

Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers

Huayong Wang, Li-Shiuan Peh, Emmanouil Koukoumidis
*Computer Science And Artificial Intelligence Lab
 MIT*

Email: huayongw@smart.mit.edu, {peh,koukou}@csail.mit.edu

Shao Tao, Mun Choon Chan
School of Computing

National University of Singapore
 Email: {shaot,chanmc}@comp.nus.edu.sg

Abstract—Large-scale failures are commonplace in commodity data centers, the major platforms for Distributed Stream Processing Systems (DSPSs). Yet, most DSPSs can only handle single-node failures. Here, we propose Meteor Shower, a new fault-tolerant DSPS that overcomes large-scale burst failures while improving overall performance. Meteor Shower is based on checkpoints. Unlike previous schemes, Meteor Shower orchestrates operators' checkpointing activities through tokens. The tokens originate from source operators, trickle down the stream graph, triggering each operator that receives these tokens to checkpoint its own state¹. Meteor Shower is a suite of three new techniques: 1) source preservation, 2) parallel, asynchronous checkpointing, and 3) application-aware checkpointing. Source preservation allows Meteor Shower to avoid the overhead of redundant tuple saving in prior schemes; parallel, asynchronous checkpointing enables Meteor Shower operators to continue processing streams during a checkpoint; while application-aware checkpointing lets Meteor Shower learn the changing pattern of operators' state size and initiate checkpoints only when the state size is minimal. All three techniques together enable Meteor Shower to improve throughput by 226% and lower latency by 57% vs prior state-of-the-art. Our results were measured on a prototype implementation running three real world applications in the Amazon EC2 Cloud.

Keywords—stream computing; fault tolerance; reliability;

I. INTRODUCTION

In recent years, many real-world applications, such as intelligent transportation and environmental monitoring, have to process a large volume of data in a timely fashion. Distributed Stream Processing Systems (DSPSs) are well-suited to support these applications since DSPSs offer high throughput and low latency for data processing. Commodity clusters, built with off-the-shelf hardware, are the major platforms for today's DSPSs. With the ballooning demands on computational resources, the size of DSPS clusters can grow from dozens to thousands of nodes (data center scale).

When DSPSs approach the scale of data centers, fault tolerance becomes a major challenge. In the past decade, several fault tolerance schemes have been proposed for DSPSs. They are either replication-based [1, 2, 3] or checkpoint-based [1, 4, 5, 6]. In replication-based schemes, a DSPS runs $k+1$ replicas of each operator to tolerate up to k

simultaneous failures (k -fault tolerant). Clearly, replication-based schemes take up substantial computational resources, and are not economically viable for large-scale failures. Checkpoint-based schemes perform two key functions: 1) periodic checkpoint (snapshot) of each operator's running state and 2) input preservation², i.e. every operator retains its output tuples until these tuples have been checkpointed by the downstream operators. When an operator fails, the operator is restarted from its Most Recent Checkpoint (MRC). Its upstream operators then resend all the tuples that the failed operator had processed since its MRC. The restarted operator rebuilds the same state as the failed operator after it processes all these tuples.

The previously proposed fault tolerance schemes do not fit well in today's stream processing environment for two main reasons. First, the failure model of a commodity data center is quite different from the assumptions made by previous work. Previous work assumes small-scale (usually single node) failures and is based on the belief that 1-safety guarantee is sufficient for most real-world applications [7]. However, after a careful study of Google's data center [8, 9, 10, 11], we found that about 10% failures in the data center are correlated and occur in bursts (Section II-B1). In this context of correlated failures, the single node failure assumption of previous work does not hold. Second, prior checkpoint-based schemes impose substantial overhead on the performance of DSPS applications. An ideal DSPS should be resilient to many faults, while adding little runtime overhead for fault tolerance.

This paper proposes a new fault tolerance scheme, called Meteor Shower, to solve the reliability problem of DSPSs in the context of commodity data centers. Meteor Shower introduces tokens to coordinate the checkpointing activities. The tokens are generated by source operators or a controller. The tokens trickle down the stream graph, triggering each operator to checkpoint its own state. We show that Meteor Shower enables DSPSs to overcome large-scale burst failures with low performance overhead: 226% increased throughput and 57% reduced latency vs prior state-of-the-art. We summarize the major differences between Meteor Shower and previous schemes as follows.

¹The sprinkling of tokens resembles a meteor shower.

²The name "input preservation" was proposed by [1].

1) **Better common case performance:** Common case refers to the situation where no failure occurs. In the common case, input preservation is costly. For a chain of n operators where operator o_i forwards tuples to o_{i+1} , every tuple has to be saved $n-1$ times by operators o_1 to o_{n-1} . Since the data stream is continuous, the burden of saving such a large volume of data is significant and degrades overall performance. In Meteor Shower, only the source operators preserve the output tuples, which is called *source preservation*. Source preservation is sufficient as Meteor Shower ensures that all operators in an application are checkpointed together in a unified way. In case of a failure, all the operators in this application are recovered simultaneously and the source operators replay the preserved tuples. Compared with input preservation, source preservation incurs less runtime overhead. Our experiments show that source preservation increases throughput by 35% and reduces latency by 9%, on average (Fig. 12 and Fig. 13).

2) **Non-disruptive checkpoint:** A common concern for checkpoint-based schemes, like Meteor Shower, is checkpoint time. Checkpoints are performed when operators are busy processing incoming tuples. Therefore, checkpointing activities can disrupt or even suspend normal stream processing, thus degrading overall performance. In Meteor Shower, the checkpointing work is done in parallel and asynchronously. When an operator starts checkpointing, it forks a new process which shares the memory with the parent process. Operating system guarantees the data consistency through the copy-on-write mechanism. The child process saves the operator's state, while the parent process continues processing streams. This asynchronous approach offloads the checkpointing task from the critical path to a helper process, helping to eliminate the negative impact of checkpointing on normal stream processing (Section IV-B, Fig. 15). Our experiments show that *parallel, asynchronous checkpointing* increases throughput by 28% and reduces latency by 33%, on average, vs. the state-of-the-art (Fig. 12 and Fig. 13).

3) **Fast recovery:** Another major concern for checkpoint-based schemes is recovery time. The recovery time may be long in Meteor Shower since it recovers all operators together. In this paper, we propose a new technique to solve this problem, called *application-aware checkpointing*. Application-aware checkpointing is based on two observations: 1) stream applications have evolved from early primitive monitoring applications to today's sophisticated data analysis systems; 2) data mining and image processing algorithms, which constitute the kernels of typical data analysis applications, manipulate data in batches. At the boundaries of the batches, the operator state is puny since the operator holds no intermediate results. Therefore, Meteor Shower monitors the state size of each operator and learns its changing pattern. The learnt patterns allow Meteor Shower to make informed decisions about the best time to perform a checkpoint resulting in minimal checkpointed state. For

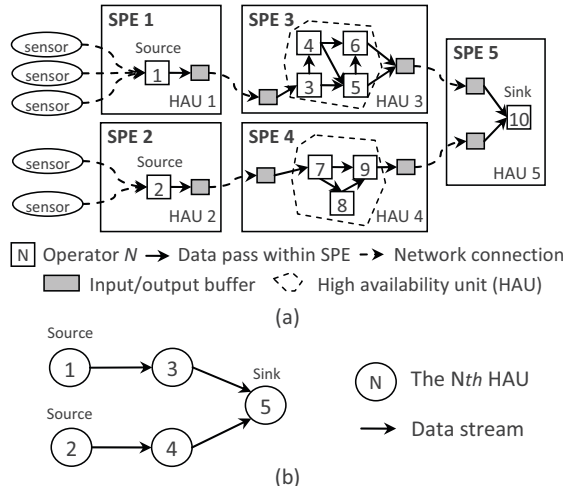


Figure 1. Stream application consisting of ten operators. (a) Stream application represented by operators and query network. (b) Stream application represented by HAUs and high-level query network.

example, application-aware checkpointing can reduce the checkpointed state in the three case study applications by about 100%, 50% and 80% respectively (Section II-B2, Fig. 5). Thanks to the small size of the checkpointed state, the recovery time is also significantly reduced.

The remainder of this paper is organized as follows. Section II introduces the background and the motivation. Section III describes Meteor Shower. Section IV dissects the experimental results. Section V compares our work with the related research work and Section VI offers our conclusions.

II. BACKGROUND AND MOTIVATION

A. Distributed Stream Processing System

A DSPS consists of operators and connections between operators. Fig. 1.a illustrates a DSPS and a stream application running on the DSPS. The application contains ten operators. Each operator is executed repeatedly to process the incoming data. Whenever an operator finishes processing a unit of input data, it produces the output data and sends them to the next operator. Each unit of data passed between operators is called a tuple. The tuples sent in a connection between two operators form a data stream. A directed acyclic graph, termed query network, specifies the producer-consumer relations between operators.

In practice, multiple operators can run on the same computing node. One or more Stream Process Engines (SPEs) on the node manage these operators. The group of operators within an SPE is called a High Availability Unit (HAU) [4]. HAU is the smallest unit of work that can be checkpointed and recovered independently. The state of an HAU is the sum of all its constituent operators' states. If multiple HAUs are on the same node, they are regarded as independent units for checkpointing. The structure of a stream application can

Table I
COMMODITY DATA CENTER FAILURE MODELS (AFN100).

Failure Sources	Google's Data Center	Abe Cluster
Network *	>300	~250
Environment *	100~150	NA
Ooops	~100	~40
Disk	1.7~8.6	2~6
Memory	1.3	NA

* The major reasons for large-scale burst failures.

be represented at a higher level based on the interaction between the HAUs (Fig. 1.b).

B. Motivation

This paper is motivated by important observations about the failure model of commodity data centers and the characteristics of stream applications.

1) *Failure Model:* Our target platform for DSPSs is commodity data centers, like Google's, that are built with off-the-shelf computers, storage and networks. Based on statistics about Google's data centers [8, 9, 10, 11], Table I describes the failure model of a Google's data center with 2400+ nodes (30+ racks and 80 blade servers per rack). In order to make comparisons, all values in Table I are translated to a common unit – Annual Failure Number per 100 nodes (AFN100). AFN100 is the average number of node failures observed across 100 nodes running through a year. AFN100 is broken down into the different causes of failure. Network failures are due to rack, switch, router and DNS malfunctions. Occasional network packet losses are not viewed as failures. Environment refers to failures caused by external reasons, such as power outage, overheating and maintenance. Ooops represents failures generated by software, man created mistakes, and other unknown reasons. Disk and memory failures, unlike the above failures, are usually correctable. For example, the majority of memory failures are single-bit soft errors, which can be rectified by error correction code (ECC) [10]. Many reported disk failures, such as scan, seek and CRC errors, are actually benign and cause no data loss [9]. Since correctable errors have no impact on applications, Table I only accounts for uncorrectable disk and memory errors.

Here is an example on how the network AFN100 is calculated in Google's data center. According to [8], in one year, there are one network rewiring (5% of nodes down), twenty rack failures (80 nodes disconnected each time), five rack unsteadiness (80 nodes affected each time, 50% packet loss), fifteen router failures or reloads, and eight network maintenances. We conservatively assume that only 10% of the nodes are affected each time in the last two cases. Therefore, there are 7640 network failures in total: $AFN100 = 7640/2400 * 100 > 300$. The actual AFN100 in the field could be higher because dozens of DNS blips are not considered in our calculation.

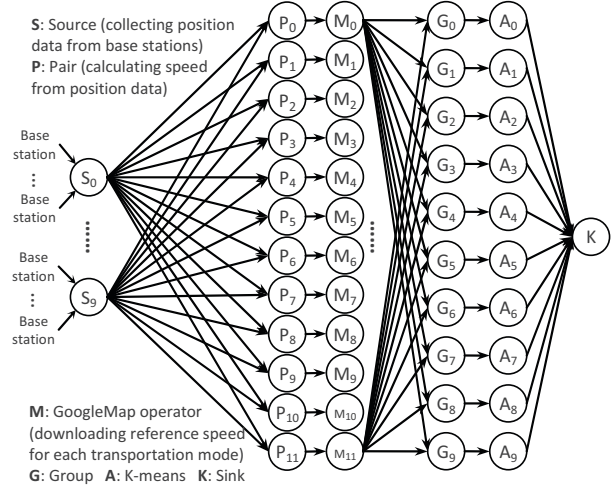


Figure 2. Query network of TMI. Each operator constitutes an HAU. Each GoogleMap operator connects to all Group operators. For clarity, only the connections starting from M_0 and M_{11} are shown.

From Google's statistics, we make two key observations. 1) The major causes for the failures in a Google's data center are network, environment and ooops. 2) Failures can be correlated. For example, a rack failure can immediately disconnect 80 nodes from the network, and takes 1~6 hours to recover. In fact, about 10% failures are part of a correlated burst, and large bursts are highly rack-correlated or power-correlated [11].

Table I gives another example of a large-scale commodity cluster – the Abe cluster at National Center for Supercomputing Applications (NCSA) [12]. Its AFN100 is lower than the Google's data center because Abe adopts InfiniBand network and RAID6 storage. Nevertheless, the same observations also apply for the Abe cluster.

In summary, large-scale burst failures are not rare in a commodity data center because of the network and environment reasons. It is thus necessary for DSPSs running in data centers to deal with large-scale burst failures.

2) *Application Characteristics:* The first application is Transportation Mode Inference (TMI) [13]. It collects the position data of mobile phones from base stations. Using the data, TMI infers the transportation mode (driving, taking bus, walking or remaining still) of mobile phone bearers in real time. Fig. 2 illustrates the query network of TMI. The kernel of TMI is the k-means clustering algorithm. The k-means operators manipulate data in batches. In each N -minute-long time window, a k-means operator retains input tuples in an internal pool and clusters the tuples at the end of the time window. After clustering, the operator discards the tuples in the pool. Therefore, the state size of TMI changes periodically. The dataset used in TMI consists of 829 millions anonymous location records collected from over 100 base stations in one month.

The second application is Bus Capacity Prediction (BCP).

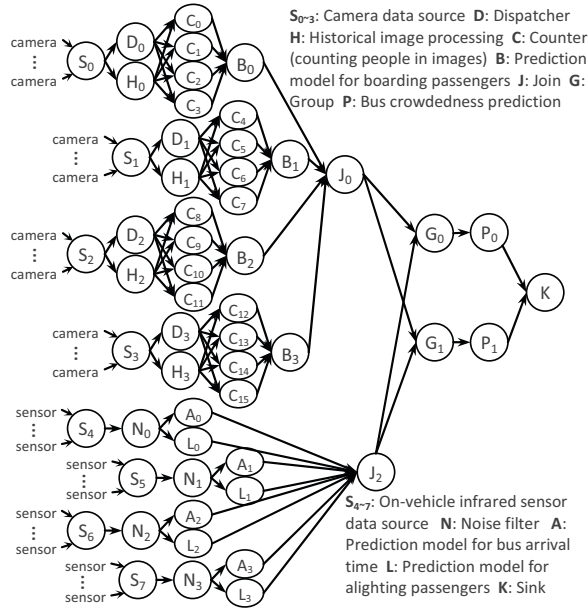


Figure 3. Query network of BCP. Each operator constitutes an HAU.

It predicts how crowded a bus will be based on the number of passengers on the bus and at the next few bus stops. BCP uses cameras installed at bus stops to take pictures and count the passengers in the pictures. In order to filter out the pedestrians who are just walking by and distinguish occluded people in the pictures, BCP’s image processing algorithm needs to analyze multiple successive images from the same camera. Therefore, BCP internally saves the images for each camera. These saved images, which are used to improve detection accuracy, become BCP’s running state. These images, however, are removed upon a bus arrival, because the passenger counts waiting at a bus stop before and after a bus arrival are usually significantly different. The image accumulation and removal cause the state size to fluctuate. Fig. 3 illustrates the query network of BCP. The H operators manage the historical images for each camera. A prototype BCP system is currently running on the National University of Singapore campus buses.

The third application is SignalGuru, a driver assistance system [14]. It predicts the transition time of a traffic light at an intersection and advises drivers on the optimal speed they should maintain so that the light is green when they arrive at the intersection. In this way, the drivers can cruise through the intersections without stopping, decreasing significantly their fuel consumption. A proof of concept SignalGuru has been implemented on iPhone devices [14]. Here, a large-scale SignalGuru is implemented on a DSPS, with inputs coming from phones and videos captured at actual deployments at ten intersections in Cambridge and Singapore. Such a DSPS implementation enables coverage of a much larger area, and can potentially increase prediction

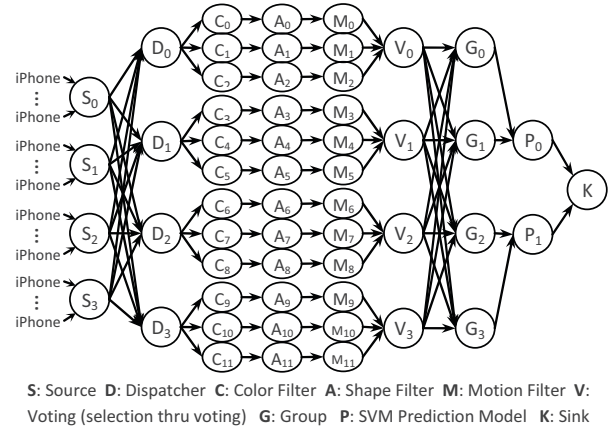


Figure 4. Query network of SignalGuru. Each operator constitutes an HAU.

accuracy since phones will have global knowledge from all other phones.

SignalGuru leverages windshield-mounted iPhones devices to take pictures of an intersection. It then detects the traffic signals in the images and learns the pattern of the signal transitions. The image processing algorithm uses motion filtering to eliminate noises because traffic lights always have fixed positions at intersections. The motion filtering operators preserve all pictures taken by an iPhone at a specific intersection, until the vehicle carrying the iPhone device leaves the intersection. The preserved images become the operator’s state as long as the vehicle remains in the vicinity of an intersection (usually 10~40 seconds). Fig. 4 illustrates the query network of SignalGuru.

Fig. 5 shows the state size of these applications over a time window. In TMI ($N=10$), the maximum size is over 300MB, while the minimum size is close to zero. If checkpoints are performed randomly, the average size of the checkpointed state is about 150MB. In BCP, the state size ranges between 100MB and 700MB with an average of 400MB. In SignalGuru, the state size ranges from 200MB to 2GB with an average of 1GB. The three applications represent low, medium and high stream workload respectively.

We summarize two key characteristics of these stream applications for data analysis. First, these applications exhibit a pattern of data aggregation. That is, they have a large number of data sources and at each data source the input data rate is low. However, the aggregated data are large. Therefore, downstream HAUs are often busy. As a result, if it is necessary to save tuples at least once, it is better to save them someplace near the sources. Second, real-time data mining and image processing algorithms usually store historical data internally for a while, then discard the data when they become stale. Consequently, the state size of the operators implementing these algorithms fluctuates significantly over time. If a checkpoint is performed just after the internal data is discarded, the checkpointed state

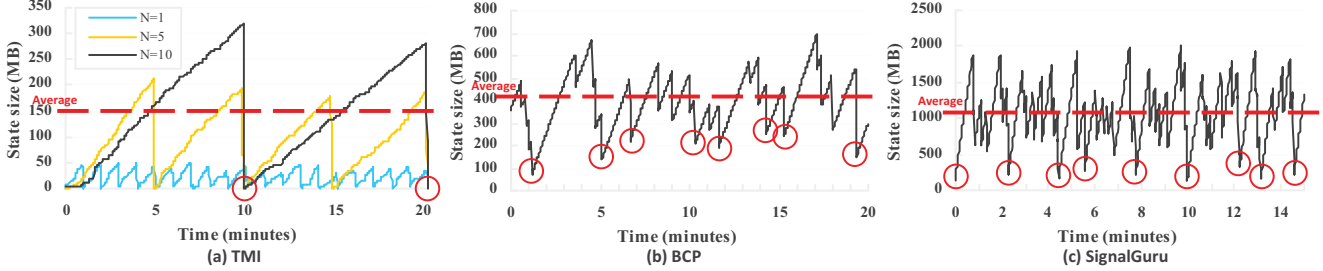


Figure 5. Fluctuation in state size. Red circles mark the local minima of the state size and red dotted lines indicate the average state sizes.

will be minimal. The data aggregation characteristic inspires the idea of source preservation, and the varying state characteristic prompts application-aware checkpointing. We believe these applications are representative since the kernels of these applications are commonly used algorithms.

3) *Fault Tolerance Overhead*: For comparison, we define a baseline system that represents the state-of-the-art checkpoint-based schemes [1, 4, 5, 6]. In the baseline system, HAUs perform checkpoints independently. Each HAU selects randomly the time for its first checkpoint. After that, each HAU checkpoints its state periodically. For all HAUs, the checkpoint period is the same, but the time for the first checkpoint is different. The default value of checkpoint period is 200 seconds. It can be adjusted as we do in the following experiments. Using input preservation, each HAU preserves output tuples in an in-memory buffer. The buffer has a limited size in order to curb memory contention. Once the buffer is full, the buffered data are dumped into the local disk. The buffer size in this implementation is 50MB. A larger buffer reduces the frequency of disk I/O, but does not reduce the amount of data written to the disk. Therefore, further enlarging buffers shows little performance improvement in our experiments. The checkpointed state is saved on a shared storage node. An HAU sends a message back to its upstream neighbors once it completes a checkpoint. The message informs the upstream neighbors of the checkpointed tuples, so these tuples are discarded from the buffer and disk of the upstream neighbors. In the baseline system, HAUs perform checkpoints synchronously. That is, an HAU suspends its normal stream processing during a checkpoint, and resumes the processing after the checkpoint is thoroughly completed. By comparing Meteor Shower and the baseline system, we hope to evaluate source preservation vs. input preservation, parallel and asynchronous checkpointing vs. synchronous checkpointing, and application-aware checkpointing vs. random checkpointing.

III. METEOR SHOWER DESIGN

This section introduces the assumptions made by Meteor Shower and three variants of Meteor Shower.

Meteor Shower makes several assumptions about the software errors, network, storage and workload. Meteor Shower handles fail-stop failures. That is, a software error is immediately detected and the operator stops functioning

without sending spurious output. Meteor Shower assumes that TCP/IP protocol is used for the network communication. Network packets are delivered in-order and will not be lost silently. According to the failure model in Section II-B1, storage is deemed reliable enough. Meteor Shower assumes that there is a shared storage system in the data center where computing nodes can share data. Furthermore, Meteor Shower assumes the storage system to be reliable except that the network between the storage system and computing nodes can fail. In practice, the shared storage system can be implemented by a central storage system or a distributed storage system like Google’s File System (GFS). The overall workload of a stream application should be below the system’s maximum processing capability. Once an application has recovered from a failure, it can process the replayed tuples faster than usual to catch up with regular processing. Meteor Shower does not consider long-term overload situations because they require load shedding [15].

A. Basic Meteor Shower

Basic Meteor Shower, denoted by MS-src, features source preservation. MS-src can be described in three steps. First, every source HAU checkpoints its state and sends a token to each of its downstream neighbors. A token is a piece of data embedded in the dataflow as an extra field in a tuple. It conveys a checkpoint command, and incurs very small overhead. Second, every HAU checkpoints its state once it receives tokens from all upstream neighbors. The checkpoint is performed synchronously. After that, the HAU forwards a token to each of its downstream neighbors. In this way, the tokens are propagated in the query network. When all HAUs have finished their own checkpoints, the checkpoint of this application is completed. To avoid confusion, we name the checkpoint of a single HAU “individual checkpoint”. An application’s checkpoint contains the individual checkpoints of all HAUs belonging to this application. Third, every source HAU preserves all the tuples generated since the Most Recent Checkpoint (MRC). The source HAU saves these tuples in stable storage before sending them out, which guarantees that the preserved tuples are still accessible even if the source HAU fails. The preserved tuples and the HAU states are saved in the shared storage system, and optionally saved again in the local disks of the corresponding nodes.

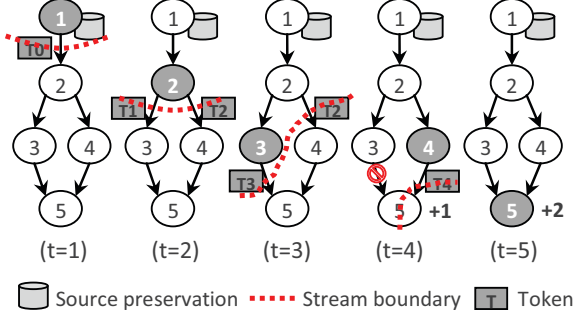


Figure 6. Execution snapshots illustrating MS-src. For clarity, the tuples preceding and succeeding the token in each stream are not shown.

Fig. 6 illustrates the execution snapshots of MS-src. At time instant 1, the source HAU checkpoints its state and sends out the token T_0 to its downstream neighbor. At time instant 2, HAU 2 receives the token, checkpoints its state and forwards the token to HAUs 3 and 4. At time instant 3, HAU 3 does the same thing and forwards the token down. Because HAU 4 runs more slowly than HAU 3, token T_2 has not been processed yet. At time instant 4, HAU 4 receives the token T_2 , checkpoints its state and forwards the token down. HAU 5 receives one token T_3 from HAU 3. Since HAU 5 has two upstream neighbors, it cannot start its checkpoint at the moment. HAU 5 then stops processing tuples from HAU 3, which guarantees that the state of HAU 5 is not corrupted by any tuple succeeding the token. HAU 5 can still process tuples from HAU 4 because HAU 5 has not received any token from HAU 4. At time instant 5, HAU 5 receives tokens from both upstream neighbors and then checkpoints its state. After HAU 5 finishes its checkpoint, the checkpoint of this application is completed.

The token plays an important role in Meteor Shower. A token indicates a stream boundary (depicted by dotted lines in Fig. 6). That is, in a stream between a pair of neighboring HAUs, the tuples preceding the token are handled by the downstream HAU while the tuples succeeding the token are handled by the upstream HAU. The stream boundary guarantees that no tuple will be missed or processed twice when the application is recovered from a failure.

Failure detection in Meteor Shower relies on a controller. The controller runs on the same node as the shared storage system. If the shared storage system is implemented by GFS, the controller runs on the master node of GFS [16]. As pointed out by [3], the controller is not necessarily a single point of failure. Hot standby architecture [17] and active standby technique [18] can provide redundancy for the controller. The controller’s job is to periodically ping the source nodes (the nodes on which source HAUs run). A source node is deemed failed if it does not respond for a timeout period. The other computing nodes in the DSPS are monitored by their upstream neighbors respectively. A node can also report its own failure actively to its upstream neighbor. For example, when a node is disconnected from

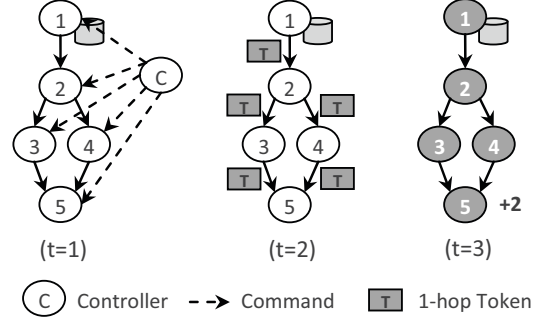


Figure 7. Execution snapshots illustrating MS-src+ap.

the shared storage system, it notifies its upstream neighbor.

If a failure has been detected, the controller sends notifications to all HAUs that are alive. All HAUs must then be restored to their MRC. When nodes fail, the HAUs on those failed nodes are restarted on other healthy nodes. Recovery starts by reading the checkpointed state from the local disk. Where the local disk cannot provide the data, the restarted HAUs will read the data from the shared storage system. Source HAUs then replay the preserved tuples. All downstream HAUs reprocess these tuples and the application therefore recovers its state.

B. Parallel, Asynchronous Meteor Shower

In MS-src, individual checkpoints are performed one after another in the order in which the token visits HAUs. A way to shorten the checkpoint time and reduce the disruption to normal stream processing is to perform individual checkpoints in parallel and asynchronously. We thus propose an enhanced design of Meteor Shower, denoted by MS-src+ap.

Fig. 7 illustrates the parallel aspect of MS-src+ap. The controller sends a token command to every HAU simultaneously. After receiving the command, the HAU generates a token for each of its downstream neighbors, and then waits for the incoming tokens from its upstream neighbors. Once an HAU receives the tokens from all upstream neighbors, the HAU checkpoints its state. Unlike MS-src, the incoming tokens are not forwarded further to downstream HAUs. Instead, they are discarded after the individual checkpoint starts. Tokens in MS-src+ap are hence called 1-hop tokens.

Fig. 8 illustrates the asynchronous aspect of MS-src+ap. It shows the token handling and asynchronous checkpointing in an HAU. The HAU has two upstream and downstream neighbors, corresponding to the two input and output buffers respectively. At time instant 1, the HAU receives the token command. Then, two 1-hop tokens T_0 and T_1 are immediately inserted to the output buffers and placed at the head of the queue. After that, the HAU keeps processing the tuples in both input buffers and waits for incoming tokens. The first incoming token T_2 is received by input buffer 1 at time instant 2. The HAU continues the normal tuple processing until it processes T_2 at time instant 3. The HAU then stops processing tuples in input buffer 1 because a token can be

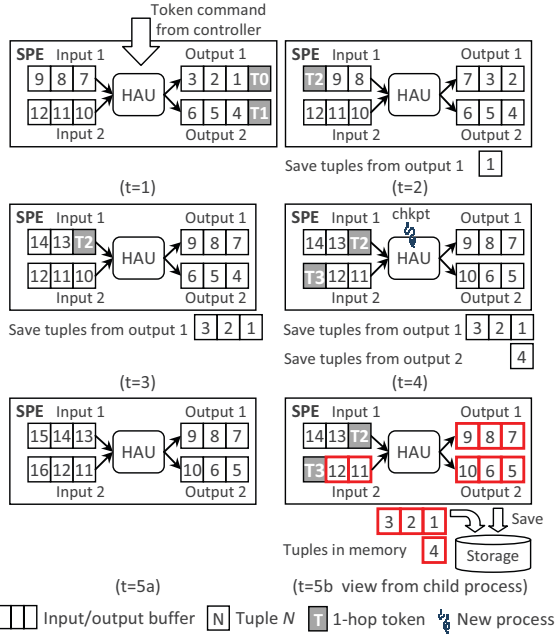


Figure 8. Token handling and asynchronous checkpointing in HAU. 1) Command arrival. 2) First token arrival. 3) Token waiting. 4) Second token arrival. 5a) Process creation and token erasure. 5b) Asynchronous checkpointing.

handled only when the tokens from all upstream neighbors have been received. The HAU can still process the tuples in input buffer 2 because input buffer 2 has not received a token yet. Another incoming token T_3 is received by input buffer 2 at time instant 4. As both tokens have been received, the HAU spawns a child process to complete the checkpointing asynchronously. This child process shares the memory with its parent process, with the operating system guaranteeing data consistency through the copy-on-write mechanism [19]. The child process then copies and saves state, while the parent process continues normal stream processing. After the child process is created, the tokens in the input buffers are erased immediately, as at time instant 5a. From now on, the HAU returns to its normal execution.

In addition to the above actions, all tuples that are sent from time instant 1 to 4 retain a local copy in memory. The retained tuples can be deleted at time instant 5a after the child process is created. Let us turn to the child process at time instant 5b. Besides the HAU’s original state, the child process also checkpoints all the tuples between the incoming tokens (T_2 and T_3) and the output tokens (T_0 and T_1). They are tuples 1~12 in this case, as shown by the red rectangles in Fig. 8. Some of them are still in the buffers. These tuples are treated as a part of the HAU state. They will be restored if the HAU recovers its state in the future. Please note that MS-src+ap is different from input preservation although they both save some tuples in each HAU. In MS-src+ap, only tuples between the input and output tokens are saved.

But in input preservation, all tuples between two adjacent checkpoints are saved. MS-src+ap saves a negligible number of tuples in comparison with input preservation.

C. Application-aware Meteor Shower

To reduce the checkpointing overhead of MS-src+ap, we propose Meteor Shower with application-aware checkpointing, denoted by MS-src+ap+aa. MS-src+ap+aa tracks applications’ state size, then chooses the right time for checkpointing, thus resulting in less checkpointed state and lower network traffic.

1) *Code Generation at Compile Time:* MS-src+ap+aa provides a precompiler that processes the source code of stream applications before the standard C++ compiler. The precompiler scans the source code, recognizes operator state and generates stub code that will be executed at runtime for the purpose of state size tracking.

A tuple is a C++ class whose data members are to be one of the following types. 1) Basic types: integer, floating-point number and character string. 2) Tuple types: another tuple or a pointer to another tuple. 3) Array types: an array whose elements are of basic types or tuple types. Our precompiler disallows cyclic recursive definitions – tuple A contains tuple B and tuple B contains tuple A. Because every data member in a tuple has an explicit type, the precompiler can generate a member function *size()* for each tuple class. The function returns the size of the memory consumed by this tuple.

An operator is a C++ class inherited from a common base class *stream_operator*, as shown in Fig. 9. Developers implement the operator’s processing logic: function *input_port_0()* processes the tuples from the first upstream neighbor, function *input_port_1()* processes the tuples from the second, and so on. An SPE calls one of these functions whenever an input tuple is received.

Operator state is the data members of an operator class. These data members are usually tuples managed by standard C++ containers. The precompiler recognizes the class members and automatically generates a member function *state_size()* for each operator class. By default, function *state_size()* samples the elements of each data structure and estimates its total size. For example, function *state_size()* in Fig. 9 takes three samples in the vector *data* and calculates the total size of the vector. We believe that sampling is suitable for estimating state size as stream applications normally process sensor data, which have fixed format and constant size. Developers can give hints to the precompiler in the comments after the variable declarations. The comment “state sample= N ” indicates that function *state_size()* should take N random samples when estimating the total size of a data structure. The element size of a data structure can also be specified explicitly if it is known a priori, like the comment “state element_size=1024” in Fig. 9. For user-defined data structures, like *my_hashtable*, developers must specify how to get the number of the elements in

```

class my_operator : public stream_operator {
/* operator state */
vector<my_tuple> data;
map<int, my_tuple*> tbl; // state element_size=1024
my_hashtable *idx; // state length="idx->count()"
// element_size="idx->first().size()"
.....
/* processing logic */
void input_port_0(my_tuple *t) {
    data.push_back(*t);
    .....
}
.....
/* automatically generated code */
int state_size() {
    .....
    /* take three samples by default */
    len = data.size();
    if (len > 0) {
        sz = (data[0].size() + data[len-1].size()
            + data[len/2].size()) / 3;
        total_size += len * sz;
    }
    /* data structure with fixed size */
    total_size += tbl.size() * 1024;
    /* user-defined data structure */
    if (idx != NULL) {
        total_size += idx->count() * idx->first().size();
    }
    return total_size;
}
};

```

Figure 9. Example of operator class and automatically generated function `state_size()`.

the data structure and the size of each element, using the parameters “length=...” and “element_size=...”. Otherwise, the precompiler ignores the user-defined data structure. Once specified, the parameters will be placed by the precompiler at appropriate locations within function `state_size()`.

2) *Profiling during Runtime*: MS-src+ap+aa profiles the state of stream applications during runtime. The profiling is done periodically so that MS-src+ap+aa can adapt to changing workloads. The purpose of the profiling is to determine s_{max} , the state size threshold for entering “alert mode”. During actual execution, whenever the state size falls below s_{max} , MS-src+ap+aa enters alert mode, during which the controller only allows the state size to decrease. Once the state size increases, the controller initiates a checkpoint immediately.

The first step of the profiling is to find dynamic HAUs, i.e. HAUs whose state size changes significantly over time, as dynamic HAUs are the main reason behind state size fluctuation in a stream application. This is done by observing each HAU’s state size through function `state_size()`. During the period of observation, HAUs whose minimum state size is less than half of its average state size are deemed dynamic HAUs. In our case study applications, the HAUs containing k-means operators (A_i of TMI), historical image processing operators (H_i of BCP) and motion filter operators (M_i of SignalGuru) are dynamic HAUs, constituting less than 20% of all HAUs.

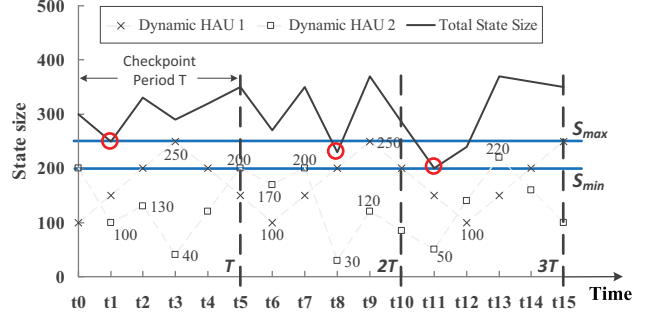


Figure 10. State size function illustrating aggregated state size of two dynamic HAUs. The state sizes at turning points are marked. Red circles indicate the best time for checkpointing in each period.

The second step is to rebuild the aggregated state size of all dynamic HAUs. Each dynamic HAU records its recent few state sizes and detects the turning points (local extrema) of the state size. For example, at time instant t_7 in Fig. 10, the recent few state sizes of HAU1 are 100(t_0), 150(t_1), 200(t_2), 250(t_3), 200(t_4), 150(t_5), 100(t_6) and 150(t_7). The turning points are 250(t_3) and 100(t_6). In order to lower network traffic, dynamic HAUs report only the turning points of their state size to the controller. The state size at any time point between two adjacent turning points can be roughly recovered by linear interpolation. The controller then sums the state sizes of dynamic HAUs. The total state size can be represented by a state size function $f(x)$, whose graph is a zigzag polyline, as shown in Fig. 10.

Finally, we determine the state size threshold for alert mode. The controller finds the time instants when the state size is minimum in each checkpoint period, as marked by the red circles in Fig. 10. These time instants are the best time for checkpointing. The y-coordinates of the highest and lowest red-circled points are called s_{max} and s_{min} respectively, and the ratio $\alpha = (s_{max} - s_{min}) / s_{min}$ is called relaxation factor. The value s_{max} is the threshold for alert mode. Based on our empirical experimental data, it is better to conservatively increase s_{max} a little, so that there are more occasions where the state size stays below s_{max} in each period. We do so by bounding the relaxation factor to a minimum of 20% relative to s_{min} .

3) *Choosing Time for Checkpointing*: After profiling, MS-src+ap+aa begins actual execution. Given s_{max} from profiling, MS-src+ap+aa enters alert mode whenever the total state size of dynamic HAUs falls below s_{max} . Based on the second characteristic of stream applications (Section II-B2), dramatic decrease in total state size is usually a result of a dramatic decrease in a *single* HAU’s state size, rather than the joint effect of several HAUs having small reduction in state size. Therefore, the method to check alert mode can be designed with less network overhead: When the system is not in alert mode, dynamic HAUs do not actively report their state sizes to the controller. Instead, they wait passively

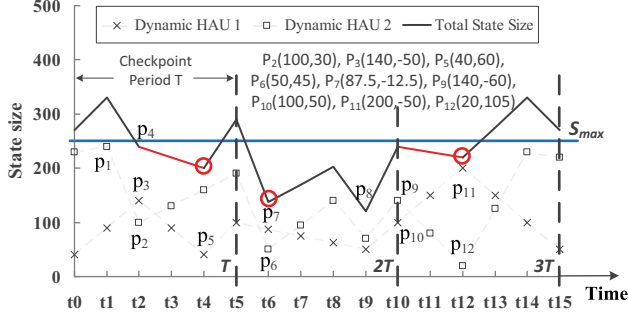


Figure 11. Choosing time for checkpointing. Red circles indicate the time chosen for checkpointing. Red line segments indicate that MS-src+ap+aa is in alert mode.

for the queries about state size from the controller. The controller sends out queries only on two occasions: 1) A new checkpoint period begins; 2) A dynamic HAU detects, at a turning point of its state size, that its state size has fallen by more than half (the HAU notifies the controller then). On these two occasions, the controller sends out queries to each dynamic HAU and obtains their state sizes. If the total state size is below the threshold, the system enters alert mode.

Take Fig. 11 as an example. In the first period, the controller collects the state size at t_0 and the total state size is larger than s_{max} . At t_2 , dynamic HAU2 detects that its state size has fallen by more than half (from p_1 to p_2). It triggers the controller to check the total state size. Since the total state size at point p_4 is below s_{max} , MS-src+ap+aa enters alert mode at t_2 . Similarly, MS-src+ap+aa enters alert mode at t_6 and t_{10} in the second and third periods.

When in alert mode, dynamic HAUs actively report to the controller at the turning points of their state size. Besides the current state sizes, dynamic HAUs also report the instantaneous change rate (ICR) of their state sizes. For example, at t_2 in Fig. 11, HAU1 reports its state size (140) and ICR (-50) to the controller. The ICR of -50 means that HAU1's state size will decrease by 50 per unit of time in the near future. In practice, HAU1 can know the ICR only shortly after t_2 . We ignore the lag in Fig. 11 since it is small. Similarly, dynamic HAU2 reports its state size (100) and ICR (30) at t_2 . The controller sums all ICRs and the result is -20. The negative result indicates that the total state size will further decrease in the future. Therefore, it is not the best time for checkpointing. The controller then waits for succeeding reports from HAUs. At t_4 , HAU1 detects another turning point p_5 , it reports its state size (40) and ICR (60). The aggregated ICR is 90 this time. The positive result indicates that the total state size will increase. Once the controller foresees a state size increase in alert mode, it initiates a checkpoint. After that, the alert mode is dismissed. In the second period, the aggregated ICR at t_6 is 32.5. Therefore, the controller initiates a checkpoint at t_6 . For the same reason, the controller initiates a checkpoint at t_{12} .

Since this method can only find the first local minimum in alert mode, it skips point p_8 , which is a better time for checkpointing in the second period. This is the reason why we need the profiling phase and require it to return a tight threshold s_{max} . In addition, in the rare case where the total state size is never below s_{max} during a period, a checkpoint will be performed anyway at the end of the period.

IV. EVALUATION

We run the experiment on Amazon EC2 Cloud and use 56 nodes (55 for HAUs and one for storage). The controller runs on the storage node. Each node has two 2.3GHz processor cores and is equipped with 1.7GB memory. The nodes are interconnected by a 1Gbps Ethernet. We evaluate Meteor Shower using the three actual transportation applications presented in Section II-B2. Each application is composed of 55 operators and each operator constitutes an HAU.

A. Common Case Performance

We first evaluate the common case performance of the three applications: the end-to-end throughput and latency. Throughput is defined as the number of tuples processed by the application within a 10-minute time window, and latency is defined as the average processing time of these tuples.

First, we compare the throughput of the baseline system and Meteor Shower. In order to examine the throughput variation at different checkpoint frequencies, we have arranged 0~8 checkpoints performed within the time window. As shown in Fig. 12, MS-src outperforms the baseline system. Since the baseline system and MS-src both adopt synchronous checkpointing, the improvement is due to source preservation. Specifically, when there is no checkpoint, MS-src increases throughput by 35%, on average, compared with the baseline system. This increase indicates the performance gain of source preservation in terms of throughput. It can also be seen that MS-src+ap offers higher throughput than MS-src. As an example, when there are 3 checkpoints in a 10-minute window, the increase in throughput from MS-src to MS-src+ap is 28%, on average. This improvement is due to parallel, asynchronous checkpointing. Among all the schemes under evaluation, MS-src+ap+aa offers the highest throughput. MS-src+ap+aa outperforms MS-src+ap because of application-aware checkpointing, which results in less checkpointed state. At 3 checkpoints, the improvement in throughput from MS-src+ap to MS-src+ap+aa is 14%, on average. Combining the three techniques, MS-src+ap+aa outperforms the baseline system by 226%, on average, at 3 checkpoints.

Second, we measure the average latency in these systems. Fig. 13 shows the results. It can be seen that MS-src+ap+aa and the baseline system perform best and worst respectively in terms of latency. When there is no checkpoint, Meteor Shower reduces latency by 9%, on average, compared with the baseline system. This decrease indicates the performance

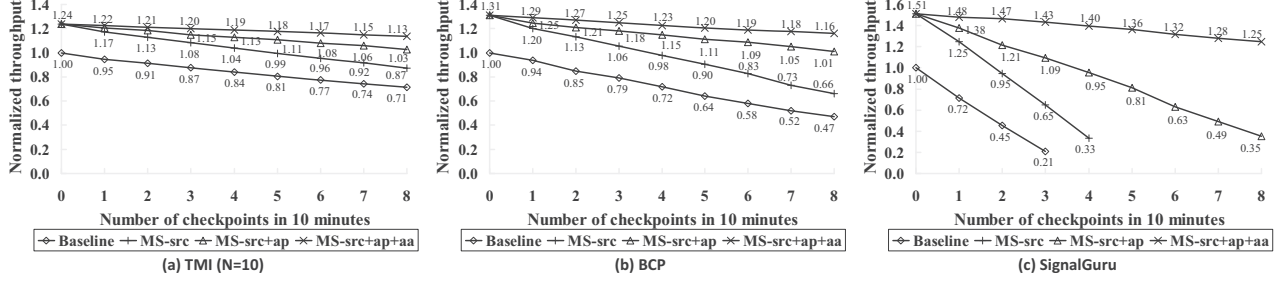


Figure 12. Throughput of baseline system, MS-src, MS-src+ap and MS-src+ap+aa. All values are normalized to the throughput of the baseline system with zero checkpoint.

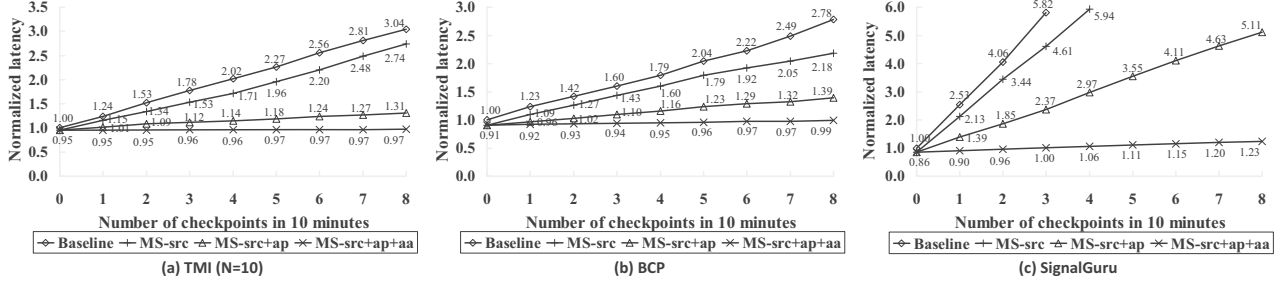


Figure 13. Latency of baseline system, MS-src, MS-src+ap and MS-src+ap+aa. All values are normalized to the latency of the baseline system with zero checkpoint.

gain of source preservation in terms of latency. At 3 checkpoints, MS-src+ap and MS-src+ap+aa reduce latency by 43% and 57% respectively, on average, compared to the baseline system.

B. Checkpointing Overhead

We evaluate the checkpointing overhead using two metrics: checkpoint time and instantaneous latency (latency jitter). Checkpoint time is the time used by a DSPS to complete a checkpoint. Instantaneous latency is the processing time of each tuple during a checkpoint. As checkpointing activities disrupt normal stream processing, these two metrics indicate the duration and extent of the disruption.

The methods for measuring checkpoint time differ in MS-src, MS-src+ap and MS-src+ap+aa. In MS-src+ap and MS-src+ap+aa, we only measure the time consumed by the slowest individual checkpoint because individual checkpoints start at the same time and are performed in parallel. The checkpoint time can be broken down into three portions: token collection, disk I/O and other. Token collection is the period of time during which an HAU waits for the tokens from all upstream neighbors (time instants 1~4 in Fig. 8). Disk I/O is the time used to write the checkpointed state to stable storage. Other includes the time for state serialization and process creation. In MS-src, however, we only measure the total checkpoint time because token propagation and individual checkpoints overlap. Besides, to evaluate application-aware checkpointing, we also measure the checkpoint time in MS-src+ap+aa when the checkpoint

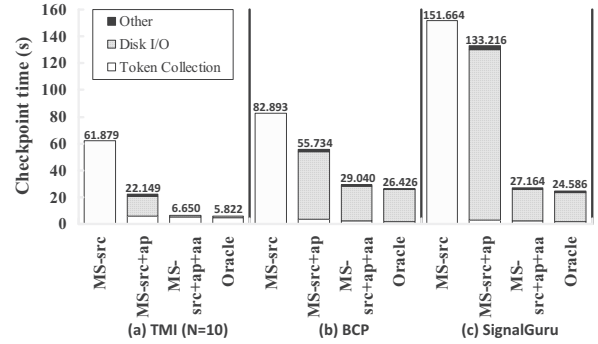


Figure 14. Checkpoint time. The value of MS-src is not broken down.

is performed exactly at the moment of the minimal state (Oracle). This checkpoint time is obtained from observing prior runs, when a complete picture of the runtime state is available. Oracle is the optimal result.

Fig. 14 shows the results. It can be seen that disk I/O dominates the checkpoint time. MS-src+ap reduces checkpoint time by 36%, on average, compared with MS-src. MS-src+ap+aa further reduces checkpoint time by 66%, on average, compared with MS-src+ap. This is close to the Oracle which reduces checkpoint time by 69%, on average, vs MS-src+ap. This shows that application-aware checkpointing can reasonably pinpoint suitable moment for checkpointing in the vicinity of the ideal moment.

We then evaluate the disruption of our checkpoints to normal stream processing by measuring instantaneous latency during a checkpoint. Fig. 15 shows the results. It

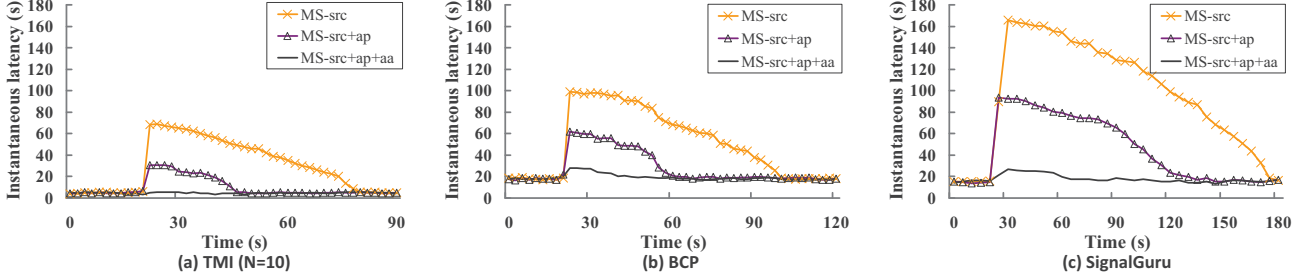


Figure 15. Instantaneous latency during checkpoint.

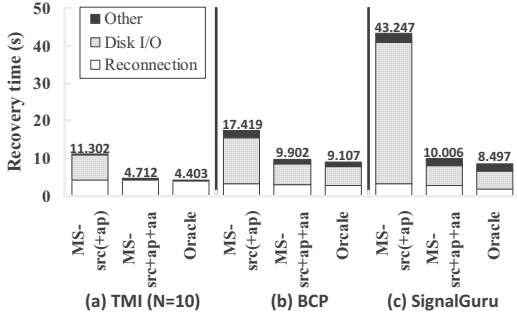


Figure 16. Recovery time. MS-src and MS-src+ap have the same recovery time.

can be seen that MS-src causes larger instantaneous latency than MS-src+ap, due to the synchronous checkpointing. MS-src+ap+aa outperforms MS-src and MS-src+ap. MS-src+ap+aa increases instantaneous latency by just 47%, on average, compared with the latency when there is no checkpointing, whereas MS-src can aggravate the latency by 5~12X. MS-src+ap+aa thus effectively hides the negative impact of checkpointing on normal stream processing.

C. Worst Case Recovery Time

Recovery time is the time used by a DSPS to recover from a failure. We measure the recovery time in the worst case, where all computing nodes on which a stream application runs fail. In this situation, all the HAU's in this application have to be restarted on other healthy nodes and read their checkpointed state from the shared storage. Since the baseline system can only handle single node failures, we do not include it in this experiment. For each HAU, the recovery proceeds in four phases: 1) the recovery node reloads the operators in the HAU; 2) the node reads the HAU's state from the shared storage; 3) the node deserializes the state and reconstructs the data structures used by the operators; and 4) the controller reconnects the recovered HAU's. The recovery time is the sum of these four phases.

Fig. 16 shows the recovery time. The data is broken down into three portions: disk I/O, reconnection and other. Corresponding to the aforementioned procedure, disk I/O is phase two, reconnection is phase four, and other is the sum of phases one and three. We also measure the

recovery time when the application is recovered from the state checkpointed by the Oracle. It can be seen that disk I/O dominates the recovery time. MS-src+ap+aa is able to reduce 59% of the recovery time, on average, compared with MS-src and MS-src+ap. The Oracle reduces 63% of the recovery time, on average, compared with MS-src and MS-src+ap. The similar ratios indicate again that application-aware checkpointing is effective.

After recovery, the source HAU's replay the preserved tuples and the application catches up the normal stream processing. Since this procedure is the same with previous schemes, we do not further evaluate it.

V. RELATED WORK

Checkpointing has been explored extensively in a wide range of domains outside DSPS. Meteor Shower leverages some of these prior art, tailoring techniques specifically to DSPS. For instance, there has been extensive prior work in checkpointing algorithms for traditional database. However, classic log-based algorithms, such as ARIES or fuzzy checkpointing [20, 21], exhibit unacceptable overheads for applications, such as DSPS, with very frequent updates [22]. Recently, Vaz Salles et al. evaluated checkpointing algorithms for highly frequent updates [19], concluding that copy-on-write leads to the best latency and recovery time. Meteor Shower therefore adopts copy-on-write for asynchronous checkpointing. In the community of distributed computing, sophisticated checkpointing methods, such as virtual machines [23, 24], have been explored. However, if used for stream applications, these heavyweight methods can lead to significantly worse performance than the stream-specific methods discussed in this paper; Virtual machines incur 10X latency in stream applications in comparison with SGuard [6].

In the field of DSPS, two main classes of fault tolerance approaches have been proposed: replication-based schemes [1, 2, 3] and checkpoint-based schemes [1, 4, 5, 6]. As mentioned before, replication-based schemes take up substantial computational resources, and are not economically viable for large-scale failures. Checkpoint-based schemes adopt periodical checkpointing and input preservation for fault tolerance, like the baseline system in this paper. The

differences between the schemes are the techniques used to reduce disk I/O and the disruption to normal stream processing. Passive standby [1] saves the checkpointed state in memory. It avoids disk I/O but limits the state size. LSS [5] sacrifices data consistency for performance. Whenever the buffer for input preservation is full, LSS drops the tuples from the buffer instead of saving them into disk. Cooperative HA Solution [4] saves each HAU's state on other computing nodes in the DSPS, thus avoiding a central storage system. It also experiments with delta-checkpointing (saving only the changed part of the state) to reduce the state size. SGuard [6] adopts asynchronous checkpointing and distributed checkpointing (scattering the checkpointed state into multiple storage nodes). We believe that distributed checkpointing and delta-checkpointing complement Meteor Shower's application-aware checkpointing and could be applied jointly.

VI. CONCLUSION

We presented a new fault tolerance scheme for DSPSs – Meteor Shower. Meteor Shower enables DSPSs to overcome large-scale burst failures in commodity data centers, and improves the overall performance of DPSPs. We evaluated Meteor Shower across three actual transportation applications and showed substantial performance improvements over the state-of-the-art.

REFERENCES

- [1] J.-H. Hwang, M. Balazinska, A. Rasin, et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, pages 779–790. IEEE, 2005.
- [2] M. Balazinska, H. Balakrishnan, S.R. Madden and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. *ACM Transactions on Database Systems*, 33(1), 2008.
- [3] M.A. Shah, J.M. Hellerstein and E. Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In *SIGMOD*. ACM, 2004.
- [4] J.-H Hwang, Y. Xing, U. Cetintemel, et al. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE*, pages 176–185. IEEE, 2007.
- [5] Q. Zhu, L.Chen and G. Agrawal. Supporting Fault-Tolerance in Streaming Grid Applications. In *IPDPS*, pages 1–12. IEEE, 2008.
- [6] Y.C. Kwon, M. Balazinska and A. Greenberg. Fault-tolerant Stream Processing Using a Distributed, Replicated File System. *PVLDB*, pages 574–584, 2008.
- [7] J. Gray and A.Reuter. *Transaction Processing – Concepts and Techniques*. Kaufmann, 1993.
- [8] J. Dean. Keynote: Designs, Lessons and Advice from Building Large Distributed Systems. In *the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*. ACM, 2009.
- [9] E. Pinheiro, W.-D. Weber and L.A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–29. USENIX, 2007.
- [10] B.Schroeder, E.Pinheiro and W.-D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, pages 193–204. ACM.
- [11] L.A. Barroso. Keynote: Warehouse-scale Computing. In *SIGMOD*. ACM, 2010.
- [12] S. Gaonkar, E. Rozier, A. Tong, et al. Scaling File Systems to Support Petascale Clusters: A Dependability Analysis to Support Informed Design Choices. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 386–391. IEEE, 2008.
- [13] H. Wang, F. Calabrese, G. Di Lorenzo and C. Ratti. Transportation Mode Inference from Anonymized and Aggregated Mobile Phone Call Detail Records. In *Proceedings of the IEEE Conference on Intelligent Transportation Systems*, pages 318–323. IEEE, 2010.
- [14] E. Koukoumidis, L. Peh and M. Martonosi. SignalGuru: Leveraging Mobile Phones for Collaborative Traffic Signal Schedule Advisory. In *MobiSys*. ACM, 2011.
- [15] N. Tatbul, U. Cetintemel, S. Zdonik, et al. Load Shedding in a Data Stream Manager. In *VLDB*, pages 309–320, 2003.
- [16] S. Ghemawat, H. Gobioff, S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, 2003.
- [17] S.-O. Hvasshovd, O. Torbjornsen, S. Bratsberg et al. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *VLDB*, 1995.
- [18] E. Lau and S. Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *VLDB*, 2006.
- [19] M.V. Salles, T. Cao, B. Sowell et al. An evaluation of checkpoint recovery for massively multiplayer online games. In *VLDB*, 2009.
- [20] C. Mohan, D. Haderle, B. Lindsay et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *TODS*. ACM, 1992.
- [21] K. Salem and H. Garcia-Molina. Checkpointing Memory-Resident Database. In *ICDE*, 1989.
- [22] T. Cao, M.V. Salles, B. Sowell et al. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *VLDB*, 2011.
- [23] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. In *ACM Symposium on Operating Systems Principles*, 1995.
- [24] J. Zhu, W. Dong, Z. Jiang et al. Improving the Performance of Hypervisor-Based Fault Tolerance. In *IPDPS*, 2010.