```

Algorithm 1. Prediction Algorithm Hotswap.

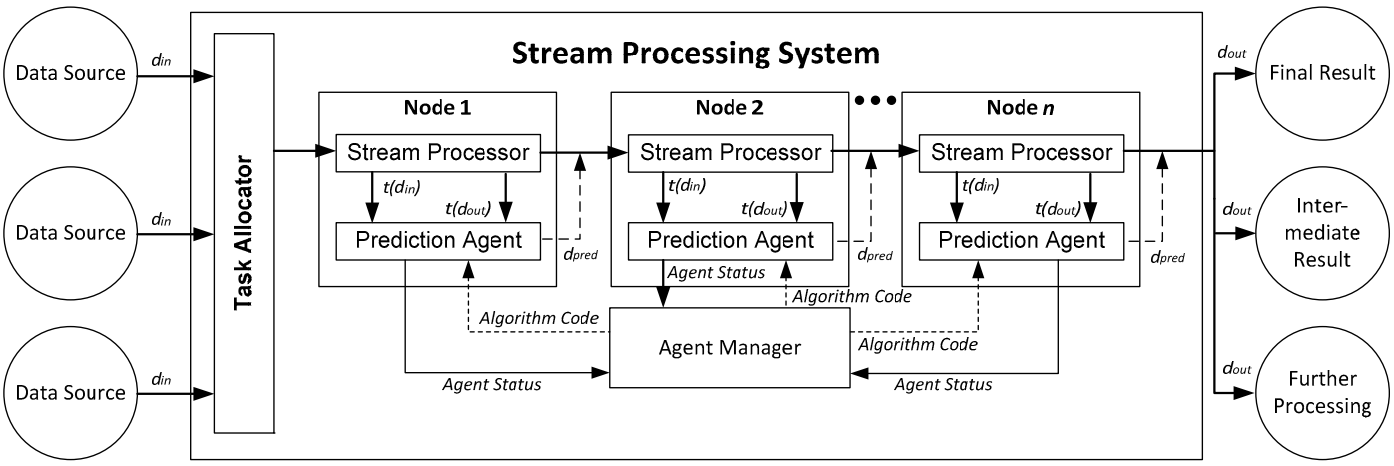

Figure 5. System architecture of fault-tolerant approach within a sequential Stream Processor.

well as sending predicted data across the network in the event of failure resulting in challenges pertaining to scalability and bottlenecks (further aggravated in the event of simultaneous failure, or multi-tenant systems). Furthermore, a decentralized approach allows for Prediction Agents to operate even if a failure occurs within another Prediction Agent.

Due to their decentralized nature, it is advantageous to construct a light weight manager which communicates with all Prediction Agents. This Agent Manager is responsible to monitoring Prediction Agents that periodically send heartbeats reporting their current status; this Agent Manager provides a complete view of the current state of the system. An additional feature of the Agent Manager is the ability to upload different data prediction algorithms into Prediction Agents. This allows for the system administrator to control the number of prediction algorithms within an individual Prediction Agent, as well as select the algorithm type of a specific parameter configuration.

We describe an example scenario of the fault-tolerant approach in operation in the event of a late-timing fault:

1) Data is sent to Stream Processor, with its time stamp sent to the Prediction Agent. The Stream Processor has a timing deadline specified by stream system QoS.
2) A data output fails to be produced prior to deadline imposed by the QoS, and is detected as a late-timing fault within the Process Monitor.
3) Prediction hot-swap compiles a list of all algorithms ranked by their accuracy and time overhead in descending and ascending order, respectively to complete model training.
4) The selected prediction algorithm executes, with the predicted value generated from previously trained data. This is sent to the output address in place of the data output for the Stream Processor. The Stream Processor clears the previous data tuple.

An advantage of this approach is the ability to provide different data prediction algorithms towards fulfilling QoS constraints in response to heterogeneous failure scenarios. For example, two Stream Processors with different data throughput affected by different failure duration are capable of selecting different data prediction algorithms in order to fulfill timing requirements imposed by the specified QoS.

V. Experiment Setup

We implemented our proposed fault-tolerant technique within Apache Storm in order to develop a Cloud datacenter monitoring system. The system reports datacenter server operation in real-time detailing resource utilization and status periodically. The abstract system concepts shown in Figure 5 can be directly mapped to implementations used by Storm. Specifically, Storm stream topologies can be represented as serial and/or parallel, use *spouts* for reading tuples from external sources (Data Source), and bolts for processing (Stream Processor).

We implemented the Prediction Agent as a library within Apache Storm available at [22]. The Prediction Agent was implemented in Java as a single object attached to each Stream Processor. The Process Monitor calculates tuple execution duration in a Stream Processor by recording input and output timestamps matched by message ID. The Prediction Model Builder is constructed by periodically training prediction algorithms on *n* data tuples stored in memory within the Data Cache, with *n* configured to 50. For experiments we implemented four data prediction algorithms comprising KNN, Stride Based, Last Outcome and Simple Markov. Model training was performed offline with metrics of accuracy, training time and execution time recorded within Algorithm Rank.

The Storm cluster was implemented into the University of Leeds Cloud Test Bed, comprising 15 Intel Xeon CPU E5-2630 v3 servers @ 2.40GHz using up to 7 VMs

Table 3. Experiment Configuration Parameters.

Metric	Configuration	Value
System Scale	2 VMs	11 Stream Processor nodes
	7 VMs	44 Stream Processor nodes
Fault Size	Small	10% Stream Processor nodes
	Large	50% Stream Process nodes
Data Input Size	Fixed	18 Spouts
Data Input Throughput	Small	250ms
	Medium	500ms
	Large	1000ms

TABLE 4. STREAM PROCESSOR END-TO-END EXECUTION TIME (ms) FOR FAULT-INJECTION EXPERIMENTS.

System Size			Baseline		Upstream		Prediction	
	Fault	Throughput	Average	Max	Average	Max	Average	Max
2 VMs	Small	Small	257.90	5264.12	183.61	3258.36	14.967	292.49
	Small	Medium	293.86	6489.63	213.77	4303.70	57.15	572.34
	Small	Large	202.72	4288.72	222.55	5832.72	129.32	1792.72
	Large	Small	378.99	5654.00	426.68	6477.08	202.41	2237.52
	Large	Medium	539.17	6204.43	524.27	6667.13	306.45	3249.78
	Large	Large	429.95	6819.42	558.76	6523.19	345.87	4110.62
7 VMs	Small	Small	182.38	3539.57	193.65	4394.59	13.86	1151.71
	Small	Medium	142.09	4460.15	159.44	3933.50	25.96	3079.77
	Small	Large	143.96	4649.28	201.83	4196.15	133.96	3584.50
	Large	Small	253.24	4832.33	322.22	4736.22	26.76	280.81
	Large	Medium	305.43	4868.63	228.35	4227.59	88.68	3794.32
	Large	Large	340.97	5217.39	180.89	5636.30	227.14	3462.26

configured with 2 VCPU and 1 GB memory. Each VM contained Ubuntu OS with multiple workers representing an individual Stream Processor.

Numerous experiments were conducted reflecting different operational and failure scenarios. We varied key parameters including data input size, fault injection percentage, and number of VMs as shown in Table 3. Data input size was altered by increasing and decreasing the velocity which data is sent to the Storm system. Faults were injected through using sleep threads to reduce data throughput and injected every 30 seconds and lasting 8 seconds. This was to simulate failures due to memory leaks, memory bloats or un-terminated threads resulting in locks. With these experiment configurations we measured several metrics including resource overhead, algorithm accuracy and throughput. Furthermore, we calculate the tuple end-to-end time for the application (i.e. first input into the system until the final result).

For evaluation against other approaches, we executed the same experiment conditions for our approach, upstream backup [13][26], and no fault tolerance (referred to as baseline). Upstream backup timeout was configured to the default value of 30 seconds. Each experiment case was executed for 150 seconds 10 times each. For our experiments we assume that there is failure isolation between VMs (which typically reside within separate physical machines), and that there is sufficient data available for prediction model training. Furthermore we assume that all faults will eventually be corrected and not exhibit crash failure characteristics (i.e. will eventually send data). Our approach is focused on fault-tolerance, therefore we do not evaluate fault correction and recovery practices. Furthermore, it is assumed that the input data itself is non-faulty and contains no missing data.

VI. EVALUATION

Table 4 shows the end-to-end execution time of data tuples within the Stream Processor in numerous experiment configurations. It is observable that data prediction minimizes execution time under the presence of failures by approximately 61% on average compared to upstream backup. In particular, the approach minimizes the maximum end-to-end time spike that occurs during failure as depicted in Figure 6 and Figure 7. The number of VM failures also impacts the effectiveness of techniques; with large faults producing greater end-to-end time on average compared to small faults across all experiment cases. We observe that while large faults impact the average execution time of all approaches, we observe that there is minimal difference in effectiveness between baseline and upstream backup for long faults compared to short faults. The exception to this is large faults and large data where upstream outperforms all methods. This is due to less burden upon a Stream Processor when re-routing tuples to other Stream Processors. In contrast, while prediction results in a considerably lower average and maximum end-to-end time its effectiveness is more sensitive to large faults. This is due to the reliance on predicting values on already data impacted by late-timing failures, requiring more time to train prediction algorithms. This results in an increase of 46.5% and 96.8% for experiments for small and large faults, respectively.

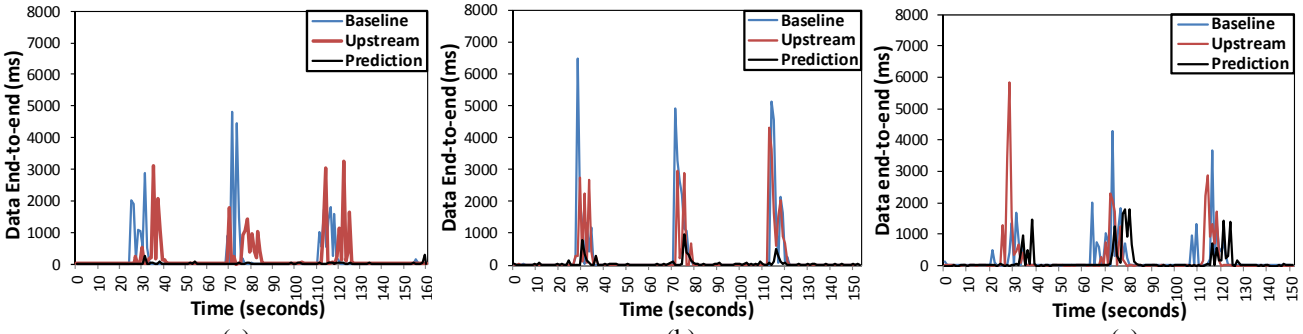

Figure 6. End-to-end execution times for small faults in 2 VMs with data size (a) small, (b) medium, (c) large.

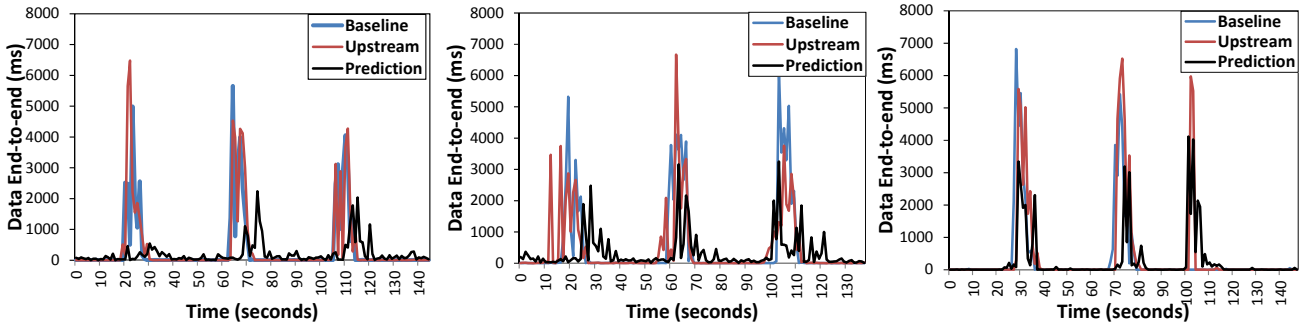

Figure 7. End-to-end execution times for large faults in 2 VMs with data size (a) small, (b) medium, (c) large.

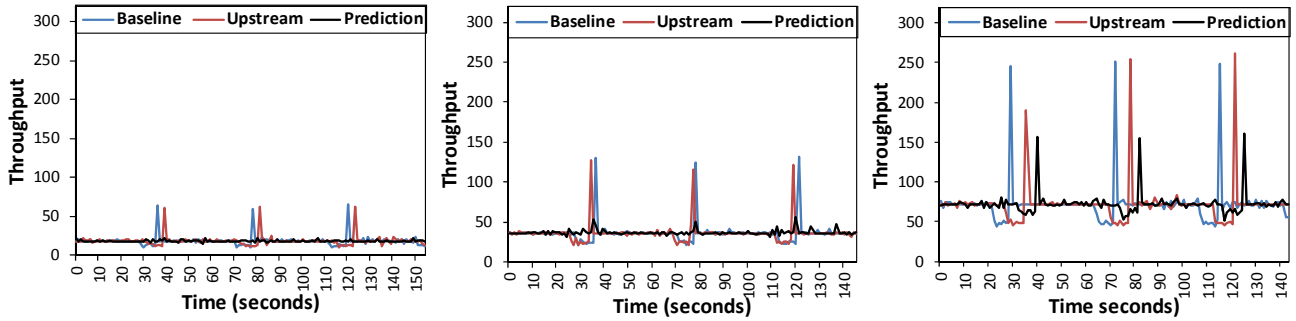

Figure 8. Throughput for small faults in 2 VMs with data size (a) small, (b) medium, (c) large.

While the prediction approach results in minimized end-to-end time under the presence of faults, it does so by potentially sending inaccurate results shown in Table 2 as discussed in Section 4a. Within all experiments we discovered that algorithm accuracy does not drastically fluctuate and ranged between 65-75%.

It is worth noting that Storm contains functionality which buffers data, and is capable of accelerating the throughput of an individual processing node. This is indicated by a slight drop and sudden increase in throughput as shown in Figure 8 for baseline and upstream backup caused by delay in fault mitigation. As prediction operates on immediate mitigation upon fault detection, this results in a minimized impact towards throughput, however increases with data size.

Figure 6 and 7 also depict the impact of faults under various data tuple sizes. We observe that increasing the data size of tuples results in increased end-to-end time of the stream processor for faults. This is resultant of increased volume of data that must be queued by the Stream Processor until model training on previous tuples has been completed. Such behavior is reflected by a larger spike in throughput for processing as shown within Figure 8(c).

Each technique also exhibits different resource characteristics in terms of CPU and network across the entire system as shown in Figure 9(a) and 9(b). It is observable that Prediction uses the highest amount of CPU utilization 12% higher in comparison to other techniques. Higher CPU utilization is due to continuous model training and evaluation to produce prediction results. In contrast, this results in reduced network usage required up to 44% and 35% less than upstream for upload and download, respectively. Upstream backup requires just under 7 MiB network usage to reroute data tuples to other processing nodes within the system. These results present an important consideration for considering network congestion, and trade-off between reducing end-to-end times. Memory usage of all technique remains stable between 6-9% across all experiments.

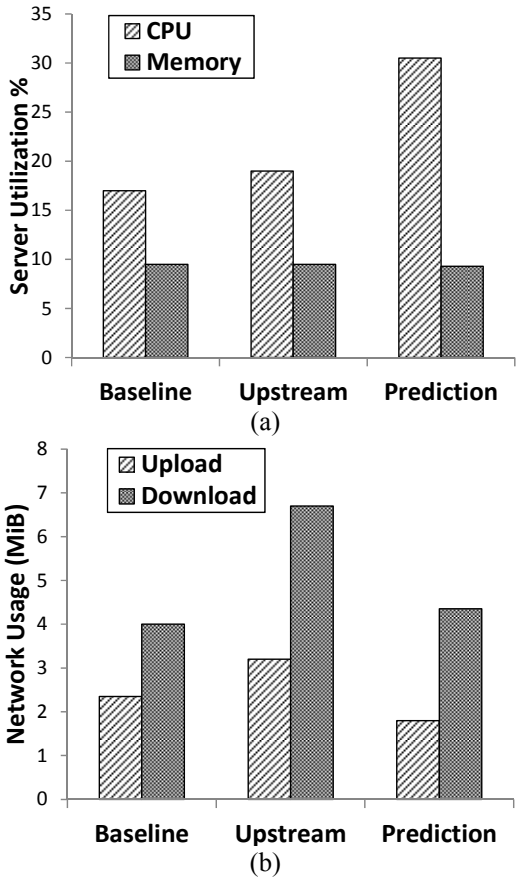

Figure 9. Stream processor resource usage. (a) CPU and memory, (b) Network.

In summation, while using prediction for fault-tolerance results in decreased end-to-end execution time, stabilized throughput and decreased network usage in contrast to other methods, it does so at the cost of increased CPU overhead and potential result inaccuracy. Furthermore, the effectiveness of upstream backup towards minimizing end-to-end time increases with increased system size, however incurs additional network resource usage. Such design trade-off must be considered when designing real-time stream processing applications within the context of system resource constraints and whether application specific QoS favors timing over potential inaccuracy.

VII. CONCLUSIONS

In this paper we have presented an approach for tolerating late-time transient failures in real-time stream processor applications. The approach is capable of applying data prediction algorithms heterogeneously in response to various failure conditions for different Stream Processors to satisfy imposed QoS. We have described core concepts of the fault-tolerant approach and presented its architecture which has been implemented within Apache Storm. Numerous experiments have been conducted to evaluate its effectiveness for different system configurations and scenarios in comparison to other fault-tolerance approaches. Our conclusions are summarized as follows:

Data prediction can improve the performance of stream processing under the presence of transient timing faults. Using data prediction for fault-tolerance reduces Stream Processor end-to-end execution time however can potentially provide inaccurate results. Such accuracy can be improved with the inclusion of more effective prediction algorithms configured to patterns within the data.

Design considerations for applying fault-tolerance with respect to failure type and resource overhead. Our experiments demonstrate that different fault-tolerant approaches exhibit different effectiveness for different data size and system scale. Furthermore, each approach produce various resource utilization characteristics predominantly within CPU and network. Such behavior is important to system administrators when designing the infrastructure.

Future work will include further investigation into additional application domains such as video streaming and online gaming. Furthermore, we believe there is opportunity to further improve the approach by investigating methods to reduce the produced resource overhead and expansion to include crash failures.

ACKNOWLEDGMENTS

This work was supported by CIATEQ Division of Information Technologies and Control.

REFERENCES

[1] S. T Allen, M. Jankowski, P. Pathirana, “Storm Applied Strategies for Real-time Event Processing”, NY:Manning Publications Co, 2015.
[2] G. Aceto, A. Botta, W. de Donato, A. Pescapè, "Cloud Monitoring: A Survey”, Computer Networks, 57(9), 2013, pp. 2093-2115.
[3] G. Vigna, W. Robertson, V. Kher, R. A. Kemmerer, "A Stateful Intrusion Detection System for World-wide Web Servers", Computer Security Applications Conference, 2003, pp. 34-43.
[4] M. Stonebraker, U. Çetintemel, S. Zdonik, "The 8 Requirements of Realtime Stream Processing", ACM SIGMOD Record 34.4, 2005, pp. 42-47.
[5] B. Schroeder, G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems", IEEE Transactions on Dependable and Secure Computing, vol. 7, 2010, pp. 337-351.
[6] M.A. Shah, J.M. Hellerstein, E. Brewer, "Highly available, fault-tolerant, parallel dataflows", ACM SIGMOD International conference on Management of Data, 2004, pp. 827-838.
[7] M. Balazinska, H. Balakrishnan, S. Madden, M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system”, ACM Transactions on Database Systems (TODS), vol. 33(1), 2008.
[8] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, S. Zdonik, "A Comparison of Stream-oriented High-availability Algorithms", Technical Report TR-03-17, Computer Science Department, Brown University, 2003.
[9] A. Brito, C. Fetzer, P. Felber, "Minimizing Latency in Fault-tolerant Distributed Stream Processing Systems", IEEE International Conference on Distributed Computing Systems, 2009, pp. 173-182.
[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, "Discretized Streams:Fault-tolerant Streaming Computation at Scale", ACM Symposium on Operating Systems Principles, 2013, pp.423-438.
[11] B. Koldehofe, R. Mayer, U. Ramachandran, K, Rothermel, M. Völz, "Rollback Recovery without Checkpoints in Distributed Event Processing Systems”, ACM International Conference on Distributed Event-based Systems, 2013, pp. 27-38.
[12] K. Lee, D. Chu, E. Cuervo, Y. Degtyarev, S. Grizan, J. Kopf, A. Wolman, J. Flinn, "Outatime: Using Speculation to Enable Low-latency Continuous Interaction for Mobile Cloud Gaming", In Proceedings of MobiSys, 2015, pp. 151 – 166.
[13] K. Wang, F. Manoj, "Highly Accurate Data Value Prediction using Hybrid Predictors", In Proceedings ACM/IEEE International Symposium on Microarchitecture, 1997, pp. 281-290.
[14] D. Robins, "Complex Event Processing”, International Workshop on Education Technology and Computer Science, 2010.
[15] S. Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing", In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2003, pp. 668-668.
[16] B Ellis, “Real-time Analytics: Techniques to Analyze and Visualize Streaming Data”, John Wiley & Sons, 2014.
[17] H. Kopetz, “Real-time Systems: Design Principles for Distributed Embedded Applications”, Springer Science & Business Media, 2011.
[18] A. Toshniwal, et al. "Storm@ Twitter", In Proceedings of ACM SIGMOD International Conference on Management of Data, 2014, pp. 147-156.
[19] Apache. Apache Spark.. Internet: http://spark.apache.org/streaming/
[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, "Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management", ACM SIGMOD International Conference on Management of Data, 2013, pp. 725-736.
[21] H. Zhou, C. Y. Fu, E. Rotenberg, T. M. Conte, "A Study of Value Speculative Execution and Misspeculation Recovery in Superscalar Microprocessors", North Carolina State University, Tech. Rep, 2000.
[22] Apache, Apache Storm, 2015 Internet: https://storm.apache.org/documentation/Powered-By.html
[23] P. Garraghan, X. Ouyang, P. Townend, J. Xu, “Timely Long Tail Identification Through Agent Based Monitoring and Analytics”, IEEE International Symposium on Real-time Computing (ISORC), 2015, pp. 19-26.
[24] J. Dean, J, Barroso "The Tail at Scale", Communications of the ACM, 56(2), 2013, pp. 74-80.
[25] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, S. Zdonik, "High-availability algorithms for distributed stream processing", International Conference in Data Engineering (ICDE), 2015, pp.779-790.
[26] I. Witten, E. Frank, “Data Mining: Practical machine learning tools and techniques”, Morgan Kaufmann, 2005.
[27] Google, Google Cluster Data V2., 2015, Internet: https://github.com/google/cluster-data
[28] A. Avizienis, J.C. Laprie , B. Randell , C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, vol. 1, 2004, pp.11-33.
[29] U. Hölzle, L. A. Barroso, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines", Synthesis Lectures on Computer Architecture, 2009.