Hone: Mitigating Stragglers in Distributed Stream Processing With Tuple Scheduling

Wenxin Li, Duowen Liu, Kai Chen, *Senior Member, IEEE*, Keqiu Li, *Senior Member, IEEE*, and Heng Qi

Abstract—Low latency stream processing on large clusters consisting of hundreds to thousands of servers is an increasingly important challenge. A crucial barrier to tackling this challenge is *stragglers*, i.e., tasks that are significantly straggling behind others in processing the stream data. However, prior straggler mitigation solutions have significant limitations. They balance streaming workloads among tasks but may incur imbalanced backlogs when the workloads exhibit variance, causing stragglers as well. Fortunately, we observe that carefully scheduling the outgoing tuples of different tasks can yield benefits for balancing backlogs, and thus avoids stragglers. To this end, we present *Hone*, a tuple scheduler that aims to minimize the maximum queue backlog of all tasks over time. *Hone* leverages an online *Largest-Backlog-First* (*LBF*) algorithm with a provable good competitive ratio to perform efficient tuple scheduling. We have implemented *Hone* based on Apache Storm and evaluated it extensively via both simulations and testbed experiments. Our results show that under the same workload balancing strategy–*shuffle grouping*, *Hone* outperforms the original Storm significantly, with the end-to-end tuple processing latency reduced by 78.7 percent on average.

Index Terms—Distributed stream processing, tuple scheduling, straggler task, backlog balancing

1 Introduction

Stream processing has recently witnessed an increasing wave of popularity across many application domains, including real-time object recognition [1], internet quality of experience prediction [2], and online social network analysis [3]. A common denominator of these applications is that the stream data is on a significant rise. As a result, leading IT companies (e.g., Google [4], Facebook [5], and Yahoo! [6]) have begun to run distributed stream processing (DSP) jobs in production clusters. As both clusters and streaming pipelines continue to grow in size and complexity, providing low latency for stream processing is the fundamental challenge to a DSP job.

DSP systems such as Storm [7] and Flink [8] represent a job as a directed acyclic graph (DAG) of operators, with each operator often being executed by many parallel tasks [9]. One crucial roadblock to the job performance is *stragglers*, i.e., the tasks that take significantly longer than others to process the stream data. At this point, past work has proposed several straggler mitigation solutions by balancing the workload (i.e., the incoming tuples[1]) assigned to each task [10], [11], [12], [13].

However, these solutions may not perform as expected. The crux is that the workloads can exhibit high variability [14], [15], [16], implying that different tuples will trigger different amounts of outputs under the same operation. For instance, a `SplitSentence` task will output two tuples for ⟨`1, Hello World`⟩ and one tuple for ⟨`2, Hello`⟩. In this case, even when tasks have an equal number of incoming tuples, they will produce different amounts of outgoing tuples. Meanwhile, the number of outgoing tuples could be enormous as compared to the scarce network resources [17], [18], [19]. Consequently, some outgoing tuples necessarily have to be backlogged, and the backlogs can vary significantly across different tasks. Further, the lack of transmission control on the outgoing tuples in mainstream DSP systems (e.g., Storm [7] and Flink [8]) could exacerbate the difference in the backlogs (see Section 2 for details). As a result, stragglers still exist.

As an example, Fig. 1 further demonstrates that the task `T1` may still be straggling behind `T2`, though they are guaranteed to get an equal number of incoming tuples with an excellent workload balancing strategy–*shuffle grouping* [7]. In this case, the upstream operator `O1` will be forced to slow down its processing speed under backpressure effect. Meanwhile, the downstream operator `O3` may be suspended to wait for the complete intermediate results from `O2`. By carefully scheduling the outgoing tuples of `T1` and `T2`, it is undoubtedly feasible to reduce or even eliminate the backlog imbalance between them, thus mitigating stragglers.

In this paper, we focus on the problem of leveraging tuple scheduling to mitigate the straggler tasks in a DSP job,

- *Wenxin Li, Duowen Liu, and Kai Chen are with the iSING Laboratory, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. E-mail: toliwenxin@gmail.com, dliual@connect.ust.hk, kaichen@cse.ust.hk.*
- *Keqiu Li is with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin 300350, China. E-mail: keqiu@tju.edu.cn.*
- *Heng Qi is with the School of Computer Science and Technology, Dalian University of Technology, Dalian 116023, China. E-mail: hengqi@dlut.edu.cn.*

Manuscript received 3 Jan. 2020; revised 12 Oct. 2020; accepted 30 Dec. 2020. Date of publication 12 Jan. 2021; date of current version 19 Feb. 2021. (Corresponding authors: Kai Chen and Keqiu Li.) Recommended for acceptance by S. Pallickara. Digital Object Identifier no. 10.1109/TPDS.2021.3051059

1. A tuple is an ordered list of values, e.g., a ⟨`Key, Value`⟩ pair.

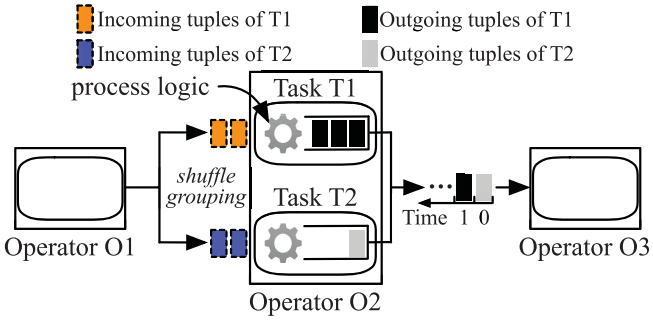


Fig. 1. An illustrating example, where there is a DSP job that has three operators, i.e., O1, O2, and O3. With *shuffle grouping* [7], the tasks T1 and T2 of O2 are guaranteed to have an equal number of incoming tuples. Assume that T1 produces two times the number of outgoing tuples as compared to T2 due to the incoming tuple variance. In this case, T1 could be straggling behind T2 when it backlogs more tuples.

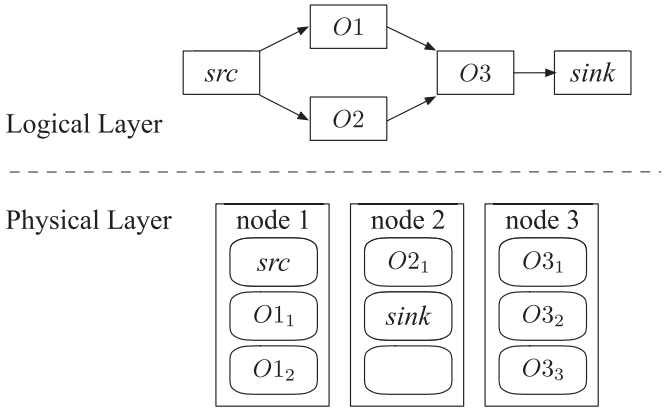


Fig. 2. The logical and physical views of a DSP job.

with the primary goal of achieving low latency stream processing. To achieve this goal, we have designed and implemented *Hone*, an efficient and practical straggler mitigation solution for DSP jobs. At the heart of Hone is an online scheduler to schedule the outgoing data tuples of all the tasks belonging to the same operator. In this scheduler design, we advocate balancing the backlogs by explicitly formulating an optimization problem of minimizing the maximum queue backlog of all tasks across all time slots. We solve this problem by making no assumptions about the *prior* knowledge of future tuple arrivals. Specifically, we propose a *Largest-Backlog-First* (*LBF*) heuristic that greedily schedules a task for tuple transmission based on its queue backlog. With the results from rigorous theoretical analysis, we demonstrate that the proposed *LBF* heuristic can achieve a $3 + \lceil \log_2 N \rceil$-competitive ratio, where N is the number of parallel tasks. We have also conducted extensive simulations to demonstrate the effectiveness of the *LBF* heuristic in minimizing the maximum queue backlog. The results show that *LBF* reduces the maximum queue backlog by up to 83.3 percent, compared to a round-robin heuristic.

To further demonstrate that it is amenable to practical implementations, we have implemented *Hone* based on Apache Storm. In our implementation, instead of overriding the default tuple transmission behavior in the Storm framework, we take a non-intrusive approach by adding a scheduling signal to each task. As a result, the tuple scheduling decisions determined by the proposed *LBF* heuristic can be carried out by controlling the signal of each task only. We proceed to quantitatively evaluate the performance of *Hone* on a small-scale testbed with a streaming application based on a real-world dataset. The experimental results demonstrate that under the same workload balancing strategy—*shuffle grouping*, *Hone* can reduce the end-to-end tuple processing latency by 78.7 percent on average, compared to the original Storm.

In summary, the main highlights of this paper include:

- We study the problem of scheduling the outgoing tuples of the tasks in a DSP job to reduce the stream processing latency.
- We present a practical solution *Hone*, which reduces the latency by minimizing the maximum queue backlogs of parallel tasks with an efficient and online approximation algorithm *LBF*.
- We implement *Hone* in Storm. The required modification for our implementation is non-intrusive and incremental to Storm framework.
- We conduct both simulations and testbed experiments to evaluate *Hone*. The results have demonstrated the superior performance of *Hone* in reducing the stream processing latency.

The rest of this paper is organized as follows. In Section 2, we present the background of DSP and show the significance of tuple scheduling. Section 3 presents an overview of *Hone*. In Section 4, we present the details about the tuple scheduling in *Hone*. We show the implementation details of *Hone* in Section 5. We present the experiment results in Section 6. Section 8 briefly discusses the related work and Section 9 concludes this paper.

2 Preliminaries

In this section, we briefly overview the background of DSP and elaborate tuple scheduling in DSP.

DSP Job. As aforementioned, a DSP job is often interpreted as a DAG, with nodes representing the operators and edges being the data flows between operators. Each DAG consists of three types of operators. The first type is *source* operator that pulls input data stream from external sources (e.g., Kafka) in the form of *tuples*. The second type is *intermediate* operator, which is a logic processing unit that applies a user-defined function to map an incoming tuple from upstream operator to a group of outgoing tuples for downstream operators. The last type is *sink* operator used to spew output tuples to display the job query results at final destinations.

DSP Cluster. A DSP cluster consists of a set of physical machines (called *worker nodes*). Each node has a limited number of slots, with each slot being used for one task instance. Once a job is submitted to the cluster, the DSP system executes it with a set of tasks distributed over the worker nodes. Fig. 2 illustrates the logical and physical views of a DSP job running on a three-node cluster. This job contains a source, a sink, and three intermediate operators. Operators O1, O2 and O3 are executed with two, one, and three tasks respectively.

DSP Tuple Transfer. The DSP system triggers inter-operator communication for streaming data tuples between the tasks of different operators. If connecting operators reside

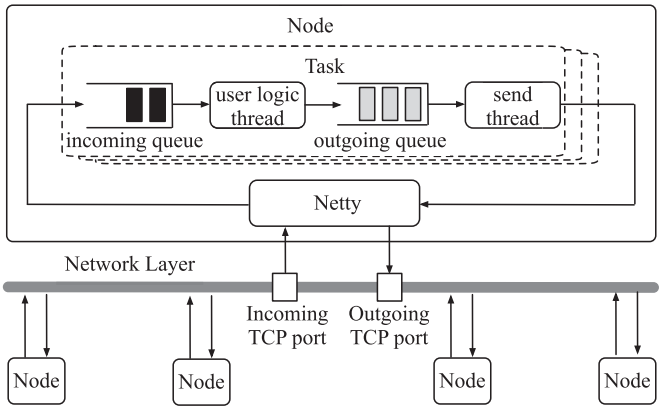


Fig. 3. The illustration of the procedure of outgoing tuple transfer in DSP.

in the same node, there exists only intra-node communication; otherwise, the communication occurs across nodes over network. The inter-node communication may make a stream processing application to involve higher network resource demands than CPU cycles [19], [20], as many applications nowadays are data-intensive [21]. Moreover, tuple transfers across the network incur high latency. In fact, a recent study has pointed out that such tuple transfers can account for 86 percent of the total latency of a DSP application [18]. As we can see from the facts above, inter-node communication is more likely to be the cause of the performance bottleneck rather than intra-node communication, and accordingly, is the main focus of this paper. One may question that such inter-node communication can be eliminated by deploying connecting operators on the same node. However, common operators such as `union`, `shuffle`, and `join` may require data to be transmitted across nodes since their inputs are from different nodes [22].

Mainstream DSP systems (e.g., Storm, Flink) mostly resort to Netty[2] for the inter-node communication described above. To be particular, in their design, each task is configured with an outgoing queue as well as an incoming queue. The outgoing tuples of a task are buffered at its outgoing queue while the incoming tuples are at the incoming queue. As shown in Fig. 3, each task has a user logic thread, which continuously fetches tuples from its incoming queue for processing. For each tuple in the incoming queue, the user logic thread will generate one or more tuples which will then be placed into the outgoing queue. To manage the outgoing tuples, each task uses a send thread to read tuples from its outgoing queue and push them to Netty for transmission over the network to downstream operators. On the other hand, Netty will listen on the binding TCP port for reading tuples from the network and placing them into the incoming queue of the relevant task.

DSP Tuple Scheduling. It is common that DSP systems usually attain high processing efficiency for a job by replicating its operators on multiple tasks [9]. Meanwhile, each task processes a subset of the incoming tuple stream to its operator and outputs an unbounded sequence of tuples to its downstream tasks. The tasks running significantly slower than others become "*stragglers*". These tasks have a significant negative impact on job performance. Balancing the input tuple stream among the tasks of an operator could help to mitigate such impact, which, however, is insufficient. The reason is that the input tuples of different tasks can have significant variance [14], [15], [16], resulting in imbalanced queue backlogs of outgoing tuples between these tasks and causing stragglers as well.

Scheduling the outgoing tuples of the tasks belonging to the same operator can help to balance the queue backlogs, and thus becomes critical for a DSP job, especially when these tasks reside in the same node. Because in this case, substantial outgoing tuples from these tasks will share the same link that has relatively limited bandwidth [17], [18], [19], increasing the effect of tuple scheduling for low latency DSP. In fact, placing the tasks of an operator in the same node is inevitable, because there are usually many DSP jobs [24] and the number of the tasks from all jobs is apparently larger than the cluster size.

Mainstream DSP systems, e.g., Storm [7] and Flink [8], are unaware of exploring tuple scheduling to balance queue backlogs and mitigate stragglers. More precisely, once tuples enter the outgoing queues of corresponding tasks, they will be directly flushed into Netty where the arriving tuples flow in a FIFO order. With FIFO, a data tuple may suffer from arbitrary long delay in Netty. Meanwhile, flushing tuples blindly may exacerbate the queue backlog imbalance for tasks in downstream operators. It may even create a cascading effect and lead to imbalanced backlogs for tasks in upstream operators when backpressure is enabled. Hence, we are inspired to design a tuple scheduler to determine the order in which the outgoing tuples of different tasks are moved to Netty. In this scheduler, we advocate making the maximum queue backlog of the tasks belonging to the same operator as small as possible, to avoid any task straggling behind.

3 Hone Overview

Hone is an online tuple scheduler applied to each worker node to optimize the DSP job performance. In this section, we present a brief overview of Hone to help the reader follow the analysis and design of our online tuple scheduling algorithm and implementation details.

3.1 Problem Statement

When improving the performance of a DSP job, given multiple tasks that run on the same worker node and execute the same operator for this job, *Hone* must decide to select *which* task's queue to output a data tuple each time. It can balance the queue backlog of data tuples among all the relevant tasks and thus achieve efficient straggler mitigation. We mainly focus on the following metric for improving DSP job performance:

Definition 1 (Latency). *The latency of a DSP job refers to the* average end-to-end tuple processing latency over a window of time, *where the end-to-end processing latency of a tuple is the time between it entering the DAG from the source and producing an output result on any sink.* Hone *must provide low latency for a DSP job.*

2. Netty [23] is a non-blocking I/O client-server framework popular for inter-operator communication in mainstream DSP systems (e.g., Storm and Flink).

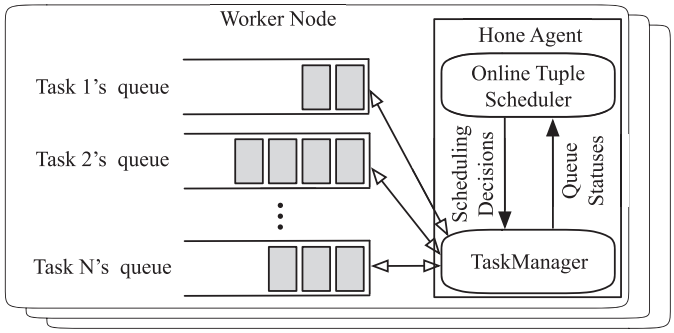


Fig. 4. The overview of Hone.

3.2 Design Overview

We design *Hone* as an agent or daemon, and run it on each worker node in the DSP cluster. As shown in Fig. 4, the core module in *Hone* is the online tuple scheduler, which interacts with the task manager module in two ways. On the one hand, it dynamically generates tuple scheduling decisions and enforces them through the task manager. On the other hand, it will receive outgoing queue statuses from the task manager. The task, upon receiving a scheduling instruction from the task manager, selects a tuple from its outgoing queue to output. Here, tuples in the same outgoing queue of a task are scheduled with default FIFO policy.

Online Tuple Scheduler. A key insight for scheduling the tuples of multiple relevant tasks is to minimize the maximum queue backlog among all the tasks over time, such that stragglers can be mitigated efficiently. To achieve this, *Hone* applies a $3 + \lceil \log_2 N \rceil$-competitive scheduling algorithm, which we will show in Section 4.

Task Manager. The task manager mainly keeps track of the queue statuses of all tasks (e.g., queue backlog information), so that the scheduler can make decisions. On the other hand, it receives the decisions from the scheduler and sends relevant instructions to tasks for enforcing the scheduling decisions. We will show the details of the task manager module in Section 5.

4 Online Tuple Scheduling

In this section, we present a tuple scheduling algorithm for minimizing the maximum queue backlog among tasks, *without* the prior knowledge of future tuple arrivals. Specifically, we first develop the mathematical model. We then formulate the tuple scheduling problem and solve it with an efficient online *LBF* algorithm. Finally, we conduct simulations to evaluate the effectiveness of *LBF*.

4.1 Mathematical Model

In our analysis, we consider a general case with single-operator and single-node. Tuple scheduling across the whole DAG over the entire cluster is achieved by scheduling tuples for each operator in each node independently. Besides, we consider a discrete-time mode where the time is divided into time slots. For simplicity, we assume that in each time slot, only one data tuple is allowed to be transmitted through the physical NIC of the node. Moreover, each tuple is assumed can be completed within one time slot. This is reasonable because a tuple is very small and can be encapsulated into an Ethernet packet frame [25]. We also assume that the number of tasks, as well as their placements, are fixed. One can change the number of tasks in the runtime to achieve better performance (as in [9]), while this is out of the scope of our work.

For brevity of exposition, we denote the set of tasks as $\mathcal{N} = \{w_1, w_2, \ldots, w_N\}$ and q_i as the outgoing queue of task w_i. Initially, at time slot $t = 0$, all the queues are empty. At each time slot $t > 0$, new tuples arrive at the N queues, while scheduled tuples leave. To be particular, we denote the number of tuples that arrive at queue q_i at time slot t as $a_i(t)$, which is a non-negative integer. Further, to indicate the tuple scheduling, we denote $x_i(t)$ as whether the queue q_i is selected to output a tuple at time t. The tuples that are not scheduled will be backlogged. We, therefore, denote $B_i(t)$ as the backlog of queue q_i at time t, which is measured by the number of tuples. We define $B_i(0) = 0$, $\forall q_i$, and then update it in each time slot t (> 0) as follows:

$$B_i(t+1) = \max\{B_i(t) + a_i(t) - x_i(t), 0\}. \tag{1}$$

This equation implies that each queue q_i takes $a_i(t)$ as input and $x_i(t)$ as output, and the unscheduled tuples at time t are backlogged to future time slots. The larger the backlog of a queue, the longer an outgoing tuple takes to wait in this queue before it can be scheduled, and accordingly, the higher the likelihood the corresponding task becomes a straggler.

4.2 Formulating the Problem

Given the model described above, we now study the problem of determining *which* queue to be scheduled to transfer one tuple each time to minimize the maximum backlog among all queues across all time slots. We denote this problem as *Tuple Scheduling Problem (TSP)*, as shown in the following:

$$\underset{\{x_i(t),\forall i,\forall t\}}{\text{Minimize}} \ \max_{0\le t\le T-1}\{\max_{1\le i\le N} B_i(t)\} \tag{2}$$

$$\textbf{Subject to}: \sum_{i=1}^{N} x_i(t) = 1, \forall t, \tag{3}$$

$$x_i(t) = \{0, 1\}, \forall q_i, \forall t. \tag{4}$$

The term $\max_{1\le i\le N} B_i(t)$ denotes the maximum backlog of all queues at time slot t. The objective function in Eq. (2) is clearly to minimize $\max_{1\le i\le N} B_i(t)$ over all time slots, enforcing that each queue has nearly balanced backlog in each time slot. Since only one tuple can be transmitted in each time slot, Eq. (3) ensures that there is only one $x_i(t) = 1$ for all q_i in each t. Eq. (4) enforces that the decision variable $x_i(t)$ can only take 0 or 1.

Despite its simple structure, the *TSP* problem is inherently challenging to be solved. The crux is that the current control decisions are coupled with future ones, i.e., $x_i(t)$ and $x_i(t+1)$. An alternative approach for solving this problem is to design an optimal offline algorithm. However, it inevitably relies on a *prior* knowledge of future tuple arrivals, i.e., $a_i(t), \forall i, \forall t$, which are readily unavailable.

4.3 Online Scheduling Algorithm

To address the challenge in solving the above *TSP* problem, we design an online scheduling algorithm that can be

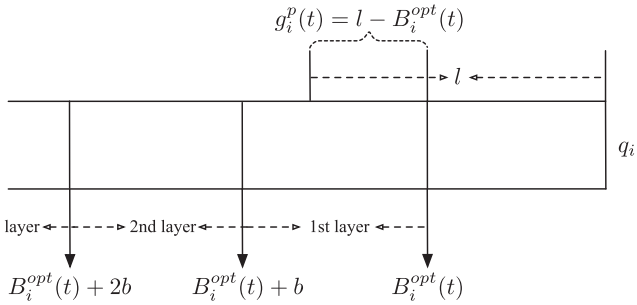


Fig. 5. Partition of queue q_i into layers at time slot t.

executed in a lightweight fashion. A key feature of this algorithm is that it decides to select which queue to output a data tuple, without any knowledge of how many tuples may be received in the future. Despite this, we will prove that our algorithm can provide a non-trivial competitive ratio when solving the *TSP* problem.

Algorithm 1. **L**argest-**B**acklog-**F**irst (**LBF**)

Input: Initial queue state: $\{B_i(0), \forall q_i\}$
Output: A feasible scheduling solution: $\{x_i(t), \forall q_i, \forall t\}$
1: In the beginning of each time slot t, observe the current queue backlog $B_i(t)$ and the number of arrived tuples $a_i(t)$ for all queues q_i;
2: Select the queue q_i with largest backlog $B_i(t)$, output one tuple from this queue and set $x_i(t) \leftarrow 1$;
3: Update $B_i(t+1)$ according to Eq. (1) and the newly determined scheduling decisions;
4: **return** $\{x_i(t), \forall q_i, \forall t\}$

We propose the Largest-Backlog-First (*LBF*) heuristic, as shown in Algorithm 1. In each time slot t, based on the *online* observation of the queue backlogs $\{B_i(t), \forall q_i\}$ and the number of newly arrived tuples $\{a_i(t), \forall q_i\}$, the *LBF* algorithm selects the queue with largest backlog and outputs its head tuple. Finally, it updates the queue backlogs according to Eq. (1) and the newly determined scheduling decisions. To verify that our algorithm can solve the *TSP* problem efficiently, we compare its effectiveness to that of the optimal offline algorithm, as shown in the following theorem.

Theorem 1. *The Largest-Backlog-First (LBF) heuristic has a competitive ratio of* $3 + \lceil \log_2 N \rceil$ *for the TSP problem.*

To facilitate the proof, we partition *LBF*'s queues into layers by introducing a metric *gap*. More formally, let $B_i^{opt}(t)$ denote the backlog of q_i at time t under an optimal off-line algorithm *OPT*. If p is the lth tuple from the top of queue q_i at t, then we calculate the gap of this tuple as $g_i^p(t) = l - B_i^{opt}(t)$, which essentially measures the differences between the height of a tuple in a queue and the length of the corresponding queue in *OPT*. According to the gap of each tuple, a queue q_i can be partitioned into multiple layers. As shown in Fig. 5, the kth layer of q_i at t consists of tuples whose gap satisfies $(k-1)b + 1 \le g_i^p(t) \le kb$ (where $b = B^{opt}$). We define the number of tuples contained in the kth layer of q_i at time t as $U_k^i(t)$. We can clearly see that for $k \ge 1$, $U_k^i(t) \ge U_{k+1}^i(t)$ and $U_k^i(t) = b$ if $U_{k+1}^i(t) > 0$. $U_k(t) = \sum_{i=1}^N U_k^i(t)$ is the total number of tuples contained in the kth layer over all queues. We define $V_k(t) = \sum_{k' > k} U_{k'}(t)$ as the total number of tuples contained in the layers strictly higher than the kth layer over all queues. The proof process of Theorem 1 relies on the following lemma.

Lemma 1. *For* $k \ge 1$ *and for any time* t, $U_k(t) \ge V_k(t)$.

Proof of Lemma 1. We prove the lemma by induction on the time t. Clearly, $U_k(t) \ge V_k(t)$ is true at $t = 0$, because no tuple is outputted and the backlog of each queue in *LBF* is same as that in *OPT*. Assuming $U_k(t) \ge V_k(t)$ holds for $t \ge 0$, it only remains to prove the following inequality:

$$U_k(t+1) \ge V_k(t+1). \tag{5}$$

To this end, we run the algorithms *OPT* and *LBF* simultaneously. Assume *OPT* and *LBF* select q_i and q_j at time t, respectively. If $i = j$, Eq. (5) is straightforward. If $i \ne j$, we have $U_k^h(t+1) = U_k^h(t)$ and $V_k^h(t+1) = V_k^h(t)$ for any queue q_h ($h \ne i, j$).

We now focus on q_i. If $B_i^{lbf}(t) < B_i^{opt}(t)$, then $B_i^{lbf}(t+1) \le B_i^{opt}(t+1)$. Thus, we get

$$U_k^i(t+1) = U_k^i(t) = 0 \text{ and } V_k^i(t+1) = V_k^i(t) = 0. \tag{6}$$

If $B_i^{lbf}(t) \ge B_i^{opt}(t)$, then let $\alpha = \lfloor (B_i^{lbf}(t) - B_i^{opt}(t))/b \rfloor + 1$. Considering the definitions of $U_k^i(t)$ and $V_k^i(t)$, we obtain

$$U_k^i(t+1) = \begin{cases} U_k^i(t), & k \ne \alpha, \\ U_k^i(t) + 1, & k = \alpha, \end{cases} \tag{7}$$

$$V_k^i(t+1) = \begin{cases} V_k^i(t) + 1, & k < \alpha, \\ U_k^i(t), & k \ge \alpha. \end{cases} \tag{8}$$

Next, we consider q_j. If $B_j^{lbf}(t) \le B_j^{opt}(t)$, then $B_j^{lbf}(t+1) < B_j^{opt}(t+1)$ and therefore we have

$$U_k^j(t+1) = U_k^j(t) = 0 \text{ and } V_k^j(t+1) = V_k^j(t) = 0. \tag{9}$$

If $B_j^{lbf}(t) > B_j^{opt}(t)$, then let $\beta = \lceil (B_j^{lbf}(t) - B_j^{opt}(t))/b \rceil$. Again by the definitions of $U_k^i(t)$ and $V_k^i(t)$, we have

$$U_k^j(t+1) = \begin{cases} U_k^j(t), & k \ne \beta, \\ U_k^j(t) - 1, & k = \beta, \end{cases} \tag{10}$$

$$V_k^j(t+1) = \begin{cases} V_k^j(t) + 1, & k < \beta, \\ U_k^j(t), & k = \beta. \end{cases} \tag{11}$$

Considering Eqs. (6)-(11), it suffices to show that Eq. (5) is true even when $U_k^j(t+1) = U_k^j(t) - 1$ or $V_k^i(t+1) = V_k^i(t) + 1$.

Now we focus on the first case where $U_k^j(t+1) = U_k^j(t) - 1$. From Eq. (10), we can infer $B_j^{lbf}(t) > B_j^{opt}(t)$ and $k = \beta$. Hence, $B_j^{lbf}(t) \le B_j^{opt}(t) + \beta b$. This leads to $B_j^{lbf}(t+1) = B_j^{lbf}(t) - 1 \le B_j^{opt}(t+1) + \beta b$, implying that the $(\beta+2)$th layer of q_j is empty at $t+1$. Assume that there exists a queue q_h ($h \ne j$) whose $(\beta+2)$th layer contains some tuples at $t+1$. Since q_h is not selected by *LBF* at t, we have

$$B_h^{lbf}(t) = B_h^{lbf}(t+1) \geq (\beta+1)b + 1 \geq B_j^{opt}(t) + \beta b \geq B_j^{lbf}(t). \tag{12}$$

This means that q_h should be selected by *LBF* at t, resulting in a contradiction. Thus, the $(\beta+2)$th layers of all queues must be empty at $t+1$. Hence, we have $U_\beta(t+1) \geq U_{\beta+1}(t+1) = V_\beta(t+1)$, implying that Eq. (5) holds in this case.

Next, we consider another case where $V_k^i(t+1) = V_k^i(t)+1$. Since $V_k^i(t+1) \geq 1$, we have $B_i^{lbf}(t) = B_i^{lbf}(t+1) \geq kb+1$. Further, we can know $k < \beta$. As *LBF* selects q_j rather than q_i at t, we have

$$B_j^{lbf}(t) \geq B_i^{lbf}(t) \geq kb+1 \geq B_j^{opt}(t) + (k-1)b + 1. \tag{13}$$

If $B_j^{opt}(t) + (k-1)b + 1 \leq B_j^{lbf}(t) \leq B_j^{opt}(t) + kb$, we can then similarly use the proof by contradiction to show that the $(k+2)$th layers of all queues are empty at $t+1$. Thus, we have $U_k(t+1) \geq U_{k+1}(t+1) = V_k(t+1)$. On the other hand, if $B_j^{lbf}(t) > B_j^{opt}(t) + kb$, then we have $V_k^j(t) > 0$. After *LBF* outputs a tuple at t from q_j, we have $V_k^j(t+1) = V_k^j(t) - 1$ and thereafter $k < \beta$. Thus, we obtain

$$\begin{aligned} V_k(t+1) &= V_k^i(t+1) + V_k^j(t+1) + \sum_{h\neq i,j} V_k^h(t+1) \\ &= (V_k^i(t)+1) + (V_k^j(t)-1) + \sum_{h\neq i,j} V_k^h(t) = V_k(t). \end{aligned} \tag{14}$$

Regarding $U_k(t)$, we can obtain $U_k^i(t+1) \geq U_k^i(t)$ from Eq. (7) and $U_k^j(t+1) = U_k^j(t)$ from Eq. (10) (as $k \neq \beta$). Thus, we have $U_k(t+1) \geq U_k(t) \geq V_k(t) = V_k(t+1)$, which means that Eq. (5) still holds in this case. Combining the above two cases, Lemma 1 can then be proved. □

Proof of Theorem 1. Using Lemma 1, we have

$$\begin{aligned} &V_{k+1}(t) = V_k(t) - U_{k+1}(t) \leq V_k(t) - V_{k+1}(t) \\ &\Longrightarrow V_{k+1}(t) \leq \frac{1}{2} V_k(t). \end{aligned} \tag{15}$$

Applying the above inequality $\lceil \log_2 N \rceil$ times, we have

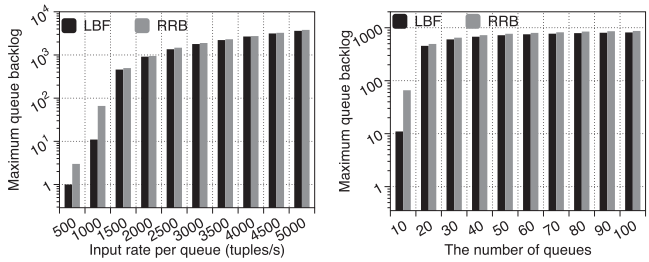
$$\begin{aligned} V_{\lceil \log_2 N \rceil + 1}(t) &\leq \left(\frac{1}{2}\right)^{\lceil \log_2 N \rceil} V_1(t) \leq \left(\frac{1}{2}\right)^{\log_2 N} V_1(t) \\ &= \frac{1}{N} V_1(t) = \frac{N \cdot b}{N} = b. \end{aligned} \tag{16}$$

The above inequality implies that the number of tuples in the layers strictly above the $(\lceil \log_2 N \rceil + 1)$th layer is at most b. The backlog at time t under *LBF* can then be bounded by

$$\begin{aligned} B^{lbf}(t) &\leq B^{opt}(t) + b + (\lceil \log_2 N \rceil + 1) \cdot b \\ &\leq (3 + \lceil \log_2 N \rceil) \cdot b. \end{aligned} \tag{17}$$

The above inequality holds at any t. Thus, proved. □

The following theorem shows that any general online algorithm is N-competitive for the *TSP*, demonstrating the superior theoretical performance of *LBF* algorithm.



(a) Maximum queue backlog vs. input rate per queue (b) Maximum queue backlog vs. num. of queues.

Fig. 6. The comparison of the maximum queue backlog across all queues and all time slots achieved by *LBF* and *RRB* under the settings of (a) different input rates and (b) different number of queues.

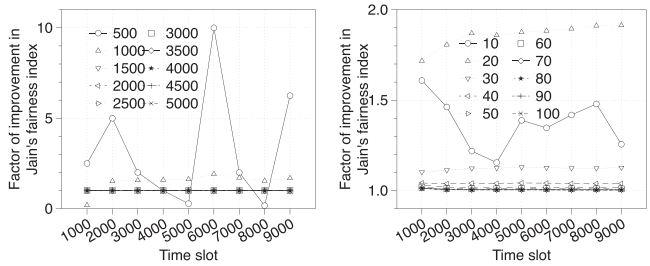
Theorem 2. *The general on-line algorithm for the TSP problem is N-competitive.*

Proof of Theorem 2. Let $C(t)$ denote the number of tuples backlogged in all the N queues at time t. Let $B^g(t) = \max_{1\leq i\leq N} B_i^g(t)$ denote the maximal queue length over the N queues at time t incurred by any on-line algorithm g. Then, we clearly have $B^g(t) \leq C(t)$. On the other hand, in the off-line optimal algorithm, $C(t)$ will be evenly distributed among the N queues. As such, for the off-line optimal algorithm, the maximum queue length at time t, $B^{opt}(t)$, should satisfy $B^{opt}(t) \geq C(t)/N$. Thus, we yield $B^g(t) \leq NB^{opt}(t)$ at any time t. Proved. □

4.4 Simulation for *LBF*

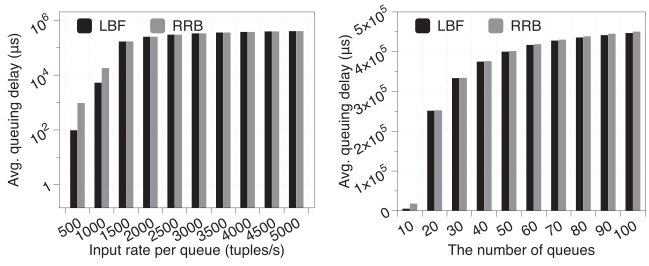
Simulation Setup. We conduct extensive simulations to evaluate the effectiveness of the *LBF* heuristic in balancing the queue backlogs. In our simulations, we assume that, for each queue, the data tuple arrival follows a Poisson distribution. All queues have the same tuple input rate. We conduct multiple simulation experiments by varying the input rate and the number of queues. Each simulation runs for 10,000 time slots, with each time slot being 100 μs. In each time slot, only one tuple can be scheduled. We use *Round-Robin* (*RRB*) as the baseline. In *RRB*, if queue q_i is selected at time slot t, then the queue selected at time slot $t+1$ is $q_{(i+1) \bmod N}$ where N is the number of queues. We consider *RRB* initially selects q_1.

Maximum Queue Backlog. Fig. 6a first depicts the maximum queue backlog among all queues across all time slots by varying the input rate from 500 to 5,000 tuples/s and fixing the number of queues to 10. It is clear that *LBF* can always achieve a smaller maximum queue backlog than *RRB*, with the maximum queue backlog reduced by up to 83.3 percent. We further observe that the maximum queue backlogs of both *LBF* and *RRB* increase as the input rate grows. The reason is that the service rate (i.e., one tuple a time) remains unchanged. One may wonder why the gap of maximum queue backlog between *LBF* and *RRB* decreases with the increasing of the input rate. The crux is that when the input rate is far more than the service rate, the system becomes highly overloaded, and hence the tuple scheduling can have little effect. Fortunately, modern DSP systems [7], [8] will trigger the backpressure mechanism to avoid high loads, implying that tuple scheduling is useful in most common cases. To further evaluate the impact of the number of



(a) Factor of improvements under different input rates. (b) Factor of improvements under different number of queues.

Fig. 7. Improvements in Jain's fairness index of queue backlogs at nine sampled time slots (i.e., 1000, 2000, ... , 9000) over *RRB* using *LBF* under (a) different input rate and (b) different number of queues.



(a) Avg. queuing delay of tuples under different input rates. (b) Avg. queuing delay of tuples under different number of queues.

Fig. 8. Average queuing delay of tuples achieved by *LBF* and *RRB* under (a) different input rates and (b) different number of queues.

queues on the maximum queue backlogs, we fix the input rate of each queue to 1,000 tuples/s and vary the number of queues from 10 to 100. As shown in Fig. 6b, the maximum queue backlog of *LBF* is always smaller than that of *RRB*, demonstrating the effectiveness of *LBF* in resolving the *TSP* problem. We can further observe that the maximum queue backlogs of both *LBF* and *RRB* first increase and then change a little, as the number of queues keeps growing. The reason is that when the number of queues increases to a sufficiently large value and each queue has the same input rate, there always exists a queue that needs to wait for a long time to be scheduled.

Backlog Balancing Among Queues. We also leverage the Jain's fairness index [26] to evaluate the backlog balancing performance among the queues. Specifically, given the backlog $B_i(t)$ of each queue q_i at a certain time slot t, the Jain's fairness index of backlogs of all queues at t is defined as $F(t)=\frac{(\sum_{i=1}^{N} B_i(t))^2}{N\cdot\sum_{i=1}^{N} B_i(t)^2}$, which is a value between $\frac{1}{N}$ and 1. A value closer to 1 means the queue backlogs are more balanced. For comparison, we define the following metric:

$$\text{Factor of Improvement}=\frac{\text{Jain's Fairness Index under } LBF}{\text{Jain's Fairness Index under } RRB}.$$

With this definition, Fig. 7a first shows the improvements in Jain's fairness index of queue backlogs at sampled time slots under different input rates. Note that the number of queues is fixed to 10. We observe that for all settings with different input rates, the factor of improvement is larger than one at most of the time. This verifies that *LBF* achieves a higher Jain's fairness index and hence results in more balanced queue backlogs than *RRB*. More specifically, *LBF* can be 10× better than *RRB*. One may question that *RRB* sometimes outperforms *LBF* in terms of Jain's fairness index, e.g., under the input rate of 500 tuples/s. Despite this, as shown in Fig. 6a, *LBF* can indeed obtain a lower maximum queue backlog under the input rate 500 tuples/s. Moreover, minimizing the maximum queue backlog can be more relevant in mitigating the stragglers for DSP jobs. We can further observe that as the input rate increases, the factor of improvement approaches to 1 gradually, implying that *LBF* and *RRB* can have nearly the same queue backlog balancing performance at high load scenarios. This is actually in line with the decreasing trend of the maximum queue backlog gap between *LBF* and *RRB* in Fig. 6a. Fig. 7b plots the factor of improvements in the Jain's fairness index of queue backlogs at sampled time slots for the cases with different number of queues and with the input rate being fixed to 1,000 tuples/s. It is obvious that *LBF* outperforms *RRB* over all sampled time slots and all cases considering the Jain's fairness index. Moreover, the factor of improvement using *LBF* over *RRB* increases to 1.9 until the number of queues reaches 20. After that, it decreases to 1 gradually. The underlying reason is that more queues can make tuple scheduling have less impact in queue backlogs considering the low service rate, and hence can lead to a more balanced backlog, especially when all queues have the same input rate.

Average Queuing Delay. The above results show that *LBF* can achieve superior queue backlog balancing performance than *RRB*. We now investigate if this can lead to low delay. Fig. 8a first shows the average queuing delay of all tuples with varying input rate per queue, under both *LBF* and *RRB*. In this figure, we set the number of queues to 10. We can see that *LBF* achieves a lower average tuple queuing delay than *RRB*. More specifically, *LBF* can reduce the average tuple queuing delay by up to 89.8 percent, compared to *RRB*. We further observe that the average tuple queuing delay of both *LBF* and *RRB* increases as the input rate per queue grows. This is reasonable because higher input rate will lead to more workload injected into the system. An interesting observation is that the reduction in the average tuple queuing delay using *LBF* over *RRB* seems to decrease with the increase of input rate per queue. The reason is that when the system is overloaded, tuple scheduling can have little space to take effect for reducing queuing delay. By fixing the input rate per queue to 1,000 tuples/s, Fig. 8b further depicts the average queuing delay achieved by both *LBF* and *RRB*, with the number of queues varying from 10 to 100. *LBF* can always outperform *RRB*, with the average queuing delay of tuples being reduced by up to 70.1 percent. As the input rate per queue is fixed, more queues lead to more workloads. That's why the average tuple queuing delay increases when the number of queues grows.

Based on the simulation results, we conclude that *LBF* can achieve superior performance than *RRB*. Therefore, in the following, we will incorporate *LBF* into a real DSP system (i.e., Storm) to see if it can benefit the low latency stream processing.

5 Implementation of *Hone*

We implement *Hone* in Storm, which is a modern framework popular for distributed stream processing. Storm is

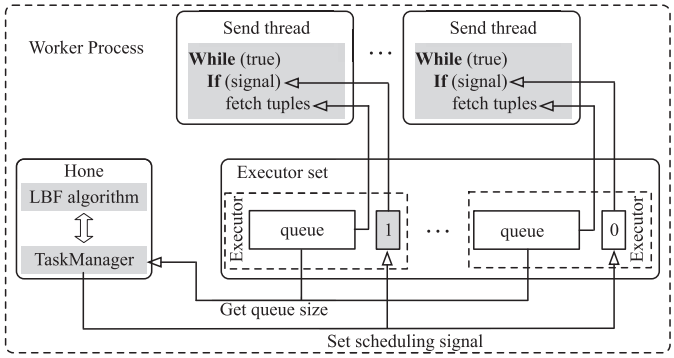


Fig. 9. *Hone* implementation in Storm.

simple, scalable, and fast in processing unbounded streams of data. Nowadays, more than 80 companies are using the Storm framework to build their streaming pipelines [27].

To incorporate our *Hone* in Storm, we take a non-intrusive approach. From a high level, *Hone* runs in the worker process, and is invoked every τ milliseconds (we will evaluate the impact of τ in Section 6). Fig. 9 shows the detailed architecture of the *Hone* implementation in Storm. As we can see from the figure, after the TaskManger module in *Hone* obtains the queue size information, the *LBF* algorithm will be invoked to make tuple scheduling decisions. After that, the TaskManger module will then set the scheduling signal for the relevant task to 1 for transmissions and 0 for otherwise. We can observe from the *Hone* implementation architecture that during the entire process, the TaskManager is the key to incorporate the *LBF* scheduling logic into Storm. More specifically, the TaskManager module involves two key steps: getting the queue length and enforcing the scheduling decisions; in what follows, we will present more details about these two steps.

Obtaining Queue Length. Before presenting how we obtain the queue length of all tasks, we first review the design of Storm. In the implementation of Storm, each worker process maintains a class called as `WorkerState`. This class stores the state of all tasks (also called as *executors* in Storm) in a worker process. For each executor, it is associated with an outgoing queue. This queue contains a variable called as `size` that stores exactly the up-to-date queue length information. Further, it exposes an API—`getQueuedCount()`, to enable users can have the ability to access such `size` information. With the above insights about Storm, we can obtain the queue length information of each executor by simply invoking the `getQueuedCount()` function. We then feed the queue length information of all executors to the *LBF* algorithm for determining tuple scheduling decisions.

Enforcing Tuple Scheduling Decisions. Recall that each task is equipped with an outgoing queue as well as a long-running send thread. Once the queue is nonempty, the corresponding thread will read tuples from this queue and push them to Netty for transmission. One intuitive approach for enforcing our tuple scheduling decisions in Storm is to replace all these send threads with a single central thread to read tuples from all the queues. In other words, if a queue is selected for transmission according to the scheduling decisions, the central thread will choose to fetch tuples from this queue and hence other queues are suspended. However, this approach is undesirable because the required modification will be too intrusive to the Storm framework. Hence, we take a non-intrusive approach without modifying the existing internal behavior of the Storm. In our implementation, we add a variable, called as `signal`, in each executor. Then, *Hone* sets the `signal` to 1 for the executor that has largest queue length and to 0 for others, based on the scheduling decisions of the LBF algorithm. Finally, each send thread checks if the `signal` equals to 1 before fetching tuples. If yes, it reads tuples and pushes them to Netty for transmission; otherwise, it runs quietly and fetches nothing.

It should be noted that our *Hone* is not limited to transmit exactly one tuple per time slot for a scheduled queue. Instead, it allows each scheduled queue to transmit tuples for the same while (which may be larger than a time slot). This aggregation has two benefits. First, it makes the scheduler to run in low frequency, thus involving low scheduling overhead. Second, it can reduce the amount of potentially wasted network resources caused by many small tuples that require less than one time slot to finish.

Since each send thread will check its scheduling signal before fetching tuples from their corresponding queues, a tuple will be either fetched and pushed to the underlying Netty framework for transmission or waiting in the queue. In other words, each tuple will be fully scheduled or non-scheduled, and there exist no tuples that have some part of data being pushed to Netty while leaving the remaining data in the queue.

6 Performance Evaluation of *Hone*

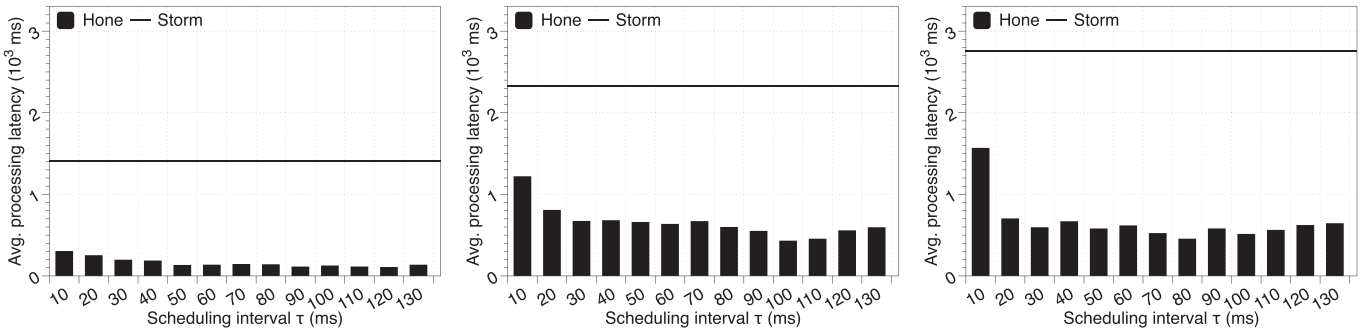
In this section, we demonstrate that our *Hone* scheduler is practical yet beneficial with a small-scale testbed evaluation. Our evaluation seeks to answer the following questions:

- *How does Hone perform in reducing the tuple processing latency?* Compared to the default Storm, *Hone* can reduce the average end-to-end tuple processing latency by 78.7 percent, even under the same workload balancing strategy.
- *What impact do different parameters have on the performance of Hone?* Our results show that *Hone* can outperform the default Storm, irrespective of the changes of the input rate and the scheduling interval.
- *Can Hone mitigate stragglers in practice?* The slowest task in *Hone* is significantly faster than that in Storm, meaning that *Hone* is efficient in straggler mitigation.

6.1 Experimental Setup

Testbed. We run all the experiments on a three-node testbed. Each node has two Intel(R) Xeon(R) E5-2630 v2 @2.60GHz CPUs and 64GB of RAM, running Ubuntu 16.04 LTS. Each CPU has two physical cores with each core having six virtual cores. The bandwidth between each pair of nodes is 1 Gbps. It should be noted that the high costs prohibit us from building a large-scale testbed. On the other hand, even though this three-node testbed is small in scale, it suffices to be used for testing *Hone*'s performance.

Applications. We use `WordCount` streaming applications for testing the performance of *Hone*. We choose `WordCount` because it is a fundamental benchmark application for



(a) Sleep time per `Spout` task = 8500 μs ⇒ Overall input rate ≈ 1170 tuples/s

(b) Sleep time per `Spout` task = 8000 μs ⇒ Overall input rate ≈ 1250 tuples/s

(c) Sleep time per `Spout` task = 7500 μs ⇒ Overall input rate ≈ 1330 tuples/s

Fig. 10. The comparison of the average end-to-end tuple processing latency between *Hone* and the original Storm under the settings of different input rates and different scheduling intervals.

distributed stream processing [9], [24], [28] and it can be used to process the real-world dataset (which we will show later) used in our experiments. The `WordCount` calculates the frequency of every distinct word appearing in the input tuple stream. It mainly involves two intermediate operators: `split` and `count`. More specifically, it treats each input tuple as a sentence, and splits each sentence into words which are then sent to the `count` operator for word frequency calculating.

Dataset. We use a publicly available 12 GB dataset from Wikipedia [29] as the input. To focus on the streaming regime, we split this Wikipedia dataset into sentences, view each sentence as a tuple, and feed one tuple to the `WordCount` application every time interval. We call this interval as *sleep time*. By controlling the sleep time, we can set the input rate of the `WordCount` flexibly. It is worth noting that tuples in this Wikipedia dataset exhibit significant variability, with the number of words per tuple varying from 1 to 32.

Baseline. We compare *Hone* with the default Storm [7] which is unaware of the tuple scheduling. As mentioned in Section 2, it pushes tuples to the underlying Netty framework for transmission, regardless of which one should be pushed first and which one later. For both *Hone* and Storm, we use *shuffle grouping* to balance the incoming tuples between the tasks of each operator. *shuffle grouping* has been shown to be an excellent workload balancing strategy that guarantees each relevant task will get an equal number of tuples [12], [13], [14].

Deployment. In our experiments, we use one node to serve as the `Spout` (i.e., *source* operator), while the other two nodes execute the `Split` and `Count` operators respectively. The number of tasks configured for each operator is 10. We mainly study the impacts of the following two parameters on the performance of *Hone*: the input rate (which is controlled via the sleep time per `Spout` task) and the scheduling interval (i.e., τ). When evaluating the impact of each parameter, we fix the other parameter. We run each experiment for 30 minutes. It should be noted that the send threads of tasks will typically batch outgoing tuples off their outgoing queues, which, however, bringing negative impact to low tuple latency. To eliminate such impact, we disable such batching (i.e., setting the batch size to 1) when running both Storm and *Hone*.

Performance Metric. Our primary performance metric for the comparison between *Hone* and the default Storm, as defined in Section 3, is the average end-to-end tuple processing latency over a window of time (i.e., 30 minutes).

6.2 Experimental Results

Overall Performance. Fig. 10 overviews the experimental results of the average end-to-end tuple processing latency for both *Hone* and Storm under the settings with different input rates and different scheduling intervals. We can clearly observe from this figure that *Hone* can drastically reduce the average end-to-end tuple processing latency, compared to Storm. To be particular, such reduction can be up to 92.2 percent when the input rate approximately equals to 1,170 tuples/s and the scheduling interval is equal to 120 *ms*. Furthermore, the average reduction across all the settings listed in Fig. 10 can be 78.7 percent. In the following, we focus on the analysis of the impacts of the input rate and the scheduling interval on *Hone*'s performance.

Impact of Input Rate. To explore the impact of input rate, we change the sleep time per `Spout` task, i.e., the interval at which we feed tuples to the `WordCount` application via one `Spout` task. We focus on three scenarios with the sleep time being 8500 μs, 8000 μs, and 7500 μs, respectively. Given that the `WordCount` application has 10 `Spout` tasks, the overall input rates for these three scenarios are approximately set to 1170 tuples/s, 1250 tuples/s, and 1330 tuples/s, respectively. By combining Figs. 10a, 10b and 10c, we observe that the average tuple processing latency achieved by both *Hone* and Storm increases with the increase of input rate. This is reasonable because that the higher the input rate, the more tuples a task needs to process. We can easily check that the average tuple processing latency incurred by Storm is higher than that achieved by *Hone*. The root reason is that our *Hone* can balance the queue backlogs well among the tasks to mitigate stragglers. Taking a step further, we find that given the same latency constraint, our *Hone* can support a higher input rate, as compared to Storm. One may wonder why the average tuple processing latency incurred by *Hone* substantially increases when the input rate scales from 1170 tuples/s to 1250 tuples/s, and remains relatively stable (or only increases little) afterward. The underlying reason may be that when there are more tuples injected into the system, the imbalance of the outgoing queue backlogs could be more severe, leaving more space for *Hone* to take effect for optimizing the average tuple processing latency.

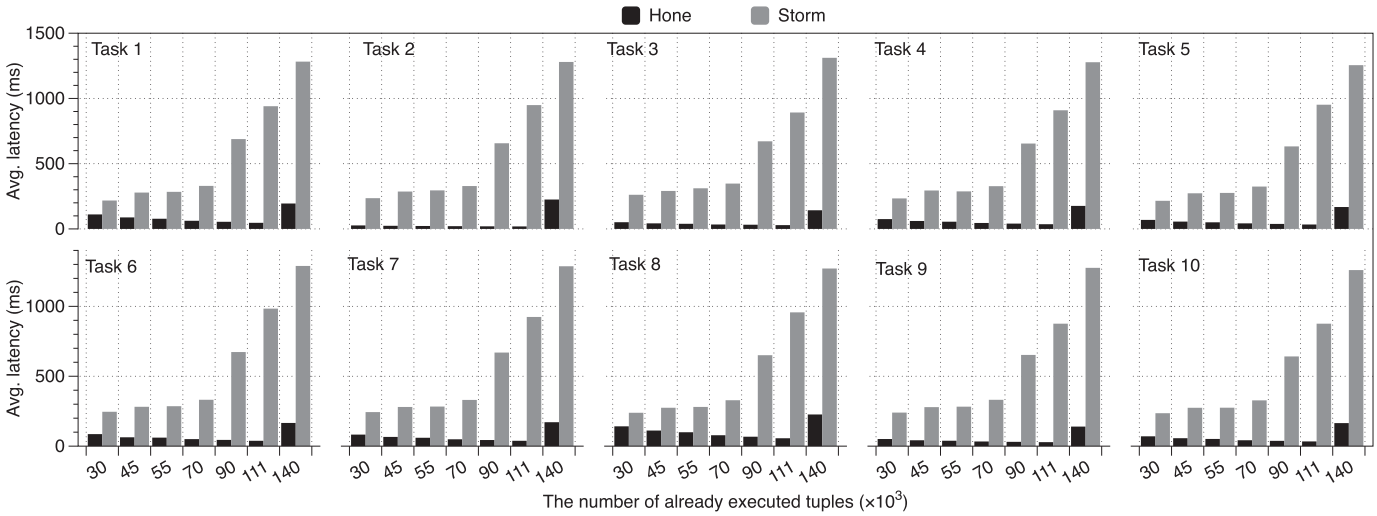


Fig. 11. The comparison of the average processing latency for each `Split` task between *Hone* (with the scheduling interval τ = 80 *ms*) and the original Storm under different number of already executed tuples, with the sleep time per `Spout` task = 8500 μs (i.e., overall input rate $\approx$ 1170 tuples/s).

Impact of Scheduling Interval τ. Recall that *Hone* is invoked every τ milliseconds to schedule tuples and hence τ is the scheduling interval. By varying τ from 10 *ms* to 130 *ms*, we study the impact of the parameter τ on *Hone*'s performance in reducing the average tuple processing latency, as shown in Fig. 10. From this figure, we make the following observations. First, across all input rates, when the scheduling interval τ is small, e.g., $\tau = 10$ *ms*, *Hone* incurs relatively high average tuple processing latency. This is because a smaller scheduling interval can lead to more thread communication overhead with respect to enforcing the frequently updated scheduling decisions. Despite this, *Hone* can still outperform Storm at small scheduling intervals. Second, as the parameter τ increases in a certain range (e.g., [10,50] *ms* in Fig. 10a), the thread communication overhead decreases and the tuple scheduling effect is getting better. As a result, the average tuple processing latency achieved by *Hone* is low when τ reaches a moderate value. Finally, when the scheduling interval τ is sufficiently large, e.g., 130 *ms*, the tuple scheduling may have less effect in balancing queue backlogs. That's why the average tuple processing latency is increasing after $\tau = 100$ *ms*.

Performance on Straggler Task Mitigation. To investigate the straggler mitigation performance, we record the average processing latency of each `Split` task of the `WordCount` application for both *Hone* and Storm, across different number of already executed tuples. The results are shown in Fig. 11. We make the following observations from this figure. First, across all tasks and all number of executed tuples, *Hone* can always achieve a lower average processing latency than Storm. This result directly demonstrates the effectiveness of our *Hone* in mitigating stragglers in DSP. Second, as the number of executed tuples increases, the average processing latency of each task grows using Storm while such latency remains within a relatively small range using *Hone*. The underlying reason is that Storm ignores the tuple scheduling, making substantial tuples to be backlogged after being executed. In contrast, *Hone* makes use of tuple scheduling to restrain the growth of the queue backlogs of each task, lowering the impact of an increasing number of executed tuples on the average processing latency.

Performance Under Partial Key Grouping. So far, our evaluation is based on shuffle grouping strategy. We now evaluate our algorithm under partial key grouping (PKG) [12] strategy to see if *Hone* can still provide superior performance. PKG is a simple stream workload partitioning strategy, which associates each key to two possible tasks, and selects the least loaded of the two whenever a tuple for a given key must be processed. In this experiment, we fix the sleep time to 8500 μs and vary the scheduling interval τ from 10 *ms* to 130 *ms*. The results are shown in Fig. 12. From this figure, we can clearly observe that *Hone* can achieve a lower average tuple processing latency than the original Storm, across all settings of scheduling intervals. More precisely, compared to original Storm, *Hone* can reduce the average tuple processing latency by 94.2 percent on average. Such results directly demonstrate that our *Hone* performs well under PKG strategy. One may wonder at this point that the reduction of *Hone* under partial key grouping is more obvious than that under shuffle grouping. The reason is that partial key grouping may balance the stream workload worse than shuffle grouping for the `WordCount` application, thus leaving more space for our *Hone* to take effect for reducing the average tuple processing latency.

Impact of Tuple Variance. We now study how the number of words per tuple impacts the performance of *Hone*. To this end, we construct three workloads based on the Wikipedia

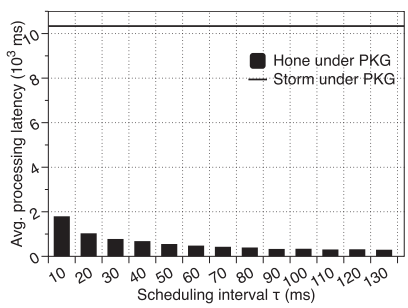


Fig. 12. The comparison of the average end-to-end tuple processing latency between *Hone* and the original Storm under partial key grouping (PKG) [12] strategy, with varying scheduling intervals and with the sleep time being 8500 μs.

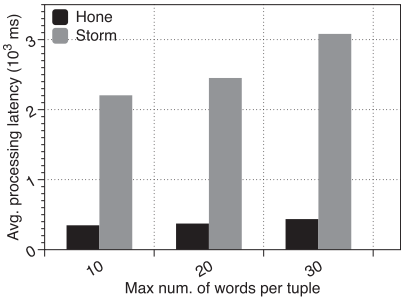


Fig. 13. The comparison of the average end-to-end tuple processing latency between *Hone* and the original Storm, under varying maximum number of words per tuple.

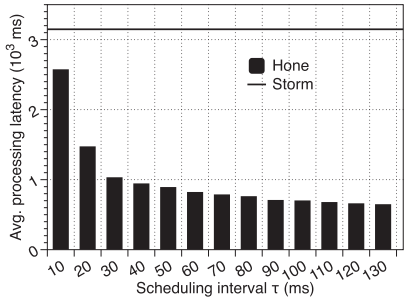


Fig. 14. The comparison of the average end-to-end tuple processing latency between *Hone* and the original Storm, under a 10-docker-container cluster.

dataset described in Section 6.1, by controlling the maximum number of words per tuple to 10, 20, 30, respectively. Fig. 13 depicts the average end-to-end tuple processing latency of both *Hone* and the original Storm, under these three workloads. We find that *Hone* can achieve a lower average tuple processing latency than the original Storm, across the three workloads. The average reduction in the average tuple processing latency achieved by *Hone* over Storm is 84.9 percent. We can further find that the reduction in the average tuple processing latency using *Hone* over Storm increase as the maximum number of words per tuple grows. This is because the larger the number of words per tuple in the workload, the more unbalanced backlogs will be experienced by relevant tasks, thus leaving more optimization space for *Hone* to take effect for reducing tuple processing latency.

Performance Under a 10-Docker-Node Cluster. So far, the above experiments are conducted in a cluster with only three nodes. We now study if *Hone* can perform well in a large cluster. To this end, we use one server in the testbed (in Section 6.1), and leverage the Docker technique to emulate a large cluster on this server. Specifically, we launch ten docker containers for this cluster. The application is still the `WordCount` application. We use three docker nodes for the `Spout` operator, four for `Split` operator and the remaining three for `Count` operator. Each operator has ten tasks. We deploy *Hone* and the original Storm on this cluster and run the `WordCount` application respectively, with the sleep time fixed to 8,500 μs and the scheduling interval τ being varied from 10 *ms* to 130 *ms*. Fig. 14 shows the results. It is clear that *Hone* outperforms the original Storm across all settings of the scheduling interval τ, with the average tuple processing latency being reduced by up to 79.4 percent. Such results demonstrate that our *Hone* can still provide performance gain under a relatively large cluster. One may wonder further why the reduction in the average tuple processing latency achieved by *Hone* over Storm decreases as the scheduling interval τ goes down. The reason is again that *Hone* incurs more scheduling overhead with a smaller scheduling interval τ.

7 Discussion

Considering Throughput. So far, our work only considers reducing latency for a DSP job. However, many DSP jobs also desire high throughput (i.e., the number of tuples processed per time unit) [24]. In fact, as indicated by the experimental results in Fig. 10, maintaining a lower average tuple processing latency can make a system to support a higher tuple injection rate. This indirectly implies that our *Hone* can achieve high throughput. To further improve throughput, one can resort to some other techniques, i.e., dynamically scaling of computational resources allocated to a DSP job [9] or balancing workload among parallel tasks with work-stealing [30].

Designing Key Grouping With Hone as a Basis. Our *Hone* mainly focuses on scheduling the outgoing tuples to balance the backlogs of parallel tasks. However, we observe that *Hone* can help to design efficient key grouping mechanisms to balance input workload among parallel tasks as long as *Hone* is enabled in the upstream operator of these tasks. Specifically, to achieve this goal, one can do the following. First, one can tag the scheduling time (in Unix time format) to each tuple whenever it is scheduled by *Hone*. Then, one can associate each key to two possible tasks: one of the two processes tuples having odd scheduling time stamp, and another one handles tuples with even scheduling time stamp. As such, the imbalance of the backlogs in upstream operator's tasks will not be cascaded to downstream operators.

Optimizing Transmission Delay. Our work only considers queuing delay of the tuples in the outgoing queues while leaving the tuple transmissions to the underlying Netty framework. Techniques such as bandwidth allocation [31], flow scheduling [32], congestion control [33], load balancing [34] can complement *Hone* by optimizing the transmission delay of tuples in the network. Alternatively, one can also pursue a cross-layer design to achieve better performance for a DSP job by dynamically adjusting flow rates according to the instantaneous information (e.g., the amount of outstanding tuples) obtained from the application layer. Such cross-layer design remains largely unexplored. We leave it to future work.

Handling Non-Data-Parallel Operators. Our *Hone* only works for the operators that can be executed with multiple parallel tasks. However, there do exist some operators that can not be parallelized such as `sort`, `sum` and `top-k`. Such non-data-parallel operators cannot benefit from the scheduling of *Hone*. One can tag such operators for *Hone* to ignore. Alternatively, one can do the following to integrate them. First, one can divide such operators into two sub-operators, with one of the two being able to be parallelized and another one being non-data-parallel. Then, one can still apply *Hone* to schedule outgoing tuples of the parallel tasks for the data-parallel sub-operator, whereas the non-

data-parallel sub-operator can run within a single task and simply merges results from parallel tasks from the data-parallel sub-operator.

Handling Stragglers in Heterogeneous Clusters. Our work only considers stragglers caused by the variability in the workloads. However, hardware heterogeneity can cause straggler tasks as well [35], [36]. To handle stragglers in heterogeneous clusters, one can use a reactive or proactive approach. The reactive approach allows all tasks to run for a while and marks slow running tasks as stragglers. It then reacts by replicating multiple copies of slow tasks or moving those tasks to fast nodes. On the other hand, the proactive approach can replica each task and only use the one that finishes first. Alternatively, it can predict straggler tasks using the promising machine learning techniques based on past workload traces and scheduling decisions, and then reschedule them.

8 Related Work

Hone mainly focuses on mitigating straggler tasks in a DSP job to speed up stream processing. We review the closely related work along the following topics:

Straggler Mitigation in Batch Processing. Mitigating straggler tasks in batch processing applications/jobs has always been a research hotspot. In order to deal with stragglers, prior solutions mainly prevent tasks from computing on slow machines or replicate stragglers on other machines–known as speculative re-execution or scheduling [37], [38], [39], [40], [41], [42], [43]. However, they detect stragglers too late and the re-schedule actions are carried out in the runtime, bringing in additional yet inevitable overhead. Hence, they are inapplicable to stream processing scenarios with the needs of continuous optimizations. There also exist some solutions on addressing stragglers in deep learning clusters with elastic parallelism control [44] or dynamic workload re-assignment [45]. They suffer from the same issues with the above solutions and can not be generalized to stream processing.

Straggler Mitigation in Stream Processing. The most effective, and indeed the most widely adopted technique for mitigating stragglers in stream processing is load balancing. To enable load balancing in DSP systems, existing work can be classified into three categories. The first category is to balance the number of tasks among distributed worker nodes to prevent tasks from running on slower nodes [10], [11], [46], [47]. Another category of work focuses on partitioning the input workload among parallel tasks based on the keys of the tuples in the stream [12], [13], [14], [15]. However, both categories of work ignore the outgoing tuple scheduling, which are insufficient for low latency DSP. Moreover, the workload-based solutions may lead to imbalanced queue backlogs due to tuple variance in the workloads, thus leading to poor performance. By contrast, *Hone* can balance the queue backlogs of different tasks. There are also other techniques for straggler mitigation, such as pre-scheduling [48], [49], which, however, rely on recurring nature of the workload and can only applicable to batched streaming processing.

Low Latency Stream Processing. Except for mitigating stragglers to achieve low latency stream processing, there exist several other techniques in literature [50], [51], [52], [53], [54], [55]. For example, Das *et al.* [50] design a control algorithm to automatically adapt batch sizes to maintain low latency for Spark Streaming. Rabkin *et al.* [51] leverage aggregation and degradation to reduce traffic over WANs to speed up the streaming analytics. Heintz *et al.* [52] use an idea of approximate processing, which sacrifices some accuracy to achieve low latency for geo-distributed streaming analytics. Tao *et al.* propose a dependable scheduling strategy with active replica placement to enhance workflow application performance [53]. Xu *et al.* [54] and Li *et al.* [55] leverage the techniques of in-memory computing and memory object caching, respectively, to speed up Internet of things real-time data processing. The techniques above can complement *Hone* to further reduce the stream processing latency.

9 Conclusion

In this paper, we have presented *Hone*, a tuple scheduler that seeks to balance the queue backlogs of different tasks to mitigate stragglers in a DSP job. To perform efficient tuple scheduling, *Hone* employs an online *LBF* heuristic that has a provable good competitive ratio in minimizing the maximum queue backlogs of tasks over time. To the best of our knowledge, *Hone* is the first work that proposes and proves the position that tuple scheduling must be well utilized for straggler mitigation in DSP. Through real implementation and testbed experiments, we have shown convincing evidence that *Hone* provides a remarkably lower average end-to-end tuple processing latency than the default Storm, when both *Hone* and Storm use the same workload balancing strategy.

Acknowledgments

This work was supported in part by the Hong Kong RGC TRS under Grant T41-603/20-R and Grant GRF-16215119; in part by NSFC under Grant 62002259, Grant 62032017, Grant 61772251, and Grant 61772112; and in part by the Science Innovation Foundation of Dalian under Grant 2019J12GX037.

References

[1] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proc. 21st Annu. Int. Conf. Mobile Comput. Netw.*, 2015, pp. 426–438.

[2] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang, "CFA: A practical prediction system for video QoE optimization," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 2016, pp. 137–150.

[3] S. A. Noghabi *et al.*, "Samza: Stateful scalable stream processing at Linkedin," in *Proc. VLDB Endowment*, vol. 10, pp. 1634–1645, 2017.

[4] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[5] G. J. Chen *et al.*, "Realtime data processing at Facebook," in *Proc. ACM Int. Conf. Manage. Data*, 2016, pp. 1087–1098.

[6] Apache Storm Roadmap at Yahoo. Accessed: Jun. 15, 2019. [Online]. Available: http://yahoohadoop.tumblr.com/post/122544585051/apache-storm-roadmap-at-yahoo

[7] Apache Storm. Accessed: Jun. 15, 2019. [Online]. Available: http://storm.apache.org/

[8] Apache Flink. Accessed: Jun. 15, 2019. [Online]. Available: https://flink.apache.org/

[9] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 783–798.

[10] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proc. 32nd Int. Conf. Very Large Data Bases*, 2006, pp. 775–786.

[11] R. Khandekar *et al.*, "COLA: Optimizing stream processing applications via graph partitioning," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2009, pp. 308–327.

[12] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 137–148.

[13] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 589–600.

[14] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola, "Online scheduling for shuffle grouping in distributed stream processing systems," in *Proc. 17th Int. Middleware Conf.*, 2016, Art. no. 11.

[15] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, "Parallel stream processing against workload skewness and variance," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 15–26.

[16] L. Mai *et al.*, "Chi: A scalable and programmable control plane for distributed stream processing systems," in *Proc. VLDB Endowment*, vol. 11, pp. 1303–1316, 2018.

[17] J. Cho, H. Chang, S. Mukherjee, T. Lakshman, and J. Van der Merwe, "Typhoon: An SDN enhanced real-time big data streaming framework," in *Proc. 13th Int. Conf. Emerg. Netw. Experiments Technol.*, 2017, pp. 310–322.

[18] S. Wu *et al.*, "TurboStream: Towards low-latency data stream processing," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 983–993.

[19] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1507–1518.

[20] W. Aljoby, X. Wang, T. Z. J. Fu, and R. T. B. Ma, "On SDN-enabled online and dynamic bandwidth allocation for stream analytics," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 8, pp. 1688–1702, Aug. 2019.

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

[22] A. Jonathan, A. Chandra, and J. B. Weissman, "Multi-query optimization in wide-area streaming analytics," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 412–425.

[23] Netty project. Accessed: Jun. 15, 2019. [Online]. Available: https://netty.io/

[24] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven multi-tenant stream processing," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 249–262.

[25] T. Buddhika and S. Pallickara, "NEPTUNE: Real time stream processing for Internet of Things and sensing environments," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 1143–1152.

[26] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination," *CoRR*, 1998. [Online]. Available: https://arxiv.org/pdf/cs/9809099.pdf

[27] Companies using apache storm. Accessed: Jun. 15, 2019. [Online]. Available: https://storm.apache.org/Powered-By.html

[28] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endowment*, vol. 10, pp. 1825–1836, 2017.

[29] Wikipedia. Accessed: Jun. 15, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Main_Page

[30] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in MapReduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.

[31] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "NUMFabric: Fast and flexible bandwidth allocation in datacenters," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 188–201.

[32] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 455–468.

[33] J. Xue, M. U. Chaudhry, B. Vamanan, T. N. Vijaykumar, and M. Thottethodi, "Dart: Divide and specialize for fast response to congestion in RDMA-based datacenter networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 1, pp. 322–335, Feb. 2020.

[34] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 503–514.

[35] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 7.

[36] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou, "Improving performance of heterogeneous MapReduce clusters with adaptive task tuning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 774–786, Mar. 2017.

[37] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[38] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using Mantri," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 265–278.

[39] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 185–198.

[40] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.

[41] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proc. ACM Conf. SIGCOMM*, 2015, pp. 379–392.

[42] Z. Fu and Z. Tang, "Optimizing speculative execution in spark heterogeneous environments," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2019.2947674.

[43] W. C. Ao and K. Psounis, "Resource-constrained replication strategies for hierarchical and heterogeneous tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 793–804, Apr. 2020.

[44] Q. Zhou *et al.*, "Falcon: Addressing stragglers in heterogeneous parameter server via multiple parallelism," *IEEE Trans. Comput.*, vol. 70, no. 1, pp. 139–155, Jan. 2021.

[45] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 98–111.

[46] J. Wolf *et al.*, "SODA: An optimizing scheduler for large-scale stream-based distributed computer systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2008, pp. 306–325.

[47] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst.*, 2013, pp. 207–218.

[48] H. Jin *et al.*, "Towards low-latency batched stream processing by pre-scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 3, pp. 710–722, Mar. 2019.

[49] S. Venkataraman *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *Proc. ACM Symp. Operating Syst. Princ.*, 2017, pp. 374–389.

[50] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

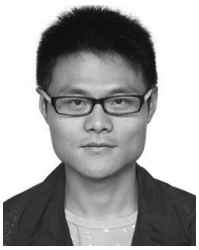
[51] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in JetStream: Streaming analytics in the wide area," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 275–288.

[52] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 361–373.

[53] M. Tao, K. Ota, and M. Dong, "DSARP: Dependable scheduling with active replica placement for workflow applications in cloud computing," *IEEE Trans. Cloud Comput.*, vol. 8, no. 4, pp. 1069–1078, Oct.–Dec. 1, 2020, doi: 10.1109/TCC.2016.2628374.

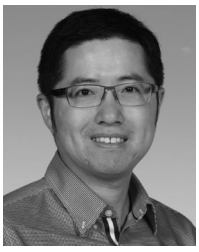
[54] J. Xu, K. Ota, and M. Dong, "Real-time awareness scheduling for multimedia big data oriented in-memory computing," *IEEE Internet of Things J.*, vol. 5, no. 5, pp. 3464–3473, Oct. 2018.

[55] D. Li, M. Dong, Y. Yuan, J. Chen, K. Ota, and Y. Tang, "SEER-MCache: A prefetchable memory object caching system for IoT real-time data processing," *IEEE Internet of Things J.*, vol. 5, no. 5, pp. 3648–3660, Oct. 2018.



Wenxin Li received the BE and PhD degrees from the School of Computer Science and Technology, Dalian University of Technology, Dalian, China, in 2012 and 2018, respectively. Currently, he is a post-doc researcher with the Hong Kong University of Science and Technology. From May 2014 to May 2015, he was a research assistant with the National University of Defense Technology. From October 2016 to September 2017, he was a visiting student with the Department of Electrical and Computer Engineering, University of Toronto. His research interests include datacenter networks and cloud computing.

Duowen Liu received the BE degree from the School of SEIEE, Shanghai Jiao Tong University, Shanghai, China, in 2017. Currently, he is working toward the Mphil degree from the Hong Kong University of Science and Technology, Hong Kong. His research interests include datacenter networks and cloud computing.



Kai Chen (Senior Member, IEEE) received the PhD degree in computer science from Northwestern University, Evanston, Illinois, in 2012. He is currently an associate professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include data center networking, machine learning systems, and privacy-preserving AI infrastructure.

Keqiu Li (Senior Member, IEEE) received the bachelor's and master's degrees from the Department of Applied Mathematics, Dalian University of Technology, Dalian, China, in 1994 and 1997, respectively, and the PhD degree from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology, Nomi, Japan, in 2005. He also has two-year postdoctoral experience with the University of Tokyo, Japan. He is currently a professor with the School of Computer Science and Technology, Dalian University of Technology, China. He has published more than 100 technical papers, such as the *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Internet Technology*, and *ACM Transactions on Multimedia Computing, Communications, and Applications*. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. His research interests include internet technology, data center networks, cloud computing, and wireless networks.

Heng Qi received the bachelor's degree from Hunan University, Changsha, China, in 2004, and the master's and doctorate degrees from the Dalian University of Technology, Dalian, China, in 2006 and 2012. He was a lecture with the School of Computer Science and Technology, Dalian University of Technology, China. He servered as a software engineer in GlobalLogic-3CIS from 2006 to 2008. His research interests include computer network, multimedia computing, and mobile cloud computing. He has published more than 20 technical papers in international journals and conferences, including the *ACM Transactions on Multimedia Computing*, *Communications and Applications* (ACM TOMCCAP), and *Pattern Recognition* (PR).