2014 IEEE 34th International Conference on Distributed Computing Systems

T-Storm: Traffic-aware Online Scheduling in Storm

Jielong Xu, Zhenhua Chen, Jian Tang and Sen Su

***Abstract*—Storm has emerged as a promising computation platform for stream data processing. In this paper, we first show inefficiencies of the current practice of Storm scheduling and challenges associated with applying traffic-aware online scheduling in Storm via experimental results and analysis. Motivated by our observations, we design and implement a new stream data processing system based on Storm, namely, *T-Storm*. Compared to Storm, T-Storm has the following desirable features: 1) based on runtime states, it accelerates data processing by leveraging effective traffic-aware scheduling for assigning/re-assigning tasks dynamically, which minimizes inter-node and inter-process traffic while ensuring no worker nodes are overloaded; 2) it enables fine-grained control over worker node consolidation such that T-Storm can achieve better performance with even fewer worker nodes; 3) it allows hot-swapping of scheduling algorithms and adjustment of scheduling parameters on the fly; and 4) it is transparent to Storm users (i.e., Storm applications can be ported to run on T-Storm without any changes). We conducted real experiments in a cluster using well-known data processing applications for performance evaluation. Extensive experimental results show that compared to Storm (with the default scheduler), T-Storm can achieve over 84% and 27% speedup on lightly and heavily loaded topologies respectively (in terms of average processing time) with 30% less number of worker nodes.**

***Index Terms*—Big Data, Stream Data Processing, Storm, Scheduling, Resource Management.**

I. Introduction

We are at the beginning of a *Big Data* era, in which both industry and academia are undergoing a profound transformation with the use of large-scale, diverse, and high-resolution datasets that allow for data-intensive decision-making, at a level never before imagined. Currently, many systems and applications involve a large volume of continuous data streams. For example, the popular online social network, Twitter, is a typical stream data system, in which thousands of short text messages (a.k.a tweets) are posted every second. The current record (as of August 2013) was set to $143,199$ tweets per second during a showing of the classic animation film "Castle in the Sky" on the weekend of August 2 in Japan [1]. In addition, over 500 million tweets are sent in a typical day, with an average of $5,700$ tweets per second. Another example is a sensor network consisting of a large number of mobile phones that utilize their sensors (e.g., digital camera, microphone, GPS, accelerometer, etc) to produce continuous sensing data (video, audio, location, moving status, etc) [2]. It is very challenging to process and analyze these big data streams in a real-time or near real-time manner.

MapReduce [3] and Hadoop [4] have become the *de facto* programming model and system for big data processing respectively. However, MapReduce and Hadoop are not suitable for stream data applications because they were designed for offline batch processing of static data, in which all input data need to be stored on a distributed file system in advance. Storm [5] has emerged as a promising computation platform for stream data processing. Storm is a distributed, reliable, and fault-tolerant system, which is designed particularly for processing unbounded streams of data in real time. A Storm application is modeled as a directed graph called a topology, which usually includes two types of components: spouts and bolts. A spout is a source of data stream, while a bolt consumes tuples from spouts or other bolts, and processes them in the way defined by user code. Spouts and bolts can be executed as many tasks in parallel on multiple physical machines (a.k.a worker nodes) in a cluster. Storm achieves reliability by providing an effective mechanism to guarantee message processing and supports fault-tolerance by monitoring worker processes and re-starting failed ones in a timely manner.

Storm includes a default scheduler, which assigns workload of a topology evenly into worker processes across the cluster using a simple round-robin algorithm, without considering inter-node and inter-process traffic which, however, may make a significant impact on performance. Another problem of the default scheduler is that it always causes Storm to use all available worker nodes in a cluster, regardless of workload. It is known that in case of light workload, the operational cost (such as electricity cost) can be reduced substantially by consolidating worker nodes and turning off idle ones (or putting them into sleep) [6]. Motivated by our observations and analysis, we design and implement a new stream data processing system based on Storm, namely *T-Storm*. Compared to Storm, T-Storm has the following desirable features: 1) based on run-time states, it accelerates data processing by leveraging effective traffic-aware scheduling for assigning/re-assigning tasks dynamically, which minimizes inter-node and inter-process traffic while ensuring no worker nodes are overloaded; 2) it enables fine-grained control over worker node consolidation such that it can achieve better performance with even fewer worker nodes; 3) it allows hot-swapping of scheduling algorithms and adjustment of scheduling parameters on the fly; 4) it is transparent to Storm users (i.e., Storm applications can be ported to run on T-Storm without any changes). We summarize our contributions in the following:

- Via experimental results and analysis, we show inefficiencies of the current practice of scheduling in Storm and challenges associated with applying traffic-aware online scheduling in Storm.
- We present design and implementation of T-Storm, which has several desirable features (mentioned above).
- We justify the superiority of T-Storm via real experiments in a cluster using well-known data processing applications such as Word Count (stream version) and Log Stream Processing.

Jielong Xu, Zhenhua Chen and Jian Tang are with Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, 13244. Email: {jxu21, zchen03, jtang02}@syr.edu. Sen Su is with the State Key Lab of Networking and Switching, Beijing University of Posts and Telecommunications, Beijing, China. Email: susen@bupt.edu.cn. This research was supported in part by NSF grants #1218203. The information reported here does not reflect the position or the policy of the federal government.

1063-6927/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDCS.2014.61

The rest of the paper is organized as follows: We briefly introduce Storm in Section II. In Section III, we discuss problems and challenges. We then present design and implementation details of T-Storm in Section IV. Experimental results are presented and analyzed in Section V. We discuss related works in Section VI and conclude the paper in Section VII.

II. Storm

For completeness of presentation, we first briefly introduce Storm. Storm [5] is a distributed, reliable, and fault-tolerant computation system for stream data processing. Unlike batch data processing systems such as Hadoop [4], Storm is designed as a real-time computation system to process unbounded streams of data. A key difference between these two is that a MapReduce job will eventually finish, whereas a Storm "job" continues on forever, unless it is killed by its user.

A stream is defined as an unbounded sequence of tuples. A Storm application is modeled as a directed graph called a topology, which includes two types of components: spouts and bolts. A spout is a source of data stream, where data are often read from external sources and then tuples are emitted into the topology. A bolt consumes tuples from spouts or other bolts, and processes them in the way defined by code provided by the user. A bolt can either persist data in storage, or pass it to some other bolts for further processing. These components are represented by vertices in the topology graph, where direct edges indicate how the streams are routed. Spouts and bolts can be executed as many tasks in parallel on multiple physical machines in a cluster. Therefore, a task can be considered as an instance of a spout or bolt. There are 5 ways for grouping in Storm, which define how to send tuples between tasks.

- Shuffle grouping: Tuples are randomly distributed across receiving bolt's tasks and each task is guaranteed to receive an equal number of tuples.
- Fields grouping: A field of a tuple is used as the key to partition the stream. Tuples with the same key will be mapped to the same task.
- All grouping: Each tuple is broadcasted to all tasks of the corresponding bolt.
- Global grouping: The entire stream is routed to one of the bolt's tasks, usually the task with the lowest ID.
- Direct grouping: The producer of the stream decides which task of the consuming bolt will receive each tuple.

Storm uses two levels of abstractions (logical and physical) to express parallelism, which is illustrated in Fig. 1. In the physical layer, a Storm cluster usually consists of a master node that serves as a central control unit and a set of physical machines (called *worker nodes*) that actually process incoming data. A topology is executed in one or multiple worker processes (called *workers* for simplicity) running on one or multiple worker nodes. Each worker node runs a daemon called *supervisor* that listens for any work assigned to it by the scheduler. Slots are configured on each worker node, which are basically ports used to receive messages. The number of slots indicates the number of workers that can be run on this worker node, and is usually pre-configured by the cluster operator. Typically, it can be set to the number of cores on

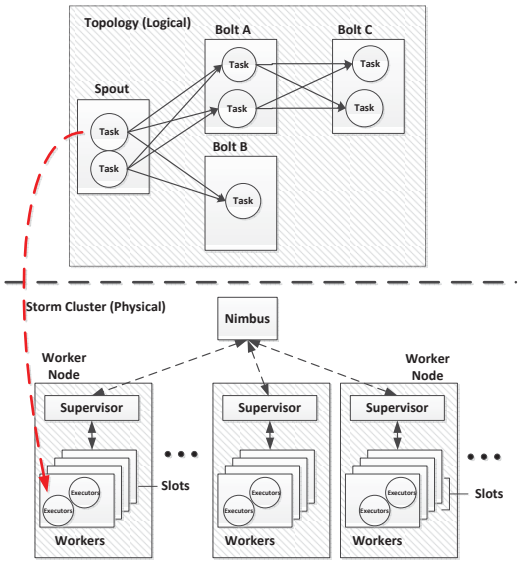


Fig. 1. The logical and physical views of Storm

that worker node. Each worker is a Java Virtual Machine (JVM) that executes a subset of tasks defined in a topology. An executor is a thread spawned by a worker, which can contain one (by default) or multiple tasks. A Storm cluster is controlled by a daemon running on the master node called *Nimbus*. Nimbus is responsible for distributing users' code around the cluster, assigning executors to workers and worker nodes, and monitoring for failures.

Storm's reliability comes from guaranteed message processing. Every tuple emitted has a *message ID* and can be tracked back to its originating spout. When the message ID of a spout tuple successfully traverses a topology, a special *acker* task is called to inform the originating spout that message processing is complete. If a tuple fails to be processed within a specified timeout (30 seconds by default in Storm [7]), the tuple is considered to be failed and can be replayed from the originating spout. Storm is also fault-tolerant. When a worker dies, its supervisor will try to restart it on the same worker node. However, if it continues to fail to start and is unable to heartbeat to Nimbus, the worker will be restarted on another worker node. In addition, in Storm, Nimbus and supervisors use *Zookeeper* [8] as a coordination service to maintain configuration information, naming, and distributed synchronization among worker nodes.

III. Problems and Challenges

In this section, we discuss the problems of the current practice of scheduling in Storm and the challenges associated with traffic-aware online scheduling.

Before running an application on a Storm cluster, a user can specify the number of workers and executors of each vertex (spout or bolt) on the corresponding topology. However, it is hard to determine what is the optimal number of workers that should be used for the input topology.

Storm includes a default scheduler, which assigns executors to pre-configured workers in a round-robin manner and then

evenly assigns those workers to available slots on worker nodes. This scheduling policy leads to almost even distribution of executors over available slots.

The default scheduler ignores inter-node and inter-process traffic, which may make a significant impact on performance. We conducted a few simple experiments on small cases in a Storm cluster to study such impact, and to show problems and challenges. Our experiments were conducted on a cluster consisting of IBM blade servers (each with two Intel Xeon dual-core 2.0GHz CPUs and 4GBs of RAM) connected by a 1Gbps network. We installed Storm 0.8.2 (obtained from Storm's repository on GitHub [9]) and used the Throughput Test topology [10] in these experiments. We used anchored bolts in the topology so that we can observe the average completion time of tuples, which is the duration between when the spout emits the tuple and when it has been acked (fully processed). We deducted the arbitrary sleep time (5ms) that spout uses when emitting each tuple for rate control to obtain the actual *tuple processing time*, which is used as the primary metric for performance evaluation. We summarize our observations and analysis in the following:

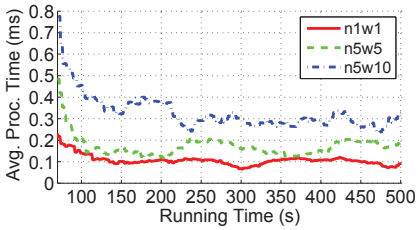
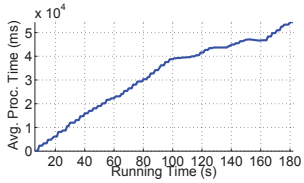


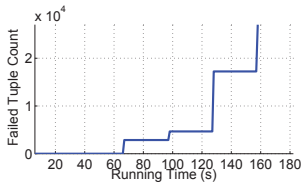
Fig. 2. Impact of inter-process and inter-node traffic

Observation 1: Inter-node/inter-process traffic makes a significant impact on performance. In the first experiment, we used a chain-like topology consisting of one spout executor, four bolts with one executor per component, and five acker executors. We tested three assignment solutions: in the first solution, we placed all executors in one worker node (with only one worker); and in the second solution, the default scheduler is used to place all executors evenly in 5 worker nodes with a total of 5 workers; in the last solution, we spread out executors as much as possible by placing each executor in its own worker on 5 worker nodes. From Fig. 2, we observe that compared to the first solution (n1w1 – 1 node, 1 worker), the second (n5w5 – 5 nodes, 5 workers) and third solution (n5w10 – 5 nodes, 10 workers) lead to worse performance. If executors were spread to different workers and nodes unnecessarily, it resulted in longer average processing time (about 35% longer after stabilization for the second solution and 67% for the third solution) because of additional delay introduced by inter-node/inter-process communications and context switching. Hence, we need a traffic-aware scheduling scheme that can minimize inter-node/inter-process traffic.

Observation 2: Overloading a worker node counteracts the benefit brought by reducing inter-node traffic. In the second experiment, we devised a way to overload a worker node. We set the number of spout executors to 5 but kept the number of bolt executors at 1. This essentially mimicked the situation where a topology is heavily loaded with incoming tuples but does not have enough bolt executors to process them. In Fig. 3, we see that once a worker node was overloaded, the processing time skyrocketed and tuples started to fail. In this case, the bolt executors could no longer process all incoming tuples from spout executors, and tuples queued up until eventually timeout (30 seconds by default in Storm [7]). This experiment demonstrates that although consolidating executors can lead to speedup, overdoing it can cause a worker node to be overloaded, which counteracts the benefit brought by reducing inter-node/inter-process traffic and causes substantial tuple failures. Hence, we need to design an effective traffic-aware scheduling scheme that can prevent worker nodes from overloading; and even when overloading occurs for some reason, it can help the system recover from it quickly.



(a) The average processing time



(b) The number of failed tuples

Fig. 3. Impact of overloading a worker node

Another problem of the default scheduler is that it always makes Storm use all of available worker nodes in a cluster. However, it is known that in the case of light workload, consolidating worker nodes and shutting down idle ones (or put them into sleep) can significantly reduce operational costs (such as electricity cost) [6]. However, overdoing it may cause performance degradation and tuple loss as shown above. Hence, there is a tradeoff between traffic load minimization (or worker node consolidation) and workload balancing. Finding the best scheduling (assignment) solution is quite challenging but worth studying.

Besides the default scheduler, two schedulers were presented for Storm in a closely related work [11]. The first scheduler is an offline scheduler, which simply analyzes the topology and identifies possible sets of bolts to be scheduled on a common node by looking at how they are connected. The second scheduler is an online scheduler, which monitors effectiveness of the schedule at runtime and re-adapts it to

reduce the inter-node traffic when it sees fit. However, these schedulers suffer from the following problems: 1) The offline scheduler is oblivious with respect to runtime workload therefore is not efficient in practice, which was even concluded by the authors of [11]. 2) Modifications to the user code in spouts and bolts are required to enable load monitoring needed by the online scheduler. So it is not completely transparent to Storm users. 3) According to our tests using their code, the online scheduler is not general enough: for some topologies that do not have a certain degree of complexity, the default scheduler was invoked instead. 4) All the experiments in [11] were performed based on their own applications/topologies without using any real data processing applications such as Word Count and Log Stream Processing. Hence, the effectiveness of the proposed online scheduler has not been well justified.

IV. Design and Implementation of T-Storm

In this section, we first present an overview for T-Storm and then describe design and implementation in details.

A. Overview

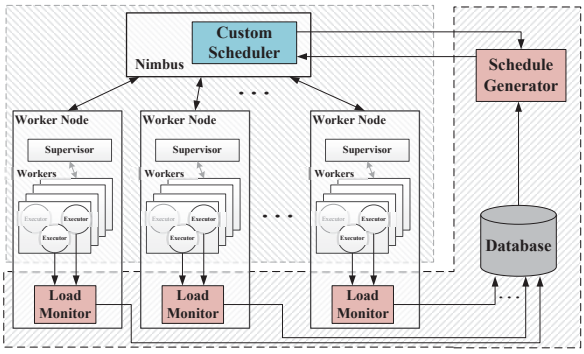


Fig. 4. The architecture of T-Storm

Motivated by the observations and analysis described above, we design and implement T-Storm (Fig. 4) based on Storm to enable traffic-aware online scheduling by revising some components of Storm and adding several new components, including *a custom scheduler, a schedule generator and load monitors*. Scheduling in T-Storm works as follows:

1) The load monitors periodically collect workload and traffic load information at runtime, and store it into a database.
2) The schedule generator periodically reads load information from the database and computes a schedule using a traffic-aware online scheduling algorithm.
3) The custom scheduler periodically fetches the current schedule generated by the schedule generator and execute it by assigning executors accordingly.

We summarize the desirable properties of T-Storm as follows and discuss how we enable these features in the following subsections.

1) *Traffic-aware Online Scheduling (Section IV-C):* Based on workload and traffic load information collected at runtime by load monitors (Section IV-B), T-Storm accelerates data processing by leveraging a traffic-aware online algorithm for assigning/re-assigning tasks (executors). Our design minimizes inter-node and inter-process traffic while ensuring no worker nodes are overloaded.
2) *Worker Node Consolidation for Cost Reduction (Section IV-C):* T-Storm enables fine-grained control over worker node consolidation such that T-Storm can achieve better performance with even fewer worker nodes (compared to Storm).
3) *Optimization for the Number of Slots (Section IV-C):* T-Storm is optimized by using the scheduling algorithm to determine the number of slots to be used for each topology instead of using a pre-configured number.
4) *Hot-Swapping of Scheduling Algorithms (Section IV-C):* It allows the current scheduling algorithm to be replaced by a new one at runtime without shutting shown the cluster.
5) *Optimization for Re-assignment Overhead (Section IV-D):* We introduce techniques and modify components of Storm to reduce overhead (such as delay, tuple loss, etc) associated with a re-assignment procedure. It also allows scheduling parameters to be adjusted on the fly.
6) *Storm User Transparency:* Storm applications can be ported to run on T-Storm without any changes since the differences between T-Storm and Storm all lie in the scheduling part.

B. Load Monitoring

A load monitor is developed to collect vital system statistics at runtime for the schedule generator. A load monitor runs in the background as a daemon on each worker node and collects the following information every 20s.

1) Workload of each executor: It is the CPU usage in MHz. It is collected using the *getThreadCpuTime(threadID)* method of the *ThreadMXBean* class in the standard Java Management Extensions (JMX) APIs. We then translate CPU time to CPU usage in MHz.
2) Workload of each worker node: The workload of each worker node is obtained by summing the CPU usages of all executor threads on that node.
3) Inter-executor traffic load: It is the number of tuples sent between a pair of executors during the sampling period (20s). We made minor modifications to Storm's executor implementation to send a message to a load monitor each time an executor sends a tuple.

Instead of storing these instantaneous readings to the database, T-Storm updates the values of these parameters by $Y = \alpha Y + (1 - \alpha)SampleY$, where Y is the value of a parameter (workload or traffic load) in the database, $SampleX$ the instantaneous reading of that parameter obtained using the method described above, and $0 \leq \alpha \leq 1$ is the coefficient that determines how sensitive the value changes with instantaneous readings (the smaller the α, the more sensitive). Then it stores the new value into the database, which will then be used by the schedule generator as the input. This is a method widely used in computer network systems [12] to estimate values of critical parameters for decision making. We set $\alpha = 0.5$ in our implementation. We try to keep our design simple and practical. However, other machine learning based

(usually more complicated) estimation/prediction methods can be easily integrated to T-Storm too, which will be our future work. Next, we discuss how T-Storm assigns executors to worker nodes according to collected workload and traffic load.

C. Traffic-aware Online Scheduling

We first summarize major notations in the following table for quick reference.

TABLE I
MAJOR NOTATIONS

Notation	Description
C_k	The CPU capacity of worker node k
E	The set of all executors, $\|E\| = N_e$
i	The index of executor, $i \in \{1, \cdots, N_e\}$
j	The index of slot, $j \in \{1, \cdots, N_s\}$
k	The index of worker node, $k \in \{1, \cdots, K\}$
l_i	Workload of executor i, $i \in \{1, \cdots, N_e\}$
$\omega(j)$	The worker node that slot j belongs to
$r_{ii'}$	Traffic load from executor i to executor i'
S	The set of available slots, $\|S\| = N_s$

In the core of our design, we came up with an algorithm to solve the following scheduling problem: Given M topologies, the corresponding set of N_e executors, the set of N_s available slots (different worker nodes may have different numbers of slots), the current workload of each executor l_i, the traffic load from executor i to executor i', $r_{ii'}$, the scheduling problem seeks an assignment $\mathbf{X} =< x_{ij} >$ that specifies how to assign N_e executors to N_s slots (if $x_{ij} = 1$, executor i is assigned to slot j) such that the total inter-node traffic load is minimized. As discussed above, runtime workload and traffic load will be collected by load monitors, which then estimate the future load and store them into a database as the input for the schedule generator. Note that the scheduling problem considered here seeks an executor-slot assignment (instead of an executor-worker assignment and then a worker-slot assignment [11]), i.e., we determine how to assign each executor to a slot on worker node in one step, because our design ensures that in every worker node, executors from a topology are placed in no more than one slot (worker). This is enforced because as discussed in Section III, assigning executors from one topology to multiple (instead of one) slots (workers) on a worker node introduces inter-process traffic and can only harm performance. Next, we present a simple but effective algorithm to solve this problem.

The algorithm first sorts the executors in the descending order of its (incoming and outgoing) traffic load. Then it keeps trying to assign executors to available slots with minimum incremental traffic load until every executor is assigned to a slot. Note that we deal with executors in the descending order of their traffic load because usually this leads to desirable performance for a combinatorial assignment/packing problem (such as bin packing). Moreover, the algorithm always ensures the following constraints are met in each worker node during assignment: (1) executors belonging to a topology are assigned to only one slot; (2) the total workload of executors does not exceed its capacity, and (3) the total number of executors assigned to it does not exceed $\gamma \frac{N_e}{K}$ (where γ is a control parameter called *consolidation factor*). This is achieved by updating Q properly such that Q only includes those slots that the current executor i can be assigned to without violating the constraints described above. Sorting takes $O(N_e \log N_e)$ time and the for loop takes $O(N_e N_s)$ time. So the overall time complexity of this algorithm is $O(N_e \log N_e + N_e N_s)$.

Algorithm 1 Traffic-aware Online Scheduling Algorithm

Input: $E, S, < r_{ii'} >, < l_i >$
Output: $\mathbf{X} =< x_{ij} >$, $i \in \{1, \cdots, N_e\}$, $j \in \{1, \cdots, N_s\}$

1: $< x_{ij} >:= 0 \ \forall i \in \{1, \cdots, N_e\}$, $j \in \{1, \cdots, N_s\}$;
2: Sort all executors in E in the descending order of the summation of their incoming and outgoing traffic loads and store the sorted list to π;
3: **for** $i = 1$ to N_e **do**
4: Update Q properly according to $< x_{ij} >$;
5: $j^* := \text{argmin}_{q \in Q} \sum_{i'=1}^{N_e} \Big((r_{i'\pi(i)} + r_{\pi(i)i'}) \sum_{\omega(j) \neq \omega(q)} x_{i'j} \Big)$;
6: $x_{ij^*} := 1$;
7: **end for**
8: **return** $\mathbf{X}$;

We realize *traffic-awareness* by Algorithm 1 since its objective is to minimize inter-node traffic load. Our scheduling algorithm also ensures that executors belonging to a topology are hosted by only one worker (slot) in each worker node. In this way, our algorithm essentially minimizes inter-process traffic and determines the number of workers (slots) to be used for executors on each worker node. However, with Storm's default scheduling algorithm or the scheduling algorithms presented in [11], the number of slots to be used for a topology is *blindly* specified by its user in advance, which may lead to poor performance. Moreover, T-Storm can handle *overloading* properly: 1) Once overloading occurs on a worker node, the schedule generator can detect it and will then calculate a new schedule using Algorithm 1 to mitigate overloading. 2) In Algorithm 1, the capacity of worker node k, C_k, can be set to a fraction of its actual capacity to prevent overloading from happening with high probability. In addition, Algorithm 1 enables fine-grained control over worker node consolidation by introducing the consolidation factor γ. Setting $\gamma = 1$ can lead to almost even distribution of executors over all worker nodes. The larger the γ, the more likely T-Storm will use less number of worker nodes (i.e., the higher level of consolidation). We will discuss how to set γ properly further in Section V.

As mentioned above, the schedule generator periodically reads the estimated load information from the database at runtime and computes a scheduling solution (an executor-slot assignment) using Algorithm 1. So scheduling in T-Storm is conducted in an *online* manner. In our current implementation, this scheduling generation period is set to a relatively large value, 5 minutes, to prevent high overhead.

Storm allows a custom scheduler to be used at runtime. T-Storm uses a custom scheduler to periodically fetch the new schedule from the schedule generator and applies it without computing its own schedule. More specifically, the custom scheduler will update the executor-to-slot assignment in Nimbus. The new assignment will then be actually implemented by the supervisors. This schedule fetching period is set to 10 seconds in our current implementation, which is usually short and less than the schedule generation period mentioned above because T-Storm should be able to fetch and apply a new assignment in a timely manner after a worker node is overloaded. In our design, the schedule generator is a software component that is independent of Storm. This design can achieve better flexibility than traditional design in which the custom scheduler generates a scheduling solution [11], due to the following reasons: 1) it enables hot-swapping of scheduling algorithms, i.e., the current scheduling algorithm can be replaced by a new one at runtime without shutting down the cluster because the code of a new scheduling algorithm can be loaded to the schedule generator without changing or stopping anything in Storm (including its custom scheduler); certainly, any scheduling parameters (such as γ) can be adjusted on the fly; 2) it makes deployment more flexible since the schedule generator can run on a machine different from the one hosting Nimbus in case that Nimbus is very busy and its host is overloaded; 3) it makes the algorithm development easier because the developer of a scheduling algorithm can focus on developing his/her algorithm without knowing all the details about Nimbus, scheduler and supervisors in Storm.

Once a new topology is launched on a cluster, T-Storm first assigns executors in almost the same way as the default scheduler in Storm. Note that the proposed traffic-aware scheduling algorithm cannot be applied initially since no runtime load information can be provided at that time. We made a minor modification to the default scheduler in T-Storm. Essentially, T-Storm determines the number of workers N_w^* first by $N_w^* = \min\{N_u, N_w\}$, where N_u is the number of workers specified by the user, while N_w is the number of worker nodes with available slots. In this way, T-Storm can ensure that in a worker node, executors from a topology are assigned to no more than one slot (worker). In our implementation, we use Storm's command *rebalance* to enforce this setting.

D. Optimization for Re-assignment Overhead

In Storm, a supervisor checks to see if there is a new assignment from ZooKeeper every 10 seconds. When a supervisor detects a new assignment that assigns different sets of executors to its slots, it will shut down those affected workers (and its executors) and start new workers and corresponding executors based on the new assignment. The unaffected workers and newly started workers will then try to establish TCP connections with other workers that they have interactions with (if necessary). Re-assignment may lead to longer processing time and tuple loss, because creation and termination of workers and TCP connections are not perfectly coordinated. Specifically, Storm suffers from the following problems during the re-assignment procedure:

1) Some workers are shut down by supervisors while their substitutions are not ready.
2) A spout executor starts to work immediately once created, however, other related executors may not be ready.

In order to minimize overhead and enable a smooth re-assignment procedure, we come up with a method to fix the problems described above in T-Storm. The basic idea is to introduce certain delay such that the system won't shut down (old) workers (and its executors) until new workers are ready. Specifically, in T-Storm, every time when a supervisor is scheduled to shut down a worker, instead of doing it immediately, it introduces a delay to this operation. In our current implementation, this delay is set to 20 seconds (2 time the checking period mentioned above). Moreover, spout executors halt for an additional delay until other bolt executors are ready. In the current implementation, the delay is set to 10 seconds. This way, T-Storm can guarantee that old workers are shut down after their substitutions are ready, and a spout executor starts to stream data to bolt executors when they are ready to take them. In other words, T-Storm can mitigate the two problems mentioned above.

In T-Storm, during the re-assignment procedure, new workers and executors and old ones may co-exist for a short period of time. This may cause an issue: Should a tuple be processed by new workers/executors or old ones? We resolve this issue by introducing a simple dispatcher for each slot in a supervisor. T-Storm uses the timestamp of an assignment as its ID and to differentiate new and old workers. A worker needs to register itself with the corresponding dispatcher and moreover needs to add the assignment ID to the header of each message (that includes a number of tuples). Every time when a worker needs to send a message, it forwards it to a dispatcher (on the intended worker node), which then dispatches it to the intended worker (old or new) according to the assignment ID (included in the message header). Essentially, tuples from old workers will be dispatched to old workers, while tuples from new workers will be directed to new workers without confusion. Note that even though more information (assignment ID) needs to be included in message headers, this will only make a very minor impact on system performance since multiple tuples are usually packed in a single message for transmission, which amortizes such overhead over a number of tuples.

V. Performance Evaluation

In this section, we first describe experimental settings and then present results.

We developed the proposed T-Storm based on Storm 0.8.2 (obtained from Storm's repository on GitHub [9]) and installed it on top of Ubuntu Linux 12.04. We performed real experiments on a cluster in Syracuse University's Green Data Center to evaluate performance of T-Storm. The cluster consists of 10 IBM blade servers (each with dual 2.0GHz Xeon CPUs, 2.0GB RAM) connected by a 1Gbps network.

The average processing time (of tuples) was used as the performance metric. We utilized Storm's default timing mechanism to track each tuple's processing time. Storm's UI can be used to collect such information but it displays it as a

10-minute average. In our implementation and experiments, we took 1-minute averages instead, which give us much better precision in real-time performance evaluation. During experiments, the worker nodes were kept in sync by using the standard NTP protocol [13] on the Ubuntu Linux. We summarize the common settings in the following table.

TABLE II
COMMON EXPERIMENTAL SETTINGS

Parameter	Value
Estimation coefficient (α)	0.5
Load monitoring and estimation period	20s
Number of available worker nodes	10
Running time of each experiment	1000s
Schedule fetching period	10s
Schedule generation period	300s

We conducted our experiments using three well-known data processing applications (topologies), namely *Throughput Test*, *Word Count* (stream version) and *Log Stream Processing*. We compared T-Storm against Storm with the default scheduler (*which will be simply labeled and called "Storm" in the following*). Note that as expected, in all experiments, Storm always used all of 10 worker nodes. In addition, we tested Storm with the online scheduler presented in [11] too. However, we found that most of time, it used the default scheduler (instead of their online scheduler) when our topologies were deployed. Note that according to [11], the authors only conducted experiments on internally developed reference topologies. In the following, we describe these topologies, and then present and analyze the corresponding experimental results.

Throughput Test Topology [10]: We started with a simple topology called Throughput Test, which has one spout and two bolts. The spout repeatedly generates random strings of a fixed size of 10K bytes as input tuples. The spout is connected to a bolt called identity bolt that simply emits any tuples it receives from the spout without changing anything. The next component in the topology is a counter bolt which holds a counter, and increments and outputs the counter value every time a tuple has been received and processed.

We tested the Throughput Test topology on 10 worker nodes, with 40 workers, 5 spout executors, 15 identity bolt executors, and 15 counter bolt executors and 10 acker executors (for acknowledgment).

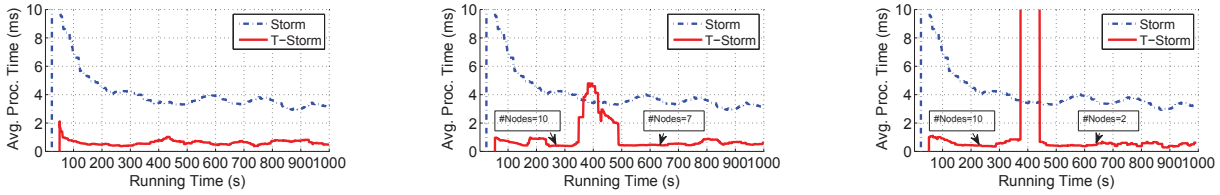
Fig. 5 shows the average processing times achieved by T-Storm and Storm (with the default scheduler) over time respectively on different experiments. Note that some very large values are not shown on the figure, which is why there are some gaps on these figures. In the first experiment, we set the consolidation factor (γ) to 1 for T-Storm, which caused T-Storm to use all worker nodes (10 nodes) as Storm. From Fig. 5(a), we can see that after an short initial period of about 62s in which T-Storm consolidated workers (NOT worker nodes) used on each of the 10 nodes, the average processing time immediately dropped to 0.99ms, while Storm gave an average processing time of 9.25ms. After about 200s, Storm stabilized too. By counting average processing times after 200s, we observe that compared to Storm, T-Storm reduced average processing time by 83% on average. In the next two experiments, we increased γ to see if T-Storm can use less number of worker nodes, while still offering good performance. In the second experiment, we set $\gamma = 1.7$ for T-Storm. During the experiment, T-Storm calculated whether it can generate a new assignment using the proposed traffic-aware online scheduling algorithm at the pre-specified schedule interval (300s). It decided to produce a new assignment that used only 7 worker nodes (compared to 10 used by Storm). As can be seen from Fig. 5(b), T-Storm amazingly preserved almost the same speedup (84%) after the system stabilized at about 500s. However, re-assignment came with a small price, which was the spike around 381s. An interesting property of the bolts of the Throughput Test topology is that they are designed to do little work. Therefore, there is a good potential to consolidate more worker nodes. In the third experiment, we made an even more aggressive move by setting $\gamma = 6$ for T-Storm, which caused it to use only 2 worker nodes (out of 10 nodes). Surprisingly, T-Storm can achieve similar speedup after re-assignment and a short spike. Note that worker node consolidation does not necessary degrade performance since it can reduce inter-node traffic, which has been shown to have an impact on performance. However, overdoing it may cause overloading of worker nodes, long processing time, which will be discussed later.

Word Count Topology (stream version) [14]: Word Count is a well-known MapReduce application [3]. We conducted another set of experiments using the stream version of Word Count application, which does almost the same work as its MapReduce counterpart. It has a chain-like topology with one spout and three bolts. The spout is basically a reader that reads in a file one line at a time. For input, we made a very large word file by concatenating the text version of Alice's Adventures in Wonderland [15] repeatedly for the duration of our experiments. We modified the reader spout to read the word file that is pushed into a Redis queue.

The reader spout is connected to a SplitSentence bolt which splits each line into words and feeds them to a WordCount bolt using fields grouping. The WordCount bolt increments counters based on distinct input word tuples. The last stage of the topology is a Mongo bolt which saves the results into a Mongo database.

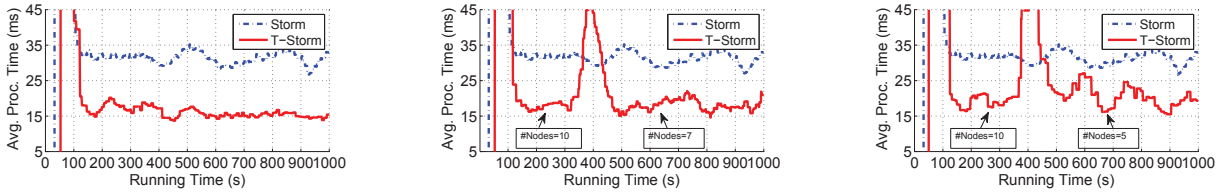
We tested the Word Count topology on 10 worker nodes, with 20 workers, 2 spout executors, 5 executors for each other bolt. Fig. 6 shows the corresponding results. Similarly, γ was set to 1 first for T-Storm, which caused it to use the same number of worker nodes (10 nodes) as Storm. From Fig. 6(a), we can see that T-Storm took a little longer (about 150s) to stabilize on this topology. After that, the average processing time given by T-Storm dropped to 16.2ms (compared to 32.6s given by Storm). By counting average processing times after 150s, T-Storm outperformed Storm by 49% on average.

In the second experiment, we set the $\gamma = 1.8$ for T-Storm. A new assignment was again computed and applied at the pre-specified schedule generation interval (300s), which caused T-Storm to use only 7 worker nodes. As can be seen from Fig. 6(b), on the Word Count topology, T-Storm still preserved similar speedup (42%) after the system stabilized at about 500s (with fewer worker nodes). Similarly, we can ob-



(a) $\gamma = 1$ (10 nodes) (b) $\gamma = 1.7$ (7 nodes). (c) $\gamma = 6$ (2 nodes).

Fig. 5. Performance on the Throughput Test topology



(a) γ=1 (10 nodes). (b) γ=1.8 (7 nodes). (c) γ=2.2 (5 nodes).

Fig. 6. Performance on the Word Count topology

serve a spike around 374s caused by re-assignment. Compared to the bolts of the Throughput Test topology, the bolts of the Word Count topology did much more substantial work, which is why the corresponding average processing time is much longer. In the third experiment, we tested whether T-Storm can do more consolidation without sacrificing performance by setting $\gamma = 2.2$ for T-Storm such that it only used half of the worker nodes (5 nodes). On this topology, T-Storm's performance became a little worse, but still achieved a 35% speedup over Storm (counting measurements after 500s). In addition, we found if we increased the consolidation factor further, T-Storm started to perform similar or even worse than Storm. When using T-Storm for applications with work-intensive bolts, the consolidation factor should not be greedily set to a large value.

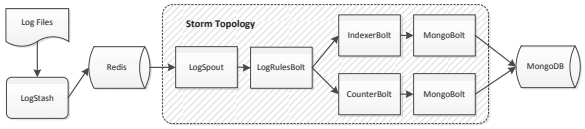
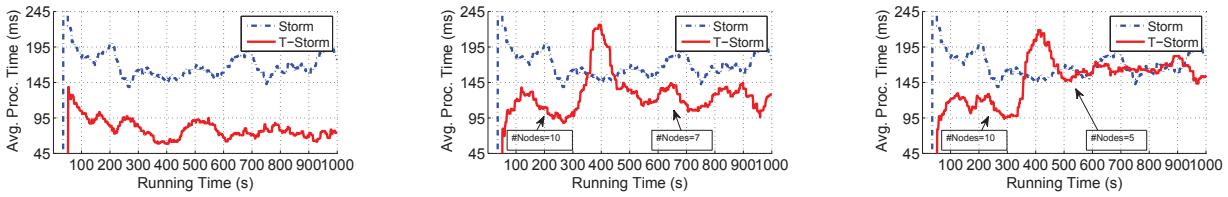


Fig. 7. Log Stream Processing Topology

Log Stream Processing Topology [16]: Log stream processing presents a great real-world use case for Storm. We performed a set of experiments using a published log stream processing topology, which is shown in Fig. 7. The topology uses an open-source log agent called LogStash to read data from log files. LogStash submits log lines as separate JSON values into a Redis queue, which are then consumed by the log spout and emitted into the topology. For input, we used Microsoft IIS log files obtained from the College of Engineering and Computer Science at Syracuse University. The log rules bolt performs rule-based analysis on the log stream and emits a single value containing a log entry instance. The log entry instance is then sent to both the indexer bolt and the counter bolt which perform useful indexing and counting actions on the log entries respectively. For testing purpose, we slightly modified the original topology by introducing Mongo bolts to simply save the results into separate collections in a Mongo database for verification. We tested this topology with 10 worker nodes, 20 workers, 5 spout executors, 5 executors for the log rules bolt, the indexer bolt, the counter bolt, and 2 executors each for the two Mongo bolts. Fig. 8 show all related results.

Similar to the experiments on the other two topologies, we set γ to 1, 1.7 and 2 in three experiments respectively for T-Storm. We can see from Fig. 8(a) that by counting numbers after 150s, T-Storm used the same number of worker nodes as Storm but it outperformed Storm by 54% on average in terms of average processing time. Moreover, it can be observed from Fig. 8(a) that by counting numbers after stabilization after re-assignment at about 500s, T-Storm used only 7 worker nodes but achieved an average of 27% speedup in terms of average processing time. When γ went up to 2, we find that compared to Storm, T-Storm offered comparable average processing time but still used much fewer number of worker nodes (only half). This is mainly because most bolt executors in the Log Stream Processing topology need to do even more intensive work than those in the Word Count topology. Results presented here are actually consistent with those corresponding to the Word Count topology.

Overloading Handling: In the following experiments, we show how T-Storm can handle overloading. As an improvement over Storm, T-Storm is able to detect an overloaded topology because of load monitoring, it will then automatically allocate more resources (workers and worker nodes) to handle



(a) $\gamma = 1$ (b) $\gamma = 1.7$ (c) $\gamma = 2$

Fig. 8. Performance on the Log Stream Processing Topology

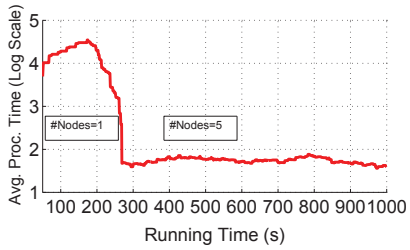


Fig. 9. Overloading handling on the Word Count topology

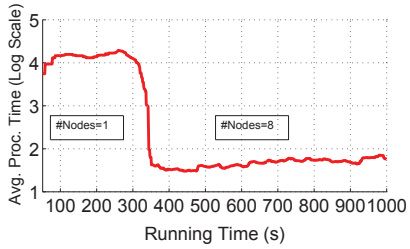


Fig. 10. Overload handling on the Log Stream Processing topology

it. We again used the Word Count and Log Stream Processing topologies for experiments. Note that since the average processing time during overloading is very large, we used the *logarithmic scale* for y axis.

For the experiment on the Word Count topology, we initially made it only use one worker on one node. We overloaded the topology by pushing two concurrent streams of word files into the topology. Once T-Storm detected overloading on executors, it calculated a new assignment utilizing more workers on more nodes to handle it. From Fig. 9, we can see after overloading was detected at about 120s, a new assignment was applied to use more worker nodes to undertake the load, and average processing time started to drop sharply to a normal value.

Similarly for the experiment on the Log Stream Processing topology, we initially set the topology to only use one worker on one node. We overloaded the topology by feeding 2 streams of IIS log files into the same Redis queue consumed by the spout. As shown by Fig. 10, T-Storm detected overloading at about 164s and produced a new assignment which utilized 8 worker nodes in the cluster to handle it. It led to a sharp drop on average processing time, which was very large during overloading.

VI. Related Work

In this section, we review the related systems and solutions. First, we briefly discuss MapReduce/Hadoop related research. In a pioneering work [3], Dean and Ghemawat introduced the MapReduce programming model and a runtime system. In [17], Fischer *et al.* studied MapReduce and Hadoop from a theoretical perspective by showing that the corresponding task scheduling problem is NP-hard and by presenting a flow-based algorithm to compute assignments that are optimal within an additive constant. In [18], Borkar *et al.* developed Hyracks, an implementation of an extensible DAG based programming model, which allows users to use a DAG to express computation and is compatible with Hadoop applications. Incoop [19] is a generic MapReduce framework for incremental computations. Incoop detects changes to the input and automatically updates the output by employing an efficient, fine-grained result reuse mechanism. In [20], Chen *et al.* presented an elastic execution engine E^3, which avoids reprocessing intermediate results and supports multi-stage jobs better than MapReduce. A resource allocation system, Purlieus, was presented in [21], which achieves high data locality for MapReduce in virtualized cloud environments. In [22], the authors considered the problem of jointly scheduling all three phases of a MapReduce process. They presented approximation algorithms and outlined several heuristics to solve the joint scheduling problem. In [23], Jin *et al.* introduced ADAPT, an availability-aware MapReduce data placement strategy to optimize the MapReduce applications performance in non-dedicated distributed computing environments.

Next, we discuss stream data processing systems. In [24], Kumar *et al.* extended IBM's System S stream processing middleware with support for MapReduce by providing language and runtime support for specifying and embedding MapReduce jobs as elements of a larger data-flow. In [25], a modified MapReduce architecture was introduced, which allows data to be pipelined between operators. This extends the MapReduce programming model beyond batch processing, and can reduce processing times and improve system utilization for batch jobs. Karve *et al.* in [26] proposed to use pipelining between the map and reduce phases to ensure the output of the map phase is made available to the reduce phase as soon as possible to speed

up the execution of MapReduce jobs. In [27], Espeland *et al.* introduced P2G, a framework developed specifically for processing distributed real-time multimedia data. In [28], Aly *et al.* presented a Hadoop-based system, M^3, which bypasses the HDFS to support main-memory-only processing and allows for continuous execution of the map and reduce phases where individual mappers and reducers never terminate. Backman *et al.* presented C-MR [29], a modified MapReduce processing model to continuously execute workflows of MapReduce jobs on unbounded data streams. Another MapReduce-like system, Muppet was presented in [30] for fast data processing, in which an update (instead of reduce) function is used to continuously update output based on data streams. A very recent work [31] presented a new model for distributed stream computation, called discretized streams, which enables fast recovery from both faults and stragglers without replication overhead.

The architecture and design of Storm are fundamentally different from either MapReduce-based batching processing system or MapReduce-like stream data processing systems. Hence, scheduling in Storm is quite different from that in Storm-based or Storm-like systems and the scheduling methods presented in the works introduced above cannot be applied to solve the problem considered here. So far, scant attention has been paid to scheduling in Storm. As mentioned above, the closest work is [11], which presented an offline scheduler and an online scheduler for Storm. However, they suffer from issues related to user transparency, topology dependency and lack of performance evaluation on real Storm applications (See Section III for details).

VII. Conclusions

In this paper, we first showed inefficiencies of the current practice of Storm scheduling and challenges associated with using traffic-aware online scheduling in Storm via experimental results and analysis. Then we presented the design, implementation and evaluation of T-Storm. T-Storm leverages traffic-aware online scheduling for speeding up data processing without overloading worker nodes. Moreover, it can consolidate workers and worker nodes such that T-Storm can achieve better performance with even less number of worker nodes. In addition, T-Storm allows hot-swapping scheduling algorithms and adjusting scheduling parameters on the fly; and it is transparent to Storm users. We conducted real experiments in a Storm cluster using well-known applications such as Throughput Test, Word Count and Log Stream Processing for performance evaluation. Experimental results have shown that compared to Storm (with the default scheduler), T-Storm can achieve over 84% and 27% speedup on lightly and heavily loaded topologies respectively (in terms of average processing time) with 30% less number of worker nodes; and 2) it can detect and recover from overloading in a timely manner.

References

[1] Twitter record, https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

[2] X. Sheng, J. Tang, X. Xiao and G. Xue, Sensing as a Service: challenges, solutions and future directions, IEEE Sensors Journal, Vol. 3, No. 10, 2013, pp. 3733–3741.

[3] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Proceedings of USENIX OSDI'2004*.

[4] Apache Hadoop, *http://hadoop.apache.org/*

[5] Storm, *http://storm-project.net/*

[6] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, Server workload analysis for power minimization using consolidation, *Proceedings of USENIX Annual Technical Conference 2009*.

[7] Guaranteeing message processing (Storm), *https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing*

[8] Apache ZooKeeper, *http://zookeeper.apache.org/*

[9] Storm 0.8.2 on Github, *https://github.com/nathanmarz/storm/tree/0.8.2*

[10] Storm Benchmark: Throughput Test, *https://github.com/stormprocessor/storm-benchmark*

[11] L. Aniello, R. Baldoni and L. Querzoni, Adaptive online scheduling in Storm, *Proceedings of ACM DEBS'2013*.

[12] L. Peterson and B. Davie, Computer Networks: A Systems Approach (5th Edition), *Morgan Kaufmann*, 2010.

[13] The Network Time Protocol, *http://www.ntp.org/*

[14] Example Topology: Word Count, *https://github.com/nathanmarz/storm-starter/blob/master/src/jvm/storm/starter/WordCountTopology.java*

[15] Alice's Adventures in Wonderland, *http://www.gutenberg.org/files/11/11-pdf.pdf*

[16] Q. Anderson, Storm real-time processing cookbook, *PACKT Publishing*, 2013.

[17] M. J. Fischer, X. Su, and Y. Yin, Assigning tasks for efficiency in hadoop: extended abstract, In *Proceedings ACM SPAA'2010*, pp. 30–39.

[18] V. Borkar, M. Carey, R. Grover, N. Onose, R. Vernica, Hyracks: a flexible and extensible foundation for data-intensive computing, *Proceedings of IEEE ICDE 2011*, pp. 1151–1162.

[19] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar and R. Pasquini, Incoop: Mapreduce for incremental computations, *Proceedings of ACM SOCC'2011*.

[20] G. Chen, K. Chen, D. Jiang, B. C. Ooi, L. Shi, H. T. Vo, S. Wu, E^3: an elastic execution engine for scalable data processing, *Journal of Information Processing*, Vol. 20, No. 1, pp. 65–76.

[21] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: locality-aware resource allocation for MapReduce in a cloud, *Proceedings of ACM SC'2011*.

[22] F. Chen, M. S. Kodialam, and T. V. Lakshman, Joint scheduling of processing and shuffle phases in mapreduce systems, *Proceedings of IEEE Infocom'2012*, pp. 1143–1151.

[23] H. Jin, X. Yang, X. Sun, I. Raicu, ADAPT: availability-aware MapReduce data placement for non-dedicated distributed computing, *Proceedings of IEEE ICDCS'2012*, pp.516–525.

[24] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, Deduce: at the intersection of Mapreduce and stream processing, *Proceedings of ACM EDBT'2010*, pp. 657–662.

[25] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy and R. Sears, Online aggregation and continuous query support in MapReduce, *Proceedings of ACM SIGMOD'2010*, pp. 1115–1118.

[26] R. Karve, D. Dahiphale and A. Chhajer, Optimizing cloud MapReduce for processing stream data using pipelining, *Proceedings of European Symposium on Computer Modeling and Simulation*, 2011, pp.344–349.

[27] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen, P2G: A framework for distributed real-time processing of multimedia data, *Proceedings of IEEE ICPPW'2011*, pp. 416–426.

[28] A.M. Aly, A. Sallam, B.M. Gnanasekaran, L. Nguyen-Dinh, W.G. Aref, M. Ouzzani, and A. Ghafoor, M^3: Stream Processing on Main-Memory MapReduce, *Proceedings of IEEE ICDE'2012*, pp. 1253–1256. 2012.

[29] N. Backman, K. Pattabiraman, R. Fonseca, U. Cetintemel, C-MR: continuously executing MapReduce workflows on multi-core processors, *Proceedings of ACM MapReduce'2012*, pp. 1–8.

[30] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, A. Doan, Muppet: MapReduce-style processing of fast data, *Proceedings of the VLDB Endowment*, vol. 5, issue 12, August 2012, pp. 1814–1825.

[31] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: fault-tolerant streaming computation at scale, *Proceedings of ACM SOSP'2013*, pp. 423–438.