Impacts of Task Placement and Bandwidth Allocation on Stream Analytics

Walid A. Y. Aljoby*†, Tom Z. J. Fu†‡, Richard T. B. Ma*†
* School of Computing, National University of Singapore
† Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd.
‡ School of Computers, Guangdong University of Technology
Email: algobi@comp.nus.edu.sg, tom.fu@adsc.com.sg, tbma@comp.nus.edu.sg

***Abstract*—We consider data intensive cloud-based stream analytics where data transmission through the underlying communication network is the cause of the performance bottleneck. Two key inter-related problems are investigated: task placement and bandwidth allocation. We seek to answer the following questions. How does task placement make impact on the application-level throughput? Does a careful bandwidth allocation among data flows traversing a bottleneck link results in better performance? In this paper, we address these questions by conducting measurement-driven analysis in a SDN-enabled computer cluster running stream processing applications on top of Apache Storm. The results reveal (i) how tasks are assigned to computing nodes make large difference in application level performance; (ii) under certain task placement, a proper bandwidth allocation helps further improve the performance as compared to the default TCP mechanism; and (iii) task placement and bandwidth allocation are collaboratively making effects in overall performance.**

I. Introduction

The widespread of today's big data stream analytics impose different requirements to the batch-based ones. They continuously receive fast data streams generated by one or multiple producers (e.g., sensor readings, video feeds, micro-blogs, server logs, etc) and act on such transient data on-the-fly to extract insightful information in real time. Hence, sophisticated cloud-based distributed stream processing engines tailored for real time stream analytics have been increasingly developed such as Apache Storm [1], Apache Samza [2], Apache Flink [3], Twitter Heron [4], Stylus [5], and so on.

We focus on data intensive applications where computation is rarely the bottleneck but data transmission is quite often leading to performance bottleneck. To what extent that data transmission degrades the overall performance of the stream data analytics applications, such as throughput, latency and so on, is largely affected by two key factors, i.e. task [1] placement and data flow bandwidth allocation.

Task placement in a distributed stream processing engine is the assignment of the tasks (application logical level) to the underlying computing nodes (physical level). Having two tasks co-located at the same node benefits the data transmission between them, e.g., higher throughput and lower latency. Nevertheless, it increases the probability of the contention of other physical resources, e.g., CPU and memory. On the other hand, assigning each task to an individual node avoids resource contention and is easy for scaling as well, however, it incurs transmission cost, which grows rapidly as the number of tasks becomes large. Therefore, optimizing task placement is challenging and an open ended problem. A large amount of previous works studied this problem from the perspective of computing resource contention, but very few studies took the data transmission cost into consideration.

Given a certain task placement, the associations between data transmission flows and the underlying physical links, e.g., a mapping from each individual flow to a subset of the physical links that it traverses, are also determined. When multiple flows traverse the same physical link of which the available bandwidth is insufficient for the aggregated bandwidth demands of all these flows, it leads to the question of how to effectively allocate the bandwidth of such bottleneck link among all its associated flows so as to maximize the overall network utility or application level performance metrics.

The well-known TCP congestion control mechanism [6] is effective in guaranteeing the fairness among all the flows traversing the same bottleneck link, however, it does not alway meet the requirement from the application's perspective. For example, flows of financial applications can be more important than flows delivering the video contents. Of course, this is not a new concept as the Differentiated services (DiffServ) has been studied widely for many years. Until recently, software defined network (SDN), particularly the open-flow related technique, sheds some light on the possibility of realizing the DiffServ in practice. The open-flow controllers together with open-flow enabled switches enables the developers to regulate the maximum bandwidth usage of each individual flow that traverses a physical link.

To reach the ultimate goal of optimizing the overall performance of the data intensive stream analytics applications from the network's perspective, it is necessary, as a first step, to have a comprehensive understanding of how the two key factors are taking effects both individually and collectively. In this paper, we have conducted a measurement-based study on top of a typical fork-join stream application and made the following observations:

- Without additional bandwidth allocation performed by openflow controllers, how tasks are assigned to the

[1]For the ease of presentation, we use task, processor and operator interchangeably in this paper.

978-1-5090-6501-1/17/$31.00

computing nodes makes noticeable difference in both tuple processing throughput and tuple complete latency of the fork-join application.
- Given a certain task placement, using openflow to properly allocate bandwidth among flows sharing the bottleneck links can produce better application level performance than the default TCP mechanism.
- Both task placement and bandwidth allocation collaboratively affect the overall application level performance.

The rest of this paper is organized as follows. Section II gives the definitions of important concepts and terms such as distributed stream processing frameworks, the task placement and bandwidth allocation problems, and open-flow controller. In Section III, we introduce both the experimental settings and the measurement results running on a typical fork-join stream application over Apache Storm. We discuss the related work in Section IV and finally we make our conclusion in section V.

II. Background

In this section, we introduce the notations and provide definitions on important concepts and problems.

A. Distributed stream data processing

Distributed stream data processing frameworks such as Apache Storm [1], Apache Samza [2], Apache Flink [3], Twitter Heron [4], etc, provide programming paradigms that cope with application requirements by acting on fast and unbounded streams, so that analysis results, in the form of answering a pre-registered continuous query, can be generated as fast as possible on every arrival of new data tuple to the system. The programming paradigm characterizes the data flow computation model for stream analytics applications, which is typically called a topology. For instance, a topology in Apache Storm can be represented as a directed graph where vertices denote the operators equipped with the user defined functions (UDF) and edges denote the data processing sequences along the operators. One special type of operators, called *spout*, act as the data source of the topology, connecting to the external data producers such as video streams recorded by surveillance cameras, measurement results by sensors, event logs by monitoring systems, tweets by Twitter APIs, and etc. Each operator can comprise one or multiple tasks running the same copy of the UDF in parallel.

B. Task placement problem

In a running topology, all tasks of which must be deployed across the worker nodes in a computer cluster. We refer to such an assignment of the tasks to the computing nodes as *task placement*. Formally, we use $\mathcal{T} = \{t_1, t_2, \ldots, t_T\}$ to denote the set of tasks and $\mathcal{N} = \{n_1, n_2, \ldots, n_N\}$ the set of computing nodes, where T and N represent the number of tasks and the number of computing nodes, respectively.

Assuming that tasks are the minimum data processing units which are indivisible, a task placement (TP) problem is to determine a $T \times N$ binary valued matrix, denoted by P, where entry $P_{i,j} = 1$ if task t_i is placed at computing node n_j, otherwise, $P_{i,j} = 0$, satisfying the constraint:

$$\forall\, t_i \in \mathcal{T},\ \sum_{j=1}^{N} P_{i,j} = 1.$$

Note there is no constraint on computing nodes, thus they can either be idle, or host any number of tasks, i.e., $\forall n_j \in \mathcal{N}, \sum_{i=1}^{T} P_{i,j} \in [0, T]$.

C. Bandwidth allocation problem

The underlying network which physically connects to all the computing nodes can be represented as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The vertices, $\mathcal{V} = \mathcal{N} \cup \mathcal{D}$, are the union of all the computing nodes and network devices, e.g. switches and routers. We use $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ to denote the set of network devices and D represents its size. The edges $\mathcal{E} = \{e_1, e_2, \ldots, e_E\}$ are the physical links connecting either a pair of a computing node and a network device or a pair of two network devices. For example, a uni-directional link from node n_j to device d_k is in the form of $e_l = (n_j, d_k)$. We use the $E \times 1$ vector $\mathbf{b} = (b_1, b_2, \ldots, b_E)$ to denote the bandwidth of each link $e_l \in \mathcal{E}$.

Next we refer to flows as the uni-directional data transmission from a source task to a destination task. We use $\mathcal{F} = \{f_1, f_2, \ldots, f_F\}$ and F to denote the flow set and the total number of flows, respectively. A flow is called an internal flow when both its source and destination tasks are deployed at the same computing nodes; otherwise we call it an external flow. Since there are no direct physical links between any two of the computing nodes, each external flow must traverse at least two uni-directional links belonging to set $\mathcal{E}$.

The flow traversal relationships are represented by a $F \times E$ binary matrix, denoted by Q, whose element $Q_{h,l}$ indicates whether the flow f_h traverses through the link e_l. f_h belongs to an internal flow only if $\sum_{l=1}^{E} Q_{h,l} = 0$; otherwise, an external flow with $\sum_{l=1}^{E} Q_{h,l} \geq 2$.

A bandwidth allocation problem is to find a rate vector $\mathbf{x} = (x_1, x_2, \ldots, x_F)$, where $x_h, h = 1, 2, \ldots, F$ is the maximum transmission rate that flow $f_h \in \mathcal{F}$ can reach, satisfying the flowing two constraints:

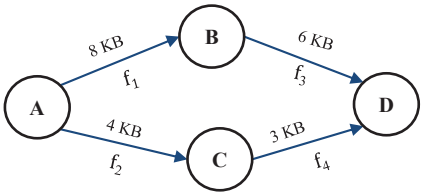
(i) $\forall\, e_l \in \mathcal{E},\ \sum_{h=1}^{F} Q_{h,l} \cdot x_h \leq b_l$ (bandwidth limitation);

(ii) $x_h > 0, h = 1, 2, \ldots, F$. (flow rate feasibility)

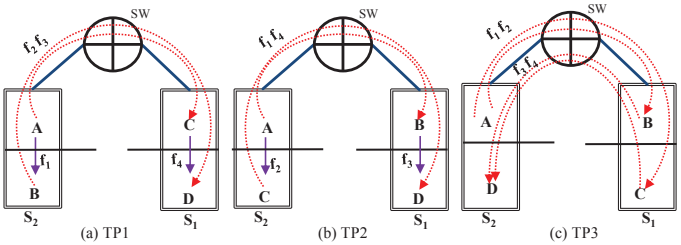
Finally, we apply SDN related tools, which will be introduced next, to handle the remaining job which is to implement the derived flow rate vector $\mathbf{x}$ into the running system.

D. SDN and open-flow controllers

SDN has been introduced as an architecture that promises a flexible and responsive network to meet the requirements from both the applications and the users [7]. It is a networking paradigm that decouples the control decision from the underlying network devices such as switches and routers, and delegates a logically centralized controller to be in charge of the network control, so the devices turn into simplified forwarding units (data plane), while the network controller is appointed



(a) Topology of a synthetic video analytics application with tuple sizes (above arrow) and flows (below arrow).



(b) Possible task placements. Purple arrows inside the servers denote the internal flows using inter-process communication between the tasks inside the same machine, while dotted red arrows denote the external flows that go across the network switch.

Fig. 1: A typical topology for streaming analytics applications and possible settings for task placement

as the container of the network control logic (control plane). Such decoupling enables network operators and application developers to abstract the lower-level functionalities of the network by means of a set of well-defined APIs such as the OpenFlow API, through which it becomes easier to manage the network services and resources as flexible as possible. In our studies, we use commercial OpenFlow switch ICX-6610 from Brocade vendor with OpenDaylight [8] controller which collectively offer a flexible architecture for managing the bandwidth involved in stream analytics. We build a bandwidth management module on top of OpenDaylight, simply combining the flows of interest with the specific Meter APIs. The metering API is a feature supported by OpenFlow which enables a rate limiting strategy at ingress ports of OpenFlow switch. Through a meter API, each flow can be assigned a predefined upper-bound rate, under which the packets belonging to the specific flows are allowed to pass through the egress ports of the switch; otherwise the rate limiter drops the packets of those flows that exceed the upper-bounds. This summaries how we utilize the metering APIs to realizing the bandwidth allocations among flows in the real running applications.

III. MEASURMENT-DRIVEN ANALYSIS

In this section, we first introduce a typical stream join application implemented atop Apache Storm. Next, we describe the experiment settings and performance metrics. Finally, we present the experiment results and our observations.

A. A representative example

We implemented the fork-join topology as a representative of the stream data analytics application and a case study in this paper [9][10]. It is a synthetic video-analytics application that involves transfer of video frames (*i.e.,* continuous tuples). The topology is depicted in Fig. 1a, which consists of 4 tasks, $\mathcal{T} = \{A, B, C, D\}$. Task A is the source that continuously transmits in parallel two different streams of frame tuples with sizes 8 KB to task B and 4 KB to task C, respectively. Tasks B and C then construct frames in sizes 6 KB and logos in size 3 KB, separately, and emit them to the aggregation task D. In the final stage of the analytics process, task D joins a frame tuple and its corresponding logo tuple, and produces the output results.

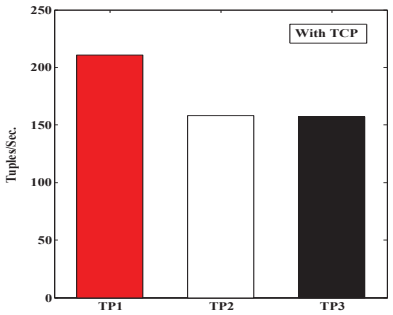


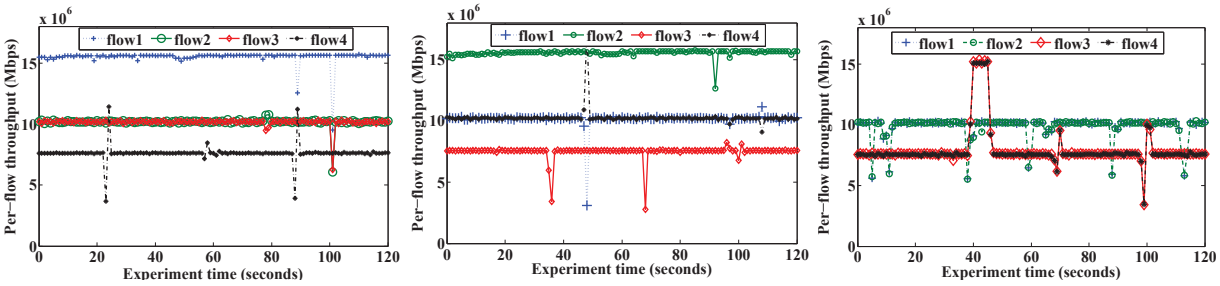
Fig. 2: Average tuple complete throughput under three task placements during the 300 seconds experiment period of the running topology shown in Fig. 1a

B. Experiment settings and performance metrics

We built a testbed that consists of 3 physical machines connected to a Brocade ICX-6610 24-port Gigabit switch that supports both Layer 2/3 networking and OpenFlow-based communication. Each machine is equipped with a 4-core Intel 2.8GHz CPU, 6 GB memory, and a Gigabit Ethernet NIC. The nodes run 16.04 Ubuntu Linux. One physical machine is dedicated to host the Nimbus (i.e., the master daemon) of the Apache Storm, the Zookeeper coordinator, and the OpenDaylight controller. The other two machines are the worker nodes hosting the running tasks, as shown in Fig. 1b. Thus, we have $\mathcal{N} = \{S_1, S_2\}$, $\mathcal{D} = \{SW\}$ and $\mathcal{E} = \{e_1, e_2, e_3, e_4\}$. In order to emulate the contention scenario, we set all the link bandwidth equal to 20 Mbps, i.e., $b_l = 20$ Mbps, $l = 1, 2, 3, 4$. In our study, we refer to the application level throughput, i.e., the number of tuples successfully processed by task D per second, as the essential performance metric.

C. Experiment results on different task placements

For load balance purpose, we only focus on the cases where each computing node hosts exactly two tasks. Due to the symmetry, there are total three distinct task placements, as shown in Fig. 1b.



(a) TP1 (b) TP2 (c) TP3

Fig. 3: Per-flow throughput on multiple task placements. Each flow corresponds to data transmission between two tasks as in Fig. 1a, flow1 (A,B), flow2 (A,C), flow3 (B,D) and flow4 (C,D)

Fig. 2 shows the application throughput under three task placements and Fig. 3 shows the throughput of the internal and external flows corresponding to each placement. As shown in Fig. 3, 4 flows are involved for each placement. In TP1 and TP2, two of the flows are internal flows and the other two external. In contrast, all flows in TP3 are external flows[2]. Notably, there are three pairs of flows sharing the bottleneck links. They are f_2 and f_3 in TP1, f_1 and f_4 in TP2, and f_1 and f_2 in TP3. As shown in Fig. 2, TP1 achieves 33.62% higher throughput, compared to both TP2 and TP3. This is because the aggregate and ratio of tuple sizes of bottlenecked flows in TP1 are the smallest amongst all. Specifically, the aggregate tuple sizes of bottlenecked flows in TP1, TP2 and TP3 correspondingly is 10 KB, 11 KB and 12 KB, with ratios between large tuple size and smaller one of 1.5, 2.6 and 2 respectively.

When aggregate sizes are smaller, the link then can transfer higher amount of stream tuples. The ratio is also important because bottlenecked flows get a fair share of the link bandwidth as approximated by TCP. Thus, when the ratio of tuple sizes belonging to bottlenecked flows is smaller, the transfer of tuples gets close to each other, which is crucial for final task in the application that consumes received tuples in a $1:1$ basis. TP1 achieves the smallest aggregate and the closest transfer of data streams among the bottlenecked flows and achieves the highest throughput amongst all placements, 15.1 Mbps. TP2 comes after and then TP3 in respect of aggregate tuple sizes, and vice versa from ratio perspective. However, both TP2 and TP3 achieve a similar throughput, 11.38 Mbps, because both bottlenecked by the same tuple size transferred by task A. Therefore, the overall throughput becomes constrained by the throughput of a tuple with a larger size in the bottlenecked flows. Regardless of aggregate bottlenecked sizes, we have observed that the smaller ratios among streams, the more constructive streams in the overall throughput of the application. To sum up, in order to obtain the optimal task placement for data intensive application where existence of link bottleneck is quite common, we allocate network link with pairs of tasks in the ascending order of their aggregate and ratio of tuple sizes. If two pairs are equivalent in the aggregate of sizes, we choose the pair of smaller ratio. Similarly, if they are equivalent in the ratio, we prefer to allocate the link for the pair with smaller aggregate. However, if the pairs have different aggregate and ratio, then we prefer allocating the link for the pair with smaller aggregate because ratio between the tuple sizes can be adjusted through explicit bandwidth allocation as explained in the next section.

[2]Note in this case, the link shared by f_3 and f_4 is not the bottleneck.

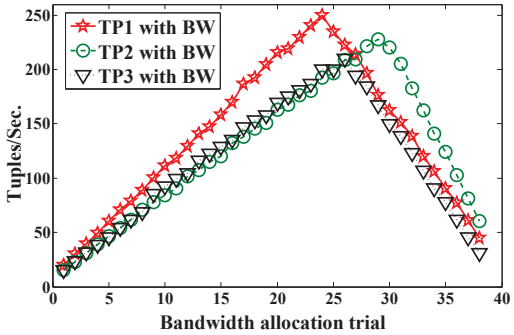
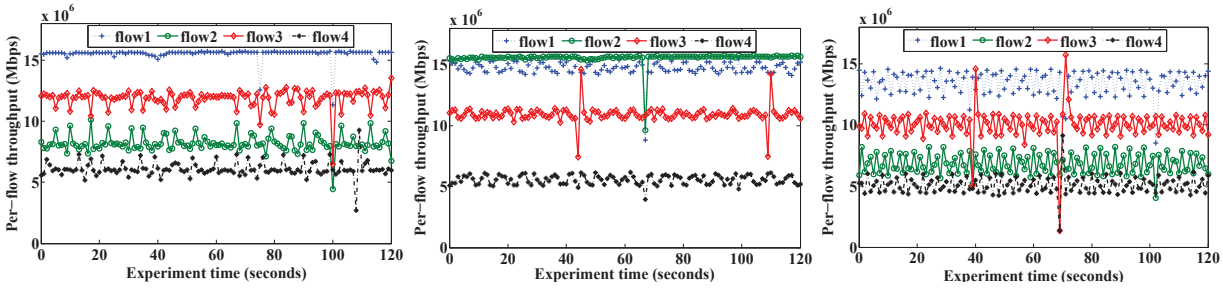


Fig. 4: Search space of finding the proper bandwidth allocation that achieves the maximum throughput over bottlenecked flows. The bandwidth ratio between bottlenecked flows that achieves a max throughput is correlated with tuple sizes ratio. For example, allocation #24 in TP1 with 8.5 Mbps for f_2 and 12.5 Mbps for f_3 is the optimal allocation that is approximately proportional to the ratio of tuple sizes of f_2:f_3 (4KB : 6KB), as depicted in Fig. 1a.

D. Experiments on bandwidth allocation

In the previous section, we tried to figure out effective settings for task placement under bandwidth allocation as instructed by default TCP. However, TCP strives to maximize network throughput while achieving fairness among flows. So it is agnostic to the application-level performance requirement, and hence this mismatch makes TCP far from the optimal allocation and may severely impact on the application-level performance. As explained previously, the task placement is influenced by aggregate and ratio of tuple sizes sharing the bottlenecked link, and the allocation with smaller aggregate



(a) TP1 (b) TP52 (c) TP3

Fig. 5: Per-flow throughput on weighted bandwidth allocations corresponding to task allocations. Each flow corresponds to data transmission between two tasks as in Fig. 1a, flow1 (A,B), flow2 (A,C), flow3 (B,D) and flow4 (C,D)

and ratio achieves better throughput. The ratio is indicated as crucial factor because bottlenecked flows get a fair share of the bandwidth as approximated by TCP. Thus, when the ratio of tuple sizes between competing flows is smaller like in TP1, the transfer of $1:1$ tuples has a higher chance to get close to each other, which is constructive for application interest that requires availability of both tuples at certain stage in the application topology. In contrast, when the ratio of tuple sizes gets larger, the flow with smaller tuple size sends much non-utilized streams as TP2, and then the application does not benefit from such transfer rather a waste of network bandwidth. In either case, it is better to equalize number of transferred tuples.

We tackle this problem by deviating from the default per-flow fairness of TCP and heuristically allocate unequal bandwidth among contending flows to find a good fit allocation that achieves the maximum throughput. Given a link bandwidth b and two bottlenecked flows f_x and f_y sharing the link, we develop an external entity that coordinates stream analytics framework with SDN controller and we delegate an algorithm to this entity that examines a range of $(xbps, ybps)$ pairs over (f_x,f_y) correspondingly, where y is initialized to a stepsize Δ_b and x to b-Δ_b, and iteratively y increases by Δ_b and x decreases by the same stepsize. The algorithm stops when y reaches b-Δ_b and x equals Δ_b. We applied this heuristic algorithm over options of task placement. Fig. 4 shows a search space of weight-based bandwidth apportion among the bottlenecked flows f_2 and f_3 in TP1, f_4 and f_1 in TP2 and f_2 and f_1 in TP3, where $b = 20Mbps$ and $\Delta_b = 0.025b$. Based on the results shown in Fig. 4, the important observation we ascertain is that there exists a unique bandwidth allocation rather than TCP-based that helps each TP to achieve a maximum throughput when it comprises bottlenecked flows. This allocation implies that bandwidth resource should be divided among the flows proportional to the respective tuple size of each flow. For example, TP1 achieves a highest throughput when the ratio of bandwidth assigned for f_2 to f_3 equals 0.68 which is approximately equivalent to ratio of transmitted tuple size by both flows.

Given allocation that achieves highest throughput for each TP, we further analyze internal and external flow pattern

TABLE I: Achieved per-flow throughput for each task placement under wight-based bandwidth allocation

	f_1	f_2	f_3	f_4
TP 1		8.20 Mbps	11.50 Mbps	
TP 2	14 Mbps			5.55 Mbps
TP 3	13 Mbps	6.73 Mbps		

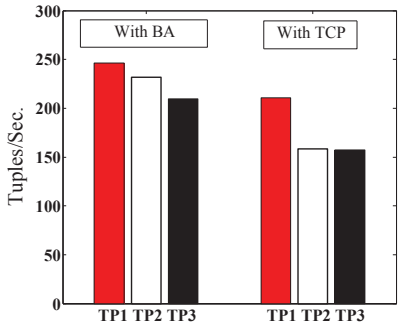


Fig. 6: Average tuple complete throughput under three task placements during the 300 seconds experiment period of the running topology shown in Fig. 1a

resulted this allocation, as depicted in Fig. 5. While the throughput of bottlenecked flows in TP under TCP are equivalent, we note that these flows achieve different throughput with weight-based bandwidth allocation as the share of link is divided unequally. With weight-based bandwidth allocation, the throughput of flows is achieved proportionally, as it can be seen in Table. I.

E. Collaborative effect of task placement and bandwidth allocation

Fig. 6 contrasts the average tuple complete throughput of task placement with TCP and with weight-based bandwidth allocation. The results convey that: 1) With default TCP, there exist some options for task placement that achieve better throughput than others. 2) For task placement based on default TCP, a proper bandwidth allocation can make further throughput improvement. In our case study, weight-based bandwidth allocation improves respectively TP1, TP2 and TP3 by 17%, 47%, and 33%, compared to TCP-based aside. Note that TP2

and TP3 achieve similar throughput based on TCP. So such options confuse the decision of which placement should be selected. However, with weight-based bandwidth allocation, TP2 achieves better throughput than TP3 because its aggregate tuple size is smaller than that of TP3.

Hence, the overall application-level performance, e.g. average tuple complete throughput, is mainly affected collaboratively by the task placement and bandwidth allocation strategies. Investigations on the joint strategy which achieves the optimality will be considered as our future work.

IV. Related Work

Recent research efforts have been emerged toward optimizing the performance of stream-analytics applications such as optimizing allocation of tasks into the physical cluster [11] [9] [12] [13], and computation resources scheduling and provisioning [14]. While in large part successful, some of these solutions are either suboptimal in optimizing network transfer [11], or assuming the network supports sufficient bandwidth resource [14], none of these solutions studies the impact of bandwidth allocation among the application tasks. In contrast, we focus on studying collectively the impact of bandwidth allocation with task placement. From application-level throughput perspective, the cross-layer approach in [12] was proposed to improve the overall throughput of the application by means of collectively determining the placement of tasks along with appropriate network route that best match end-to-end flows in that placement. This approach is orthogonal to our work, where the chosen appropriate network route can be further improved with weight-based bandwidth allocation across the tasks.

With regard to SDN, it has been recently employed to utilize network resources wisely through improving network flows scheduling and routing [7]. Further, Wang et al. [15], address the potential of integrating SDN controller with big-data application tracker to facilitate more informed scheduling and placement decisions. Xiong et al [16] propose SDN-based approach to improve the performance of queries over distributed relational databases. In contrast, we integrate SDN with stream analytics framework to enable a runtime bandwidth allocation mechanism among the application tasks.

V. Conclusion

We studied the impact on the application throughput of the sharing of link bandwidth among tasks comprising distributed stream-analytics application running data-parallel framework in a private cluster, and we focused on fork-join as common transfer pattern on stream analytics. We demonstrate measurement-driven results for two principles influencing the performance of the application: task placement and bandwidth allocation. Multiple options for task placement have been analyzed, where the results indicate that task placement is influenced by aggregate and ratio of tuple sizes sharing the bottlenecked link, such that allocating of network link among the flow in the ascending order of their aggregate and ratio of tuple sizes, achieves a better throughput. As network allocation is as instructed by native TCP, we then demonstrate how each option of task placement can be further improved of a weight-based bandwidth allocation instead of TCP per-flow fair allocation. Given explicit information about tuple sizes, we reveal that adoption of weight-based bandwidth allocation makes task placement decision much easier, where allocating the link with flows is preferred to be according to the ascending order of aggregate tuple sizes, while the ratio is no longer a matter as it can be adjusted through unequal allocation of link bandwidth.

Acknowledgment

This work is supported by Advanced Digital Sciences Center from Singapores A*STAR. The authors would like to thank the reviewers for their helpful remarks. Tom Fu was partially supported by the Natural Science Foundation of China (61702113) and China Postdoctoral Science Foundation (2017M612613). Richard Ma was partially supported by the Ministry of Education of Singapore AcRF under Grant R-252-000-572-112.

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] Apache Samza, http://samza.apache.org/.

[3] 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.

[4] 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.

[5] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proc. ACM SIGMOD*, 2016, pp. 1087–1098.

[6] 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.

[7] 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.

[8] OpenDaylight controller, https://wiki.opendaylight.org/view/Project_list.

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

[10] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: Continuous reliable distributed processing of big data streams." in *NSDI*, 2016, pp. 439–453.

[11] 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.

[12] H. Alkaff, I. Gupta, and L. M. Leslie, "Cross-layer scheduling in cloud systems," in *Proc. IEEE IC2E*, 2015, pp. 236–245.

[13] 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.

[14] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: dynamic resource scheduling for real-time analytics over fast streams," in *Proc. IEEE ICDCS*, 2015, pp. 411–420.

[15] G. Wang, T. Ng, and A. Shaikh, "Programming your network at runtime for big data applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 103–108.

[16] P. Xiong, H. Hacigumus, and J. F. Naughton, "A software-defined networking based approach for performance management of analytical queries on distributed data stores," in *Proc. ACM SIGMOD*, 2014, pp. 955–966.