GOVERNOR: Smoother Stream Processing Through Smarter Backpressure

Xin Chen[†], Ymir Vigfusson[♭], Douglas M. Blough[†], Fang Zheng[‡], Kun-Lung Wu[‡], Liting Hu[§]
[†]Georgia Tech, [‡]IBM T. J. Watson Research Center, [♭]Emory University, [§]Florida International University

***Abstract*—Distributed micro-batch streaming systems, such as Spark Streaming, employ backpressure mechanisms to maintain a stable, high throughput stream of results that is robust to runtime dynamics. Checkpointing in stream processing systems is a process that creates periodic snapshots of the data flow for fault tolerance. These checkpoints can be expensive to produce and add significant delay to the data processing. The checkpointing latencies are also variable at runtime, which in turn compounds the challenges for the backpressure mechanism to maintain stable performance. Consequently, the interferences caused by the checkpointing may degrade system performance significantly, even leading to exhaustion of resources or system crash.**

This paper describes GOVERNOR, a controller that factors the checkpointing costs into the backpressure mechanism. It not only guarantees a smooth execution of the stream processing but also reduces the throughput loss caused by interferences of the checkpointing. Our experimental results on four stateful streaming operators with real-world data sources demonstrate that Governor implemented in Spark Streaming can achieve 26% throughput improvement, and lower the risk of system crash, with negligible overhead.

I. INTRODUCTION

Big data systems have evolved beyond scalable storage and rudimentary processing to supporting complex data analytics in near real-time, such as Apache Spark Streaming [31], Comet [14], Incremental Hadoop [17], MapReduce Online [7], Apache Storm [28], StreamScope [19], and IBM Streams [1]. These systems are particularly challenging to build owing to two requirements: low latency and fault tolerance. Many of the above systems evolved from a batch processing design and are thus architected to break down a steady stream of input events into a series of *micro-batches* and then perform batch-like computations on each successive micro-batch as a *micro-batch job*. In terms of latency, the systems are expected to respond to each micro-batch in seconds with an output The constant operation further entails that the systems must be robust to hardware, software and network-level failures. To incorporate fault-tolerance, the common approach is to use checkpointing and rollback recovery, whereby a streaming application periodically saves its in-memory state to persistent storage.

These two primary requirements, however, can interfere with one another and consequently harm the system performance. Specifically, note that many real-world streaming applications maintain large state in memory, such as sliding windows. The large in-memory state in turn produces large checkpoints, which leads to long checkpointing time owing to greater data serialization/deserialization times and other I/O. The protracted checkpointing time delays the blocking time for processing, which affects and potentially violates the low latency and throughput requirements. Figure 1 illustrates how a checkpointing delay of one job cascades to subsequent jobs, causing the streaming system to slash throughput to prevent enduring delays.

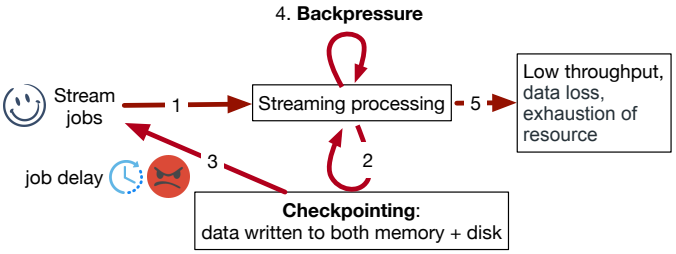


Fig. 1: ***Motivating scenario.*** *Large checkpoint (3) slows down processing (4), causing throughput degradation (5).*

One natural approach to overcome the problem of system slowdown due to checkpointing is to reduce the cost of taking checkpoints [2, 13, 18**?** , 21, 22]. However, checkpoints tend to touch disk and do other I/O-bound operations for persistence and thus complicate such an approach. Another alternative is to perform checkpointing asynchronously with data processing. This, however, complicates the scheduling of checkpointing and normal execution and may cause resource contention. Besides, it is difficult to guarantee the consistency of the global snapshot, which requires management of the separation of dirty state and state consolidation [11] that need to modify low-level state structure. As a result, most current streaming systems only implement synchronous checkpointing, such as Spark Streaming [31], Naiad[20], Flink [12] , and Storm [28].

These micro-batch systems, such as Apache Spark Streaming [9], deploy backpressure mechanisms to dynamically adjust the input rate of topics. For example, in Spark Streaming, the mechanism follows the classic Proportional-Integral-Derivative (PID) controller model in which the PID controller responds to delays introduced by checkpointing reactively, and then passively adjusts the input ingestion rate in the same way as when delays are caused by slow processing. Consequently, the backpressure controller causes the input size of jobs to fluctuate, which can degrade the system stability, lower the throughput, and in some cases even cause resource exhaustion or a system crash.

In this paper, we propose GOVERNOR: a smarter con-

Most of the work was done while the first author was an intern at IBM.

troller that can achieve high stability and high throughput simultaneously, rather than sacrificing throughput for stability as PID controller does. It estimates future checkpointing costs and then factors these costs into a backpressure mechanism to minimize checkpointing interference on the system performance. In contrast to approaches that focus on how checkpointing costs can be reduced, GOVERNOR is a complementary approach that can achieve a stable execution and a high throughput. Under the hood, GOVERNOR exposes a new channel between the controller and receiver that can configure the input size of a specific job, allowing granular adjustment of job processing times to quickly mitigate delays due to checkpointing. For instance, if the predictions foresee that a large snapshot will need to be taken, GOVERNOR would give a small input size to mitigate the checkpointing effects and help the follow-up jobs to experience shorter delays, thus improving the throughput as well as lowering the risk of a system crash.

Note that GOVERNOR is a set of backpressure techniques that can be applied to general micro-batch streaming systems with little code changes. GOVERNOR is expected to work in all other micro-batch streaming systems, since our backpressure controller is completely transparent to any specifics of the processing component and checkpointing data structure in streaming systems.

Contributions. Our paper has the following contributions.

- We empirically study and demonstrate the impact of checkpointing and backpressure mechanisms on throughput and delays in streaming systems.
- We design and implement GOVERNOR: a backpressure controller which predicts the future cost of checkpointing and dynamically adjusts the flow rate to accurately control the input sizes.
- We experimentally evaluate our implementation of GOVERNOR within Apache Spark Streaming using representative streaming window operators. Our results on a realistic financial workload [26] using different kinds of operators demonstrate that compared to a standard PID controller, GOVERNOR can improve the throughput of the system for some continuous queries by up to 26%. Moreover, GOVERNOR can reduce delays which further improves the stability of the streaming system.

Roadmap. The rest of the paper is organized as follows. We next present background and an empirical study to demonstrate the need to coordinate checkpointing and backpressure handling. Section 3 presents a naïve approach that predicts the checkpointing. Section 4 presents the design of our GOVERNOR backpressure algorithm, and discusses both important implementation specifics of our algorithm within Spark Streaming and surveys our experimental results. Finally, Section 6 summarizes related work before we conclude the paper in Section 7.

II. BACKGROUND AND MOTIVATION

Although backpressure mechanisms are critical to finding the optimal flow rate in feedback controllers, they can cause performance degradation or even lead to system crash if handled inappropriately. Before elaborating on this point, we begin with background on back pressure mechanisms and checkpointing processes in streaming systems.

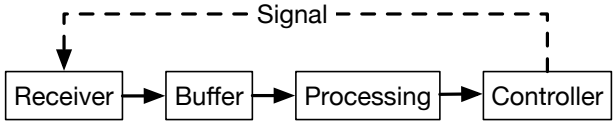


Fig. 2: *Typical back pressure in streaming systems.*

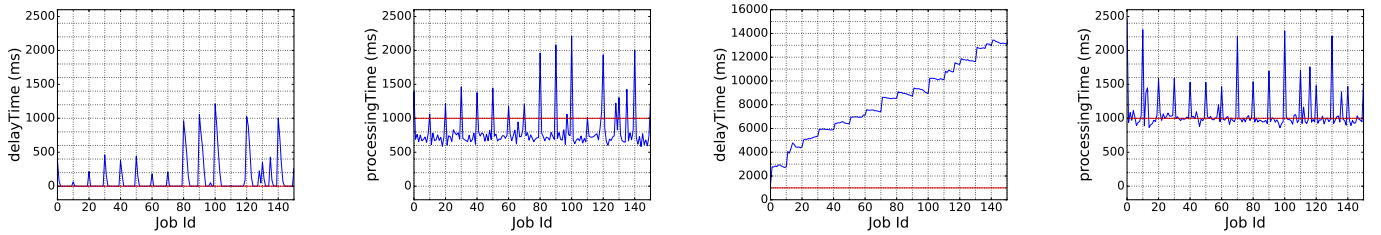
A. Backpressure Mechanisms

Backpressure is a feedback mechanism for rate limiting input based on characteristics of the output that allows a dynamic system to gracefully respond to variations in its input workload in order to achieve smoother execution and better performance. On one hand, when a system is heavily loaded, the backpressure mechanism signals that the input sizes of future jobs should be reduced. Without such provision, a system under stress may begin to drop messages in an uncontrolled way or fail catastrophically, which is unacceptable in practice. On the other hand, when system is lightly loaded, the backpressure mechanism lets the input sizes grow accordingly to prevent resources to be needlessly wasted.

As with all dynamical systems based on control theory, the responsibility of the backpressure is to maintain the system in a stable state: neither heavily loaded nor lightly loaded. To make this precise, we introduce some quantifiable metrics. High loads are reflected by a high *delayTime*, whereas light loads are reflected by a short *processingTime*. Streaming applications require that streaming systems should return results to users in a specified *interval*, also called a *deadline*. Rate is the number of tuples per second. For instance, every 1 second users expect to receive a result, so the *interval* is 1 second. A high *delayTime* implies that *processingTime* of micro-batch jobs is larger than the *interval*, indicating that the user is not receiving the results by the set deadlines. Note that the *delayTime* is cumulative metric as presented in equation 1. If the *delayTime* increases to a certain extent, system would trigger some signal to indicate data loss, possibly leading to the exhaustion of resources or system crash. A short *processingTime* means the system could have ingested more tuples for processing, while also meeting the required deadlines. If we think of these metrics as equations over jobs $1, 2, \ldots, j, j+1, \ldots$, they are related as follows.

$$\text{delayTime(j+1)} = \text{delayTime(j)} + \big(\text{processingTime(j+1)} - \text{interval}\big) \quad (1)$$

The underlying architecture of a back pressure mechanism is illustrated in Figure 2. The processing component is normally considered a black box which receives tuples from a buffer and sends the feedback signal to adjust future input size. The buffer component takes in the tuples from the external world and emits tuples for the processing based on the feedback



(a) *The $delayTime$ on the window size 40.*

(b) *The $processingTime$ on the window size 40.*

(c) *The $delayTime$ on the window size 120.*

(d) *The $processingTime$ on the window size 120.*

Fig. 3: ***Demonstration of Checkpointing Interference.*** *Here, jobs arrive at 1 second intervals in succession on a financial workload (Section V-C) and checkpointing is done every 10 seconds. The red lines denote averages.*

signal. A key component of the backpressure mechanism is the controller that can adjust the size of the buffer in terms of rate according to runtime dynamics.

The Proportional-Integral-Derivative (PID) controller is a well-known and one of the most-used feedback design in control theory [3, 4]. The idea of how PID controllers work in streaming systems can be related back to the two metrics mentioned earlier: processingRate and *delayTime*. When the processingRate increases or decreases, the rate output increases or decreases proportionally based on the processingRate of the previous job. Upon detecting *delayTime* to have grown, the controller also needs to cut the rate proportionally. PID controllers do not seek to make control decisions based on swift or sudden transitions, which is an appropriate choice for checkpointing interference due to their periodicity. This gradual effect stems from PID controllers maintaining a continuous function that estimates the state of the world, and the model makes minimal adjustments to the model based on changes from the previous state. We use PID controllers as the underlying backpressure mechanism throughout this paper.

B. Checkpointing in Stream Processing

Following a normal execution, checkpointing may produce *delayTime* that can cause jobs to miss their deadlines. Due to the cumulative characteristics of *delayTime* 1, system performance may in turn degrade significantly if the *delayTime* component is not handled carefully. To quantify the impact of the checkpointing on delays, we conducted an empirical study within Apache Spark Streaming[31], using PID controller for backpressure. Spark Streaming is a streaming system built on top of Spark engine. A Spark streaming application receives input data streams from external sources, partitions the streams into batches based on a time interval, and submits the batches to a processing engine.

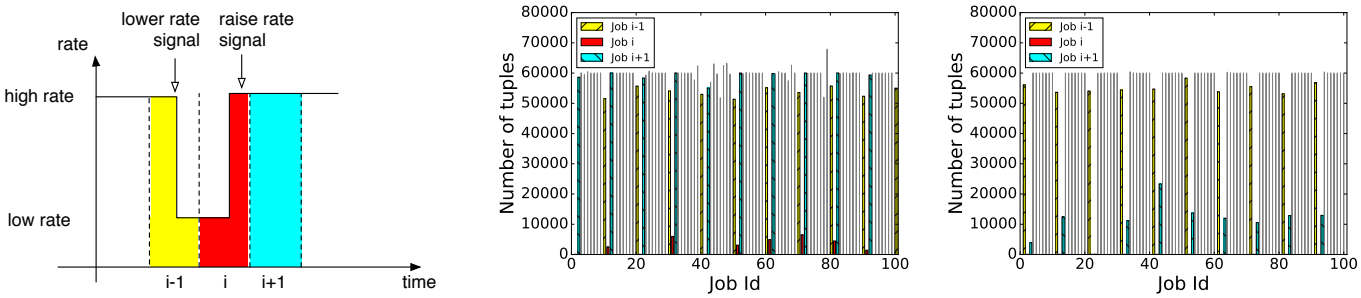
In our experiments, the streaming application is computing the average of numbers across a window, where the size of the window is therefore a proxy for the cost of doing checkpointing: a large window size indicates that a large volume of data needs to be written from memory to storage. The *interval* after which users expect the results to be complete is 1 second, and the checkpointing interval is set to 10 seconds, meaning that a checkpointing job should be launched on average once between every 10 regular jobs. In the following we report the stable executions of 150 seconds, on both a small window size 40 and a large window size 120. Window size is the number of tuples stored in window.

We study the *delayTime* under the configuration of window size 40 and window size 120, shown in figure 3a and figure 3c. We can see that the pattern of the *delayTime* matches the interval of the checkpointing very well in 10 seconds. Although the *delayTime* emerges once every checkpointing interval, the system is stable at low latency throughout the run since the built-up delay can be eliminated before the next checkpoint. However, in figure 3c, *delayTime* increases constantly out of control: the system is unable to keep up with an increasing number of jobs, which are being delayed and being buffered in memory. The situation is likely to trigger an exception due to a buffer overrun or possible crash due to exhaustion of memory if the *delayTime* keeps on increasing.

Another way to investigate the causes of *delayTime* is to study the *processingTime*. Figure 3b and figure 3d shows the *processingTime* of the window size 40 and 120. In the case of size 40, all of the following normal jobs have a shorter processing time than 1 second to counteract the *delayTime*, whereas in the case of size 120 most normal jobs reach the deadline 1 second on the *processingTime*, without giving much time to reduce the *delayTime*, resulting in an ever-increasing built-up delay.

From these tests, we can see that the backpressure would reduce the input size of jobs whenever signaled with a *delayTime*, resulting in the degradation of overall performance. The backpressure behavior directly determines the performance and even influences the system stability. It is evident from these observations that the backpressure mechanism plays a critical role in the health and efficiency of a stream processing system.

Our challenge lies in how we can reduce the *delayTime* accrued by the checkpointing in the backpressure, thus mitigating the interference of the checkpointing, to improve stability and increase throughput. GOVERNOR is a new backpressure mechanism aiming to address this problem. To the best of our



(a) *The area is the input size determined by both rate and time.*

(b) *The signal for rate reduction arrives before deadline.*

(c) *The signal for rate increase arrives after deadline.*

Fig. 4: ***Naïve approach*** *using a small rate signal in backpressure to configure the input size of job i.*

knowledge, this is the first paper that considers the influences of the checkpointing in backpressure for streaming systems.

III. A NAÏVE APPROACH TO PREDICT CHECKPOINTING

Since the checkpointing jobs in stream processing are periodic, it is easy to accurately predict which job is the checkpointing job. Instead of doing the PID estimation, we can proactively cut the input size of checkpointing job in an attempt to reduce the *delayTime* accrued by the checkpointing mechanisms. A basic algorithm is to modify the rate signal to configure a small input size for the checkpointing job. Specifically, the key logic is to seek to reduce the input size of the checkpointing job before its execution using the rate signal, and then raise the rate signal after its completion.

In our experience, we discovered that controlling the input size of a specific job using the rate signal is difficult to do as the parameters depend on precise prediction for when a new rate arrives. Note that new signals being received is determined by the time of completion of a job, since rate signals are always sent out after the jobs have executed. However, the start of generating input tuples for a job is in a constant interval regardless of the arrival of the rate signal.

Figure 4a presents how the rate and the time work to determine the input size of job. The horizontal axis represents the time and the vertical axis represents the rate; the area is the number of tuples collected during 1 second. Job i is a job for which we are trying to set to a small input size with a low rate signal. However, the input size of job i is larger than what is expected , because the high rate signal arrives earlier than the deadline. Similarly, because the lower rate signal arrives before the deadline, the input size of job $i-1$ is smaller than the high rate.

We ran experiments to determine whether we can configure a specific job with a small input size using the rate signal without influencing other jobs. We expect to be able to set the input size of a job i to a small number, by sending a low rate signal. In the following two experiments, the low rate we configure are 60 tuples/sec for one job i and 60,000 tuples/sec for the others. The checkpointing *interval* is 10 seconds. We report the measured numbers of tuples received by the jobs over the 100 seconds in Figures 4b and 4c.

As Figure 4b shows, although the rate is constant on 60, the number of the tuples of job i is highly variable, denoted with the middle red rectangle. The input size of job $i-1$ also fluctuates because the low rate signal arrives before the deadline presented in yellow area in Figure 4a. In the next experiment, shown in the Figure 4c, the number of tuples for job i is more stable than in the previous experiment. However, we observed that the number of tuples for both jobs $i-1$ and i become unstable due to the influences from the small rate signal.

In summary, this approach does not work as the input size is uncontrollable. The input size of the jobs is determined not only by the rate signal, but also by when jobs finish, which has proven to be difficult to predict precisely owing to various complex dynamic factors. Nevertheless, the rate is still an essential signal in the backpressure controller of streaming systems. Using the rate signal indicates that the systems keep digesting the data streams at the previous rate if not adjusted, which plays an important role for streaming applications that require results to be returned to users at an specified interval in a smooth and predictable fashion despite any uncertainty. This only poses a higher requirement for the management of the rate signal.

IV. DESIGN OF GOVERNOR

Instead of relying only on a rate signal for feedback, GOVERNOR introduces a new signal: $(timestamp, \#tuples)$, which offers the fine-grain control over the input sizes of jobs. We now describe the architecture and the main algorithm of GOVERNOR.

A. System Architecture

Figure 5 shows the high-level architecture of GOVERNOR within a streaming system. There are two key components: the feedback calculation component (controller), and the Fetcher. The feedback calculation component implements the main logic of our algorithm, such as how to calculate the rate and how many tuples are contained in a specific job. The Fetcher

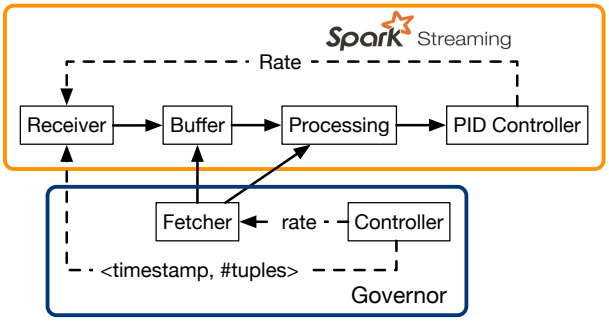


Fig. 5: GOVERNOR *Architecture*.

is mainly responsible for retrieving the tuples from the queue buffer as the input, generating a job, and then submitting it to the processing engine.

There are two signals: the rate and (timestamp, #tuples). The first signal, rate, is sent by GOVERNOR to notify the receiver of the number of tuples per second ingested from the external input sources. The second signal tells the Fetcher the input size for a specific job. Overall, rate represents the maximal achieveable throughput of the system that it could obtain. The extended signal provides a fine-grained control over the input size of jobs, which aims to reduce the *delayTime* by configuring a small input size. We believe that GOVERNOR can achieve a high rate for the throughput improvement through fine-grained adjustment of input sizes of certain jobs using (timestamp, #tuples).

Example. We illustrate the idea of our algorithm with an example. Normally, there is a minimum input size provided by streaming application, indicating that every interval at least the minimum input size should be processed regardless of anything. In this example, the checkpointing interval is 10 seconds, and every 1 second there is a micro-batch job submitted for processing. The checkpointing job takes 2 seconds with a large input size, and takes 1.3 seconds with the minimum size. Normal jobs take 1 second to process the large size, and take 0.6 second to process the minimum size. One sudden normal job consumes 1.5 seconds. The main logic of our algorithm contains three parts: Region Partition, Reducing Delay and Estimation of region rate.

B. Region Partition

To capture the dynamic nature of the execution, we use the checkpointing as a marker to partition the job flows into *regions*. Here, a region is defined as a sequence of jobs that always begins with a checkpointing job, and ends before the next checkpointing job. This is feasible because we can predict accurately when the checkpointing happens since the checkpointing is assumed to be explicit and periodic. Following the completion of the normal jobs, the checkpointing has a wide variability on its time cost, and thus our approach considers the *delayTime* caused by the checkpointing explicitly for the purpose of minimizing the interferences. The duration of the region is supposed to equal the interval of checkpointing. In the simple example, the region contains 10 jobs, including 1 checkpointing job and 9 normal jobs.

C. Collection of historical records

Our approach collects the historical records to predict the future executions. Given an estimated execution time, we need to determine an input size to let the job finish on time roughly, so an expected *processingTime* can be converted to a reasonable input size.

There are several types of jobs our approach maintains with the historical information: the checkpointing jobs, the normal jobs specified with the minimum input size, called as the small job and the normal jobs with the full interval time. The checkpointing job is the main source that produces the *delayTime*, so we can know the *delayTime* for the next region in advance. The jobs specified with the minimal input size are the jobs following the checkpointing jobs, used to reduce the *delayTime* by proactively configuring the minimum input size. With the collection of this information, we can predict how much delay can be reduced for each small job. Collecting the information of the normal jobs with full time is used to predict the input size for the normal jobs, in order to further estimate the overall rate of one region.

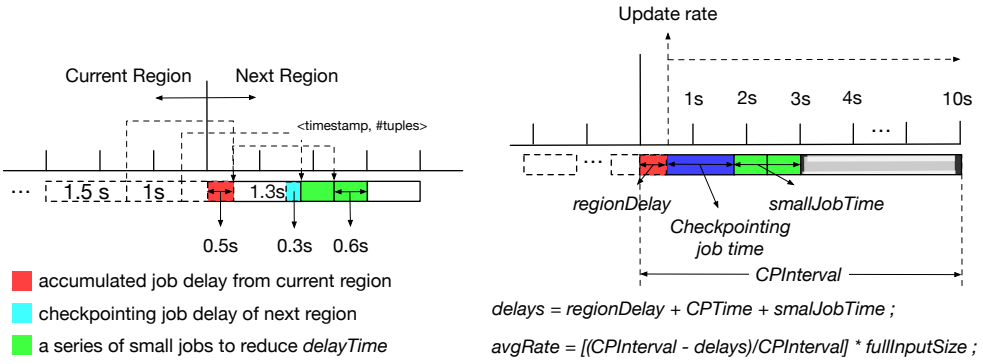
As the streaming application runs for a long time, runtime and the workload may vary widely over time. Our online algorithm maintains timeliness by only storing the records within certain past duration. For example, the duration is 1 minute, which means that the historical information only includes the records of the past 1 minute. Any records older than 1 minute would be popped out when the latest record gets memorized.

Note that we are not guaranteeing any precise accuracy of the prediction on specific jobs, because there are too many factors that might influence the results, or even some executions are virtually unpredictable because of content-dependence. However, we believe that for most streaming applications, the cost of the executions may not vary dramatically during a certain amount of time. Thus the prediction is simply implemented as doing an average on the collected records. In the simple example, 1.3 second of the checkpointing job and 0.6 second of the normal job with the minimum input size are predicted based on the collected historical records.

D. Reducing Delay

The backpressure mechanism needs to entail that the *delayTime* does not constantly increase. It is crucial to make sure the *delayTime* is controllable, otherwise the system would suffer from data loss, exhaustion of resources, or system crash. Our approach tends to reduce the *delayTime* by configuring the minimum input size.

For each region, the *delayTime* we need to predict for the next region is of two types: the delay inherited from the current region and the delay produced by the checkpointing job in the next region. Both checkpointing jobs and normal jobs can produce *delayTime*. The first delay is the time the whole region gets delayed. The sum of the two delays would be converted



(a) *Reduce the delay based on the execution of current region.*

(b) *Update the rate at the end of each region.*

Fig. 6: GOVERNOR *Design*.

into the first delay of the next region if not eliminated before the arrival of the next checkpointing. Both delays are equally important due to the characteristic of accumulation 1.

The two types of delays have different characteristics: the checkpointing delay is more predictable than the delay accumulated from the current region. Thus we deal with the two types of delay differently: The checkpointing delay is predicted based on the historical records. The accumulated delay from current region is calculated as the following. At the point of issuing the signal $(timestamp, \#tuples)$, all small jobs used to reduce *delayTime* in current region have completed and all other jobs in the current region should have a *processingTime* roughly equal to or larger than *interval*. If any of the other jobs produces delay, this *delayTime* would remain for the rest of current region until the reduction of *delayTime* in next region. We use the actual *delayTime* of the current job in current region as the accumulated delay of the next region.

After the value of two delays are predicted, we propose to specify a small input size for a fast reduction of the *delayTime*. Although the two types of delays are predicted differently, the sum of the two delays is calculated as the *delayTime* to be reduced without difference. This approach could not guarantee that the *delayTime* would disappear immediately after one small job with the small input size. This may take a series of the small jobs to reduce. In short, our approach does not reduce the time of the checkpointing, but to reduce the delay time caused by the checkpointing faster, compared to PID.

As the simple example resented in the figure 6a, the checkpointing job configured with the minimum input size is predicted to produce 0.3 second delay. There is one sudden normal job that has a *processingTime* of 1.5 second, causing 0.5 delay. At the point of the completion of the sudden job, the *delayTime* is the sum of the two delays, 0.8 second, which takes the following 2 small jobs to clear the *delayTime*, as each small job can reduce 0.4 $(1 - 0.6)$ second. Therefore, GOVERNOR needs to send 3 signals $(timestamp, miniSize)$ to reduce the *delayTime*.

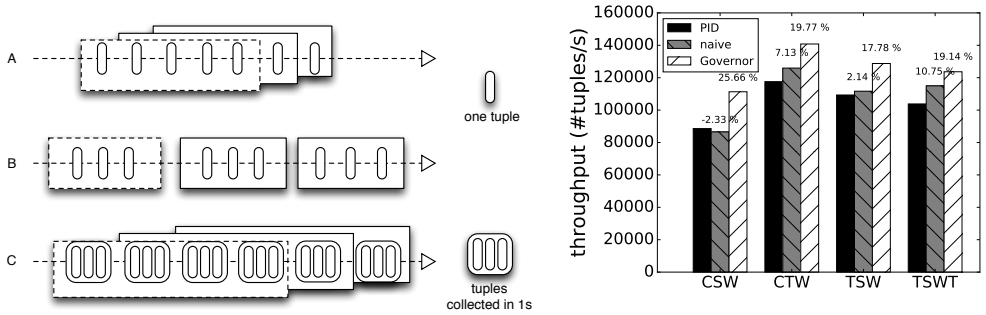
E. Estimation of Region Rate

The $rate$ is the average input size of all jobs for one region. All jobs within one region share one rate, whereas the jobs actually have different input sizes. As shown in the figure 6b, there are three factors that determine the *rate* of a region: *smallJobTime*, *regionDelay*, and *CPTime*. where *smallJobTime* is the total time consumed by the jobs specified with the minimum input size in the procedure of the delay reduction, *regionDelay* is the *delayTime* of the checkpointing job in the region, and *CPTime* is the predicted time of the checkpointing. The *regionDelay* actually can be calculated after the completion of the last job in the current region. The jobs with the minimum input size can only process the minimal tuples, and the *regionDelay* is intended to be reduced to overcome the *delayTime* from increasing. The *rate* is updated when a region ends and next region starts. In the simple example, the sum of the 3 types of delay is 3 seconds, so the rate is updated as $(10 - 3)/10$ of the full input size.

F. Design Comparison

The key difference between GOVERNOR and Spark Streaming's PID controller lies in that GOVERNOR can achieve low *delayTime* and high throughput simultaneously through fine-grain adjustment of batch sizes, whereas PID controller sacrifices throughput for the reduction of *delayTime*. Therefore, GOVERNOR has the following benefits.

Lowering the risk of instability (Low latency). GOVERNOR proactively copes with the *delayTime* so as to avoid future job delay, rather than passively considering how to handle the *delayTime* after it has grown, like the PID controller. Therefore, GOVERNOR can always maintain a low latency, mitigate interferences from the checkpointing and enhance system stability. Another different point is the rapid reduction of the *delayTime* by giving a small batch size to certain jobs, leaving other jobs uninfluenced by *delayTime* in contrast with what PID controller does to reduce the *delayTime* proportionally with the *delayTime* amortized on multiple jobs, leaving system a high risk of accumulating *delayTime*.



(a) *3 window operators on applications A (CSW); B (CTW) ad C (TSW).*

(b) *Overall throughput of the 4 streaming applications.*

Fig. 7: *Streaming applications: A. Count-based Sliding Window (CSW); B. Count-based Tumbling Window (CTW); C. Time-based Sliding Window (TSW); D. Time-based Sliding Window of Top-k (TSWT).*

Improving throughput. After the *delayTime* is cleared, job can utilize the full *interval* for processing, leading to the throughput improvement. We can see that the throughput improvement of GOVERNOR comes from the uninfluenced jobs. The high efficiency of micro-batch processing with a relative long duration is also the main reason why the micro-batch model was introduced into stream processing. However, in PID controller, the *processingTime* of jobs always goes up and down, and thus wastes the efficiency of batch processing.

V. EVALUATION

We now evaluate the efficacy of GOVERNOR experimentally and compare the results against a standard PID controller as well as the naïve approach described in Section III. Our evaluation seeks to answer the following questions:

- **Throughput.** How does GOVERNOR perform on various real-world streaming applications?
- **Overhead.** How large is the overhead of using GOVERNOR?
- **Dynamics.** How does GOVERNOR handle the delay time, improve the throughput, and enhance the system stability?
- **Versatility.** How does GOVERNOR behave under various configurations?

A. Implementation

We have implemented GOVERNOR algorithm within Apache Spark Streaming. The changes to connect GOVERNOR to the underlying codebase were minimal, requiring modification of about 20 Scala classes of the over 1000 implemented by the system. These mainly include the BlockGenerator, ReceivedBlockTracker, RateController classes to allow for blocks of a specific size. The GOVERNOR controller is implemented in about 500 LOC of Scala. For a clear comparison, the GOVERNOR and the baseline approach are implemented independently. We note that GOVERNOR is expected to work in all other micro-batch streaming systems as well, since our backpressure controller is completely transparent to any specifics of the processing component in streaming systems.

B. Experimental setup

In our experiments, we use 6 nodes, each with 8 cores Intel(R) Xeon(R) and 8GiB of DRAM. The version of Spark Streaming we use in our experiments is 1.5, released at the end of 2015. The latest version (Spark Streaming 2.0) released during the preparation of this paper uses the same implementation of the PID controller and so the same changes should apply seamlessly.

The Hadoop version is 2.6 which we use as the storage for the checkpointing on HDFS. We make each run last approximately 10 minutes to ensure a meaningful result. The data we report in our results are the average value across 10 runs. The throughput is calculated as a fixed number of tuples divided by the corresponding processing time.

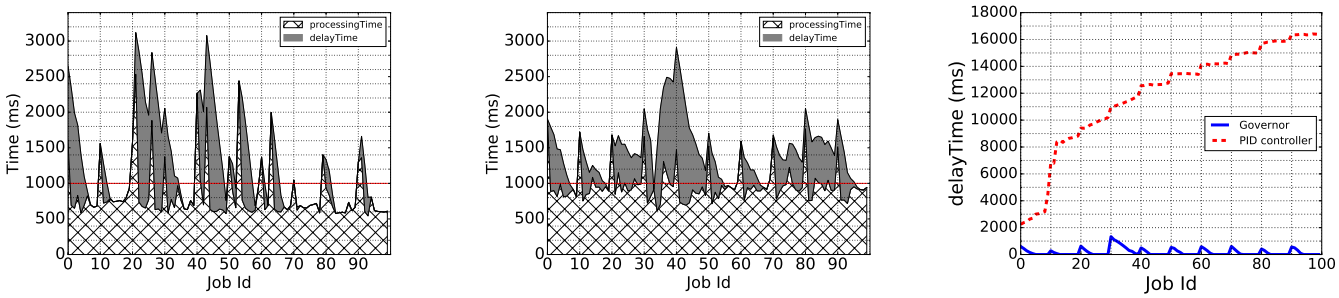
C. Streaming workloads

To assess GOVERNOR comprehensively, we choose several representative streaming applications, described in Figure 7a. The first one is the count-based sliding window (CSW). Every time the window advances by one tuple, with one tuple dequeued and another tuple enqueued, and then aggregation is performed on the tuples that remain in the window. The second one is count-based tumbling window (CTW), which differs from CSW only in that no overlap can exist between windows. The third one is a time-based sliding window (TSW), which is defined by those tuples collected during one time interval. The last one, time-based sliding window of top-k (TSWT), uses the same window as TSW, but where the aggregation is done for top-k. We implemented these applications in Apache Spark Streaming using stateful operators.

The input is real-world data collected from VWAP(Volume-Weighted Average Price) application at IBM Streams group [26]. They are from a trace recorded during a trading day and there are 46 million trade and quote messages for 3032 stock symbols.

D. Overall performance of GOVERNOR

We now study the overall performance impact of GOVERNOR with the 4 streaming applications described above.



(a) processingTime *and* delayTime *of PID controller.*

(b) processingTime *and* delayTime *of Governor.*

(c) delayTime *on window size 120.*

Fig. 8: ***Handling delay.*** delayTime *analysis of PID and* GOVERNOR.

Given the same size of tuples, the sliding window is more computationally intensive than the tumbling window, and the count-based window is more computationally intensive than the time-based window, because the number of aggregations of the former is larger than the latter. Figure 7b shows the overall throughput on the 4 streaming applications, using the three approaches: the baseline PID controller, naïve approach and GOVERNOR. We can see that GOVERNOR always performs better than PID with up to 26% improvement in throughput on CSW.

The variance of the improvement on GOVERNOR is caused by the intensity of the computation involved. If the application is more computationally intensive, the processing consumes more time per tuple, making the result more sensitive to the *delayTime*. By adapting the input size in a fine-grained fashion, GOVERNOR improves the performance of this class of of streaming applications. The flexible control is also why the count-based sliding window obtains the highest improvement.

In contrast, the performance of the baseline approach is less predictable and unstable. On some workload the naïve approach gains 11% improvement; on others it performs worse than the standard PID controller (-2.3%). As discussed in Section III, the input size is difficult to control when only the rate signal is used as feedback.

E. Analysis of delay time

We repeat the previous experiment of count-based sliding window and collect the *delayTime* and *processingTime* for a comparison between PID controller and GOVERNOR.

Figure 8a shows a gap between the *processingTime* and the *interval* on PID controller, which implies the system could potentially process more tuples and boost the throughput. The discrepancy occurs because the rate is forced to decrease dramatically due to the checkpointing delay. Subsequently, the adjustment from a low rate to a normal rate requires many steps because the step-wise updates are conservative and always based on the previous rate. Figure 8b shows that our GOVERNOR makes full use of the *interval*, leaving no separation between the *processingTime* and the *interval*. This demonstrates how GOVERNOR achieves throughput improvement over the standard PID controller.

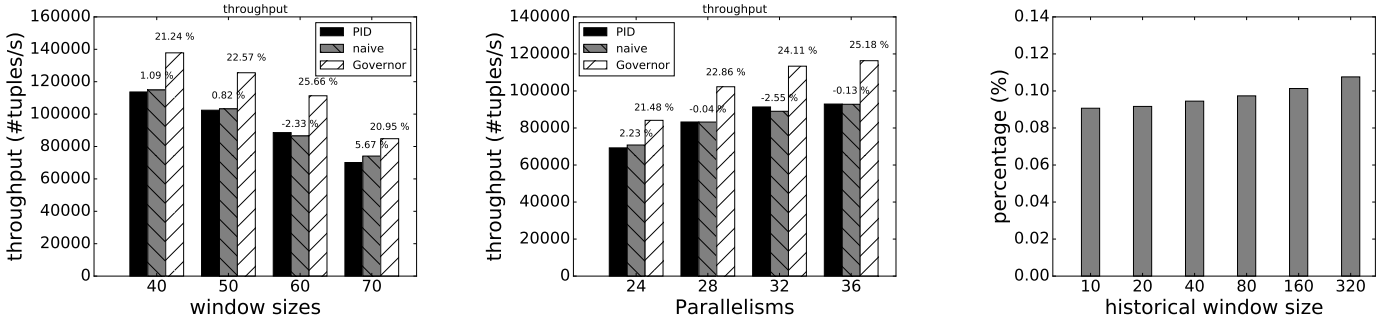
Figure 8c presents the *delayTime* on a large window size 120. We can see that *delayTime* keeps increasing at the PID controller. Conceivably, this results in an ever-increasing set of jobs is buffered in memory, whereas GOVERNOR maintains a controllable *delayTime*. For an apples-to-apples comparison, we configure the minimum input sizes for the two controllers to be the same: 10 tuples. The PID controller constantly maintains the minimum input size after a duration of high *delayTime*, yet the *delayTime* still keeps increasing. The reason is not because the input size is insufficiently small, but rather because the accumulated *delayTime* is too large. This point also motivates our design point for handling *delayTime* proactively, rather than considering the metric only after it has snowballed above a threshold. The throughput of the PID controller is the minimum input size, 10 tuples/sec on average, while GOVERNOR processes about 4000 tuples/sec.

Note that the *delayTime* is, as would be expected at full utilization, always consistent with the throughput: a high *delayTime* implies a low throughput and a low delay implies a high throughput. Both the PID controller and GOVERNOR force a low *processingTime* when detecting looming increases of *delayTime* by reducing the input size of their jobs – an essential task of the backpressure controller. Consequently, for the remainder of the evaluation section, we focus the discussion around throughput.

F. Robustness to various configurations

In this part, we evaluate the impact of two configuration parameters: the window size and the degree of parallelism. These two attributes are important in streaming applications since the window size indicates the cost of the checkpointing, and the degree of parallelism represents the resources available to computation. We use the count-based sliding window (CSW) to study the impact of the two attributes.

Figure 9a depicts the throughput of the three controllers. We can see that the throughput always decreases as the window size grows for each controller. The explanation is that a large window size means a large workload for the



(a) *Throughput with various window sizes.*

(b) *Throughput at various levels of parallelism.*

(c) *Time overhead of* GOVERNOR.

Fig. 9: ***Robustness.*** *Throughput under two configurations and overhead analysis of* GOVERNOR.

checkpointing, which leads to less remaining capacity for the real computation, thus resulting in throughput degradation. As the window size grows, the improvement that GOVERNOR can achieve increases until window size of 70. The reason behind the improvement is that a greater interference from the checkpointing processes provides more opportunity for GOVERNOR. At 70, the improvement stops since the cost the checkpointing is now high enough that GOVERNOR practically reaches its limit. For the naïve approach, the performance is still unstable and unpredictable: sometimes it provides a modest improvement, sometimes it performs worse than PID.

In the next experiment, we study the impact of the resources in terms of the parallelism. The window size is fixed to 60 in this test, so the checkpointing overhead is predictable. We can see in Figure 9b that as the parallelism increases, the throughput also increases for all three controllers. The throughput improvement of GOVERNOR compared to PID keeps growing as parallelism increases since more resources yield more potential for the computation and GOVERNOR exploits those opportunities for optimization.

G. Overhead analysis

Because the backpressure controller is invoked after each job completes, it is important to ensure the cost of GOVERNOR is low. We repeat the previous experiments and collect the time cost of GOVERNOR by adding two timers to measure the start and the end times of the algorithm execution. We vary the historical window size to see how it influences the results, with the outcomes presented in Figure 9c. Note that for most streaming applications, GOVERNOR unnecessarily maintains a large window size as the workload fluctuates over time. The vertical axis represents the average ratio of GOVERNOR time to the *interval* time – as the window size increases from 10 to 320, the ratio increases (less than 0.12%.) We deduce that the overhead of GOVERNOR is negligible. Note that all experimental results above have already included the overhead.

VI. RELATED WORK

In this section, we compare and contrast Governor with related work in the literature.

A. Backpressure algorithms

In network area, backpressure mechanisms [29] [10] [27] have been used as a scheduling policy that maximizes the throughput of multi-hop networks. In systems area, backpressure is also an important technique used to indicate performance bottleneck to better balance load. Flexible Filter [6] aims to improve the system throughput by efficient mapping of the stream tasks and dynamic load balance. Sanchez *et al.* [25] present a scheduler for pipeline-parallel programs that performs fine-grain dynamic load balancing efficiently based on backpressure. Unlike these backpressure techniques above, GOVERNOR focuses on the adjustment of ingestion rate to improve the throughput, and sustain a high stability for streaming systems.

B. Reducing the cost of checkpointing

Checkpointing optimizations have been an active area of research for several decades. Among the many techniques being exploited are efficient writing of the checkpointed data, fast recovery, configuration tuning [13, 22] and to apply various compression techniques to checkpointing. Incremental checkpointing is a canonical way to optimize the checkpointing by only operating on the checkpoint difference [2] [21] [18]. Fast recovery [5] [18] mechanisms focus on the performance of reading checkpoints to speed up the recovery. Configuration tuning includes the checkpoint time interval [30] – the size of incremental checkpoint [18] [**?**]. Our approach coordinates backpressure with checkpointing, which is complementary to the reduction of checkpointing costs. GOVERNOR can work synergistically with these techniques.

C. Streaming Systems and extensible backpressure

Dynamic batch sizing [8] achieves a high stability and a good throughput through the adjustment of batch interval, which may result in a high user-perceived latency. Instead, GOVERNOR focuses on the adjustment of input size of jobs to guarantee a constant latency. Many streaming systems, including StreamScope [19], Naiad [20], TimeStream [23], S4 [24], IBM Streams [15], Apache Flink [12], Apache Storm

[28], and Twitter Heron [16], employ a topology backpressure mechanism relying on the TCP windowing mechanism to detect any slowing down. While piggybacking on TCP is straightforward and simple, it leaves most of the complexity of backpressure configuration to the users, such as buffer size of the buffer and the "watermark". These parameters directly determine the number of messages that the system maintains in flight during the runtime, which are tricky for the users to tune as they are application-dependent. GOVERNOR controller frees the users from manually tweaking these parameters.

VII. CONCLUSION

Stream processing systems have become the backbone of big data analytics. Here, we described how the need for persistence drives periodic checkpointing, and how this checkpointing process can affect performance of the stream processing systems.

To overcome the performance degradation, we sketched our design for GOVERNOR: a backpressure controller for stream processing systems that can cope with the dynamically varying checkpointing overheads. By collecting historical information of the checkpointing jobs and the normal micro-batch jobs, GOVERNOR predicts and reduces the processing delay caused by checkpointing proactively. The feedback from the smarter controller in turn lowers the risk of system instability and improves overall throughput. Experiments with an implementation of GOVERNOR in Apache Spark Streaming have shown an overall performance improvement of up to 26% for representative streaming operators and real-world workloads, with negligible overhead. In our future work, we plan to extend GOVERNOR to the backpressure mechanism of streaming systems using continuous operator model.

VIII. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for constructive feedback on the paper. Our work is partially supported by NSF CAREER award #1553579 and funds from Georgia Institute of Technology and Emory University.

REFERENCES

[1] IBM Corporation. Streams. http://www-03.ibm.com/software/products/en/ibm-streams.

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *18th SC*, pages 277–286. ACM, 2004.

[3] J. Basilio and S. Matos. Design of pi and pid controllers with transient performance specification. *IEEE Transactions on Education*, 45(4):364–370, 2002.

[4] S. Bennett. Development of the pid controller. *IEEE Control Systems*, 13(6):58–62, 1993.

[5] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *2011 SIGMOD*, pages 265–276. ACM, 2011.

[6] R. L. Collins and L. P. Carloni. Flexible filters: load balancing through backpressure for stream programs. In *The seventh ACM international conference on Embedded software*, pages 205–214. ACM, 2009.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[8] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *SOCC*, pages 1–13. ACM, 2014.

[9] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *SOCC*, SOCC '14, pages 16:1–16:13. 2014.

[10] A. Dvir and A. V. Vasilakos. Backpressure-based routing protocol for dtns. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 405–406. ACM, 2010.

[11] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *2014 USENIX ATC)*, pages 49–60. 2014.

[12] Flink. http://flink.apache.org/.

[13] B. Gedik. Discriminative fine-grained mixing for adaptive compression of data streams. *IEEE Transactions on Computers*, 63(9):2228–2244, 2014.

[14] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SOCC*, pages 63–74. ACM, 2010.

[15] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, et al. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[16] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *2015 SIGMOD*, pages 239–250. ACM, 2015.

[17] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[18] H. Li, L. Pang, and Z. Wang. Two-level incremental checkpoint recovery scheme for reducing system total overheads. *PloS one*, 9(8):e104591, 2014.

[19] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI 16*, pages 439–453, 2016.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[21] B. Nicolae and F. Cappello. Ai-ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *The 22nd HPDC*, pages 155–166. ACM, 2013.

[22] J. S. Plank, J. Xu, and R. H. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical report, Citeseer, 1995.

[23] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[24] S4. http://incubator.apache.org/s4/.

[25] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *PACT 2011 International Conference on*, pages 22–32. IEEE, 2011.

[26] P. Selo, Y. Park, S. Parekh, C. Venkatramani, H. K. Pyla, and F. Zheng. Adding stream processing system flexibility to exploit low-overhead communication systems. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–8. IEEE, 2010.

[27] L. Shenghui, W. Chenqing, and C. Nan. Research on media stream transmission based on back-pressure in mobile wireless network. *International Journal of Future Generation Communication and Networking*, 6(5):53–64, 2013.

[28] Storm. http://storm.apache.org/.

[29] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE transactions on automatic control*, 37 (12):1936–1948, 1992.

[30] S. Yi, J. Heo, Y. Cho, and J. Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *The 2006 ACM symposium on Applied computing*, pages 1472–1476. ACM, 2006.

[31] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.