2018 IEEE 26th International Conference on Network Protocols

On SDN-Enabled Online and Dynamic Bandwidth Allocation for Stream Analytics

Walid A. Y. Aljoby*†, Xin Wang*, Tom Z. J. Fu†, Richard T. B. Ma*†
*School of Computing, National University of Singapore
†Advanced Digital Sciences Center, Illinois at Singapore Pte Ltd
*†{algobi, tbma}@comp.nus.edu.sg, *xin.wang@comp.nus.edu.sg, †tom.fu@adsc-create.edu.sg

***Abstract*—Data communication in cloud-based distributed stream data analytics often involves a collection of parallel and pipelined TCP flows. As the standard TCP congestion control mechanism is designed for achieving "fairness" among competing flows and is agnostic to the application layer contexts, the bandwidth allocation among a set of TCP flows traversing bottleneck links often leads to sub-optimal application-layer performance measures, e.g., stream processing throughput or average tuple complete latency. Motivated by this and enabled by the rapid development of the Software-Defined Networking (SDN) techniques, in this paper, we re-investigate the design space of the bandwidth allocation problem and propose a cross-layer framework which utilizes the additional information obtained from the application layer and provides on-the-fly and dynamic bandwidth adjustment algorithms for helping the stream analytics applications achieving better performance during the runtime. We implement a prototype cross-layer bandwidth allocation framework based on a popular open-source distributed stream processing platform, Apache Storm, together with the OpenDaylight controller, and carry out extensive experiments with real-world analytical workloads on top of a local cluster consisting of 10 workstations interconnected by a SDN-enabled switch. The experiment results clearly validate the effectiveness and efficiency of our proposed framework and algorithms.**

I. Introduction

Large-scale stream processing has recently gained high importance due to a large variety of supported applications such as business intelligence, video analytics, machine learning, and event monitoring and detection. These applications process high volumes of unbounded with unpredictable and variable data streams. Nevertheless, they entail a variety of processing requirements, and thus researchers and practitioners have been developing a diverse array of stream processing frameworks (e.g., Storm [1], Heron [2], Samza [3], and Flink [4]) to meet the increasing demands of these applications.

For streaming applications, the essential performance indicator of how the application reacts to the incoming data streams is the time needed for each of them to be completely processed. To help the application to achieve a desirable performance characteristics (e.g., delivering real-time response), stream processing frameworks need to effectively and dynamically allocate system resources including CPU, memory, and bandwidth among application components (instances and their flows) in order to expose a highly-optimized pipeline for execution[5][6].

Yet, many applications today are data-intensive, as opposed to compute-intensive [7]. Indeed, in data-intensive applications, stream processing involves a higher network resource demands than CPU cycles, particularity when the data stream ingestion rates or derived tuples rates from these streams are higher than provisioned network bandwidth. As such, transfer across the network might be the cause of performance bottleneck rather than CPU cycles, therefore managing and optimizing network activity is important to improving and delivering real-time responses in these applications. In this context, there has been flurry of research attempts toward optimizing streaming applications. While in large part successful, however, their focus mainly has centered to schedule and provision computation resources of the applications or limited to minimizing traffic across the network. Hence, these solutions have largely overlooked allocation and provision of network bandwidth. As a result, they are either suboptimal in optimizing network transfer [6][8][9], or assuming the network with sufficient bandwidth resource [10].

In current stream processing frameworks, the share of network bandwidth has left to the mercy of the underlying transport mechanisms (e.g., TCP [11], DCTCP [12]). Nonetheless, such mechanisms are designed mainly for end-to-end data delivery in an application agnostic manner, i.e., flows traversing the bottleneck links sharing equal portion of the bandwidth. This, with high probability, will lead to sub-optimality in the overall application-level performance because some flows can be of paramount importance than others flows of the same application.

In this paper, we explore the design space of the bandwidth allocation, formulate it as a utility maximization problem, and propose a heuristic algorithm to derive the close-to-optimal solutions. This whole procedure is encapsulated into a cross-layer framework which utilizes the additional information measured from the running applications and quickly deploys the new allocation decisions to the physical network layer. The latter is enabled by the rapid development of the Software-Defined Networking (SDN) techniques and toolkits and realized through plugging in a control plane module developed (in ODL controller [13]) by us. The main contributions we have made in this paper are listed as follows:

1. We formulate the bandwidth allocation among flows belonging to a stream processing application as an optimization problem and design a heuristic algorithm to seeking for the optimal allocation solution.
2. Leveraging the SDN capabilities, we develop a native

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00030

SDN control plane application that deploys and updates the bandwidth allocation results derived by our proposed heuristic algorithm.
3. We implement a prototype of the proposed cross-layer bandwidth allocation framework based on a popular open-source stream processing platform, Apache Storm, integrating with the OpenDaylight SDN controller.
4. We carry out comprehensive performance evaluation through running stream data applications with real world workloads in a local cluster composed of 10 workstations interconnected by a hardware SDN-enabled switch.

The rest of the paper is organized as follows. Section II presents briefly an overview of stream processing, model of datacenter fabric, communication flow, and an example stream application motivating our contribution. In Section III, we introduce a formulation of bandwidth allocation as utility maximization problem and highlight surrounding challenges. We then present the details of solution model and optimization framework in Section IV. Section V describes briefly cross-layer SDN-based implementation of proposed solution. Experimental results are presented in Section VI. Finally, we explore related works in Section VII and conclude the paper and mention directions for future work in Section VIII.

II. Background and Motivation

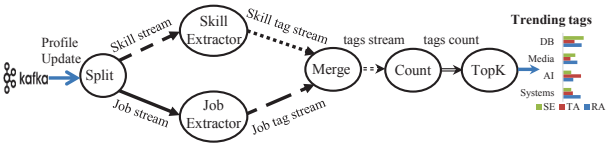
A. Stream Processing

Distributed Stream Processing Frameworks. Distributed stream processing frameworks, such as Storm [1], Twitter [2], and Samza [3], have been widely adopted in cloud-based data analytics to enable stream processing in a distributed manner with low latency. Using these frameworks, a variety of stream applications are developed for processing continuously arrived data streams from external producers (e.g., web logs, software logs, scalar sensors, and video cameras) through a pipeline of processing stages. Towards this, multiple models such as one-at-a-time and micro-batched have been proposed to cope with diverse stream application requirements [14]. In this paper, however, our focus is particularity pointed towards the one-at-a-time model which accomplishes processing on an individual tuple basis, for delay-sensitive stream applications.

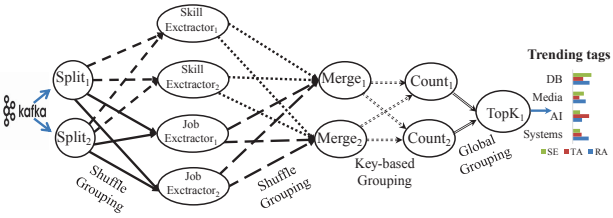
Application Model. Our conceptual viewpoint in designing our bandwidth allocation mechanism is to abstract out the entire application as a sequence of Fork-Join stages. The application is typically characterized by a logical topology, that defines a dataflow programming paradigm in a form of directed acyclic graph (DAG) of operators (i.e., vertices), through which data streams (i.e., edges) are constantly produced and consumed. Despite differences in DAG structure, we observe that stream applications have one thing in common: Fork-Join pattern, in which each edge starts with a fork operator and end with a join operator. Additionally, each operator can typically be classified according to number of input and output streams into a) *1:1* operator, b) *m:1* operator, and c) *1:m* operator. This variety in operators enables application programmer to flexibly chain them according to the logic of stream application. In Figure 1a, we show an example topology of finding trending tags at LinkedIn [3], which consists of six operators. In this topology, *Split* is a *1:m* type operator constitutes the entry point of the application, typically called source operator. It consumes streams of user profile updates, splits them into skill and job updates and then emits them to the two downstream operators named *Skill Extractor* and *Job Extractor*. Both *Skill Extractor* and *Job Extractor* are of the type *1:1* operators performing tag extraction on the input streams and sending the processing results to the *Merge* operator. The latter operator is a *m:1* operator combining skill and job tag streams into one and sending them to a *1:1 Count* operator which maintains the frequency of each distinct tag. Lastly, *TopK*, as the last operator in the topology, typically called sink operator, partitions received tag counts into windows (e.g., find top k tags over a 5-second window) and keeps updating application statistics (e.g., trending tags such as user skills and job positions, in this example).

Furthermore, stream applications exhibit a wide diversity in terms of scale, state, and lifetime. In particular, the scaling of stream application is important to cope with computational need of time-constrained applications. As such, processing frameworks offer users a set of APIs to configure multiple instances per operator in order to execute user-defined logic concurrently. Figure 1b presents a parallelization of DAG shown in Figure 1a where each operator is replicated twice (e.g., $Split_1$ and $Split_2$ are instances of the *Split* operator), except the sink operator has only one instance, $TopK_1$.

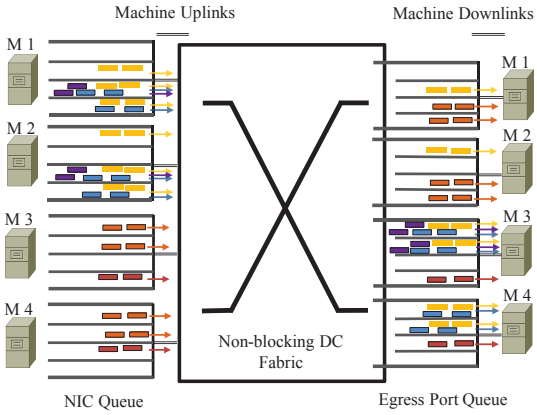
Stream Grouping Policies. As parallelizing operators is a key factor to speeding up stream processing, the transfer of data streams between the instances must follow appropriate grouping policies, sometimes called data routing, to meet different application requirements. The main grouping policies of regulating how data tuples are forwarded to target instance set of the downstream operators are categorized as follows.
Shuffle grouping: data tuples are sent in a round-robin fashion to the instances of the downstream operator. This policy ensures that the processing workloads are evenly distributed among these receiving instances.
Key-based grouping: the destination instances to which data tuples are sent is determined by applying some well defined projection function, e.g., hashing, on the key (or signature) of the data tuple. This grouping policy has the property that a) the tuples sharing the same key are guaranteed to send to the same instance, i.e., avoiding an additional step for key-based result aggregation; and b) roughly even-partition the key space among all the destination instances, but this is insufficient to guarantee workload balance under the skewed key distributions, e.g., heavy tail distributions.
Global grouping: data tuples are sent to a dedicated instance of a downstream operator, typically for results aggregation in the final stage of the topology. As an example, $TopK_1$ instance gathers tag counts from all instances of operator *Count*.
All grouping: data tuples are duplicated and sent to all instances of the downstream operator, equivalent to data broadcasting.



(a) Example logical topology of streaming application



(b) Illustration of parallelism with grouping policies



(c) A 4×4 datacenter as a big switch

Figure 1: Bandwidth allocation of streaming application over a 4×4 datacenter fabric with four uplinks/downlinks (a) Operators component of trending tags logical topology (b) Parallelism of logical topology into instances and data grouping policies among them (c) Application external flows competing for machine uplinks and downlinks. These flows are resulted from instance placements as: M1$\leftarrow$ $Split_1, Job\ Extractor_1, Count_1$; M2$\leftarrow$ $Split_2, Job\ Extractor_2, Count_2$; M3$\leftarrow$ $Skill\ Extractor_1, Merge_1, TopK_1$; M4$\leftarrow$ $Skill\ Extractor_2, Merge_2$. The instances are color-coded in operator-based, Split in yellow/dark, Skill Extractor in red/dark, Job Extractor in blue/dark, Merge in orange, and Count in purple. Subsequently, the flows are organized at the NIC queue (rest. egress port queue) by destinations (rest. sources).

Instance Placement. Given that user has configured parallelism information of the application operators in terms of instances, however, to realize this into practice, a framework scheduler will then accomplish this throughout a mapping between operator instances and physical computing machines which will host them. We refer to such mapping as an instance placement, which is usually carried out based on specific strategies including those simple ones such as random/round-robin assignment, or more sophisticated ones such as traffic-aware assignment [8][15]. Once the instances are mapped and ran on the hosting machines, the data communication patterns between each pair of instances are fixed. For example, we show in Figure 1 the results of applying round-robin placement strategy to parallel version of our example application over 4 interconnected compute machines M1, M2, M3, and M4.

B. Network Model

In our analysis and implementation, we model the entire datacenter fabric as one big switch. The reason for this is because today's datacenters are predominantly being built with the aid of rich interconnectivity, such as Fat-Tree [16] and Leaf-Spine multi-rooted Clos [17][18] modern network topologies. This in turn enables datacenter architecture to support full-bisection bandwidth and subsequently makes capacity of the fabric's internal links is largely bottleneck-free [19]–[22]. As a result, datacenter fabric can be thought of as resembling a non-blocking big switch interconnecting all machines, as depicted in Figure 1c.

With this model, the ingress and egress ports of the big switch directly connect to machines' uplinks and downlinks (i.e., machines NICs) [23][24]. We define the *uplink* (res. *downlink*) of each machine as its communication channels to (res. from) the big switch. This model is attractive for its simplicity, yet practicability. Instead of focusing on the individual switches in the fabric, we focus only on one big switch where machines' uplinks and downlinks are the only potential places where the bottlenecks happen. However, if the network is not a non-blocking switch, then our algorithm can be extended to allocate rack-to-core bandwidth instead of machines uplink and downlink bandwidth. We leave this for future work.

C. Communication Flows

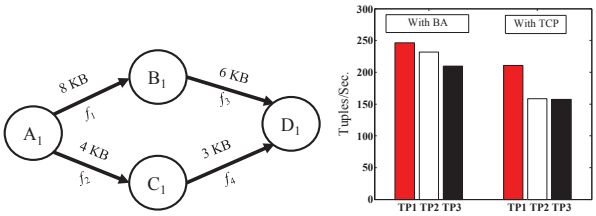
We refer to a uni-directional data transfer between any given pair of instances as a flow, denoted by f. A flow is called an *internal* flow if its two communication instances are placed in the same machine, otherwise it is called an *external* flow. Recall the application we illustrate in Figure 1 and its corresponding instance placement, the data transmission from $Split_1$ in M1 to $Job\ Extractor_1$ in M1 is an example of internal flow, while the flow from $Split_1$ in M1 to $Skill\ Extractor_1$ in M3 is an external flow.

As we adopt the big switch model, this leads to the fact that all the external flows traverse exactly two uni-directional links. In addition, Figure 1c demonstrates how the bandwidth contention and packet dropping happen when multiple flows are traversing the same bottleneck link. For the ease of exposition, flows are attached to virtual queues at machines' NICs connecting the uplinks at ingress side; and to virtual queues at the egress side of the fabric linking the NIC of the destination machine. It is also worth noting that flows competing for the uplinks belong to a fork stage, while those flows competing for the downlinks belong to a join stage. For example, the $Job\ Extractor_1$ instance in M1 has two flows color-coded in

blue/dark at the M1's uplink destined to $Merge_1$ and $Merge_2$ instances, and one flow color-coded in yellow/dark at the M1's downlink from $Split_2$ instance. Given that queues are used to buffer the packets of flows competing for bandwidth on uplinks and downlinks over the big switch, the natural question to ask is how much bandwidth shall be allocated to each flow for its packet transfer. The key contribution of this paper is to allocate link bandwidth among these competing flows with aim of maximizing application welfare that leads to low latency and high throughput streaming application.

D. Motivation Example

In our preliminary study [25], we have measured and analyzed the impact and importance of bandwidth allocation on streaming applications. Measurement-driven analysis has been conducted on a bandwidth-limited cluster running streaming application of 4 operators with parallelism set to 1 for each operator (Figure 2a). In summary, we have evaluated application performance in terms of application-level throughput, that is expressed as total number of completely processed tuples per second (averaged over 300-second experiments).



(a) Application topology (b) Throughput comparison

Figure 2: A motivation example: under three options for instance placements, we show the performance of bandwidth allocation (with TCP) versus the optimal (with BA).

Figure 2b compares the overall throughput of the application under multiple placements, as denoted by TP1, TP2 and TP3. Under each placement strategy, we compared two different bandwidth allocation mechanisms, the default TCP congestion control (i.e., With TCP) and the best allocation (i.e., With BA) we obtained throughout brute-force search. We observed that a proper bandwidth allocation rather than vanilla congestion control can make further improvement on the overall application throughput. In particular, our observation is that instead of achieving max-min fair *rate* allocation as approximated by TCP, the proper allocation is to look for max-min fair *utility* allocation in which flow utility is inversely proportional to its volume, e.g. proportional allocation as in [25], that takes into account application welfare, at which concurrent flows are processed altogether in a reasonable time window. Overall, in our example study in Figure 2, we found that the best bandwidth allocation has achieved 17%, 47%, and 33% improvement in placement TP1, TP2 and TP3, respectively. Nonetheless, brute-force search for the best allocation, e.g. as in [25], is too costly to be affordable in practice. Therefore, in this paper, we develop smart linear programs to achieve max-min fair utility allocation in order to entitle bandwidth allocation to the optimality over the lifetime of stream analytics.

III. Bandwidth Allocation in Streaming Applications

In this section, we first formulate a bandwidth allocation problem of streaming application, followed by a discussion of main challenges to resolving this problem.

A. Problem Formulation

Consider a datacenter network that consists of a set $\mathcal{L} = \{1, ..., L\}$ of links of capacity C_l, $l \in \mathcal{L}$. The network is shared by a set $\mathcal{F} = \{1, ..., F\}$ of flows. We denote the rate of any network flow $f \in \mathcal{F}$ by x_f. Our goal is to find a vector $\mathbf{x} = (x_1, x_2, ..., x_F)$ of flow rates that maximizes the overall application welfare $U(\cdot)$, s.t. constraints of link capacities:

$$\max_{\mathbf{x}} \quad U(\mathbf{x}) = U(x_1, x_2, ..., x_F) \tag{1}$$

$$\text{s.t.} \quad \mathbf{Rx} \leq \mathbf{C}, \tag{1a}$$

where, $\mathbf{R}$ is a binary valued routing matrix, of which $\mathbf{R}(f, l)$ = 1 if and only if flow f traverses link l; and $\mathbf{C}$ is the vector of link capacities. The constraint (1a) means that the aggregate flow rate at any link l cannot exceed its capacity C_l.

In our context, the overall application welfare U can be quantified by the aggregate processing rate of the streaming application and is associated with managing bandwidth allocation among application's *network* flows deployed over a big switch interconnecting a set $\mathcal{M} = \{1, ..., M\}$ of machines. We denote the uplink starting with machine i by u_i and the downlink ending with machine j by d_j. Furthermore, as the application involves active flows for unbounded time, hence the optimal control of flow rates might change over time. We therefore denote the vector $\mathbf{x}$ of flow rates at time t by $\mathbf{x}(t)$ and re-describe the problem in (1) as to find the optimal $\mathbf{x}^*(t)$ at time t that maximizes U, s.t. capacity constraints of machine uplinks and downlinks are satisfied.

$$\max_{\mathbf{x(t)}} \quad U\big(\mathbf{x}(t)\big) = U\big(x_1(t), x_2(t), ..., x_F(t)\big) \tag{2}$$

$$\text{s.t.} \quad \sum_{s(f)=i, d(f)\neq i} x_f(t) < C_{u_i}, \ \forall i \in \mathcal{M}, \tag{2a}$$

$$\sum_{d(f)=j, s(f)\neq j} x_f(t) < C_{d_j}, \ \forall j \in \mathcal{M}, \tag{2b}$$

where, $s(f)$ and $d(f)$ denote the source and destination of flow f. The constraints (2a) and (2b) ensure that the aggregate flow rates at any machine uplink u_i and downlink d_j do not exceed the uplink capacity C_{u_i} and the downlink capacity C_{d_j}, respectively. However, the key challenge is in defining Non-Clairvoyant flow utilities $U\big(\mathbf{x}(t)\big)$ that through them network flows are allocated proportional rates to their volumes [25], without prior knowledge of flow volume; meanwhile, the flow volume should be estimated in the presence of continuously- and timely-varying load on processing pipeline. Before we show how to derive subtle flow utility functions and how to apply bandwidth allocation based on them, we present following specific challenges to flows in stream analytics.

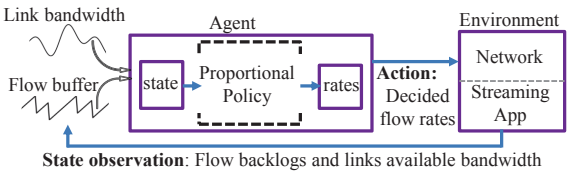


Figure 3: The bandwidth allocation optimization framework for stream analytics.

B. Challenges

- **Unbounded flows transfer**: A data stream is an unbounded sequence of events over time. This unbounded nature of data streams makes their corresponding network flows unbounded as well. Most existing state-of-the-art approaches are focused primarily on management of bounded network flows such as network flows of MapReduce jobs or search queries. For example, scheduling flows based on flow's remaining size [19], or bytes sent by each flow [26] or coflow [24] have been introduced to minimize completion time by mimicking the *Shortest Job First* (SJF) approach. However, these techniques remain hard to adapt to unbounded network flows of stream application.
- **Undetermined flows volume**: Also, the unpredictability and variability are common attributes of data stream sources. The variability in data streams makes it hard to obtain accurate flow information because flow volume in terms of tuple unit size and sending rate usually change with time. Consequently, bandwidth allocation policies that rely on prior knowledge of flows [19] or coflows [23] remain also inapplicable for stream applications. In result, bandwidth allocation policy has to capture flow updates and be adaptable to changes.

Given these challenges, we introduce a cross-layer bandwidth control framework under which streaming application and network can be flexibly characterized to each other. We particularity show that bandwidth allocation policy we aspire to find, can be seamlessly transpired and materialized.

IV. Bandwidth Allocation Optimization Framework

As we mentioned earlier, solving bandwidth allocation problem requires deriving flow utilities that through them network flows are allocated proportional rates to their volumes. To address this problem, we propose an *agent-environment* optimization framework similar to those used in reinforcement learning problems, as shown in Figure 3. In this framework, the agent observes a set of metrics, measured from the environment, including flow backlog from the application and network links' available capacities from the network. We use flow backlog to characterize flow state each time interval in order to enable determining flow utility in a non-clairvoyant manner. The agent then feeds these measurement values to a bandwidth allocation algorithm, to compose flow utilities and to perform an optimization. Finally, the agent takes an action, that is a rate vector for regulating network flows in the next time interval.

In the following, we describe the components of proposed bandwidth allocation optimization framework (Figure 3).

A. Input Details

To control each flow rate, it is necessary to obtain at the first place the association relationships between data transmission flows and the underlying physical links. In streaming application, this can be achieved by sorting out the parallel version of application's logical topology, together with its placement map into the physical machines. Luckily, this information is easily to retrieve from master of the stream processing platform (e.g., Nimbus of Apache Storm) or from the cluster manager (e.g., Application Manager of YARN or Mesos in Apache Heron and Samza). Once the association between flows and physical links are known, the second step then is to record flow backlogs and to determine links' available capacities. The flow backlogs are particularly important to characterize the state of each flow in order to use them in estimating each flow utility. Meanwhile, available links capacity are required to optimize flow rates accordingly.

*1) **Flow buffer**:* To discern what bandwidth is needed for each flow f in streaming application, instead of using flow rate, we model flow state by profiling the actual data transfer of the flow and its status at endpoints (i.e., sender and receiver). The reason of not using flow rate to express flow need is because rate observed presumably depends on cross-traffic in the network which does not reflect what bandwidth is really needed for this flow.

To obtain actual data transfer of the flow, we keep recording amount of sent tuples at the application-level sender function, expressed in MB; meanwhile, for flow status, we keep track flow backlogs at flow endpoints via maintaining and measuring each endpoint with a dedicated queue. The sender endpoint backlog is computed as a queue length in MB of data tuples that are waiting in the queue for transfer service across network link. We use this metric to indicate how sending rate of the flow's sender endpoint is higher than available bandwidth for this flow. This case is used in each fork instance of the application. On contrary, the receiver endpoint backlog is computed as a queue length in MB of received data tuples that have been waiting in a queue for processing service. However, the latter case is used in each join instance of the application. For example, consider a receiving instance that performs join processing of tuples originating from flows of multiple sending instances. In such a case, the receiving instance might be *stalled* if some of the required flow's data tuples have not received yet due to network congestion. This leads to that flow of available tuples can not be further processed, and thereby not only this flow processing will be delayed, but also the entire pipeline processing. Due to also constant streaming of data tuples, we observed that the delay of flow processing overruns instance's memory capacity which might largely cause memory overflow (e.g., OOM). Therefore, we use queue length in this context to indicate the degree of

instance stalling. Thus, to alleviate stalling, the unavailable flow's data tuples should be brought much faster by allocating it higher bandwidth than delayed flow.

The output of buffer profiling is a *5-metric* tuple characterizing state of each flow $f \in \mathcal{F}$ in a time interval $(t, t+\Delta_t)$. The metrics include queue length of the flow f at the sender and receiver at time t denoted by $L^s_f(t)$ and $L^r_f(t)$; the queue length of the flow f at the sender and receiver at time $t+\Delta_t$ denoted by $L^s_f(t+\Delta_t)$ and $L^r_f(t+\Delta_t)$, and the actual flow's data size transferred within t and $t+\Delta_t$ denoted by $V_f(t, t+\Delta t)$. A 3-metric tuple $< L^s_f(t), V_f(t, t+\Delta t), L^s_f(t+\Delta_t) >$ is used for characterizing the flow state at the uplink and another 3-metric tuple $< L^r_f(t), V_f(t, t+\Delta t), L^r_f(t+\Delta_t) >$ is used for characterizing the flow state at the downlink. For the flows contending for uplink, the higher values queue length of the flow, the higher demands of link bandwidth; while the flows contending for downlink the lower values queue length of the flow, the higher demands of link bandwidth. The profiling of flow states is thereby useful to compose flow utilities in order to serve each network flow need not only according to underlying network available capacity but also based on flow importance to application performance. Shortly, we shall see how each flow state, expressed by the *5-metric* tuple, is used to compose flow utilities and to define optimization policy over these utilities that allocates bandwidth in proportional to each flow importance in the application performance.

*2) **Link Bandwidth**:* The bandwidth allocation algorithm has to know in addition to flow state, the network links allocatable capacities (i.e., available bandwidth). However, link allocatable capacity can be computed as a difference between its total capacity and the active flows rates over this link. Strikingly, in a cooperative SDN-based cluster in which a single administrative entity controls the network, all packets are transferred as instructed by forwarding rules stored in the network devices. This ensures that all network flows belong to multiple applications should be known over all machines links. Therefore, we leverage OpenFlow statistics features to collect flow statistics in order to estimate used bandwidth of each link. We use symbols C_{u_i} and C_{d_j} to correspond respectively to allocatable capacities of uplink u_i and downlink d_j.

B. Bandwidth Allocation Algorithm

In Algorithm 1, we summarize the procedure of optimization of bandwidth allocation for maximizing the performance of stream processing application. Given the big switch model in which uplinks and downlinks (line 2) subject to their available bandwidth (line 6) are our focus and given that flow assignment over these links are determined (lines 4 and 5) and each flow state is also obtained (line 16), thus the mechanism of bandwidth allocation is as follows.

As we mentioned earlier, from our analysis of mapping of application's flows into machines uplinks and downlinks, we observed that stream applications follow a *Fork-Join* communication pattern in common. In the *Fork* stage, network flows of instance(s) co-located at the same machine compete(s) for machine's uplink, while in the *Join* stage, the instance(s) at

Algorithm 1 Bandwidth Allocation

1: **Input**:
2: Uplinks $\{u_i : i \in \mathcal{M}\}$ and downlinks $\{d_j : j \in \mathcal{M}\}$
3: Application network flows $\mathcal{F} = \{1, ..., F\}$
4: Uplinks flow set $\{\mathcal{F}_{u_i} : i \in \mathcal{M}\}$
5: Downlinks flow set $\{\mathcal{F}_{d_j} : j \in \mathcal{M}\}$
6: Allocatable capacities of links $\{C_l : l \in \mathcal{L}\}$
7: **Output**:
8: Proportional fair share x_f for each flow $f \in \mathcal{F}^b$
9: **Initialization**:
10: Bottlenecked uplinks, downlinks, and flows are $\omega^b \subset \omega$, $\phi^b \subset \phi$, and $\mathcal{F}^b \subset \mathcal{F}$ respectively;
11: $t \leftarrow 0$;
12: $L^s_f(t) \leftarrow 0$, $L^r_f(t) \leftarrow 0$, $\forall f \in \mathcal{F}^b$
13: **do**
14: run the streaming system for time Δ_t;
15: record $V_f(t, t+\Delta_t)$, $L^s_f(t+\Delta_t)$ and $L^r_f(t+\Delta_t)$, $\forall f \in \mathcal{F}^b$
16: **for** each uplink $u_i \in \omega^b$ **do**
17: Solve the optimization problem (3)
18: **end for**
19: **for** each downlink $d_j \in \phi^b$ **do**
20: Solve the optimization problem (4)
21: $x_f = \min\{(x^u_f(t+\Delta_t), (x^d_f(t+\Delta_t)\}$.
22: **end for**
23: $t \leftarrow t + \Delta_t$;
24: **While** ($\exists f |$ $f \in \mathcal{F}^b \wedge (\ L^s_f(t) \neq 0 \vee L^r_f(t) \neq 0)$)

certain machine, receive(s) multiple flows from some other instances that compete for machine's downlink. One more important observation is that majority of streaming applications flows sharing network bandwidth have to be concurrent within a reasonable time window in order to be processed together. We use these two observations to derive bandwidth allocation policy to maximize the aggregate processing rate of streaming application, based on optimization problem (2). Therefore, to achieve this, the bandwidth allocation policy should make sure that if the input speed of data generated by the sender machines keeps unchanged during the next period of time, then the system can finish handling all flow backlogs in all machines in the shortest time.

$$\min_{x^u_f(t+\Delta_t)} \max_{f \in F_{u_i}} \frac{V_f(t, t+\Delta_t) + 2L^s_f(t+\Delta_t) - L^s_f(t)}{x^u_f(t+\Delta_t)} \tag{3}$$

$$\text{s.t.} \sum_{f \in F_{u_i}} x^u_f(t+\Delta_t) = C_{u_i}, x^u_f(t+\Delta_t) \geq 0 \tag{3a}$$

$$\min_{x^d_f(t+\Delta t)} \max_{f \in F_{d_j}} \frac{L^r_f(t+\Delta_t) + x^d_f(t+\Delta_t)\Delta_t}{[V_f(t, t+\Delta_t) - L^r_f(t+\Delta_t) + L^r_f(t)]/\Delta_t} \tag{4}$$

$$\text{s.t.} \sum_{f \in F_{d_j}} x^d_f(t+\Delta_t) = C_{d_j}, x^d_f(t+\Delta_t) \geq 0 \tag{4a}$$

Next, we use this idea to explain the derivation of the optimization problems (3) and (4) of our algorithm. If an uplink u_i is shared by multiple flows whose set is denoted by F_{u_i}, then it is under the *Fork* pattern. Thus to define flow utility for any flows $f \in F_{u_i}$, the data amount of the flow f generated by the sender machine i during the time interval $(t, t+\Delta_t)$ is $V_f(t, t+\Delta_t) + L_f^s(t+\Delta_t) - L_f^s(t)$, i.e., the data size $V_f(t, t+\Delta_t)$ of the flow f transferred plus the variation $L_f^s(t+\Delta_t) - L_f^s(t)$ of the queue length of the flow f in the sender machine i during the time interval $(t, t+\Delta_t)$. If the generating speed of the data of the flow f keeps unchanged during the next time interval $(t+\Delta_t, t+2\Delta_t)$, then there will exist the data of the size $V_f(t, t+\Delta_t) + 2L_f^s(t+\Delta_t) - L_f^s(t)$ needed to be transferred during the time interval $(t+\Delta_t, t+2\Delta_t)$. Under the bandwidth $x_f^u(t+\Delta_t)$ allocated to the flow f during the time interval $(t+\Delta_t, t+2\Delta_t)$, the total time to finish transferring the data is at least $\left[V_f(t, t+\Delta_t) + 2L_f^s(t+\Delta_t) - L_f^s(t)\right] / x_f^u(t+\Delta_t)$. Because our goal is to make the system finish processing all backlogs of the flows sharing the link i in the shortest time, which usually occurs when the maximum of the transferring times $\left[V_f(t, t+\Delta_t) + 2L_f^s(t+\Delta_t) - L_f^s(t)\right] / x_f^u(t+\Delta_t)$ among the flows of sharing the uplink i is minimized, we derive the optimization problem (3). Based on the same idea, we can also derive the optimization problem (4).

Now, by optimization problems (3) and (4) we succinctly reveal the details of the problem's abstract formulation in (2), and solving them should readily fulfill our objectives. The constraints, $(3a)$ ensures that aggregate of allocated flow rates at any machine uplink u_i does not exceed uplink capacity C_{u_i}, and $(4a)$ ensures that aggregate of allocated flow rates at any machine downlink d_j does not exceed downlink capacity $\mathcal{C}_{d_i}$. The symbols x_u and x_d are per-flow f allocated rate at flow's uplink and downlink, respectively, and the flow will be given the minimum allocated rate x_f of either (line 21).

Overall, when network flows sharing bottleneck uplink or downlink are unequal in their volumes (i.e., estimated by flow state) and are required to be processed concurrently, the default transport (e.g., TCP) in processing frameworks is obviously ill-suited because it is unaware of the application atop how such variabilities impact its performance. On contrary, our algorithm checks state of each flow at each congested uplink and downlink and rigorously solves optimization problems to distill the optimal rate for each flow over respective links (lines 10-24). This continues alongside any of the flow buffers are not empty and in each time interval outputs per-flow optimal allocation.

V. Implementation

Figure 4 presents a multi-tier architecture for management and implementation of bandwidth resource allocation based on our proposed optimization framework (described in Section IV). The implementation involves dividing solution architecture into three tiers: Streaming application management plane (tier 1), SDN-based control plane (tier 2), and SDN-based forwarding devices plane (tier 3). We materialize streaming application manager in Storm framework [1], the popular open-source stream processing platform. Meanwhile, we implement SDN-based control in OpenDaylight SDN controller [13], the largest community-led and industry-supported open source SDN framework. For the SDN-based forwarding plane, this is the tier wherein network bandwidth resources reside and in need of effective management to maximize streaming application performance. In tier 3, to interoperate with above tiers, we adopt SDN-based devices which dispense with arriving data packets of streaming application according to the desired performance goals elaborated by the management plane. Taken together, the streaming application manager in tier 1 will then quickly and dynamically program the network devices in tier 3 via some of the abstract interfaces supported by network controller in tier 2, as follows.

A. Tier 1: Streaming Application Manager

This tier is the core of our solution, it implements the agent decision-maker in above agent-environment framework. By observing the state of all network flows encompass streaming application and the available links bandwidth, streaming application manager then solves optimization problems based on proposed bandwidth allocation algorithm and provides the optimal rates allocated to network flows for next time period. The main components of this manager are summarized as follows.

*1) **Flow Mapper**:* To determine the optimal bandwidth for each flow, bandwidth allocation algorithm should know as a first step application flows to network links mapping over all machine uplinks and downlinks. We built a customized scheduler instead of Storm default scheduler in order to maintain a deterministic map of applications instance over the compute machines.

*2) **Application Profiler**:* This entity is also known as flow buffer, it aims to monitoring and collecting the state of network flows as driven by our model and to feed them to the bandwidth optimizer component. Fortunately, we leverage Storm metric interface to collect the *5-metric* tuple we are interested in, based on a certain defined sampling rate. Furthermore, to update bandwidth optimizer with these metrics, we employ a pull-based mechanism under which bandwidth optimizer makes order request for these metrics each Δ_t time interval.

*3) **Bandwidth optimizer**:* The entity is the heart of tier 1. At each time interval, bandwidth optimizer pulls the application and the network. In particular, the optimizer uses the input reported by flow mapper, application profiler, and link stats collector to execute bandwidth allocation algorithm and to solve linear optimization problems and subsequently sends the output of the algorithm (i.e., the decided optimal flow rates) to the network controller. For this purpose, we develop a client-server based interfaces to enable 2-way communication between bandwidth optimizer resides at tier 1 and bandwidth enforcer resides in SDN-based control at tier 2.

B. Tier 2: Network Controller

This tier is implemented in the world's largest open source SDN platform, an OpenDaylight (ODL) controller, which acts

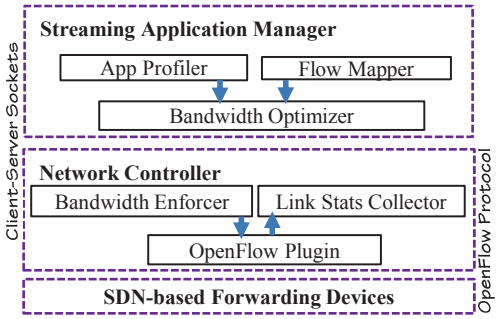


Figure 4: SDN-based implementation of proposed bandwidth allocation algorithm.

as middleware that orchestrates and facilitates exchange of control messages between application manager (tier 1) and Openflow-based physical network (tier 3). For our purpose, in addition to the core components of ODL controller, we develop a native MD-SAL compliant network application within the controller to primarily implement bandwidth allocation and to collect link measurements and statistics information. In particular, this application consists of bandwidth enforcer and link statistics collector modules. Technically speaking, our development of such a native application in ODL encompass different technologies such as OSGI, Karaf, YANG modeling, blueprint container, and a set of distinctive messaging patterns including RPC, publish-subscribe, and datastores accesses [13]. We skip the modular design and implementation details for the sake of brevity.

*1) **Bandwidth Enforcer**:* This module is built to dynamically update the rates of network flows according to the output of optimization algorithm. Instead of directly modifying the underlying transport mechanisms (such as TCP), we leverage the metering API feature supported in OpenFlow-based networks to enforce the output rates received by bandwidth optimizer into the datapath of the underlying network devices. Specifically, we associate each network flow with a meter under which the packets belonging to this flow are allowed to pass through the egress ports of the switch for up to specific upper-bound rate (e.g., the rate decided by bandwidth optimizer). Subsequently, the bandwidth enforcer instructs OpenFlow plugin to translate allocated rates into OpenFlow messages and to install them into the networking devices.

*2) **Link Statistics Collector**:* We implement this module to estimate the available bandwidth of the link in case the network is injected with cross traffic from multiple applications. In OpenFlow network, all packets belong to network flows are transferred based on flow forwarding rules stored in the network devices. Meanwhile, we registering all these rules with statistics service module in ODL. This module uses service APIs implemented by OpenFlow plugin to send statistics requests to network devices to report flow statistics including packets count, bytes count, and duration.

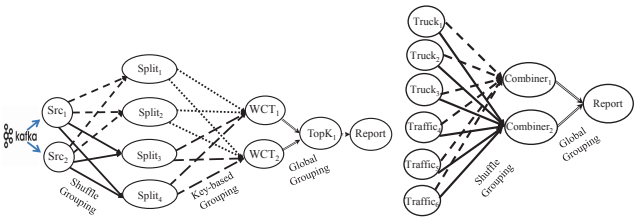
*3) **OpenFlow Plugin**:* This component is one of the essential components in ODL which implements OpenFlow protocol to mediate the communication between underlying network devices and network control applications. It is used in tandem with functions of network application developed in tier 2 to interact with the underlying network devices supporting OpenFlow protocol.

C. Tier 3: SDN-based Forwarding devices

This tier is primarily the data plane of the network interconnecting machines using SDN-enabled devices. These devices are directly programmable by means of OpenFlow protocol from a centralized network controller in tier 2. Each device contains a pipeline of flow tables used to store flow rules installed by the controller under which the main set of network functions such as forwarding, rate control, and routing are supported. More recently, some SDN-enabled devices also support a meter table to store meter entries to implement bandwidth-limiting during ingress processing of received packets. Each entry works like a token bucket policer which measures and controls the rate of packets belongs to each flow for up to specific pre-defined rate (e.g., the rate advised by bandwidth optimizer).

VI. Experimental Evaluation

In this section, we evaluate the performance of proposed bandwidth allocation model with a real hardware testbed. Our testbed experiments illustrate proposed model's good performance for realistic and synthetic workloads of real streaming applications. Our detailed experiments confirm that our model also achieves close-to-TCP in terms of network utilization.



(a) Trending Topics (TT) (b) Trucking IoT (TI)

Figure 5: Topologies of two test streaming applications.

A. Setup

*1) **Testbed Experiments**:* We built a testbed that consists of 10 Ubuntu Linux physical machines connected to a Brocade ICX-6610 24-port Gigabit SDN-enabled hardware switch with 1Gbps links including uplinks and downlinks. The machines have 4-core Intel Xeon E5-1620 3.5GHz CPUs, 16GB of RAM, and 1TB HDDs. In the testbed, we set up 9 machines Storm cluster with 1 machine hosting Storm master (i.e., Nimbus), Zookeeper, and bandwidth optimizer, and 8 machines hosting Storm worker nodes (i.e., supervisors) which run the instances of experimental streaming applications. During experiments, the worker nodes were kept in sync by using the standard NTP protocol on the Ubuntu Linux. We also configure 1 machine to host Opendaylight SDN controller including our proposed SDN control plane bandwidth enforcer application.

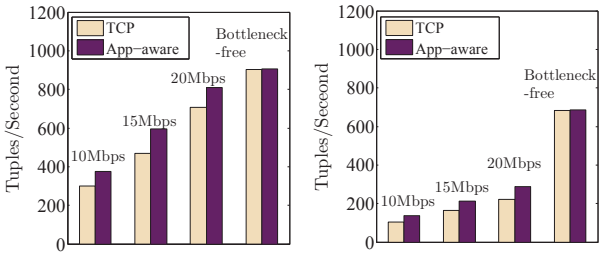
*2) **Test Applications and Benchmarks**:* We have implemented two real-time stream analytics applications on top of Apache Storm: Trending Topics (TT) and Trucking IoT (TI). Figure 5 shows the topologies of the TT and TI applications. TT application considers a topic as trending when it has been among the top K topics in a given window of time. We implement a topology of this application which consists of chain of five operators, in which each operator has one or more instances. The first operator is source of the topology that emits unbounded sequence of data streams to next operator to split them into words and emits them to the next operator, word count operator (WCT), to perform and maintain word counting. Then, WCT waits a time equals to K arrivals and performs emitting to an aggregator operator. The latter will jointly process the receiving tuples from multiple WCT instances, extracts the top K trending topics from all of them, and emits the results to report operator. For TT, we use a real dataset from twitter which contains millions of tweets for about 3 years. We set the sliding window to 30 seconds and emulate average arrival rate of 1000 tweets per second.

The TI application, is the widely used one in its topology design by many of streaming applications such as frequent pattern detection [10] and distributed computer-vision pipelines [27]. Trucking IoT service performs real-time analysis on IoT data streams coming from multiple sources. This topology receives two different streams, one stream reports data about truck status while the other one about traffic congestion status. The application processes both streams concurrently in a way that data from the truck is combined with the most up-to-date congestion data and reports a timely action that should happen accordingly. For TI, we use two synthetic datasets of different data tuple sizes which emulate the sizes of real tuple sizes reported by each of the truck sensor and traffic congestion online source service. We also emulate the arrival rate of 250 tuples per second of each stream.

With the TT and TI applications, we evaluate the performance of streaming application when the network bandwidth is the bottleneck possibly, i.e., the derived tuples rate from the application or data stream ingestion rate is higher than available network bandwidth, respectively.

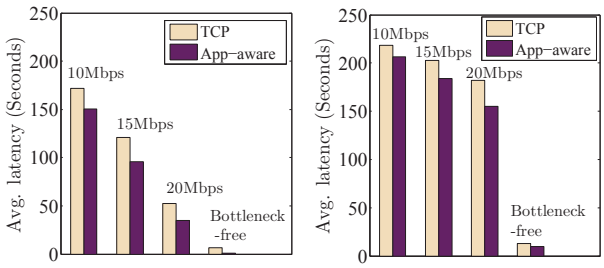
*3) **Baseline**:* We use the standard TCP bandwidth allocation model as our baseline. It is the default model used by streaming frameworks like Storm [1], Heron [2], and Flink [4]. We compare TCP against our proposed bandwidth allocation for streaming applications, which we call it, *App-ware*, for purpose of identification.

We ran a series of experiments each with 600 seconds. In all experiments, we set a sampling rate for the per flow's 5-metric tuple to 1 per second in order to report flows backlog to application profiler. Also, we set the timer interval Δ_t to 5 seconds to periodically perform new bandwidth allocation by bandwidth optimizer. We repeat each experiment 4 times over the cluster of different available bandwidth. In particular, we set link bandwidth to 10Mbps, 15Mbps, and 20Mbps to evaluate impact of different bandwidth bottlenecks for running application. We also run the applications on the cluster with bottleneck-free setting (i.e., sufficient available capacity).



(a) Trending Topics (TT) (b) Trucking IoT (TI)

Figure 6: Application Throughput.



(a) Trending Topics (TT) (b) Trucking IoT (TI)

Figure 7: End-to-end Latency.

B. Performance Improvement

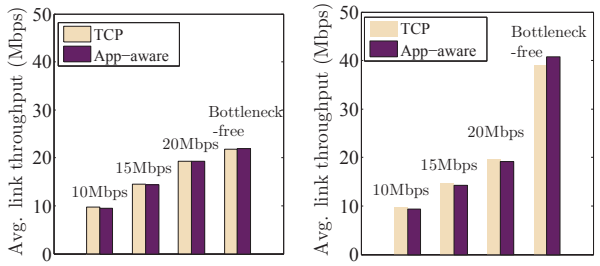
For evaluating the performance of streaming application, we use the widely used metrics of interest by streaming frameworks, *application throughput* and *average end-to-end latency*. Application throughput is the average number of successfully processed tuples per unit time by the sink operator of streaming application, while average end-to-end latency is the average time taken over all tuples from the point each tuple leaves the source until it gets completely processed by sink operator.

Application Throughput: Figure 6 contrasts throughput of TT and TI based on App-aware versus TCP over each of 10Mbps, 15Mbps and 20Mbps settings. The experimental results confirm that App-aware outperforms TCP by 25%, 27%, and 15% in TT, and by 30.93%, 30.27%, and 30.80% in TI.

End-to-end Latency: Figure 7 contrasts average end-to-end latency of TT and TI based on App-aware versus TCP over each of 10Mbps, 15Mbps and 20Mbps settings. The experimental results confirm that App-aware outperforms TCP by 14.27%, 26.24%, and 50.17% in TT and by 5.72%, 10.10%, and 17.28% in TI.

In link bottleneck setting, the improvement in application throughput and end-to-end latency can be interpreted by the fact that App-aware strives to to allocate bandwidth to the flows proportionally to their importance in application performance rather than based on bandwidth fairness as what TCP does. In TI, each combiner instance requires existence of data tuples from both source instances. Therefore, in TCP

large data tuples flows often get throttled by some other very frequent small data tuples flows which leads to processing stall of the combiner. In contrary, App-aware relies on flow metrics and then smartly allocates the flows proportional bandwidth, which in turns alleviates instance's stalled time and speeds up needed data tuples arrival. For TT, key-based grouping along with accumulating top K word at the WCT instances create flows with unbalanced sizes. Those flows are required by TopK aggregator to decide the final Top K. Hence, TCP falls short to express such imbalance, while App-aware captures such imbalance and allocates the proper bandwidth which greatly helps application achieves better performance. In bottlenck-free setting, for both applications, the performance of App-aware is much similar to TCP and sometimes it performs better.



(a) Trending Topics (TT) (b) Trucking IoT (TI)

Figure 8: Link Utilization.

C. Link Utilization

Besides, link utilization is an important property for any bandwidth allocation algorithm to use the entire available bandwidth. To evaluate network utilization due to dynamic bandwidth allocation by our model, we use *average link throughput*, the average of aggregate throughput over all bottlenecked links in the cluster.

Though TCP congestion mechanism is application-agnostic, it utilizes bottleneck links very well. Figure 8 contrasts average link throughput of TT and TI as results of bandwidth allocation based on App-aware and TCP model. App-aware provides an average of 99% and 97% link utilization much the same as TCP link utilization for TT and TI, respectively. The model utilizes all available bandwidth and in case some bandwidth remains, the model performs backfilling pass to allocate the remaining bandwidth among the flows according to their proportional share in the previous pass.

D. Overhead of bandwidth optimizer and enforcer

The most important components that influence solution decision of the whole framework are computation and communication overhead of bandwidth optimizer and enforcer, respectively. To evaluate the computation overhead of bandwidth optimizer, we report CPU time at time step of new allocation in Trucking IoT application running for 600 seconds. We observed that optimizer took 6 milliseconds on average to extract flow statistics and to calculate the optimal bandwidth allocation. Further, to evaluate the communication overhead of bandwidth enforcer, we log the completion time of each flow rate update at one time step of the new allocation. We observed that average time for the controller to completely update the switch with new flow's rate (i.e., meter table update) was 200 microseconds. As a result, this is negligible and therefore our model is able to cope with dynamic changes during application optimization.

VII. Related Work

Scheduling and Management in Datacenter There has been a plethora of recent work on scheduling and management of tasks of datacenter applications with various performance objectives. Most of them aim to minimize completion time of flows belong to user-facing applications (e.g., web search) [19][26], while others aim to minimize the completion time of data-intensive jobs (known as coflows, e.g., MapReduce jobs) as whole [23][24][28]. In principle, the basis behind introducing these solutions is driven based on the intuition of segregation (co)flows into short and long (co)flows, and then schedule each of them, broadly speaking, on a SJF basis. However, this kind of classification can not be directly applied for those applications do not have transfer boundary (e.g., streaming applications). On contrary, one aspect of our model design is addressed to overcome this limitation. It has the ability to inform the state of the flow based on its current and historic attributes regardless of its length.

SDN-based Traffic Management Much like typical programming languages offer APIs to manage workload of computation resources, SDN interestingly offers APIs (e.g., OpenFlow) to manage workload of networking resources. Hence, SDN-based traffic management [29] has been adopted to enable efficient and dynamic management of network resources in datacenters during runtime of the applications. Hedera [30] and MicroTE [31] manage network flows using centralized network-wide scheduler in order to increase network throughput. To our knowledge, no recent work integrate SDN with streaming analytics to enable network responsiveness to the application-level performance requirements.

Resources Auto-Scaling in Stream Applications DRS [10] has been introduced to schedule and provision computation resources to meet a real-time constraints. On contrary, we address I/O-bound stream applications and introduce a bandwidth scaling model that is able to dynamically increase or decrease bandwidth allocation on a performance-centric basis of the application. To our knowledge, no recent work addresses network bandwidth allocation matter in streaming applications.

Traffic-aware Placement in Stream Applications Several proposals [8][9][32] have been proposed to avoid network transfers as far as possible. They aim primarily to collocate application instances in a few machines in order to minimize inter-machine communication. However, while these solutions to some extent improve the performance of the application, it is inevitable to distribute the applications into many more machines to not overload the CPUs of particular machines. Nonetheless, optimization of bandwidth allocation is orthogonal to traffic-ware solutions.

VIII. Conclusion

As we have observed that TCP-based bandwidth allocation of bottleneck links largely hurts application-level performance of streaming applications. As opposed to TCP, we have introduced a novel bandwidth allocation model that performs well with awareness of the application layer performance requirements. To make the proposed model more practical, we have developed a cross-layer SDN-based framework which utilizes smartly backlog information obtained from the application layer and provides on-the-fly and dynamic bandwidth allocation during the runtime of the streaming applications.

We have thoroughly investigated the performance of proposed solution through a series of real testbed experiments with real stream analytics. The results indicate that applications performance resulted from our solution outperforms the standard TCP-based bandwidth allocation employed by most of streaming frameworks, with negligible overhead. It also performs comparable to TCP in terms of network utilization.

We believe that proposed model can be used not only by streaming analytics, but it can also be employed by several platforms involving parallel network flows.

Until now, our focus was only on the performance improvement of individual real-time streaming application. Not surprisingly, we intend to develop a performance model to regulate sharing of cluster resources among multiple streaming applications with different performance objectives, such as IoT analytics applications run at Fog's network edge.

Acknowledgment

This work was funded by the research grant from Singapore Agency for Science, Technology and Research (A*STAR) through the SINGA program.

References

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proc. of ACM SIGMOD*, 2014, pp. 147–156.

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

[3] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[5] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[6] J. Cho, H. Chang, S. Mukherjee, T. Lakshman, and J. Van der Merwe, "Typhoon: An sdn enhanced real-time big data streaming framework," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2017, pp. 310–322.

[7] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media Inc., 2017.

[8] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, 2014, pp. 535–544.

[9] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[10] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: auto-scaling for real-time stream analytics," *IEEE/ACM Transactions on Networking (TON).*, vol. 25, no. 6, pp. 3338–3352, 2017.

[11] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review (CCR)*, vol. 40, no. 4, 2010, pp. 63–74.

[13] OpenDaylight controller, https://wiki.opendaylight.org/view/Main_Page.

[14] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. Manning Publications Co., 2015.

[15] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 149–161.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[18] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *ACM SIGCOMM computer communication review*, vol. 45, no. 4. ACM, 2015, pp. 183–197.

[19] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013, pp. 435–446.

[20] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[21] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 13–24.

[22] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 187–198, 2012.

[23] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 443–454.

[24] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 393–406.

[25] W. A. Aljoby, T. Z. J. Fu, and R. T. B. Ma, "Impacts of task placement and bandwidth allocation on stream analytics," in *Proceedings of International Workshop on Networking BigData Oriented System (NetBOS)*, 2017, pp. 1–6.

[26] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian, "Information-agnostic flow scheduling for commodity data centers." in *NSDI*, 2015, pp. 455–468.

[27] StormCV, https://github.com/sensorstorm/StormCV.

[28] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[29] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[30] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, 2010, pp. 19–19.

[31] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedingsnetwork of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[32] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.