Liang HL, Wu J, Jiang Z *et al.* A task allocation method for stream processing with recovery latency constraint. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(6): 1125–1139 Nov. 2018. DOI 10.1007/s11390-018-1876-6

A Task Allocation Method for Stream Processing with Recovery Latency Constraint

Hong-Liang Li[1,2,3], *Member, CCF, IEEE*, Jie Wu[3], *Fellow, IEEE*, Zhen Jiang[4], *Member, ACM, IEEE*
Xiang Li[1], and Xiao-Hui Wei[1,2,*], *Distinguished Member, CCF, Member, IEEE*

[1] *College of Computer Science and Technology, Jilin University, Changchun 130012, China*

[2] *Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education Changchun 130012, China*

[3] *Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, U.S.A.*

[4] *Department of Computer Science, West Chester University of Pennsylvania, West Chester, PA 19383, U.S.A.*

E-mail: lihongliang@jlu.edu.cn; jiewu@temple.edu; zjiang@wcupa.edu; {lxiang, weixh}@jlu.edu.cn

Received July 14, 2017; revised September 26, 2018.

Abstract Stream processing applications continuously process large amounts of online streaming data in real time or near real time. They have strict latency constraints. However, the continuous processing makes them vulnerable to any failures, and the recoveries may slow down the entire processing pipeline and break latency constraints. The upstream backup scheme is one of the most widely applied fault-tolerant schemes for stream processing systems. It introduces complex backup dependencies to tasks, which increases the difficulty of controlling recovery latencies. Moreover, when dependent tasks are located on the same processor, they fail at the same time in processor-level failures, bringing extra recovery latencies that increase the impacts of failures. This paper studies the relationship between the task allocation and the recovery latency of a stream processing application. We present a correlated failure effect model to describe the recovery latency of a stream topology in processor-level failures under a task allocation plan. We introduce a recovery-latency aware task allocation problem (RTAP) that seeks task allocation plans for stream topologies that will achieve guaranteed recovery latencies. We discuss the difference between RTAP and classic task allocation problems and present a heuristic algorithm with a computational complexity of $O(n \log^2 n)$ to solve the problem. Extensive experiments were conducted to verify the correctness and effectiveness of our approach. It improves the resource usage by 15%–20% on average.

Keywords stream processing, task allocation, fault-tolerance, upstream backup, recovery latency

1 Introduction

Stream processing applications are a novel class of "Big Data" applications that are able to continuously process and analyze large-scale online data streams in real time or near real time. High demands of such applications can be found in various areas, including analyzing social networks, trading high-frequency stocks, and monitoring and controlling production lines. This has led to a rapid increase in the popularity of a new computing paradigm known as the stream processing model (SPM)[1−3].

Over the last decade, various stream processing systems have been proposed in both academia and industry[2,4−9] to implement SPM. In these systems, stream processing applications take one or more data streams as an input, perform a series of predefined functions, and generate output in the form of data streams again. A stream processing application is usually modeled as a directed acyclic graph (DAG) of tasks and

Regular Paper

A preliminary version of the paper was published in the Proceedings of IEEE CLUSTER 2017.

This work is supported by the National Key Research and Development Program of China under Grant No. 2017YFC1502306, the National Natural Science Foundation of China under Grant No. 61602205, the China Scholarship Council, and the National Science Foundation of the U.S. under Grant Nos. CNS 1629746, CNS 1564128, CNS 149860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, and ECCS 1231461.

*Corresponding Author

©2018 Springer Science + Business Media, LLC & Science Press, China

their interconnections, i.e., stream topology[10]. Each task consumes the data from upstream task(s), processes the data, and emits results as data stream(s) to downstream task(s)[11]. A task without an upstream task is called a source, and one without a downstream task is called a sink. The tasks in a stream topology accomplish in a pipeline manner and form one or multiple paths from a source to a sink.

These stream processing applications share a common characteristic of the strict latency constraint, that is, these applications must provide fast and accurate response. The processing latency of a stream topology is the end-to-end elapsed time from the entrance of a set of input data to the emission of the corresponding output data[10,12]. It equals the largest processing latency of all paths in the stream topology[10]. Existing work focuses on balancing the latencies among multiple paths by assigning each task with appropriate physical resources, which is known as the task allocation problem[10,13−16] for SPM. However, when extra time is needed for the recovery of any failed task, especially when a hardware issue, such as a core dump, occurs to force the recovery of all tasks allocated to the same processor, the slowdown of a single task and the possible accumulated impact of all delayed tasks will suspend the processing. This will slow down the entire processing and may break the latency constraints, which is the focus of this paper.

The inability to obtain complete data beforehand has led to a computing paradigm entirely different from the traditional "process-after-store" mode where SPM performs "one-pass" processing over stream data on the fly without storing them. Therefore, SPM can be more vulnerable to failures than other big-data processing schemes[6,7,17,18]. This unique characteristic poses a novel fault-tolerant problem[1,19] due to new challenges like strict recovery latency constraints[20] and complex recovery dependencies[21−24]. In recent years, the upstream backup scheme[11,22,23,25] has been widely applied due to its smaller fault-tolerant (FT) overhead, compared with the active replication scheme[26,27], and reasonable recovery latency.

In the upstream backup model, each task is able to maintain a backup of the output data for its downstream tasks. Upon failure, all of the upstream backup tasks replay backup data, and the recovering task reprocesses those data to recover to its previous status, which introduces task recovery latency[11,22]. The recovery latency of a stream topology is the largest recovery latency of its tasks[17,27]. Fig.1 shows some examples of the recovery latencies of a stream topology under different failure scenarios. Fig.1(a) displays the required recovery latency for each task. In the ideal situation, the recovering task can obtain all the backup data from its upstream task(s) (see the recovering task c and its upstream task b in Fig.1(b)). However, the impact of correlated failures is not considered here. During task allocation, an allocation plan may put multiple tasks on one processor to share the physical resource[10,28]. It is considered very common in practice because a modern CPU core can host a few stream processing tasks at one time. In this case, tasks located on the same processor fail together during a processor-level failure (see the correlated failures of tasks b and c in Fig.1(c)). In such a case, the recovering task c must wait for its dependent upstream task b to finish the recovery and then it can obtain the backup data to start its own. The recovery of these downstream tasks can be prolonged and causes extra latency (see the accumulated delay in the recovery of task c, $h_c = 6 > 2$).

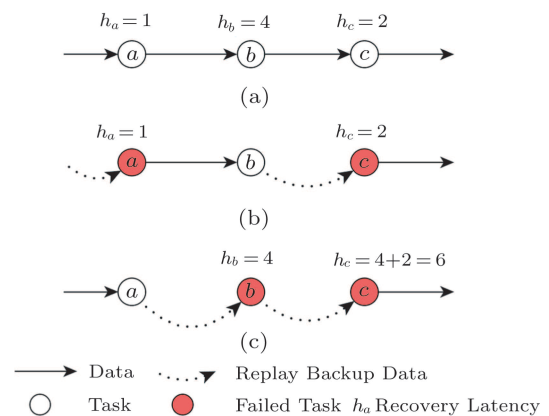


Fig.1. Examples of recovery latencies in different failure cases. (a) Recovery latency. (b) Unrelated failures $\max\{1, 2\} = 2$. (c) Correlated failures $\max\{4, 6\} = 6$.

The purpose of this paper is to study the relationship between recovery latency and task allocation plan and to present a comprehensive approach to compute task allocation plans that provide recovery latency guarantee. The main contributions are summarized as follows.

- We present a quantitative model that describes the relationship between the recovery latency and the task allocation plans of a stream topology. We introduce the recovery-latency aware task allocation problem (RTAP) and discuss how it differs from classic task allocation problems.
- We propose a heuristic algorithm based on the topology information to compute task allocation plans with recovery latency guarantee.

• We conduct extensive simulations to verify the correctness and effectiveness of our approach with different applications and setups.

This paper further explores the RTAP problem based on our earlier conference version[20]. We propose an efficient approach to solve the problem and provide extensive experimental results and analysis of difference approaches. The remaining of the paper is organized as follows. In Section 2, we summarize related work. Section 3 presents the problem model and analysis. We propose our approach in Section 4 and Section 5 discusses the experimental results. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Task Allocation for Stream Topology

A stream topology is usually modeled as a directed acyclic graph (DAG) $G(V, A)$ of tasks (V) and directed connections (A). The task allocation problem is one of the fundamental issues of stream processing systems that allocate resources for each task according to its resource requirement, avoiding either performance bottleneck (under-provisioning) or the waste of resources (over-provisioning). Earlier work focuses on the modeling of task resource requirements and the relationship between assigned resources and processing performances (throughput and latency)[2,12,28]. The resource requirement of each task, hereafter referred to as the weight of a task, represents the share of resource (computational, memory, and/or bandwidth capacity) that is required to ensure the processing performance according to its input speed. Eidenbenz and Locher[9] gave a theoretical analysis of this problem and proved its NP-hardness. They proposed an approach to compute optimal resource assignments for each task in a given stream topology when the stream topology is a series-parallel decomposable graph.

Assuming the resource requirements of each task are given as the input, other studies[13,14,29] investigated the problem of allocating resources for tasks from available resource pools. Chatzistergiou and Viglas[14] presented a fast heuristic algorithm considering both computational and bandwidth resource requirements and used throughput as the performance metric. Recent work focuses on enhancing the processing latency for both static[13] and dynamic[29] task weights.

Most of these studies formulize the task allocation problem based on the bin packing problem (BPP), which is a well-studied combinatorial optimization problem. We discuss related models and approaches of BPP in Subsection 2.4. Related work has been focusing on task allocation problem in a failure-free scenario that does not take failures effects into account.

2.2 Reliable Stream Processing

Active replication and checkpoint/recovery are two traditional FT mechanisms that have been widely studied in distributed systems[30]. They both have applications in distributed stream processing systems. Active replication maintains at least one active replica instance to enable instant switches from its primary instance to its replication when failure occurs[31]. This ensures minimum response time but suffers from a high overhead, at least doubling resource consumptions. It is applied in earlier stream processing systems or data engines[1,26] that are hosted by a cluster of a small number of machines. With the application scales increasing rapidly[8], the active replication model becomes inefficient or even impractical to distributed stream processing systems (DSPS)[6,11], which is why most recent researches explore FT approaches based on checkpoint/recovery[5,25,27,32].

Hwang *et al.*[11] introduced an upstream backup model that takes advantage of the close upstream-downstream dependencies. Upstream tasks keep output buffers as backups for downstream tasks. If a downstream task fails, the backup data is replayed to generate correct results. It is an efficient approach for the stream processing model but only supports applications that depend on recent data rather than support those that depend on the complete history of previous data. Therefore, recent work improves the upstream backup with the combination of checkpoint/recovery to solve this problem[22,25,27,33], which becomes the most commonly used FT method for SPM.

2.3 Processing/Recovery Latency Modeling

Chain[11] is one of the earliest researches that studied the processing latency model and task allocation strategies. It presents a solution for minimizing the makespan of a stream processing job in a single processor. In recent years, the task allocation problem for DSPS has been widely studied[10,13,14]. These studies use similar processing latency models and stream topology models, which provide the background for our work. Eidenbenz and Locher[10] presented strong theoretical results for a common type of stream topology

(i.e., SPD). They proposed solutions to compute optimal resource "slices" for stream processing tasks under a continuous resource partitioning scenario. However, they did not consider the extra cost from the failures that occur dynamically. The recovery incurs latency that would further trigger performance bottleneck.

The recovery latency of stream processing time is related to multiple parameters, such as state size, queue length, window size, and checkpoint interval[27]. There are two approaches to estimating recovery latency: experimental method and theoretical method. Heinze *et al.*[27] designed a clustering method based on historical samples to estimate the recovery time. Salama *et al.*[34] used a reliability model[35] to estimate the recovery time. Given the failure rate of a task, one can compute optimal checkpoint interval based on the method in [35]. The reprocessing time can be estimated (approximately equal to 1/2 checkpoint interval)[11,34,35]. However, they assumed that there is no correlation between any pair of tasks. The estimation of reprocessing time is independent. We studied the relationship between recovery latency and FT overheads[17]. A task-level failure effect model was proposed, assuming tasks are with independent failure rates.

These studies provide methods to estimate the recovery latency of individual tasks independently, but are not suitable for stream processing applications with correlated task failures. In the upstream backup model, each task is able to maintain the backup data for its downstream tasks, which introduces complex dependencies among tasks. When tasks with dependencies fail together, they have to recover sequentially according to their dependency on the data existence. The recovery latencies must be accumulated along such relation. Moreover, tasks on the same processor will fail at the same time in a processor-level failure. The recovery latencies are also related to the task allocation plans.

In our earlier paper[20], we introduced the task allocation problem considering recovery latency constraints. This paper further explores the problem and proposes an new efficient approach to solve the problem. To the best of our knowledge, this is the first time to study the relationship between recovery latencies and task allocation plans.

2.4 Bin Packing Problem

In the classic bin packing problem (BP), we are given a list of items and the goal is to place them in a minimal number of bins so that no bins are over-packed. General BP and its variants are used to model resource allocation[36] and job scheduling[37] problems. For the general BP problem and the two-dimensional BP (2BP) problem in particular, please refer to surveys of [38, 39].

If we consider the resource requirement (weight) of a task as the first dimension of the bin packing problem, and the recovery latency of a task as the second dimension, the problem we are facing is closely related to the two-dimensional packing problems, such as the two-dimensional bin packing problem (2BP) and the two-dimensional strip packing problem (2SP). Both problems are strongly NP-hard. One of the most widely-applied approaches is "level-oriented" heuristics[40], such as next-fit decreasing height (NFDH), first-fit decreasing height (FFDH), and best-fit decreasing height (BFDH). These algorithms pack items in rows forming levels[39] to solve the 2SP problem. They differ in the ways of choosing bins to put an item in. For details of these heuristics, please refer to [39]. Moreover, these levels can be further packed into bins using one-dimensional packing approaches[41] to solve the 2BP problem[42].

We refer to the target problem as a single-level two-dimensional bin packing problem (SL2BP), which is derived from 2BP and 2SP when the width of a bin represents the computational capacity of a processor in which there is only one level allowed in each bin.

3 Problem Formulation

3.1 Task Allocation Problem for Stream Topology

The stream processing application is usually modeled as a directed acyclic graph (DAG) $G(V, A)$, i.e., stream topology[10]. The vertices $V = \{v_i | i \in 1, ..., n\}$ represent tasks (also denoted as processing elements[7]) and arcs $A = \{a(v_i, v_j) | v_i, v_j \in V\}$ represent connections between tasks. For each arc $a(v_i, v_j)$, v_i and v_j are adjacent tasks, v_i is v_j's upstream task and v_j is v_i's downstream task. Each task consumes data from upstream task(s), processes the data, and emits results as data streams to downstream task(s)[11]. The task without an upstream task is called source, and the one without a downstream task is called sink. Note that we assume, without loss of generality, there are one source and one sink in each stream topology. When there are multiple sources (sinks), we can add a dummy source (sink) as the upstream (downstream) task of all sources (sinks).

For each task in the stream topology, its resource requirement (w_v) is given, reflexing the computing capacity needed by a task according to the input data rate[10]. When all tasks' resource requirements are fulfilled, all tasks can keep up with the pace of the input data so that there is no data loss caused by delayed processing. We consider the following problem that seeks to map all tasks in a stream topology to a set of parallel processors, with all the resource requirements of the tasks satisfied.

Problem 1 (Task Allocation Problem). *Given a set of parallel processors,* $P = \{p_i | i \in 1, ..., m\}$ *with a certain capacity. The task allocation of a stream topology* $G(V, A)$ *is to find a mapping* $\Phi = V \to P$ *according to task weight* w_v.

Problem 1 can be simplified as a one-dimensional bin packing problem (1 BP)[41,43], where the resource capacity of a processor corresponds to the bin capacity ($(0, 1]$) and the task weight corresponds to the item width (nominalized to 1). The goal is to place items in a minimal number of bins so that the total weight of items in each bin does not exceed 1.

Note that we only consider the computational costs of tasks (weights of vertexes in G)[10] but not the communications cost among tasks (weights of arcs in G)[44]. This represents the case when the stream application is computational-constrained[10] or the scenario when the parallel processors are connected with high speed/throughput networks that the communication costs are relatively lower than the computational costs.

3.2 Fault-Tolerant Setups

Each task performs upstream backup[11,22,23,25], i.e., upstream tasks back up output data for adjacent downstream task(s). The input stream of the source of the topology is assumed to be backed up into a log system[32], such as Apache Kafka[45]. Both stateless tasks and stateful tasks can perform upstream backup. Each stateful task performs a periodical checkpoint of its internal state into durable storage, e.g., in-memory databases or file systems, which triggers a backup trimming process on all adjacent upstream tasks to delete outdated backup data.

Upon the failure, a task will get restarted and a stateful task will get reset to its previous checkpointed state. Corresponding backup data are then replayed from upstream task(s) to the recovering task.

3.3 Correlated Failure Effect Model

The recovery process of a task introduces a recovery latency (h_v) that consists of two parts[17]: 1) upstream latency (r_v), the time consumed on retrieving backup data, and 2) reprocessing latency (t_v), the time spent on reprocessing data.

$$h_v = r_v + t_v. \tag{1}$$

Note that this mode considers neither the restart time nor the reset time of a failed task, because they are much shorter compared with reprocessing latency.

The reprocessing latency t_v of a task is related to its checkpoint interval. Given the failure arrival rate and the checkpoint cost (depending on the size of the internal state), the optimal checkpoint interval of a task can be computed[35]. Both stateless tasks and stateful tasks can be involved in the recovering process and cause recovery latencies. The reprocessing latency t_v can be estimated (approximately half the checkpoint interval)[11,34,35]. In this paper, we assume the reprocessing latencies of each task are given as an input.

We propose the correlated failure effect model. Let U_v denote the set of adjacent upstream tasks of task v. When all tasks in U_v are healthy in the moment as v is recovering, task v is able to obtain backup data right away, i.e., there is no upstream latency ($r_v = 0$). Otherwise, when upstream task $u \in U_v$ fails at the same time as v, the backup data on u is lost. Upon such correlated failures, downstream task (v) must wait for its dependent upstream task (u) to finish recovery first before the necessary backup data can become available again. Let binary variable f_v denote the healthy status of a task whose value is equal to 1 when task v has failed, and 0 otherwise. The upstream latency is defined recursively as (2).

$$r_v := \begin{cases} 0, & \text{if } \forall u \in U_v : f_u = 0, \\ \max\limits_{u \in U_v, f_u = 1} h_u, & \text{otherwise.} \end{cases} \tag{2}$$

When multiple upstream tasks in U_v fail at the same time as v, the upstream latency of v is equal to the largest recovery latency among those tasks. Moreover, when multiple adjacent tasks on the same path fail together, the effect of delay can be cascading. In Fig.2, task e must wait until both c and d get recovered, that is, when the reprocessing latencies of tasks c and d are 1 and 2 respectively, we have $h_e = h_d + 4$ and $h_d = h_c + 2$. The notations used in this paper are listed in Table 1.

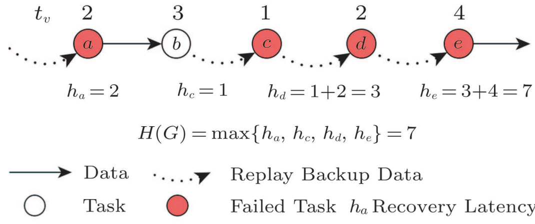


Fig.2. Recovery latency of a stream topology under multiple correlated task failures.

Table 1. Notations

Notation	Description
$G(V, A)$	DAG that represents a stream topology
w_v	Weight of $v \in V$, reflexing the computational capacity needed by a task
P	Set of processors
Φ	Task allocation function, $\Phi(v) = p, v \in V, p \in P$
t_v	Reprocessing latency of $v \in V$
r_v	Upstream latency of $v \in V$
h_v	Task recovery latency of $v \in V$
$H(G)$	Recover latency of a stream topology
U_v	Set of adjacent upstream tasks of v
K_p	Set of tasks assigned to processor p

3.4 Recovery Latency Under Processor Failure

When a processor fails, all tasks located on the same processor fail altogether. We assume that the processors have independent failure rates and multiple processors may fail at the same time. K_p denotes the set of tasks placed on processor p, $K_p = \{v|v \in V, \Phi(v) = p, p \in P\}$. Since tasks in K_p always fail together when processor p fails, $r_v = \max\{\max_{u \in U_v, \Phi(u)=\Phi(v)} h_u, 0\}$. The recovery latency of a processor p under a failure is

$$H(p) = \max_{v \in K_p} h_v. \quad (3)$$

The reprocessing latency of a stream topology is equal to the largest recovery latency of any processor, e.g., $H(G) = \max_{p \in P} H(p) = \max_{v \in V} h_v$.

As shown in Fig.3, different task allocation plans may lead to different recovery latencies. In case (a), all tasks are packed into one processor $K = \{a, b, c, d\}$, and it introduces no waste of the width of a bin. It illustrates how the 2SP approaches would pack items. However, putting adjacent tasks on one processor causes correlated task failures. It introduces high upstream latencies and increase the recovery latency of the processor. In cases (b) and (c) tasks are spread to two processors to avoid high upstream latencies. Case (c) achieves the smallest recovery latency. As illustrated in the figure, there is a trade-off between the recovery latency of the stream topology and the number of resources used. Using more processors gives the resource manager more space to keep tasks with backup dependencies apart and avoid extra recovery latencies but introduces resource waste.

3.5 Task Allocation with Recovery Latency Guarantees

Problem 2 (Recovery-Latency-Aware Task Allocaiton Problem (RTAP)). *Given a stream topology graph $G(V, A)$, the available processor set P, and the recovery latency upper bound $\bar{H}$, find the task allocation Φ that occupies the minimum number of processors while satisfying recovery latency constraints.*

A processor and its resource capacity are hereafter referred to as a bin and the width of the bin respectively. A task and its weight are hereafter referred to as an item and the width of the item respectively. The height of a bin represents the recovery latency threshold

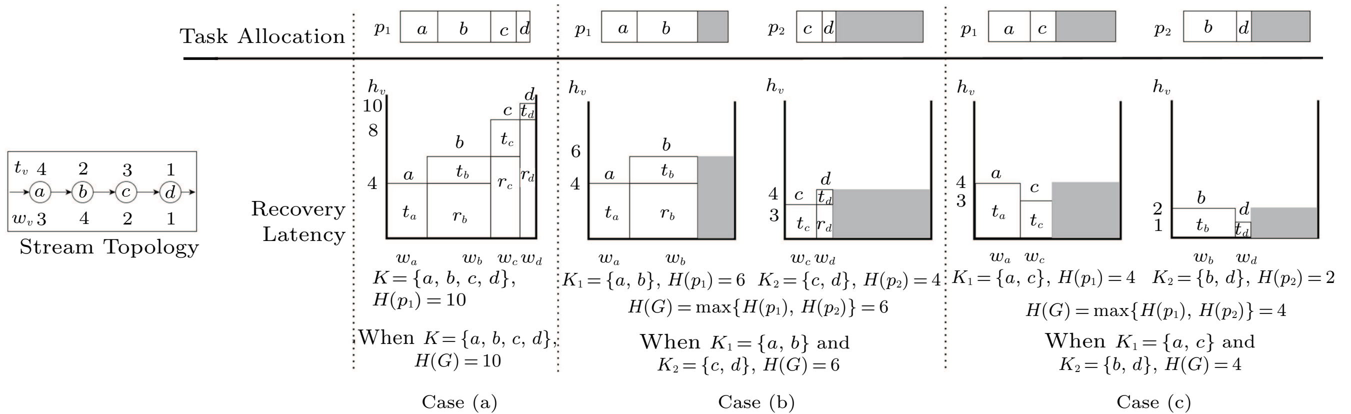


Fig.3. Different task allocation plans.

$\bar{H}$. The height of an item corresponds to the reprocessing latency (t_v) of a task. We assume, without loss of generality, that the height and the width of a bin (item) are both normalized to 1. We use a set of binary decision variables x_{ij} to represent task assignments. x_{ij} is equal to 1 if task v_i is assigned to processor p_j, i.e., $\Phi(v_i) = p_j$, and 0 otherwise. We use a set of auxiliary binary variables y_j to represent whether a processor p_j is assigned with at least one task. Then a valid model for the RTAP corresponds to (4)–(9).

$$\min \quad Y = \sum_{j=1}^{m} y_j, \tag{4}$$

subject to

$$\sum_{i=1}^{n} w_i x_{ij} \leqslant 1, \qquad j \in \{1, ..., m\}, \tag{5}$$

$$\sum_{j=1}^{m} x_{ij} = 1, \qquad i \in \{1, ..., n\}, \tag{6}$$

$$H(G) = \max_{v \in V} h_v \leqslant \bar{H}, \tag{7}$$

$$y_j \in \{0, 1\}, \qquad \forall j \in \{1, ..., m\}, \tag{8}$$

$$x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., n\}, \forall j \in \{1, ..., m\}. \tag{9}$$

The objective function seeks to minimize the number of occupied processors. Constraint (5) guarantees that if a processor is used, then the sum of the weights of the tasks allocated to that processor does not exceed its capacity. Constraint (6) states that each task must be assigned to exactly one processor. Constraint (7) ensures that the recovery latency of the stream topology does not exceed the upper bound $\bar{H}$. The recovery latency of a processor is computed based on the failure effect model proposed in Subsection 3.4. Constraints (8) and (9) impose variables to be binary.

For the sake of better understanding, we use items (bins) and task (processors) interchangeably hereafter. The width of a bin corresponds to the resource capacity of a processor. The width of an item corresponds to the weight of a task. The height of a bin represents the recovery latency threshold $\bar{H}$. The height of an item corresponds to the reprocessing latency (t_v) of a task. We assume, without loss of generality, that the height and the width of a bin (item) are all normalized to 1.

3.6 Assumptions

In this paper we make the following assumptions. Firstly, we assume the reprocessing latency of each task is given as input. They can be estimated based on methods proposed in [11, 34, 35], given certain system parameters, based on the method proposed in [27] using historical data. Secondly, we do not consider the FT overhead and its relationship with the recovery latency, which is studied in our previous work[17]. Finally, we only consider the computational costs of tasks in this paper. This is motivated from the related work[10,44] for computational-constrained applications and the low communication cost scenario. We plan to further consider communication costs among processors in our future work.

4 Approach

4.1 Overview

The RTAP problem is NP-hard since it generates the classic bin packing problem (BPP) when $\bar{H}$ in (7) is sufficiently big. The RTAP problem also generates the bin packing problem with conflicts (BPPC)[46], when $\forall A(i,j) \in A,\ t_i + t_j > \max_{v \in V} t_v$ and $\bar{H} = \max_{v \in V} t_v$. However, RTAP is different from BPCC where packing conflicts in RTAP are non-rigid and tasks can be put into the same processor when the recovery latency constraint (7) holds. For example in case (b) of Fig.3, putting a and b on the same processor increases the recover latency of task b. But a and b can be put together without breaking the overall recovery latency constraint.

In this paper, we aim to solve the RTAP in the general case. Let C_G denote the set of all paths from the source to the sink in stream topology G. Let $H^{MaxPath}(G) = \max_{C \in C_G} \sum_{v \in C} t_v$ denote the largest recovery latency of a path. When $\min_{v \in V} t_v \leqslant \bar{H} \leqslant H^{MaxPath}(G)$, the most similar variation of the packing problem to RTAP is the two-dimensional strip packing problem (2SP)[39]. The main difference between RTAP and 2SP is two-fold: 1) there are no height constraints on 2SP on each level but RTAP puts an extra constraint of maximum height; 2) the recovery height of each item in RTAP is not fixed but related to the task allocation plan. In RTAP, each bin contains at most one level; therefore we call it a single-level two-dimensional bin packing problem (SL2BP).

There are few efficient exact algorithms for BPP. Any exhaust search algorithm for RTAP will also be impractically time-consuming. We focus on the approach of the problem using a fast heuristic algorithm for the offline scenario, where the width and the initial height of each item are given as input. The main challenge is that the item heights (recovery latency) are related to

both the partial solution (task allocation plan) and the stream topology (G). We first propose three greedy algorithms based on well-known 2SP approaches (NFDH, FFDH and BFDH)[39], which are used as benchmarks in later experiments, and then propose a topology-aware heuristic algorithm.

4.2 2SP-Based Algorithms

Next-fit decreasing height (NFDH), first-fit decreasing height (FFDH), and best-fit decreasing height (BFDH) are widely-applied "level-oriented" algorithms[40] for the 2SP problem. For details of these algorithms, please refer to [39]. We design greedy algorithms based on these 2SP algorithms. These algorithms sort items in descending order according to their heights, put them into a queue, and pack one item at a time into rows, forming levels. They differ in the ways of choosing bins to put an item in.

We use the similar sorting methods to sort items and break ties by decreasing the width. These algorithms then pack one item at each step based on the 2SP strategies. At each round, before putting an item into a bin, a function is applied to update the current level height according to (3), which is the height of the highest item in the bin. This function introduces an extra $O(\log n)$ time complexity. Then, the estimated recovery latency of the current item is examined according to (7).

These greedy algorithms are called A^{NFDH}, A^{FFDH}, and A^{BFDH} according to the 2SP strategies they use. The time complexities of these algorithms are $O(n\log^2 n)$. Note that these greedy algorithms ensure recovery latency.

4.3 Heuristic Algorithms

The proposed approach is based on similar framework as popular approaches for 2SP where items are sorted and put into a queue and the item at the head of the queue is packed to a bin at each round. The proposed heuristic $Algorithm^{RTAP}$ (Algorithm 1) sorts items in descending order according to packing "hardness". It is designed according to the observation that tasks with more adjacent tasks are more likely to cause correlated failures.

We introduce a new metric for sorting the items: the weighted upstream degree (WUD) ((10)). Fig.4 displays the WUD values of the tasks in the exampled stream topology. The sorting metrics are listed in Table 2. Item 3 is ordered in front of item 2 because it is harder to pack.

$$WUD = \frac{|U_v|}{n} \times \sum_{u\in U_v} t_u. \quad (10)$$

Algorithm 1. $Algorithm^{RTAP}(G, P, \bar{H})$

Input: a stream topology graph $G(V, E)$, processor set P, and recovery latency constraint $\bar{H}$
Output: task allocation Φ

1 Sort items according to (10)
2 Group items into $\{G'1, G'2, ..., G'_q\}$
3 **foreach** $i \in \{1, ..., q\}$, *Group* G'_i **do**
4 Apply 1SP algorithm to G'_i according to (1) and constraint (7) to compute Φ_i
5 $\Phi := \Phi \cup \Phi_i$
6 **end**
7 **return** Φ

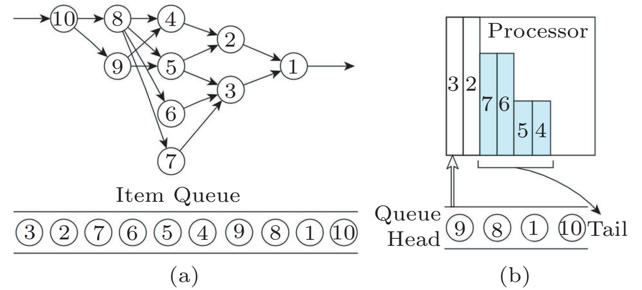


Fig.4. Example of stream topology and packing process. (a) Stream topology and item queue. (b) Packing.

Table 2. Sorting Metrics in Fig.4

v	t_v	WUD_v
1	0.5	0.08
2	0.2	0.32
3	0.2	0.48
4	0.8	0.20
5	0.8	0.20
6	0.5	0.20
7	0.5	0.20
8	0.5	0.90
9	0.5	0.90
10	0.9	0.00

When a task is packed into a bin with its adjacent upstream tasks, its upstream latency increases and extra height is introduced into itself according to (1), which may make it unfit for the bin. Even worse, when a task is packed with its adjacent downstream tasks, extra heights are introduced to downstream those tasks.

This may result in changes in the current partial solution and in the backtracking of tasks as they become unfit for the bin. Fig.4(b) shows an example of this situation. In order to pack item 9 into the current bin, extra heights will be added to items 7, 6, 5, and 4, which makes them unfit for the bin. They will be picked out and put back into the item queue for further decisions.

Note that this backtracking is caused by both correlated failures and the recovery latency constraints, as shown in (7). In order to avoid these situations and accelerate the packing process, items are first ordered by their WUD values, breaking ties by breadth-first traversal orders, and then partitioned into groups. The partitioning seeks to avoid putting tasks that may break recovery latency in the same group. Fig.5 illustrates two cases where tasks are partitioned into one group and two groups respectively according to different latency constraints. We will discuss the effects of the number of groups on packing results in Subsection 5.3.3.

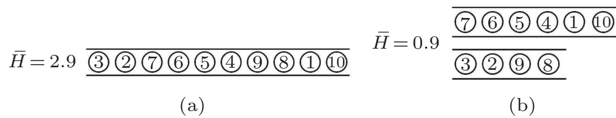


Fig.5. Examples of sorting and partitioning items. (a) One group. (b) Two groups.

Computing the weighted upstream degree for each item and sorting items in step 1 of Algorithm 1 introduce $O(\log n)$ computations. It also takes $O(\log n)$ computations to pack each item, including examining current packing solution and updating items' estimated heights. Therefore, the computational complexity of Algorithm 1 is $O(n \log^2 n)$. We omit the details in this paper.

5 Experimental Results

We conduct several simulations to illustrate: 1) the performance of the proposed algorithms compared with 2SP-based algorithms, 2) the efficiency of our approach for different types of stream topologies and various parameter settings (resource requirements and recovery latencies), and 3) the scalability of the proposed approach and how it can cope with real-world applications.

5.1 Experimental Settings

In this subsection, we use both 2SP-based greedy algorithms and group-based heuristics, as listed in Table 3. For the complexity of the BP problem, we use 2SP-based algorithms ($A^{*\mathrm{DH}}$) as benchmarks. Moreover, we implement the heuristic algorithm in combination with three level-oriented strategies to explore the effects of different packing methods.

Table 3. Compared Algorithms

Algorithm	Description
A^{Exhaust}	An exact search algorithm
A^{NFDH}	Next fit decreasing height
A^{FFDH}	First fit decreasing height
A^{BFDH}	Best fit decreasing height
A^{RTAPNF}	Next-fit group
A^{RTAPFF}	First-fit group
A^{RTAPBF}	Best-fit group

We choose four types of stream topologies for the experiments. "Tree" topology (Fig.6(a)) is a dummy topology where each task has at least one upstream task and exactly one downstream task. The degrees

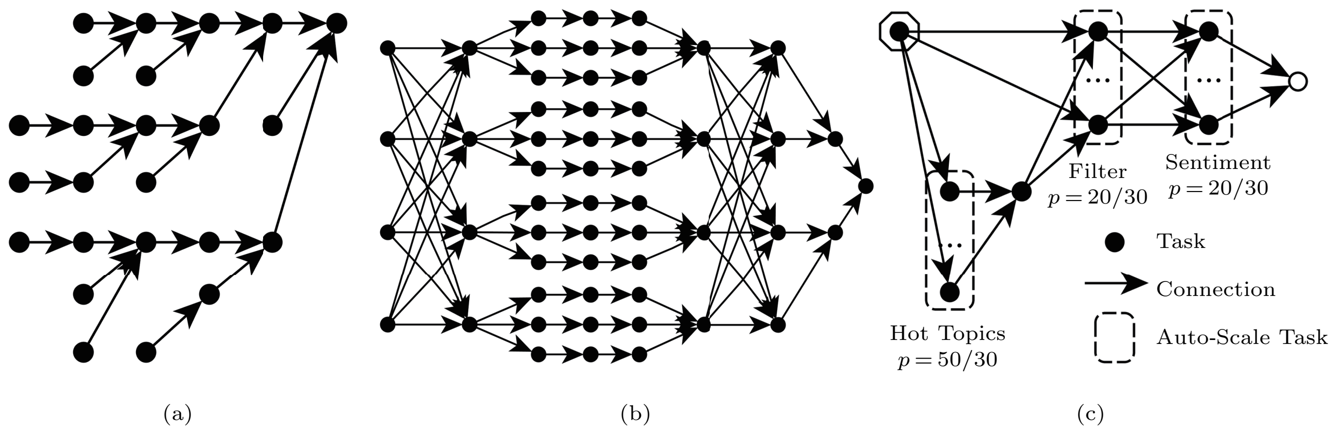


Fig.6. Stream topology examples. (a) Tree topology. (b) Sequential-dominated topology. (c) Parallel-dominated topology.

of each task can be easily controlled. This topology can be used to collect data in a monitor network. We choose two topologies from real stream processing applications in the literature. "Guru" topology (SignalGuru[47]) is a sequential-dominated topology. It collects traffic signals from mobile phones and predicts traffic signal schedules. As shown in Fig.6(a), it has longer paths than other test instances. We use this topology to demonstrate how our approach copes with scenarios where multiple adjacent tasks placed on the same processor introduce large upstream latencies. "Senti" topology (Twitter Sentiment[15]) is a parallel-dominated topology with a large number of parallel tasks and edges caused by auto-scaling. This topology is used to demonstrate the scenarios where tasks in the stream topology DAG have higher in-degrees or out-degrees and are therefore more likely to introduce extra upstream latencies. As shown in Fig.6(c), "Senti" topology contains three auto-scale tasks (hot topics, filter, and sentiment). Finally, we generate random topology instances to test the same topology with different item sizes.

Each of the four topologies is further extended into two test cases with different numbers of tasks ($|V|$) and edges ($|A|$). We use task auto-scale to generate test cases for "Guru" and "Senti" topologies. We test two instances for "Senti" topology, where the auto-scale parameters are set to (50, 20, 20) and (30, 30, 30) resulting in 560 edges and 1 050 edges respectively. Note that the numbers of tasks in both cases are kept the same to illustrate the effect of different cases of parallelism.

Furthermore, we introduce three more parameters to control the topology:

• the average width of tasks, denoted by $\alpha = Ave(w_v)$;

• the average height of tasks, denoted by $\beta = Ave(t_v)$;

• the average degree of tasks, denoted by γ.

A topology with smaller α and β means the items have smaller sizes. They are more likely to be placed in a small number of bins. As discussed in Subsection 3.5, the bin's width and height are set to 1 for simplicity and the item's width and height are normalized to 1 accordingly. Instead of changing the maximum recovery latency (7), we fix the bin height to 1, representing the recovery latency threshold, and change different item heights (β) in each test instance. Note that β denotes the original height (reprocessing cost) of an item. Putting tasks with backup dependency on the same processor will introduce upstream latency and increase the heights of items. Parameter γ represents the possibility of such a case, which is decided by $|A|$ and $|V|$. The test instances we use are listed in Table 4.

Table 4. Stream Topology Instances

Type	$\|V\|$	$\|A\|$	α	β	γ
S-Tree	33	32	0.3	0.4	2.0
L-Tree	220	219	0.2	0.3	2.0
S-Guru	55	95	0.3	0.5	3.1
L-Guru	127	239	0.2	0.2	3.6
S-Senti	93	560	0.2	0.2	7.0
L-Senti	93	1 050	0.2	0.2	22.5
S-Rnd	200	400	0.2	0.2	4.0
L-Rnd	200	400	0.6	0.6	4.0

5.2 Simulation Data

We test the proposed algorithms on all test instances to measure performance, in terms of the number of processors (#b) used and the algorithm execution time (AET), as shown in Table 5. Then, we show the quality of the proposed algorithms with the resource waste of each test case. We offer detailed results on the wastes of width, height, and area of each bin. Note that in our tests, item width and item height, as well as bin width and bin height are normalized to 1 to simplify the results. Moreover, we show the results and highlight the relationship between the number of groups and the number of processors to illustrate the effect of the grouping function (step 1 of Algorithm 1).

5.3 Results and Analysis

5.3.1 *Number of Processors*

The objective of RTAP is to compute a task allocation plan (Φ) for the given stream topology under recovery latency constraints. The number of processors used by Φ is the primary evaluation metric for different algorithms. Table 5 lists the performance results of all six algorithms on eight topology instances. Among the three greedy 2SP-based algorithms, $A^{\rm NFDH}$ performs the worst while $A^{\rm BFDN}$ has the best result. Using a group function based on topology information, the heuristic algorithms proposed in this paper outperform the 2SP-based algorithms. $A^{\rm RTAPBF}$ uses the best-fit strategy after a grouping process and has the best results. Overall, the proposed group-based heuristics use 15%–25% fewer processors than the 2SP-based benchmarks.

Table 5. Performance of Different Algorithms

Algorithm	S-Tree		L-Tree		S-Guru		L-Guru		S-Senti		L-Senti		S-Rnd		L-Rnd	
	#b	AET	#b	AET	#b	AET	#b	AET	#b	AET	#b	AET	#b	AET	#b	AET
$A^{\rm FFDN}$	30	22	125	130	32	22	82	14	38	22	45	23	101	45	184	49
$A^{\rm NFDN}$	36	98	149	93	38	12	89	12	47	13	54	13	125	34	198	35
$A^{\rm BFDN}$	29	11	113	131	32	17	81	14	38	18	45	16	100	49	178	50
$A^{\rm RTAPFF}$	23	37	135	329	27	68	72	37	32	33	40	48	92	147	179	139
$A^{\rm RTAPNF}$	44	51	165	268	39	40	84	49	43	38	56	47	142	131	201	130
$A^{\rm RTAPBF}$	21	37	101	305	25	33	71	42	30	35	41	48	90	141	169	128

Note: The unit of algorithm execution time (AET) is millisecond. #b: the number of bins.

The grouping function involves an extra searching procedure in each task, and this can be time-consuming when the degree of the topology (γ) is large. However, the proposed algorithms have a polynomial time complexity so that they can compute task allocation plans for the test instances in milliseconds.

5.3.2 *Resource Waste*

Next we analyze the resource waste introduced by each algorithm. We assume each bin has a unit width, height, and area. Item width and item height are given as parameters α and β respectively in Table 4. We measure three waste results, W, H and A denoting the waste of bin width, height, and area respectively. Table 6 and Table 7 show the result on each topology instance. $A^{\rm RTAP*}$ has better #b results in Table 5 and has less width and area waste. For example, $A^{\rm RTAPFF}$ and $A^{\rm RTAPBF}$ both introduce a small resource waste. On the contrary, $A^{\rm RTAPNF}$ and $A^{\rm NFDH}$ do not look through current bins for the best result, which leads to more resource waste. Note that S-Senti and L-Senti topologies have the same number of vertices but different numbers of edges. More edges lead to larger γ, which means tasks on the same processors are more likely to have backup dependencies. Therefore, L-Senti wastes more resources than S-Senti on all algorithms.

We can find similar results in S-Rnd and L-Rnd for a different reason. These two topologies are the same except that the items in L-Rnd have larger sizes than the ones in S-Rnd, which makes L-Rnd harder to pack and introduces more resource waste.

Table 6. Resource Utilization (Tree and Guru Topologies)

Algorithm	S-Tree			L-Tree			S-Guru			L-Guru		
	W	H	A	W	H	A	W	H	A	W	H	A
$A^{\rm FFDN}$	14	78	11	6	84	6	12	68	9	12	81	9
$A^{\rm NFDN}$	28	75	21	22	84	19	24	72	17	26	84	22
$A^{\rm BFDN}$	11	79	9	5	84	4	10	74	7	10	82	8
$A^{\rm RTAPFF}$	11	78	9	5	84	5	10	78	7	10	83	8
$A^{\rm RTAPNF}$	41	81	34	38	85	33	26	68	18	28	84	23
$A^{\rm RTAPBF}$	11	72	8	3	85	2	7	74	5	7	81	5

Note: W, H and A denote the resource waste percentage of the width, height and area of a bin respectively.

Table 7. Resource Utilization (Senti and Random Topologies)

Algorithm	S-Senti			L-Senti			S-Rnd			L-Rnd		
	W	H	A	W	H	A	W	H	A	W	H	A
$A^{\rm FFDN}$	2	43	1	4	83	5	7	76	4	6	91	6
$A^{\rm NFDN}$	34	44	14	22	84	18	24	70	17	23	89	21
$A^{\rm BFDN}$	2	44	1	4	82	4	4	75	3	6	81	4
$A^{\rm RTAPFF}$	2	41	1	3	80	2	5	73	3	3	89	4
$A^{\rm RTAPNF}$	37	43	16	20	80	19	32	76	24	32	89	29
$A^{\rm RTAPBF}$	2	41	1	3	75	2	3	62	2	4	79	3

Note: W, H and A denote the resource waste percentage of the width, height and area of a bin respectively.

5.3.3 Number of Groups

The grouping procedure in step 1 of Algorithm 1 partitions items into different groups that share no interconnections with one another. Only items in the same group can be packed into one bin. Fig.7 illustrates the relationship between the number of groups and the number of bins. Four topologies are tested and we show the best and the worst case (with *) of each test instance. We show the results from large topologies. As we can see, the packing results are worse when items are partitioned into more groups. This is because tasks that are partitioned into different groups will not be packed into the same processor. Let G'_g denote a group, and then v_i can be put into G'_g if and only if $\forall v_j \in G'_g, h_j + t_i \leqslant \bar{H}$. Two tasks in two groups are not necessarily backup-dependent and there is a possibility that they can be packed into the same processor. As a result, more partition groups lead to much stricter constraints and therefore can waste more resource.

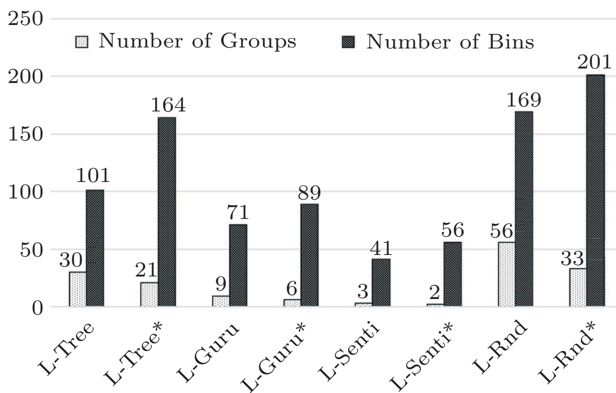


Fig.7. Relationship between the number of groups and the number of bins. $*$ indicates the worst cases.

5.4 Application Senario

The algorithms proposed in this paper are able to compute results within milliseconds, which makes them practical in real production environments. First, they can be used to generate efficient task allocation decisions with guaranteed reliability. For example, if the user wants to analyze the sentiment of the latest tweets using "Senti" topology[15], the user can submit a DAG topology representing the requirements and the approach proposed in this paper can be used by a stream processing platform to make task allocation decisions. Second, although our method is based on static parameters, such as task weight and reprocessing latency, the proposed algorithms can also be applied to make fast reallocation decisions on the fly in dynamic environments, e.g., data streams with fluctuate workloads. The proposed algorithms are time-efficient. They can be executed periodically to evaluate the task allocation and make fast adjustments accordingly. Finally, the proposed method can also be used as a tool to analyze the performance of a system. For example, when we fix the number of processors, representing limited resource, there is a trade-off between the failure-free processing performance and the recovery performance when a failure occurs. We plan to study this issue in our future work.

6 Conclusions

This paper focuses on a task allocation strategy for distributed stream processing systems. We proposed a novel, quantitative, correlated failure effect model to describe the relationship between the recovery latency and task allocation plans (the packing of tasks into processors) of a stream topology. We introduced the recovery-latency aware rask allocation problem (RTAP) based on the failure effect model. We showed that RTAP is closely related to 2BP and 2SP but with unique characteristic of variable item heights. We proposed an approach to compute the task allocation plan with a recovery latency guarantee and a time complexity of $O(n \log^2 n)$. We showed that the proposed method is more efficient (15%–20% on average) compared with related 2BP and 2SP methods.

References

[1] Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 2005, 34(4): 42-47.

[2] Arasu A, Babcock B, Babu S, Datar M, Ito K, Nishizawa I *et al.* STREAM: The Stanford stream data manager (demonstration description). In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2003, pp.665-665.

[3] Hesse G, Lorenz M. Conceptual survey on data stream processing systems. In *Proc. the 21st IEEE International Conference on Parallel and Distributed Systems*, January 2015, pp.797-802.

[4] Chandrasekaran S, Cooper O, Deshpande A *et al.* TelegraphCQ: Continuous dataflow processing. In *Proc. the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003, pp.668-668.

[5] Akidau T, Balikov A, Bekiroğlu K, Chernyak S *et al.* MillWheel: Fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment*, 2013, 6(11): 1033-1044.

[6] Toshniwal A, Taneja S, Shukla A, Ramasamy K *et al.* Storm@ Twitter. In *Proc. ACM SIGMOD International*

Conference on Management of Data, June 2014, pp.147-156.

[7] Neumeyer L, Robbins B, Nair A, Kesari A. S4: Distributed stream computing platform. In *Proc. IEEE International Conference on Data Mining Workshops*, Dec. 2010, pp.170-177.

[8] Kulkarni S, Bhagat N, Fu M, Kedigehalli V *et al.* Twitter heron: Stream processing at scale. In *Proc. ACM SIGMOD International Conference on Management of Data*, May 2015, pp.239-250.

[9] Zhao J, Ou S, Hu L, Ding Y, Xu G. A heuristic placement selection approach of partitions of mobile applications in mobile cloud computing model based on community collaboration. *Cluster Computing*, 2017, 20(4): 3131-3146.

[10] Eidenbenz R, Locher T. Task allocation for distributed stream processing. In *Proc. the 35th Annual IEEE International Conference on Computer Communications*, April 2016.

[11] Hwang J, Balazinska M, Rasin A, Cetintemel U, Stonebraker M, Zdonik S. High-availability algorithms for distributed stream processing. In *Proc. the 21st IEEE International Conference on Data Engineering*, April 2005, pp.779-790.

[12] Babcock B, Babu S, Motwani R, Datar M. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2003, pp.253-264.

[13] Cardellini V, Grassi C, Presti L F, Nardelli M. Optimal operator placement for distributed stream processing applications. In *Proc. the 10th ACM International Conference on Distributed and Event-Based Systems*, June 2016, pp.69-80.

[14] Chatzistergiou A, Viglas S D. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proc. the 23rd ACM International Conference on Information and Knowledge Management*, Nov. 2014, pp.1579-1588.

[15] Lohrmann B, Janacik P, Kao O. Elastic stream processing with latency guarantees. In *Proc. the 35th IEEE Distributed Computing Systems*, June 2015, pp.399-410.

[16] Li H, Wu J, Jiang Z, Li X, Wei X, Zhuang Y. Integrated recovery and task allocation for stream processing. In *Proc. the 36th IEEE Performance Computing and Communications Conference*, Dec. 2017.

[17] Li H, Wu J, Jiang Z, Li X, Wei X. Minimum backups for stream processing with recovery latency guarantees. *IEEE Transactions on Reliability*, 2017, 66(3): 783-94.

[18] Sun D, Zhang G, Wu C, Li K, Zheng W. Building a fault tolerant framework with deadline guarantee in big data stream computing environments. *Journal of Computer and System Sciences*, 2017, 89: 4-23.

[19] Krempl G, Žliobaite I, Brzeziński D, Hüllermeier E *et al.* Open challenges for data stream mining research. *ACM SIGKDD Explorations Newsletter*, 2014, 16(1): 1-10.

[20] Li H, Wu J, Jiang Z, Li X, Wei X. Task allocation for stream processing with recovery latency guarantee. In *Proc. IEEE International Conference on Cluster Computing*, Sept. 2017, pp.379-383.

[21] Ananthanarayanan R, Basker V, Das S, Gupta A, Jiang H, Qiu T, Reznichenko A, Ryabkov D, Singh M, Venkataraman S. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2013, pp.577-588.

[22] Qian Z, He Y, Su C, Wu Z, Zhu H, Zhang T, Zhou L, Yu Y, Zhang Z. TimeStream: Reliable stream computation in the cloud. In *Proc. the 8th ACM European Conference on Computer Systems*, Apr. 2013, pp.1-14.

[23] Su L, Zhou Y. Tolerating correlated failures in massively parallel stream processing engines. In *Proc. the 32nd IEEE International Conference on Data Engineering*, May 2016, pp.517-528.

[24] Upadhyaya P, Kwon Y, Balazinska M. A latency and fault-tolerance optimizer for online parallel query plans. In *Proc. ACM SIGMOD International Conference on Management of Data*, Jun. 2011, pp.241-252.

[25] Fernandez C R, Migliavacca M, Kalyvianaki E, Pietzuch P. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2013, pp.725-736.

[26] Balazinska M, Balakrishnan H, Madden S R, Stonebraker M. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 2008, 33(1): Article No. 3.

[27] Heinze T, Zia M, Krahn R, Jerzak Z, Fetzer C. An adaptive replication scheme for elastic data stream processing systems. In *Proc. the 9th ACM International Conference on Distributed Event-Based Systems*, June 2015, pp.150-161.

[28] Stanoi I, Mihaila G, Palpanas T, Lang C. WhiteWater: Distributed processing of fast streams. *IEEE Transactions on Knowledge and Data Engineering*, 2007, 19(9): 1214-1226.

[29] de Matteis T, Mencagli G. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software*, 2017, 127: 302-319.

[30] Wu J. Distributed System Design. CRC Press, Inc., 2017.

[31] Wu J, Huang K. The balanced hypercube: A cube-based system for fault-tolerant applications. *IEEE Transactions on Computers*, 1997, 46(4): 484-90.

[32] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink$^{\text{TM}}$: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015, 36(4): 28-38.

[33] Zhuang Y, Wei X, Li H, Wang Y, He X. An optimal checkpointing model with online OCI adjustment for stream processing applications. In *Proc. the 27th IEEE International Conference on Computer Communication and Networks*, July 2018.

[34] Salama A, Binnig C, Kraska T, Zamanian E. Cost-based fault-tolerance for parallel data processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, May 2015, pp.285-297.

[35] Young J W. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 1974, 17(9): 530-531.

[36] Xu H, Xing L, Huang L. Regional science and technology resource allocation optimization based on improved genetic algorithm. *KSII Transactions on Internet & Information Systems*, 2017, 11(4): 1972-1986.

[37] Jin Z, Xu G, Li Y, Liu P. A novel cloud scheduling algorithm optimization for energy consumption of data centres based on user QoS priori knowledge under the background of WSN and mobile communication. *Cluster Computing*, 2017, 20(2): 1587-1597.

[38] Christensen H I, Khan A, Pokutta S, Tetali P. Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review*, 2017, 24: 63-79.

[39] Lodi A, Martello S, Monaci M. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 2002, 141(2): 241-252.

[40] Coffman Jr E G, Csirik J, Galambos G, Martello S, Vigo D. Bin packing approximation algorithms: Survey and classification. In *Handbook of Combinatorial Optimization*, Pardalos P A, Du D Z, Graham R L (eds.), Springer, 2013, pp.455-531.

[41] Johnson D S. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 1974, 9(3): 256-278.

[42] Chung F R, Garey M R, Johnson D S. On packing two-dimensional bins. *SIAM Journal on Algebraic Discrete Methods*, 1982, 3(1): 66-76.

[43] Garey M R, Graham R L, Ullman J D. Worst-case analysis of memory allocation algorithms. In *Proc. the 4th ACM Symposium on Theory of computing*, May 1972, pp.143-150.

[44] Jiang Y, Huang Z, Tsang D H. Towards max-min fair resource allocation for stream big data analytics in shared clouds. *IEEE Transactions on Big Data*, 2016, 4(1): 130-137.

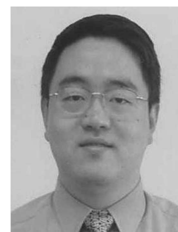
[45] Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing. In *Proc. the 6th Workshop on Networking Meets Databases*, June 2011.

[46] Sadykov R, Vanderbeck F. Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, 2013, 25(2): 244-255.

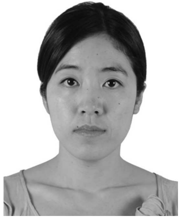
[47] Koukoumidis E, Peh L S, Martonosi M R. SignalGuru: Leveraging mobile phones for collaborative traffic signal schedule advisory. In *Proc. the 9th ACM International Conference on Mobile Systems, Applications, and Services*, June 2011, pp.127-140.

Hong-Liang Li received his Ph.D. degree in computer application technologies from the College of Computer Science and Technology (CCST), Jilin University, Changchun, in 2012. He is currently an associate professor of CCST and a visiting scholar in the Department of Computer and Information Sciences at Temple University, Philadelphia. His research interests include resource scheduling and fault tolerance in HPC systems. He is a member of CCF and IEEE.

Jie Wu received his Ph.D. degree in computer engineering from Florida Atlantic University, Boca Raton, in 1989. Dr. Wu is the Associate Vice Provost for International Affairs at Temple University, Philadelphia. He also serves as the Chair and Laura H. Carnell professor in the Department of Computer and Information Sciences, Temple University, Philadelphia. Prior to joining Tempe University, he was a program director at the National Science Foundation of USA and was a distinguished professor at Florida Atlantic University, Boca Raton. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Service Computing and Journal of Parallel and Distributed Computing. Dr. Wu was general co-chair/chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, and ACM MobiHoc 2014, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker, an overseas board member of CCF, and a fellow of IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.



Zhen Jiang received his B.S. degree in computer engineering from Shanghai Jiao Tong University, Shanghai, in 1992, his M.S. degree in computer engineering from Nanjing University, Nanjing, in 1998, and his Ph.D. degree in computer engineering from Florida Atlantic University, Boca Raton, in 2002. Currently, he is an associate professor in the Department of Computer Science at West Chester University of Pennsylvania (WCU), West Chester, the director of National Security Agency (NSA) Certified Information Security Center at WCU, and an adjunct professor at Temple University, Philadelphia. His research interests are in information system development and wireless communication. He won the Best Paper Award for Protocols and Algorithms in the 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems in 2010. Dr. Jiang is also active in many committees, and he holds membership in IEEE and ACM where he is involved in the organization of many conferences and workshops.



Xiang Li is a Ph.D. candidate and a faculty member of the College of Computer Science and Technology (CCST), Jilin University, Changchun. Her research interests include cloud computing and distributed systems.

Xiao-Hui Wei is a professor and the dean of the College of Computer Science and Technology (CCST), Jilin University, Changchun. He is currently the director of the High Performance Computing Center of Jilin University, Changchun. His current research interests include resource scheduling for large distributed systems, infrastructure level virtualization, large-scale data processing systems, and fault-tolerant computing. He has published more than 50 journal and conference papers in the above areas. He is a distinguished member of CCF and a member of IEEE.