Cost-Effective Sharing of Streaming Dataflows for IoT Applications

Shilpa Chaturvedi, *Student Member, IEEE*, Sahil Tyagi, and Yogesh Simmhan, *Senior Member, IEEE*

Abstract—Internet of Things (IoT) applications are often designed as dataflows that analyze sensor data in real-time to make decisions. Stream processing systems like *Apache Storm* execute these on Cloud infrastructure. As IoT applications within shared data environments like smart cities grow, they will duplicate tasks like pre-processing and analytics. This offers the opportunity to collaboratively reuse the outputs of overlapping dataflows, improving the resource efficiency on Clouds. We propose *dataflow reuse algorithms* that when given a submitted dataflow, identify the intersection of reusable tasks and streams from existing dataflows to form a *merged dataflow*, with guaranteed equivalence of their output streams. Algorithms to unmerge dataflows when they are removed, and defragment partially reused dataflows are also proposed. We implement these algorithms for the Storm fast-data platform, and validate their performance and resource savings using 86 real and synthetic dataflows from eScience and IoT domains. Our reuse strategies reduce the number of running tasks by 34–45 percent and the cumulative CPU usage by 29–63 percent. Including defragmentation of incremental dataflows achieves a monetary savings on Cloud resources of 36–44 percent compared to dataflows without reuse, and has limited redeployment overheads.

Index Terms—Distributed stream processing, dataflow reuse, cloud elasticity, internet of things, fast data

1 Introduction

One of the fast growing sources of data is from *Internet of Things (IoT)* deployments, where sensors and actuators collect observational data from and enact control signals on physical and virtual infrastructure [1]. Besides consumer IoT devices like wearables and smart thermostats [2], Smart Cities offer a canonical use of IoT to provide effective citizen services, and improve the efficiency of city utilities. Such Cyber-Physical Systems (CPS) include *smart power grids* where real-time load data from consumers help with demand-response optimization [3], [4] and *intelligent transportation* where street sensors and camera feeds are used to manage traffic lights, transit frequency, and pricing [5].

Smart City deployments make available *urban streaming data* from millions of sensors,[1] with the need to analyze them to make real-time decisions or provide services. *Distributed Stream Processing Systems (DSPS)* offer a *fast data* platform to compose *continuous dataflow* applications that execute constantly over one or more streams. DSPS like *Apache Storm*, *Flink* and *Spark Streaming* [6], [7], [8] are designed to scale-out across commodity clusters and Cloud Virtual Machines (VMs), and operate on 1000's of *events/sec*. They are often used to compose IoT and Smart City applications hosted on the Cloud, and access sensor streams pulled from the edge into the data-center [9], [10]. E.g., IoT gateway services from Microsoft Azure and Amazon AWS give the ability to push streams to their Cloud-hosted stream processing platforms from edge devices.[2,3]

Motivation. As Smart City installations expand, thousands of public data streams on traffic, pollution, weather, etc. from diverse domains will be available for integration and analysis on the Cloud. One can expect an explosion of innovative services and "apps" that perform online analytics over these streams, and even personalize it for individuals. E.g., an app may correlate weather observation streams (*turning cloudy*) with power grid generation streams (*solar output drop*) to predict when surge-pricing might be triggered by the utility to offset demand. This can help users (or their digital agents) decide to, say, recharge their electric vehicle or start their smart washing machine [4], [10].

Cloud-hosted DSPS will form the scalable analytics engine for composing and executing these continuous dataflows, collocated with the data streams [11]. These compositions are enabled through distributed streaming platforms [12], Functions as a Service (FaaS) [13], and IoT application platforms [14], [15]. At the same time, the numerous dataflows that operate on these shared streams, are likely to duplicate common tasks like data pre-processing (*parsing*, *reformat*, *unit conversion*), quality checks (*cleaning*, *outlier*

1. Urban Data Platform, EU Science Hub, http://urban.jrc.ec.europa.eu

2. Microsoft Azure IoT, https://azure.microsoft.com/en-in/suites/iot-suite/

3. Amazon AWS Greengrass, https://aws.amazon.com/greengrass/

• S. Chaturvedi is with the NetApp Inc., Bangalore, Karnataka 560048, India. E-mail: shilpac@iisc.ac.in.
• S. Tyagi is with the Indiana University, Bloomington, IN 47405 USA. E-mail: sahiltyagi@iisc.ac.in.
• Y. Simmhan is with the Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, Karnataka 560012, India. E-mail: simmhan@iisc.ac.in.

Manuscript received 6 Apr. 2018; revised 24 Mar. 2019; accepted 2 May 2019. Date of publication 6 June 2019; date of current version 6 Dec. 2021. (Corresponding author: Shilpa Chaturvedi.) Recommended for acceptance by Y. Wu. Digital Object Identifier no. 10.1109/TCC.2019.2921371

2168-7161 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

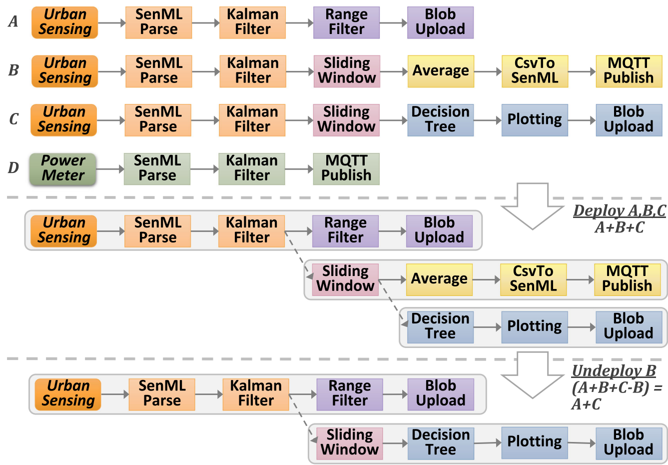


Fig. 1. Illustration of dataflows being merged for reuse on submission, and unmerged on removal.

detection, interpolation), and even analytics (*ARIMA time-series predictions, moving window averages*) [16], [17]. This offers the opportunity to reuse parts of the logic among different dataflows to avoid recomputation, thereby reducing the costs for using Cloud resources for app developers and end-users, and the time to deployment as well.

Gaps. Such scenarios are common in collaboratory scientific and Enterprise environments where datasets and workflows are reused [18], [19]. Scientific projects often make Level 1/2/3 datasets, which have been pre-processed to different degrees using standard routines, available to their user groups. Similarly, repositories like *myExperiment* and *OPMW* allow the definition and reuse of scientific workflows by the broader community [20], [21]. Provenance collected from workflow runs have also been leveraged for data and workflow reuse [22], [23]. Even *Apache Spark* uses lineage to avoid recomputing RDDs [8]. Others have examined stream reuse in wide area networks [24].

While related, the problem we address differs from these prior works in a key aspect. Reuse of workflows and their outputs happens *after* their execution. We instead focus on streaming applications that are *actively running and generating transient data streams*. This requires a greater awareness of the platform runtime, and is performance sensitive.

Contributions. A preliminary version of this article introduced the formalism and algorithms for the reuse of streaming dataflows by merging/unmerging them, and validated it for two sets of dataflows [25]. We extend these here, and include two defragmentation approaches for partially used dataflows, offer a billing model for shared dataflows on Clouds, and include a more detailed evaluation. Specifically, we make the following contributions in this article:

1) We motivate (Section 2) and formally define (Section 3) the problem of streaming dataflow reuse, including the *equivalence* between tasks present in dataflows.
2) We propose algorithms for *merging* a submitted dataflow with the deployed ones at specific overlap points, and *unmerging* a merged dataflow when removed, while guaranteeing their output stream equivalence, in Section 4.
3) We offer *defragmentation strategies* to unmerged partially used dataflows while balancing the cost of (re) deployment in Section 5, and include a user cost model for billing shared dataflows on Clouds in Section 6.
4) We implement these strategies in *Apache Storm*, and *evaluate* their resource and cost savings for real and synthetic workloads of Smart Utility and eScience dataflows in Section 7.

We also review related literature in Section 8, and present our conclusions and future work in Section 9.

2 Problem Description

Continuous or *streaming dataflows* are composed as a *Directed Acyclic Graph (DAG)*, where vertices are user logic or *tasks* and directed edges are *streams* that transfer *events* from the output of a task to the input of a downstream task. Tasks execute once per input event to generate zero or more output events, with the ability to maintain state and aggregate multiple events. Once *deployed* to a DSPS, dataflows execute continuously on their input stream(s) till *undeployed*.

Streaming dataflows are used to compose IoT applications [5], [11], [16]. They operate over streams from *physical sensors* that are publicly available, and publish the output streams in real-time, or persist them to storage. The output streams from each task in the dataflow can be considered as an intermediate stream that has been partially processed through the preceding dataflow tasks. We call these output and intermediate streams that have been processed as *derived streams*, in contrast to the *raw streams* from sensors.

DSPS like Apache Storm and Flink can run multiple concurrently dataflows on a shared commodity cluster or Cloud VMs. Dataflows submitted to Storm execute independently on hosts of the Storm cluster. Tasks from multiple dataflows can be collocated on the same machine, but there is no implicit sharing of events or tasks between dataflows.

IoT dataflows that use the same raw streams as inputs are likely to have similar pre-processing and analytics tasks. As Smart Cities make many urban observations public, startups and citizen scientists will design novel applications for residents, hosted on public or private city-owned Clouds. Hence, it is expected that dataflows with significant overlaps, but running independently, will duplicate efforts.

Fig. 1 illustrates such a scenario where dataflows *A, B, C* and *D* are performing *Extract-Transform-Load (ETL)* and *Statistical Summarization (STATS)* on two streams, from urban sensing and smart power meters [16]. The dataflows differ in overall structure but share similar prefix tasks. E.g., *A, B* and *C* share the raw stream source and the next two tasks, while *B* and *C* share an additional third task. These three dataflows can be "merged" into one dataflow, *A+B+C*, where *B* reuses the derived stream from *A's Kalman Filter* output, and *C* reuses the derived stream from *B's Sliding Window* output. This is equivalent to running the three independently, but avoids duplicate execution of the prefix tasks. While *D* has an overlap with *A*, the source stream is different, and hence they cannot be merged. Similarly, when *B* is undeployed, then an "unmerge" should bring it to *A+C*.

These dataflows may be owned by different users in the IoT community, and collaboratively wish to reuse their execution in a Cloud data center to reduce costs. While these examples show simple sequential dataflows being merged and unmerged, more realistic scenarios will have forks and

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

joins, and dataflows added and removed regularly. While dataflows simplify application composition, manually identifying the overlaps with existing dataflows is infeasible in a Smart City ecosystem with 100's of users and applications.

In this article, we explore algorithms to transparently reuse derived streams in submitted dataflows to reduce resource utilization while guaranteeing that the outputs of the dataflows are identical to the original ones, even when the reused dataflows are removed. There are specific challenges on *correctness and efficiency* that must be addressed.

- We need to *automatically identify* the derived streams in existing dataflows that offer the logical equivalent of a stream in the submitted dataflow. This requires checking that the *ancestors* of the derived stream matches the one in the new dataflow. The raw input stream(s), the task types and their configurations must be identical.
- We must ensure that this *reuse is maximal*, and as far downstream as possible, to take the best advantage of the deployed dataflows.
- We should support the reuse of *multiple derived streams from different dataflows* by the same incoming dataflow.
- When a dataflow is removed, the *unmerging* should retain the correctness of the remaining (merged) dataflows while minimizing the disruption to existing applications.
- We should track the degree of reuse between dataflows to allow *accurate billing* of Cloud resources to users.

Next, we formalize this problem and propose dataflow merge and unmerge algorithms to meet these requirements.

3 Problem Formulation

3.1 Tasks, Streams and Dataflows

An *event* is a discrete unit of data that is uniquely identified by an *event id*, and has a *payload* whose contents are opaque to the DSPS platform. An *abstract task* $\tau = \langle type, config\rangle$ is a user-defined logic defined by its *type*, which executes on one event at a time, and may generate zero or more events for each event consumed. The behavior of the user logic is controlled by the *config* property for the task, such as the window size for an aggregation task or the NoSQL URL for an event storage task. A *stream* transfers events generated from one task to another downstream task for consumption.

Let $\mathcal{T} = \{\tau\}$ be the universal set of abstract tasks. Two such tasks are identical if their type and config are the same,

$$\tau_i = \tau_j \Rightarrow \tau_i.type = \tau_j.type \wedge \tau_i.config = \tau_j.config.$$

Source and *sink tasks* are special abstract tasks that solely generate and consume events, respectively. A source task does not consume an input stream, but produces (raw) events on its output stream based on its logic (e.g., read from a physical sensor), while a sink task consumes an input stream but does not produce an output stream (e.g., persist to a database). Their *type* uniquely identifies the logical name of the source or sink while their *config* has a constant value of 'SOURCE' or 'SINK'. The sets $\mathcal{R} \subset \mathcal{T}$ and $\mathcal{N} \subset \mathcal{T}$ are the universal set of source and sink tasks, with $\mathcal{R} \cap \mathcal{N} = \varnothing$.

Users compose streaming applications as a *dataflow* defined as a DAG, $D = \langle \mathbb{T}, \mathbb{S}\rangle$, where $\mathbb{T} = \{t_1, \dots, t_n\}$ is the set of n *concrete tasks* (or just "tasks") that are the vertices of the DAG, and $\mathbb{S} = \{s_1, \dots, s_m\}$ is the set of m *streams* that are the edges of the DAG. Each task $t_i \in \mathbb{T}$ has a globally unique *id* and matches an abstract task's *type* and *config*,

$$t_i = \langle id, type_p, config_q\rangle \mid \exists \tau = \langle type_p, config_q\rangle \in \mathcal{T}.$$

The same abstract task can appear multiple times as different concrete tasks in the DAG with different *id*'s. A stream $s_k \in \mathbb{S}$ that transfers output events from an upstream task t_i to the input of a downstream task t_j is defined as,

$$s_k = \langle t_i.id, t_j.id\rangle \mid t_i, t_j \in \mathbb{T}.$$

We allow flexible *routing semantics* when multiple streams are incident on the same task or when multiple streams leave the same task, as long as the routing decisions are made locally at that task. E.g., a task may execute once for each event arriving from two streams incident on it in any arbitrary order (interleaved), or events from the streams may be pre-processed (say, pairs joined into one event) before being executed by the task. Similarly, a copy of each output event from a task may be sent on all its output streams (duplicated), or they may be sent only on one based on some event partitioning function.

Two convenience functions return the upstream and downstream tasks an edge is incident on,

$$up(s) = \{t_i \mid s = \langle t_i.id, t_j.id\rangle \in \mathbb{S},\ t_i, t_j \in \mathbb{T}\}$$
$$down(s) = \{t_j \mid s = \langle t_i.id, t_j.id\rangle \in \mathbb{S},\ t_i, t_j \in \mathbb{T}\}.$$

A dataflow has a set of *input tasks*, $\mathbb{I}$, and a set of *output tasks*, $\mathbb{O}$, that form the start and the end boundaries of the DAG, and are from the universal set of abstract source and sink tasks. Given $t_i \in \mathbb{T}, \tau_R \in \mathcal{R}$ and $\tau_N \in \mathcal{N}$, we have,

$$\mathbb{I} = \{t_i \mid t_i.config = \text{SOURCE} \wedge t_i.type = \tau_R.type\}$$
$$\mathbb{O} = \{t_i \mid t_i.config = \text{SINK} \wedge t_i.type = \tau_N.type\}.$$

A DSPS engine continuously executes tasks of the dataflow on distributed resources and orchestrates the event transfer. While our definition makes no assumptions on the runtime or scheduling, our techniques are well-suited for dataflows executed in a single Cloud data center.

3.2 Equivalence

Similarity between Tasks. Say we have two concrete tasks t_i and t_j. They are said to be *type-similar* if $t_i.type = t_j.type$, and denoted as $t_i \overset{\text{T}}{\approx} t_j$. They are said to be *config-similar* if they are type similar, and also $t_i.config = t_j.config$, shown as $t_i \overset{\text{C}}{\approx} t_j$. The tasks are said to be *identical* if $t_i.id = t_j.id$, and given as $t_i = t_j$. Being identical implies that these are the same tasks, and so are config similar too.

Parent of a Task. For a dataflow $D\langle \mathbb{T}, \mathbb{S}\rangle$, we define a *parent function*, $\pi_D : \mathbb{T} \rightarrow \mathscr{P}(\mathbb{T})$, that takes a task as input and returns its *parent set*, which is the set of tasks that are its immediate upstream predecessors in the DAG. The function's range is the power set $\mathscr{P}$ of all tasks. There are no parents for the input task(s) to the dataflow. For $t \in \mathbb{T}$,

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

$$\pi_D(t) = \begin{cases} \{p \mid s \in \mathbb{S}, p \in \mathbb{T}, \\ \ p = up(s), t = down(s)\} & \text{if } t \in \mathbb{T} \setminus \mathbb{I} \\ \varnothing & \text{if } t \in \mathbb{I} \end{cases}.$$

Ancestor Graph. An *Ancestor Graph* for a task in a dataflow is a DAG formed from the task and all its ancestors, along with the streams that connect these tasks within the original dataflow. For a task $t \in \mathbb{T}$ for a dataflow $D\langle\mathbb{T}, \mathbb{S}\rangle$ we have the *ancestor graph recurrence function*, $\alpha_D(t) \rightarrow A\langle\overleftarrow{\mathbb{T}}, \overleftarrow{\mathbb{S}}\rangle \mid \overleftarrow{\mathbb{T}} \subset \mathbb{T}, \overleftarrow{\mathbb{S}} \subset \mathbb{S}$, defined as,

$$\alpha_D(t) = A\langle\{t\},\ \{s \mid s \in \mathbb{S}, down(s) = t\}\rangle\ \cup \bigcup_{p \in \pi_D(t)} \alpha_D(p).$$

Here, we include the current task and its incoming streams in the ancestor graph, recursively apply the ancestor function on the parent set of the task and take the union of the parent's ancestor graph. This will recur till we reach the DAG's input tasks, which do not have input streams. The union of two ancestor graphs $A_i\langle\mathbb{T}_i, \mathbb{S}_i\rangle$ and $A_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle$ is

$$A_k\langle\mathbb{T}_k, \mathbb{S}_k\rangle = A_i \cup A_j = \langle\mathbb{T}_i \cup \mathbb{T}_j, \mathbb{S}_i \cup \mathbb{S}_j\rangle.$$

The ancestor graph for a task indicates the operations that were performed on one or more input tasks of the DAG to derive the input streams to the task. It is similar to the *prospective provenance* of the events generated from that task [22]. Each ancestor graph is connected and forms a DAG. Every task in the dataflow has a unique ancestor graph, and it contains at least one of the input tasks to that dataflow. In a dataflow with a single sink task, the ancestor graph of the sink task is the entire dataflow.

Maximal Ancestor Graph Set. The *ancestor graph set*, $\mathbb{A}$, for a dataflow $D\langle\mathbb{T}, \mathbb{S}\rangle$ is given by, $\mathbb{A} = \{\alpha_D(t) \mid t \in \mathbb{T}\}$.

An ancestor graph $A_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle$ is said to be a *sub-ancestor* of another ancestor graph $A_i\langle\mathbb{T}_i, \mathbb{S}_i\rangle$ if $\mathbb{T}_j \subset \mathbb{T}_i$ and $\mathbb{S}_j \subset \mathbb{S}_i$, and we say that $A_j \subset A_i$. The function Ω gives the *maximal ancestor graph set*, $\widehat{\mathbb{A}}$, for a given dataflow which is the ancestor graph set that only has ancestor graphs that are not sub-ancestors of any other ancestor graph in that dataflow.

$$\widehat{\mathbb{A}} = \Omega(\mathbb{A}) = \{A \mid A \not\subset A', \quad A, A' \in \mathbb{A}\}.$$

Intuitively, the number of ancestor graphs in the maximal ancestor graph set equals the number of sink tasks in that dataflow. This is because the sink being the most downstream of the tasks in the DAG will not appear in any other ancestor graph besides its own. It will also have the most number of tasks in its ancestor graph.

Task and Ancestor Graph Equivalence. If we have $A_i\langle\mathbb{T}_i, \mathbb{S}_i\rangle$ and $A_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle$ as the ancestor graphs for tasks $t_i, t_j \in \mathbb{T}$ in a dataflow $D\langle\mathbb{T}, \mathbb{S}\rangle$, we say that the *ancestors graphs are equivalent*, denoted as $A_i \leftrightarrow A_j$, if there exist two *bijective* functions $\epsilon : \mathbb{T}_i \rightarrow \mathbb{T}_j$ and $\bar{\epsilon} : \mathbb{S}_i \rightarrow \mathbb{S}_j$ such that,

$$\begin{aligned} &\epsilon(t'_i) = t'_j \Rightarrow t'_i \overset{\text{C}}{\approx} t'_j \qquad \mid\ t'_i \in \mathbb{T}_i, t'_j \in \mathbb{T}_j, s'_i \in \mathbb{S}_i, s'_j \in \mathbb{S}_j \\ &\bar{\epsilon}(s'_i) = s'_j \Rightarrow \epsilon(up(s'_i)) = up(s'_j),\ \epsilon(down(s'_i)) = down(s'_j). \end{aligned}$$

In other words, for each task in the ancestor graph of t_i, there should be a distinct task in the ancestor graph of t_j that is config-similar, and vice versa, and their input and output streams should be incident on matching tasks.

Two tasks t_i and t_j are *equivalent*, denoted as $t_i \leftrightarrow t_j$ if their ancestor graphs are equivalent. The output streams of such tasks are identical, and one task can replace the other.

De-Duplicated DAG (De-dup DAG). A *De-Duplicated DAG* $D\langle\mathbb{T}, \mathbb{S}\rangle$ is one in which there exists no two task $t_i, t_j \in \mathbb{T}$ that are equivalent. Each dataflow submitted by the user for execution should be a de-dup DAG.

Disjoint and Overlapping DAGs. Two dataflows $D_i\langle\mathbb{T}_i, \mathbb{S}_i\rangle$ and $D_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle$ are said to be *disjoint*, denoted as $D_i \not\leftrightarrow D_j$, if they do not share any equivalent tasks,

$$D_i \not\leftrightarrow D_j \Rightarrow \nexists t_i \in \mathbb{T}_i, t_j \in \mathbb{T}_j \mid t_i \leftrightarrow t_j.$$

Disjoint dataflows have no tasks that are mutually reusable. Dataflows that are not disjoint are called *overlapping*.

Ancestor Intersection of DAGs. We define the *ancestor intersection* of two DAGs, given as a function $\Lambda(D_i, D_j)$, as the set of ancestor graphs for tasks in each of the DAGs that are ancestor equivalent. Without loss of generality, we choose the ancestor graph from the task in the first DAG for inclusion in the intersection set. Given $D_i\langle\mathbb{T}_i, \mathbb{S}_i\rangle$ and $D_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle$, we have,

$$\Lambda(D_i, D_j) = \{\alpha_{D_i}(t_i) \mid t_i \leftrightarrow t_j\ \forall\ t_i \in \mathbb{T}_i, t_j \in \mathbb{T}_j\}.$$

The ancestor intersection of disjoint DAGs is an empty set.

The *maximal ancestor intersection* finds the maximal set from the returned set of intersecting ancestor graphs,

$$\widehat{\Lambda}(D_i, D_j) = \Omega(\Lambda(D_i, D_j)).$$

This indicates the largest set of equivalent tasks in the two dataflows, and is an upper bound on the reusable tasks.

3.3 Problem Definition

There are *running* DAGs executing in the DSPS, and a user *submits* a DAG for deployment. Our goal is to optimize the set of running DAGs after a submitted DAG is deployed, while ensuring that the outputs from the running DAGs are indistinguishable from those of all the submitted DAGs so far. Users may also *remove* a DAG submitted previously, and here too a similar optimization is required after removal, without affecting the outputs of the remaining DAGs.

Formally, say we have a set of m disjoint and de-dup DAGs, $\overline{\mathbb{D}} = \{\overline{D}_i\langle\overline{\mathbb{T}}_i, \overline{\mathbb{S}}_i\rangle\}$ that are currently *running*, and together represent a collection of $n \geq m$ de-dup DAGs, $\mathbb{D} = \{D_j\langle\mathbb{T}_j, \mathbb{S}_j\rangle\}$, that were *submitted* by users for deployment. The following two *constraints* hold for the system:

1) *Sink Task Coverage*. For each sink task t_p in the dataflows $\mathbb{D}$ submitted by the users, there exists some *equivalent* task t_q in the running dataflows $\overline{\mathbb{D}}$,

$$\forall t_p \in \mathbb{O}_j, \quad \exists t_q \in \overline{\mathbb{T}}_i \mid t_p \leftrightarrow t_q. \tag{1}$$

2) *Task and Stream Minimization*. The running dataflows $\overline{\mathbb{D}}$ must be disjoint and de-dup DAGs, and each task t_q and stream s_r in them, must appear in the ancestor graph for some sink task t_p in the submitted dataflows $\mathbb{D}$,

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

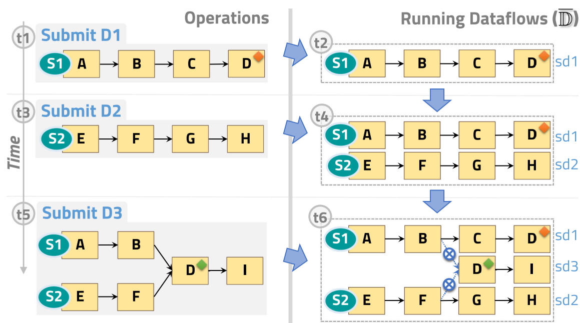


Fig. 2. Example of dataflow *submission* and *merge*.

$$\forall t_q \in \overline{\mathbb{T}}_i, s_r \in \overline{\mathbb{S}}_i \quad \exists t_p \in \mathbb{O}_j \,|\, t_q \in \overleftarrow{\mathbb{T}_p} \wedge s_r \in \overleftarrow{\mathbb{S}_p} \\ \text{where } A_p\langle \overleftarrow{\mathbb{T}_p}, \overleftarrow{\mathbb{S}_p}\rangle = \alpha_{D_j}(t_p). \tag{2}$$

Here, the first constraint guarantees that the running dataflows can produce the identical output streams as the submitted dataflows. The second constraint ensures that there are no more running tasks, and streams connecting them, than what is absolutely needed to satisfy the first constraint. This, coupled with the running dataflows being disjoint, ensures that we execute the least number of tasks while *maximizing reuse*. Given this, our *problems* are,

1) *Merging DAGs*. When a new de-dup DAG D_n is submitted by a user, update the set of running DAGs $\overline{\mathbb{D}}$ such that Constraints 1 and 2 hold for the new set of submitted DAGs $\mathbb{D} \cup D_n$, while ensuring that tasks equivalent to the output tasks of D_n are present in $\overline{\mathbb{D}}$.
2) *Unmerging DAGs*. When a dataflow $D_r \in \mathbb{D}$ that was earlier submitted is now requested to be removed, update the set of running DAGs $\overline{\mathbb{D}}$ such that Constraints 1 and 2 hold for the remaining submitted DAGs $\mathbb{D} \setminus D_r$.

4 Merging and Unmerging Dataflows

4.1 Merging Algorithm

When a dataflow is submitted by the user, we need to check if it overlaps with any running dataflow. If not, then there is no possibility of reuse and the submitted dataflow has to be run independently (Fig. 2, $D1, D2$). If there are overlaps with one or more existing dataflows, we need to identify the *maximum overlapping tasks and streams* that will be reused. We should also locate the *non-overlapping parts* of the submitted dataflow that have to be run afresh, and connected to the upstream tasks being reused.

Running dataflows are disjoint with each other, i.e., they do not share any source tasks. A submitted dataflow with multiple source tasks (and optionally their successors) that are present in different running dataflows may reuse more than one of them. In this case, these running dataflows will be *merged* with the new dataflow's non-overlapping tasks and streams that are instantiated, and connected to form a single running DAG (Fig. 2, $D3$).

We also need to identify the tasks in the running dataflow that correspond to the sink tasks in the submitted dataflow to provide as output to the user. This should be maintained for all submitted dataflows even when we merge running dataflows. Next, we detail these operations for merging a submitted DAG with running ones.

Algorithm 1. Merge Algorithm

1: **procedure** MergeAndDeploy($D_n\langle \mathbb{T}_n, \mathbb{S}_n\rangle$, $\overline{\mathbb{D}}$)
2: Add new DAG to set of *submitted DAGs*, $\mathbb{D} = \mathbb{D} \cup \{D_n\}$
3: Identify DAGs $\mathbb{Y} \subseteq \overline{\mathbb{D}}$ *overlapping* with D_n, where
$\mathbb{Y} = \{\overline{D_i}\langle \overline{\mathbb{T}}_i, \overline{\mathbb{S}}_i\rangle \,|\, t_i \in \overline{\mathbb{T}}_i,\ t_n \in \mathbb{T}_n,$
$t_n.config = \text{Source} \wedge t_n \overset{C}{\approx} t_i\}$
4: **if** $|\mathbb{Y}| = 0$ **then**
5: Deploy dataflow D_n directly, and set $\overline{\mathbb{D}} = \overline{\mathbb{D}} \cup \{D_n\}$.
▷ *Construct overlapping DAGs to merge and reuse*
6: Create *merged DAG* $\overline{D_m}\langle \overline{\mathbb{T}_m}, \overline{\mathbb{S}_m}\rangle$, where
$\overline{\mathbb{T}_m} = \bigcup_{\overline{D_i}\langle \overline{\mathbb{T}_i}, \overline{\mathbb{S}_i}\rangle \in \mathbb{Y}} \overline{\mathbb{T}_i}$ and $\overline{\mathbb{S}_m} = \bigcup_{\overline{D_i}\langle \overline{\mathbb{T}_i}, \overline{\mathbb{S}_i}\rangle \in \mathbb{Y}} \overline{\mathbb{S}_i}$
7: Find *maximal ancestor graph set* between D_n and $\overline{D_m}$,
$\widehat{\mathbb{A}} = \widehat{\Lambda}(\overline{D_m}, D_n) = \Omega(\Lambda(\overline{D_m}, D_n))$
8: Find *task overlaps* $\mathbb{T}_o$ between D_n and $\overline{D_m}$, where
$\mathbb{T}_o = \bigcup_{A_k\langle \overleftarrow{\mathbb{T}}_k, \overleftarrow{\mathbb{S}}_k\rangle \in \widehat{\mathbb{A}}} \overleftarrow{\mathbb{T}}_k$
9: Find *stream overlaps* $\mathbb{S}_o$ between D_n and $\overline{D_m}$, where
$\mathbb{S}_o = \bigcup_{A_k\langle \overleftarrow{\mathbb{T}}_k, \overleftarrow{\mathbb{S}}_k\rangle \in \widehat{\mathbb{A}}} \overleftarrow{\mathbb{S}}_k$
▷ *Identify non-overlapping tasks in input DAG to include*
10: Find *non-overlapping* tasks $\mathbb{T}_x$ and streams $\mathbb{S}_x$, where
$\mathbb{T}_x = \mathbb{T}_n \setminus T_o \qquad \mathbb{S}_x = \mathbb{S}^*_x \cup \mathbb{S}^+_x$
$\mathbb{S}^*_x = \{s_n \,|\, up(s_n), down(s_n) \notin \mathbb{T}_o\}$
$\mathbb{S}^+_x = \{s_n \,|\, up(s_n) \in \mathbb{T}_o, down(s_n) \notin \mathbb{T}_o\}, \forall s_n \in \mathbb{S}_n$
11: Add these tasks and streams to merged DAG,
$\overline{D_m}\langle \overline{\mathbb{T}_m}, \overline{\mathbb{S}_m}\rangle \Rightarrow \overline{\mathbb{T}_m} = \overline{\mathbb{T}_m} \cup \mathbb{T}_x,\quad \overline{\mathbb{S}_m} = \overline{\mathbb{S}_m} \cup \mathbb{S}_x$
▷ *Perform deployment*
12: Undeploy DAGs $D_i \in \mathbb{Y}$ and deploy merged DAG $\overline{D_m}$
13: Update set of running DAGs as $\overline{\mathbb{D}} = \overline{\mathbb{D}} \setminus \mathbb{Y} \cup \{\overline{D_m}\}$
14: **end procedure**

Algorithm 1 formally describes these operations for merging a *newly submitted DAG* $D_n\langle \mathbb{T}_n, \mathbb{S}_n\rangle$ with the *set of currently running DAGs* $\overline{\mathbb{D}}$. These are also illustrated with an example in Fig. 2. Dataflows submitted by users are shown in gray shaded box on the left side and the running dataflows are in a dashed box on the right, along with the logical time in circles going from top to bottom. The task label are their *type* and a diamond if present (e.g., Task $D^{\blacklozenge}$) shows additional *config*.

Identifying Overlaps. In lines 3–5, we first identify running DAGs $\mathbb{Y} = \{\overline{D_i}\} \subseteq \overline{\mathbb{D}}$ that *overlap* with the input dataflow D_n. While we can test the ancestor equivalence for every pair of tasks in the submitted DAG and the running ones, this is costly. Instead, we prune this search-space to just consider the running DAGs that share a source task with the input DAG to ensure at least a minimal overlap. Note that an input DAG can overlap with at most as many running DAGs as the number of source tasks it contains. In contrast, running DAGs that share no source task with the new DAG will be disjoint with it. If there are *no overlaps* with running DAGs, we go ahead and deploy the new DAG independently.

E.g., in Fig. 2, when dataflow $D1$ is submitted at time $t1$, it is the first dataflow and there is no reuse possible. So it is deployed and run as a new DAG, $sd1$, at time $t2$. Next, when $D2$ is submitted at $t3$, it is compared with the running

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

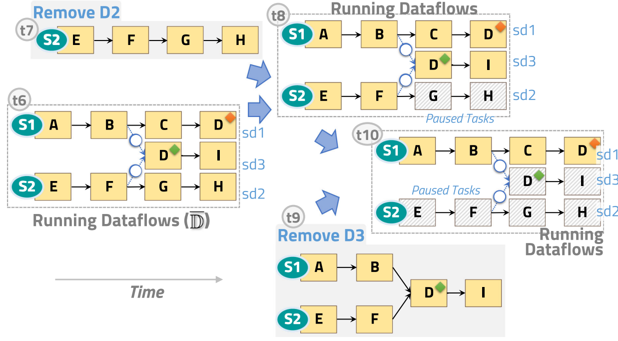


Fig. 3. Example of dataflow *unmerge* and *removal*.

dataflow $sd1$ and there is no overlap even in any source tasks. Hence, $D2$ is deployed as a new running DAG $sd2$ at $t4$ without any reuse. When dataflow $D3$ is submitted at $t5$, its source tasks A and E respectively overlap with the two running DAGs, $sd1$ and $sd2$. So we proceed to the subsequent steps for doing a merge and reuse.

Merging and Reusing Overlaps. In case there are overlaps, we construct a new *merged DAG* $\overline{D_m}$ from these overlapping DAGs and the new one, in lines 6–9. We first create a *merged DAG* $\overline{D_m}$ as the the union of all tasks and streams of these overlapping DAGs $\mathbb{Y}$. We then identify the prefix of the submitted DAG in this partially merged DAG by checking their maximal ancestor graph set, $\widehat{\mathbb{A}}$. This set of disjoint and maximal ancestor graphs contain the set of tasks $\mathbb{T}_o$ and edges $\mathbb{S}_o$ of the submitted DAG that are present in the overlapping DAGs and can be reused.

E.g., in Fig. 2, when adding the DAG $D3$ having overlaps with $sd1$ and $sd2$, the partially merged DAG initially contains tasks A–H and their corresponding streams. Now, we check the maximal ancestor graph between this and the new DAG. This gives the overlapping tasks as A, B, E and F, and streams as $\langle A, B\rangle$ and $\langle E, F\rangle$. So this subset of the new DAG is already present and can be reused.

Including Non-Overlapping Tasks. Then, in lines 10–11, we find the parts of the new dataflow that are not present in the running DAGs, and have to be *newly instantiated*. This has the set of new non-overlapping *tasks* to be created, $\mathbb{T}_x$, and the set of new *streams* $\mathbb{S}_x$, which includes $\mathbb{S}_x^*$ that connect tasks *fully within* the non-overlapping tasks, and $\mathbb{S}_x^+$ at the *boundary* between the reused tasks and non-overlapping tasks down-stream. These new entities are added to the merged DAG D_m.

E.g., in Fig. 2, when adding the DAG $D3$, tasks $D^{\blacklozenge}$ and I are non-overlapping tasks that will be incrementally added to the merged dataflow. Also, stream $\langle D^{\blacklozenge}, I\rangle$ which is fully present in the new tasks, and streams $\langle B, D^{\blacklozenge}\rangle$ and $\langle F, D^{\blacklozenge}\rangle$ that link the existing and the new tasks will be added as well. These latter streams will pass through a broker-indirection (shown with a $\otimes$), as discussed in Section 4.3.

Deployment. Lastly, in lines 12–13, we replace the overlapping DAGs in $\mathbb{Y}$ with this newly merged DAG to get the updated set of running DAGs, $\overline{\mathbb{D}}$. We also add the user's DAG to the set of submitted DAGs for book-keeping, $\mathbb{D}$. E.g., in Fig. 2, when adding the DAG $D3$, an *incremental dataflow* $sd3$ containing the non-overlapping tasks and streams is instantiated at time $t6$, and connected with the existing dataflows $sd1$ and $sd2$ being merged. As we discuss in the implementation, Section 4.3, starting an incremental dataflow $sd3$ and linking it with existing reused tasks through broker-based boundary streams minimizes the disruption, compared to stopping $sd1$ and $sd2$ and starting a single new merged dataflow that is composed of $sd1$, $sd2$ and $sd3$.

Algorithm 2. Unmerge Algorithm

1: **procedure** UNMERGEANDREMOVE($D_r\langle\mathbb{T}_r, \mathbb{S}_r\rangle$, $\overline{\mathbb{D}}$)
2: Find *running merged DAG* $\overline{D_i} \in \overline{\mathbb{D}}$ containing D_r, $\overline{D_i}\langle\overline{\mathbb{T}_i}, \overline{\mathbb{S}_i}\rangle = \Phi(D_r)$
3: Find *retained dataflows* $\mathbb{D}_s$ supported by the DAG $\overline{D_i}$, $\mathbb{D}_s = \Delta(\overline{D_i}) \setminus D_r$
4: Find *ancestor graph set* $\mathbb{A}$ for the *sink tasks* of retained dataflows $D_k\langle\mathbb{T}_k, \mathbb{S}_k\rangle \in \mathbb{D}_s$, where $\mathbb{A} = \{A_s \,|\, A_s = \alpha_{D_k}(t_p),\ \forall t_p \in \mathbb{T}_k, t_p.config = \text{SINK}\}$
5: Find *tasks to deactivate* $\mathbb{T}_d$ that do not appear in ancestor graph set, where $\mathbb{T}_d = \{t_q \,|\, t_q \in \overline{\mathbb{T}_i},\ t_q \notin \overleftarrow{\mathbb{T}_p}, A_p\langle\overleftarrow{\mathbb{T}_p}, \overleftarrow{\mathbb{S}_p}\rangle \in \mathbb{A}\}$
6: Find *streams to deactivate* $\mathbb{S}_d$ that are incident on a task from $\mathbb{T}_d$, $\mathbb{S}_d = \{s \,|\, s \in \overline{\mathbb{S}_i},\ up(s) = t \vee down(s) = t, t \in \mathbb{T}_d\}$
7: Create an *updated merged DAG* $\overline{\mathbb{D}_j}\langle\overline{\mathbb{T}_j}, \overline{\mathbb{S}_j}\rangle$, from the running DAG $\overline{D_i}$, where $\overline{\mathbb{T}_j} = \overline{\mathbb{T}_i} \setminus \mathbb{T}_d$ and $\overline{\mathbb{S}_j} = \overline{\mathbb{S}_i} \setminus \mathbb{S}_d$.
8: Identify *replacement DAGs* $\overline{\mathbb{D}_j^m}$ from the updated DAG $\overline{\mathbb{D}_j}$ as *distinct maximal connected component*.
9: *Replace* the running DAG $\overline{D_i}$ with the replacements $\overline{\mathbb{D}_j^m}$
10: Update set of *running DAGs*, $\overline{\mathbb{D}} = \overline{\mathbb{D}} \setminus \{\overline{D_i}\} \cup \overline{\mathbb{D}_j^m}$.
11: Remove D_r from set of *submitted DAGs*, $\mathbb{D} = \mathbb{D} \setminus \{D_r\}$
12: **end procedure**

4.2 Unmerging Algorithm

Users can request a previously submitted dataflow for removal, and this request can come in any arbitrary order, irrespective of the order of the dataflow's submission. When a removal request arrives, we need to first identify the single running (possibly merged) DAG that contains this dataflow. We then determine the tasks and streams in this running DAG that can be removed such that the correctness of the remaining submitted dataflows is not affected. This may cause some tasks in the running DAG to be deactivated (Fig. 3, $D2$), or a single merged DAG to be unmerged into multiple DAGs if the components get disconnected (Fig. 3, $D3$).

Algorithm 2 describes the algorithm to unmerge and remove a given DAG D_r. Fig. 3 illustrates this with examples, and these resume from the end state of Fig. 2 where dataflows $D1$, $D2$ and $D3$ form a single logical merged DAG consisting of $sd1, sd2$ and $sd3$. The logical time in circles goes from left to right.

Let $\Delta : \overline{\mathbb{D}} \to \mathscr{P}(\mathbb{D})$ be a *decomposition function* that maps from a running merged DAG to a set of submitted DAGs it supports, where $\mathscr{P}$ is the power set. Similarly, let $\Phi : \mathbb{D} \to \overline{\mathbb{D}}$ be an *inverse mapping function* that given a submitted DAG, returns the running merged DAG that it is contained in. These functions are maintained as part of the merge algorithm. When a DAG, $D_r \in \mathbb{D}$ is being removed, in lines 2–3, we first identify the single running DAG that contains this dataflow, $\overline{D_i} = \Phi(D_r)$. From this, we can identify the set of dataflows $\mathbb{D}_s = \Delta(\overline{D_i}) \setminus D_r$ that will be *retained*, and continue to be supported by $\overline{D_i}$, even after the removal of D_r. It is

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

possible that the removed DAG was the only one supported by this merged DAG, and no dataflows are retained.

E.g., in Fig. 3, when we remove $D2$ at time $t7$, this is contained in the running merged dataflow logically formed from $sd1, sd2$ and $sd3$. Since this merged dataflow also contains the submitted DAGs $D1$ and $D3$, they must continue to be supported after $D2$ is removed.

Next, in lines 4–6, we identify the subset of tasks $\mathbb{T}_d$ and streams $\mathbb{S}_d$ in the running merged DAG that must be *deactivated*. These tasks will not appear in the ancestor graph of any of the retained dataflows $\mathbb{D}_s$. Specifically, we find the ancestor graphs $\mathbb{A}$ for the *sink tasks* of these retained DAGs, and find tasks $\mathbb{T}_d$ that are not present in this ancestor set. Similarly, we find streams $\mathbb{S}_d$ to be disconnected as ones incident on any of these tasks being deactivated.

E.g., in Fig. 3, when we remove $D2$, the sink tasks of the retained DAGs $D1$ and $D3$ are $D^{\blacklozenge}$ and I. The ancestor graph for these tasks do not contain the tasks G and H, and these can be deactivated. Similarly, the streams $\langle F, G\rangle$ and $\langle G, H\rangle$ connected to these two tasks can be removed.

In lines 7–8, we logically remove these deactivated tasks and streams from the running merged DAG $\overline{\mathbb{D}_i}$ to form an updated merged DAG $\overline{\mathbb{D}_j}$. However, this can cause the merged DAG to get *disconnected into separate DAGs* as the removed DAG may have served as the link between different components. So we identify the maximal connected component(s) $\overline{\mathbb{D}_j^m}$ that are present in this updated DAG $\overline{\mathbb{D}_j}$. This is done through a forward traversal from each source task in the retained DAGs. The upper bound on the resulting number of unmerged DAGs is the number of source tasks in the dataflow being removed – this is a corollary to the merge operation that can at most cause these many DAGs to be merged.

E.g., in Fig. 3, when we remove $D2$, the remaining tasks continue to form a single connected component. However, instead, of $D2$, if we were to remove $D3$ (example not shown), then tasks $D^{\blacklozenge}$ and I and their incident edges would be deactivated. This would create two maximal components that correspond to $sd1$ and $sd3$, and would form two unmerged DAGs that remain.

Lastly, we replace the single merged DAG $\overline{\mathbb{D}_i}$ with zero (if the DAG being removed is the only one supported by the merged DAG) or more merged DAGs in $\overline{\mathbb{D}_j^m}$. We also update the set of running and merged DAGs. In practice, as we describe in Section 4.3 for our Apache Storm implementation, we may *pause* the relevant tasks and streams rather than disconnect and remove them from the merged DAG to avoid interruption.

E.g., in Fig. 3, when we remove $D2$, we *pause* tasks G and H (shaded in gray), and stop sending events on the stream $\langle F, G\rangle$ from task F. $sd1, sd2$ and $sd3$ are form the logical merged DAG continue to remain running. Subsequently, when we remove $D3$ at time $t9$, tasks $E, F, D^{\blacklozenge}$ and I are paused. While $sd1, sd2$ and $sd3$ remain deployed, only tasks from $sd1$ are active and support the retained dataflow $D1$.

4.3 Implementation

We develop a *Dataflow Reuse Manager* that offers a generic implementation of the merge and unmerge algorithms, with bindings to the *Apache Storm DSPS* to enact the dataflows and coordinate their reuse. Fig. 4 shows the major architectural components. Users submit a dataflow to our *Reuse Manager* as

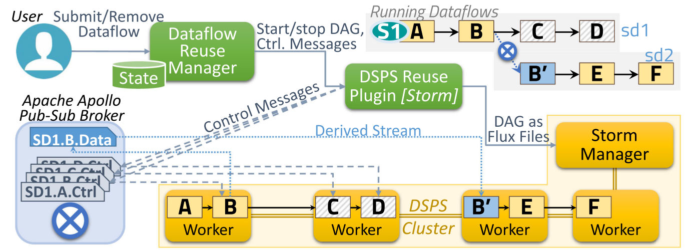


Fig. 4. Dataflow reuse architecture using apache storm.

a JSON file which captures the DAG, including the task ID, type, config and connectivity. The Reuse Manger tracks the *state* of the submitted and running DAGs. It runs the merge algorithm to identify running DAGs which can be reused and merged to support the submitted dataflow, and new tasks and streams to be created. The discrete algebra definitions for the algorithms are translated into efficient graph-based implementations. We also perform book-keeping to maintain the mappings between submitted and merged DAG, the paused/active states of tasks, and the broker-indirection of streams.

A *Reuse Plugin* translates these operations for a specific DSPS. An Storm cluster consists of a *Manager* and several *Workers* running on different VMs. Each Worker is a JVM that executes one or more tasks from the same DAG concurrently, and is typically assigned one CPU core. A DAG is assigned a specific number of workers on launch, and its tasks placed in a round-robin manner on these. We allocate $\frac{n}{p}$ workers to a dataflow, where n is the number of tasks and p is the *packing factor* – the concurrent tasks per slot. We use Storm's *Flux* JSON interface to create and launch dataflows.

DSPS like Storm do not allow the DAG structure to be modified after launch, instead requiring it be stopped and a new one launched with the updated structure. This will be disruptive to all dataflows reusing a running DAG, and will affect IoT applications that are latency sensitive. Instead, we develop a mechanism to run the merged dataflows as partial DAGs that can be incrementally launched, and use a *publish-subscribe broker* to externally connect them.

When the manager identifies multiple running dataflows to be merged ($\mathbb{Y}$) and new non-overlapping tasks and streams to be created $(\mathbb{T}_x, \mathbb{S}_x)$, it takes the following steps. It first launches a new dataflow corresponding to the non-overlapping tasks and their local streams, $\langle \mathbb{T}_x, \mathbb{S}_x^*\rangle$. Further, for each boundary stream being reused from an existing dataflow $(\mathbb{S}_x^+)$, a *proxy task* is included in the new dataflow to subscribe to the broker, and receive the remote events.

For this, each task in the Storm dataflow extends our wrapper class, which subscribes to a unique *control topic* on the broker. The Reuse Manager uses this topic to notify a task in a running DAG to *forward* a copy of its output stream to a unique *data topic*, which is subscribed to by the proxy task in the new DAG. Thus, the topic is a derived stream to connect tasks in different Storm dataflows being merged.

E.g., in Fig. 4, the single running merged dataflow consists of two Storm DAGs, $sd1$ and $sd2$ that are connected through the $SD1.B.Data$ topic. Task B of $sd1$ publishes its output stream to this topic, while B' in $sd2$ acts as a proxy task that subscribes to it, and forwards the received events to E. The control topic to which each task subscribes is used to signal the start/end of publishing the output stream.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

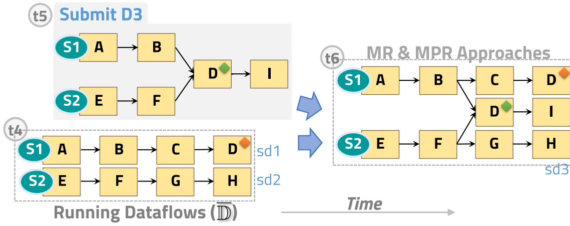


Fig. 5. Example of *defrag* on dataflow *submission*.

TABLE 1
Cost-Benefit Matrix for Defragmentation

Metric	Reuse	MR Defrag.	MPR Defrag.
Defrag. Overhead	$\Downarrow$	$\uparrow$	$\Uparrow$
Slots Used	$\uparrow$	$\downarrow$	$\Downarrow$
DAG Latency	$\uparrow$	$\downarrow$	$\downarrow$
Speed of Resubmit	$\uparrow$	$\uparrow$	$\downarrow$

Similarly, when a dataflow removal causes a running merged dataflow to be unmerged, the algorithm first identifies tasks and streams to deactivate, $\langle \mathbb{T}_d, \mathbb{S}_t \rangle$. This will be in a single merged DAG, but may span multiple Storm DAGs connected by the broker. The Reuse Manager does the book-keeping. The tasks to be deactivated may require termination of a subset of a running DAG, which Storm disallows. Instead, the Manager signals these tasks to instead *pause* their execution, using the control topic. These tasks will not process future incoming messages, which in effect, frees up the resources of those tasks without disrupting the DAGs they belong to. E.g., tasks C and D in $sd1$ have been paused.

This approach can introduce *latency overheads* due to the indirection in forwarding events between tasks through the broker. But it does not limit scaling since modern brokers like Kafka are designed for distributed scaling. A bigger concern is the *fragmentation* caused by repeated merge/demerge. This can result in many small Storm DAGs to support a few large merged DAGs, and numerous proxy and paused tasks. These affect the resource usage. Next, we discuss *defragmentation* strategies to address these concerns.

5 Dataflow Defragmentation

Frequently adding and removing overlapping DAGs can cause many fragmented DAGs to be running, even as they are part of a few logical merged dataflows. This design avoids stopping and starting DAGs repeatedly to modify their structure. However, it results in three types of overheads. One, for each stream from a running DAG being reused by a new DAG ($\mathbb{S}_x^+$), a *proxy task* uses additional compute resources, besides contributing to the latency through broker indirection (Fig. 4, B'). Two, *paused tasks* in a merged DAG consume incremental computing resources (Fig. 3, $t10$ with 6 tasks). Lastly, the Storm only allows tasks from a single DAG to be hosted in a worker, which can cause *poor bin-packing efficiency* of tasks from many DAGs to workers.

Defragmentation (defrag) is an approach to consolidate a single merged dataflow that is fragmented into DAGs, or has paused tasks within it, and addresses these short-comings. We propose two defragmentation approaches.

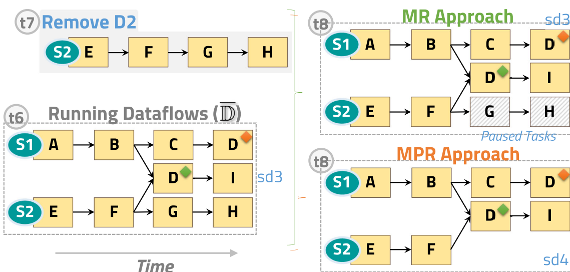


Fig. 6. Example of *defrag* on dataflow *removal*.

Merge and Replace (MR). In this, we consolidate all DAGs that are part of the same merged dataflow but are running as separate instances in the DSPS, into a single DAG. We first identify all the running component DAGs for a merged dataflow, terminate them, create a single DAG instance, and redeploy the new DAG. This will retain all tasks and streams in the original set of DAGs, *except for the proxy tasks* that linked them through the broker. The dataflow will include any paused tasks that were present. So this defragmentation is beneficial only when dataflows are added, and avoid creating smaller DAG fragments, but not when dataflows are removed since deactivated tasks continue to remain, in the paused state.

Here, we reclaim resources used by the proxy tasks, avoid the latency due to stream indirection between DAGs, and can achieve better packing of tasks to workers. By retaining the paused tasks, we pay incremental overheads for them even as they are not executing. As a minor benefit, we reduce the time to deploy an earlier DAG that was terminated and resubmitted, since we just need to unpause the tasks (faster) than defrag and deploy a new DAG (slower).

Merge, Prune and Replace (MPR). This extends the above approach. In addition, it prunes the paused tasks in the DAG instances before deploying the single DAG that exactly matches merged dataflow. Hence, the number of tasks in the new DAG will reduce by the number of proxy tasks and the paused tasks present earlier. This defrag approach is relevant both when adding and removing dataflows since removal can result in paused tasks, which this avoids. Besides the benefits of MR, MPR also eliminates the resources used by the deactivated tasks and improves the packing efficiency even more. It does lose the ability to rapidly unpause tasks when an earlier dataflow is resubmitted.

Fig. 5 shows the behavior of MR and MPR, corresponding to time $t5$ in Fig. 2 when $D3$ is submitted for reuse. Unlike the reuse approach without defragmentation, which we term RU, we see that both MR and MPR consolidate the dataflows $sd1$ and $sd2$, along with the new tasks and streams for $D3$, into a single DAG $sd3$ that is deployed at $t6$. There are no proxy tasks and broker indirection.

Similarly, Fig. 6 shows dataflow $D2$ being removed at $t7$ with MR and MPR, similar to the RU approach in Fig. 3. Here, MR pauses the tasks G, H at time $t8$, similar to RU, while otherwise not changing the running DAG $sd3$. MPR removes these tasks and deploys a new pruned DAG $sd4$ that exactly matches the merged dataflow.

The relative benefits of the two defrag approaches, relative to RU, are summarized in Table 1. At the same time, there is also cost associated with performing the defragmentation. This is primarily in the form of the redeployment latency for draining events from the dataflow(s), stopping

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

them, and starting a new dataflow with the merged/pruned tasks. This is a *one-time cost* at the time of defragmentation, while the benefits listed above *accumulate over time* as the DAGs continue to run. Since streaming applications are typically time-sensitive, this redeployment time due to defrag needs to be offset by adequate benefits.

E.g., the median time taken to deploy a dataflow in Storm is $40.4\ secs$, and for stopping a dataflow is $9.4\ secs$, based on experiments we describe later in Section 7. We leave it to users to determine when to initiate a defrag, but one can define a utility function that trades-off the time cost of defragmentation with its resource benefits to decide when to defrag. This decision is also affected by the order in which the dataflow submissions and removals arrive as the benefits of defrag accrue over time.

6 Proportional Billing

The advantages of reusing dataflows is evident when running the DSPS on elastic Cloud resources. The pay-as-you-go model of Cloud VMs at fine billing granularities of $minutes$ or $seconds$, combined with their on-demand acquisition and release, helps shape the resource cost paid. One can conceivably have a DSPS PaaS provider that supports reusable DAGs, and in turn intelligently uses IaaS VMs to run the DSPS workers and the merged dataflows. Here, we examine a proportional billing model for assigning costs to the submitted dataflows for their usage of shared resources.

We consider several practical factors in this billing calculation. IaaS providers charge for a VM that is *acquired*, irrespective of its utilization. This means that depending on the *number of workers* in a VM, and the *packing of tasks* into a worker, the cost incurred per task can vary. Cloud providers also have a *billing granularity* which determines the smallest unit of time that the VM is charged for. Billing *per-minute* has been common but recently, providers like Amazon AWS, Microsoft Azure and Google Compute have moved to a *per-second* billing for VMs. An active task running on a worker may be reused by more than one submitted dataflow, and its costs attributable to more than one user. Here, we offer a billing model to estimate the proportional cost per submitted dataflow of a user, under such a reuse scenario.

Say we have a billing interval of δ for VMs, and a cost of γ_r for a VM that can run r workers on it, with equal resource capacity each. For simplicity, we do not differentiate the quantum of actual CPU, memory, etc. resources consumed by a task within a worker. Let the dataflows be submitted only at integer increments of δ. At a given time θ that is a multiple of δ, let $\mathbb{V}_\theta = \{v_i\}$ indicate the set of n VMs that have been acquired for the DSPS cluster, and $\mathbb{W}_\theta = \{w_i^j\}$ be the set of $\sum_{i \in n} r_i$ workers on those VMs, where VM v_i has r_i workers. Let the set of active tasks deployed on those workers at time θ be given by $\mathbb{T}_\theta = \{t_k\}$, and the (merged) DAGs running at that point in time be $\overline{\mathbb{D}_\theta} = \{D_l\}$.

Say, we have a mapping function from each task to the worker it is executing on, $\mathcal{W}_\theta : \mathbb{T}_\theta \rightarrow \mathbb{W}_\theta$, and similarly its inverse mapping function, $\overline{\mathcal{W}_\theta} : \mathbb{W}_\theta \rightarrow \mathbb{T}_\theta$, from the worker to all tasks placed on it. Let another function give the mapping from a task to all the DAG(s) that are (re)using it, $\mathcal{D}_\theta : \mathbb{T}_\theta \rightarrow \mathbb{D}_\theta$, which, from the prior definitions, is,

$$\mathcal{D}_\theta(t_k) = \{t_j \mid t_j \leftrightarrow t_k,\ t_j \in \mathbb{T}'_\theta, D'_\theta\langle \mathbb{T}'_\theta, \mathbb{S}'_\theta\rangle \in \mathbb{D}_\theta\}.$$

All of these are available to the Reuse and DSPS Managers. With these, we can calculate the proportional cost for executing a dataflow $D'_\theta \in \mathbb{D}_\theta$ having tasks $\mathbb{T}'_\theta$ as follows. The cost γ_r paid in interval δ for a VM is proportionally divided among all *active workers* on that VM, where an active worker is one which has at least one *active task* placed within it. All these tasks equally share the cost of that worker. The task itself may be shared by multiple DAGs, and all those dataflows equally share the cost paid for this task. Based on this, we calculate the *incremental cost* for a task $t_k \in \mathbb{T}'_\theta$ mapped to worker $w_i^j = \mathcal{W}_\theta(t_k)$ at time interval θ as:

$$c_k = \frac{\gamma_{r_i}}{r_i^* \times |\overline{\mathcal{W}_\theta}(w_i^j)| \times |\mathcal{D}_\theta(t_k)|},$$

where v_i is the VM on which the task is running with a capacity for r_i workers, γ_{r_i} is the cost of this VM for every δ, and r_i^* is the actual number of active workers in that VM. $|\overline{\mathcal{W}_\theta}(w_i^j)|$ is the number of active tasks present on that worker, and $|\mathcal{D}_\theta(t_k)|$ is the number of dataflows reusing this task. Hence, the *cumulative cost* for all tasks of the DAG D'_θ for the time interval $[\theta, \theta+\delta)$ is $\sum_{t_k \in \mathbb{T}'_\theta} c_k$, and this may change in the next interval based on the DAGs that are submitted, removed and defraged. The total cost for a dataflow is the sum of all δ intervals it stays submitted for.

E.g., say we have a VM type that can run 2 workers on it, and costs $\gamma_2 = 1¢$ per minute, using a $\delta = 1\ min$ billing interval. Let each worker be able to run 2 tasks within it. Say, when we submit DAGs $D1, D2$ and $D3$ in Fig. 2, the tasks get mapped to 3 different VMs, v_1, v_2 and v_3, and on 5 workers $w_1^1, w_1^2, w_2^1, w_2^2$ and w_3^1 present in them – 2 workers each in the first 2 VMs and 1 worker on the third VM. Let the worker to task mapping $\overline{\mathcal{W}_\theta}$ at the end of time $t6$ be $w_1^1 : A, B;\ w_1^2 : C, D^\blacklozenge;\ w_2^1 : E, F;\ w_2^2 : G, H$ and $w_3^1 : D^\blacklozenge, I$. Now, the cost per minute for task A is $c_A = \frac{1¢}{2\times2\times2}$, since its VM v_1 has 2 workers (w_1^1, w_1^2), its worker w_1^1 runs 2 active tasks (A, B), and this task A is shared by 2 DAGs $(D1, D3)$. Similarly, the per-minute cost for task C it is $c_C = \frac{1¢}{2\times2\times1}$, and for task I it is $c_I = \frac{1¢}{1\times2\times1}$. These costs per task can then be summed up to estimate the cost of each DAG.

7 Experiments

Our experiments are designed to evaluate the relative benefits of the *merge and demerge algorithms* that enable reuse of overlapping dataflows, compared to the default configuration where *no reuse* is exploited (Section 7.2). To this end, we use multiple dataflow *workloads* (Section 7.1.1), and different *traces* for how these dataflows are submitted and removed to the DSPS, across time (Section 7.1.2). The quantitative metrics we report are the *number of active tasks* over time as dataflows are submitted and removed (Section 7.2.1), its consequence on the *CPU utilization* within the machines (Section 7.2.2), and insights into *how often* tasks get reused (Section 7.2.3).

In addition, we also validate the benefits of our *MR and MPR defragmentation strategies* in mitigating the effects of tasks that are paused but not removed (Section 7.3). Here, the metrics evaluated are the relative *number of workers used*

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Dataflows Used in Workload

Properties	OPMW	RIoT
Total # of Dataflows	65	21
Total # of Tasks	828	138
Total # of Edges	824	126
Unique Tasks	291	19
Unique Source Tasks	82	3
Unique Task Logic Implemented	1	16
Minimum Tasks in a Dataflow	3	4
Maximum Tasks in a Dataflow	34	8
Average Tasks in a Dataflow	12	6
Minimum Edges in a Dataflow	2	6
Maximum Edges in a Dataflow	34	7
Average Edges in a Dataflow	12	6
Maximum Dataflow Selectivity	10	2

(Section 7.3.1), the *reduction* in the worker count (Section 7.3.2) and the *latency* to perform the dataflow submission or removal operation (Section 7.3.3). Lastly, we also compare the proportional *billing costs* for the various strategies (Section 7.4).

7.1 Experiment Setup

7.1.1 Dataflow Workload

We use *two dataflow workloads* in our evaluation. One is from the *Open Provenance Models for Workflows (OPMW)* repository [21] which hosts ontology-based scientific workflow DAGs. These DAGs describe the structure with the task types and stream connectivity, which are adequate to identify reusability, but lack the tasks' executables for actually invoking the science logic. These workflows span different domains, and are shared publicly by the science community. As there are few openly available streaming IoT dataflows, these OPMW workflows are a proxy for future collaboratory IoT dataflow collections and indicative of the overlaps likely to occur within public IoT repositories. Of the 74 usable workflows in the repository, we pick 65 that can run within the resources available in our Storm cluster. In the absence of the original task executables for these worfklows, we instead use Viète's numerical π computation over 1600 iterations as the uniform compute-intensive logic for all these tasks. This ensures that each input event triggers a CPU-intensive operation rather than a trivial sleep.

The second workload is based on real IoT applications that are part of our *Real-time IoT Benchmark (RIoTBench)* [16], based on a smart utility project [26]. It has 27 IoT tasks with their associated execution logic, and 4 streaming IoT dataflows for performing Extract-Transform-Load, Statistical Summarization, Predictive Modeling and Analytics. We permute these datflows with these tasks to get 21 unique DAGs composed from 16 real task logic. So these are based on real IoT compositions and user logic, with the overlaps expected within shared IoT repositories being simulated. They use 3 IoT source task streams – Smart Grid, Urban Sensing, and NY Taxi Cab.

Table 2 shows statistics on these dataflow collections. We see that OPMW has more diverse DAG sizes and structure with $3-34$ tasks per DAG, 82 distinct source tasks, and 291 unique tasks that influence reuse. The peak selectivity of 10 implies that the output DAG rate can be $10\times$ the input rate. RIoT has 138 total tasks over 21 DAGs, with more similar sizes of $4-8$ tasks. But all 16 unique tasks implement real IoT logic. An extra sink task that connects to all sinks of the DAGs is used to collect performance metrics.

We use an input rate of $5\ events/sec$ for each OPMW source task. This rate can be amplified $10\times$ downstream due to their high selectivity, and helps us stay within the available resources at the peak rate. For RIoT, we support a higher event rate of $100\ events/sec$ due to the smaller DAGs. Events are 4–$380\ bytes$ in size, based on the source.

7.1.2 Dataflow Traces

We generate *3 DAG traces* each for the OPMW and RIoT dataflows to simulate submission and removal. For one trace, we use a *Sequential Submit/Remove (SEQ)* model to first incrementally *submit* a random dataflow from the workload with uniform probability, without replacement, in each time step. Once all DAGs in the workload are added, we switch to a *remove* phase where a random DAG that is present is removed in each step. This takes $65 \times 2 = 130$ steps for OPMW and $21 \times 2 = 42$ for RIoT. This trace simulates the behavior when only add or remove operations occur; the maximum reuse happens when all DAGs are submitted.

For the two other traces, we generate two *Random Walks (RW)* where we perform an add or a remove with equal probability at each time step, and repeat this 100 times. The DAGs to add/remove are chosen at random from the available/submitted pool – a submitted DAG is not resubmitted (unless removed) to avoid the whole DAG being reused by our algorithm to unfairly inflate its benefits. We initially populate the system with 40 DAGs for OPMW and 15 DAGs for RIoTBench at random, which form $\approx \frac{2}{3}^{rds}$ of the available dataflows, before the 100 random walk starts, and similarly drain the DAGs after the walk. This evaluates the system performance with repeated merge and unmerge.

7.1.3 System Setup

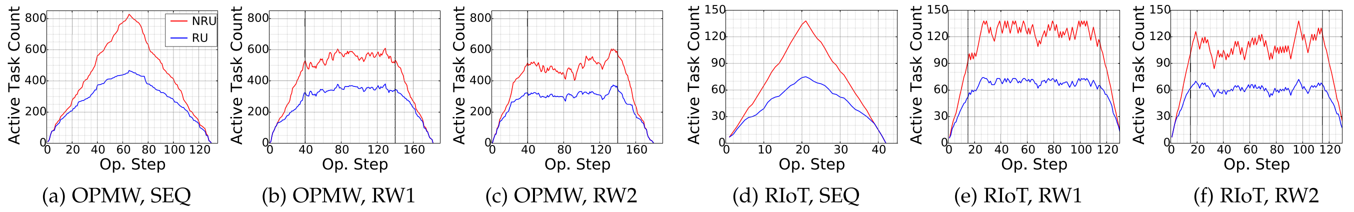
We run our experiments on *Apache Storm v1.0.2* DSPS that is setup on a commodity cluster, with each node having an AMD Opteron 3380 8-core CPU@2.6 GHz, 32 GB RAM, a 256 GB SSD, and GigaBit Ethernet, running CentOS v7. Storm runs on JRE v1.8 with the *Flux* JSON interface used for dataflow submission, and logging enabled for capturing metrics. *Apache Apollo v1.7.1* is our publish-subscribe broker using the MQTT protocol. Our *Manager* is implemented in Java and talks to Storm from a local node. We retain Storm's defaults of 1 worker (JVM) per core, 1 thread per task, and the round-robin scheduler. We see that up to $p = 8$ tasks can run concurrently on a worker without interference, allowing us to run up to 64 tasks per node. However, each Storm worker can have tasks from only one DAG. The Storm cluster is assigned as many nodes as required at the peak of a given trace, and this ranges from 3–16 machines. An *operation step* in a trace – DAG submission or removal – is generated at a $1\ min$ intervals to let the system and resource usage stabilize. Each trace takes 42–$180\ mins$ to run.

7.2 Results for Dataflow Reuse (RU)

7.2.1 Number of Active Tasks

We capture various metrics from executing these DAGs based on the traces on a Storm cluster. Fig. 7 shows the *number of*

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 7. *Number of active tasks* across operation steps for the 6 traces, with NoReuse (NRU) and Reuse (RU) approaches.

active tasks on the Y axis as the operation steps (logical time) increases along the X axis, for the default No Reuse (NRU) and the Reuse (RU) approaches. These exclude inactive and proxy tasks. The active task count should be lower when reuse occurs, and this indicates that we have avoided redundant computation.

In the *OPMW SEQ trace* Fig. 7a, there is a modest difference initially between NRU and RU, until step 20 where the NRU approach has 273 active tasks while our RU has 207. Submitting only a few DAGs from a large population limits their overlaps. However, as more dataflows are added till the peak of 65 DAGs, we see the difference widen to 828 active tasks for NRU and only 467 tasks for RU. This difference at the peak highlights the maximum possible reuse being exploited when all dataflows are submitted. In the drain phase, the gap stays wide since the random sampling happens to remove DAGs with less reuse first. *Overall, we see an average reduction of 34 percent in active task count using the reuse strategy, with a peak reduction of 44 percent tasks.*

A similar pattern of active task counts is observed for *OPMW's Random Walk traces (RW1, RW2)* in Figs. 7b and 7c. The first and last 40 steps are the submit and drain phases, and the 100 steps from 41–140 form the random walk. In RW1, we see the task count at $551 \pm 14\%$ for NRU while is only $346 \pm 13\%$ for RU during the 100 steps of the random walk phase. The matching numbers for RW2 are $501 \pm 21\%$ for NRU $314 \pm 19\%$ for RU. RW2 has more variability in the active task counts as a consequence of the random sampling. There are ≈ 40 dataflows that are running at any time. *This trace too exhibits an average reduction in tasks count of about 37 percent with reuse, and is also more representative of reality.*

For the *SEQ trace of RIoT dataflows*, we see a similar gap between the active task counts for the NRU and RU (Fig. 7d). From steps 5–21, the active tasks for NRU grows from 33–138 while the growth is muted for RU at 26–75 tasks. The average task count drop of RU from NRU is 38 percent, with a peak of 46 percent. For the *two random walks of RIoT*, the 100 steps are sandwiched between the 15 submit and drain steps at the start and end (Figs. 7e and 7f). *Like OPMW, the active tasks are fewer for RU relative to NRU with a mean drop of 44–45 percent for the two traces.* The task count variability is also smoother for RU, avoiding sudden spikes in the deployment. Nearly 15 dataflows remain deployed during the random walk. RW1 has a count of $125 \pm 25\%$ and $69 \pm 20\%$ active tasks for NRU and RU, while RW2 has $110 \pm 25\%$ and $62 \pm 16\%$ tasks.

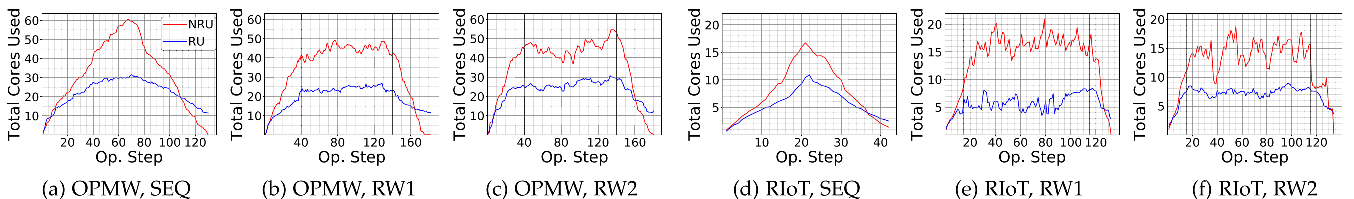
7.2.2 Cumulative CPU Utilization

We correlate the active task count with the matching CPU usage, which is the actual resource consumed with reuse. Fig. 8 shows the *cumulative cores used*, which is the sum of the CPU utilization (0.0–1.0, per core) on each active host in the Storm cluster; a value of 1.0 implies 100 percent use of 1 core. We see a strong correlation between Figs. 7 and 8, with a correlation coefficient of $\rho > 0.9$ in all but 2 of the 12 pairs of plots – RIoT RW1 with reuse and RIoT RW2 with no reuse. *OPMW's SEQ trace* has a core usage that grows and shrinks, but has a flatter resource usage for RU compared to its sharper slopes for the task count. *The peak resource reduction is also higher than the task count drop, with RU taking 48 percent fewer CPU resources than NRU; the mean reduction is 31 percent.*

When the dataflows are drained, the core usage drops for both approaches. It reaches 0 for NRU at the end. But, the usage for RU does not reach 0 since 467 tasks that were running are now paused, using 11 cores. Step 112 sees a crossover where NRU uses 15 cores and RU 19 cores, even though RU has fewer active tasks. Since the cores used impacts VM billing, this motivates the need to defragment. The cores used by *OPMW's Random Walk traces* (Figs. 8b and 8c) match the active task count during the 100 steps. *NRU's cores used is $44 \pm 15\%$ and $45 \pm 21\%$ for RW1 and RW2, while RU's usage is much lower at $24 \pm 11\%$ and $26 \pm 16\%$.* Here too RU uses 12 cores after all DAGs are drained.

Unlike OPMW DAGs whose tasks all run π compute, the RIoT workload's IoT tasks are heterogeneous, with varying compute, memory, I/O and network use. This diversity is reflected in the resource usage of its traces. Fig. 8d shows the cores used by *RIoT's SEQ trace*. We see a linear increase and decrease in cores used as seen in tasks count, but with the slopes changing occasionally to reflect the variable resource costs for the tasks. *The resource reduction for RU at the peak submission is 35 percent below NRU with an average reduction of 29 percent.*

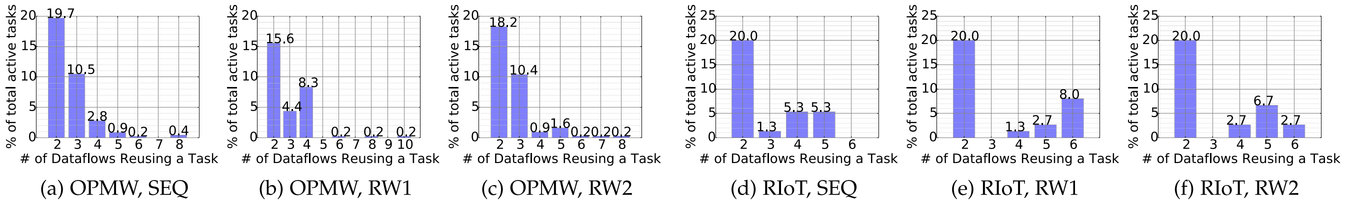
For the *RW1 and RW2 traces of RIoT* (Figs. 8e and 8f), the cores used during the 100 random add or remove operations using NRU is $16 \pm 36\%$ and $14 \pm 39\%$, and using RU is lower at $6 \pm 41\%$ and $8 \pm 19\%$. RW1 and RW2 have a poor



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 8. *Cumulative Cores Used* across operation steps for the 6 traces, with NRU and RU.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 9. *Frequency of reuse* plots showing the fraction of time that tasks were reused by more than dataflow, using RU.

correlation between task counts and CPU use, with $\rho = 0.52$ for RW1 RU and $\rho = 0.85$ for RW2 NRU, due to diverse tasks. All three of these traces have 2–4 core usage due to paused tasks, even after all DAGs are drained.

7.2.3 Frequency of Task Reuse

Lastly, we examine the *frequency of task reuse* for the different workloads and traces using RU. Ths helps us understand how many DAGs share a reused task. Fig. 9 shows the histogram of the time-weighted fraction of all running tasks over all steps (Y axis) that were reused by $[1, \underline{2})$ DAGs, $[2, \underline{3})$ DAGs, etc. ($\underline{X}$ axis). We omit the frequency of tasks used just once, which is the residual of all these frequencies.

OPMW's SEQ trace has an average of 20 percent of all running tasks reused by > 1 and ≤ 2 dataflows during the trace, with another 13 percent reused by [2,4) dataflows (Fig. 9a). The reuse fraction is marginally lower for its *RW1 and RW2 traces*, with nearly 16–18 percent of task being reused by [1,2) dataflows, while another 12 percent are reused by [2,4) dataflows. *For* $\approx 1\%$ *of tasks, as many as 10 dataflows share them in all traces*.

RIoT's SEQ, RW1 and RW2 traces in Figs. 9d, 9e, and 9f show 20 percent of active tasks reused by > 1 and ≤ 2 dataflows, and the reuse is above 5 percent even until 6 tasks. In RIoT, there are only 3 unique input tasks while OPMW has 82. As reuse requires the prefix sub-DAG to match, RIoT has a greater chance of its source and even 1–2 subsequent tasks often being common across dataflows. *Overall, 32 percent of tasks are reused by more than 1 dataflows in all three of its traces*.

7.3 Results for Defragmentation (MR, MPR)

Given the complexity of implementing the defragmentation strategies in Storm, we instead run simulations to understand the benefits and costs of these strategies. We use runtime metrics from real runs from the above setup for these simulation runs, and replicate Storm's round-robin scheduling strategy of tasks to workers.

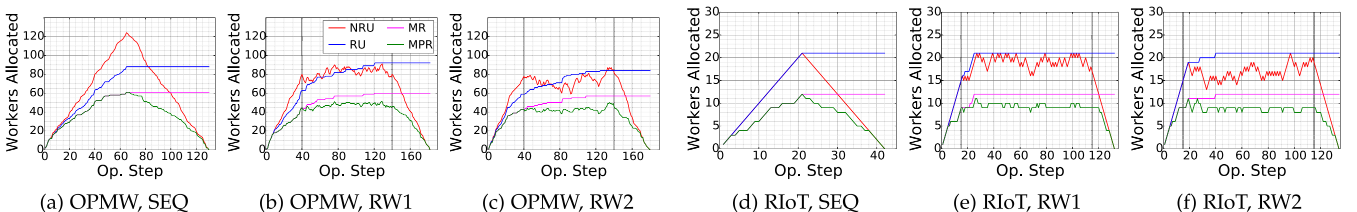
7.3.1 Number of Workers Used

Fig. 10 shows the *cumulative workers used* on the Y axis over different steps of the trace for: without reuse (NRU), with reuse but without defrag (RU), and with reuse and defrag using MR and MPR. This indicates the number of VM-cores that are *allocated* by Storm's scheduler, in contrast to the *measured* core usage in Fig. 8. The scheduler may over-allocate, causing lower usage, and this would motivate smarter DSPS scheduling strategies for better worker utilization [27], [28]. For MR and MPR, the defrag is done after every DAG submission or removal operation.

When we compare the worker allocation for the NRU approach with the core usage, it matches (e.g., RIoT SEQ with 21 workers allocated and 17 cores used at the peak) or is higher (e.g., OPMW SEQ with 124 workers allocated but only 60 cores used at the peak), but otherwise has a similar trend between Figs. 8 and 10. More pertinent is the *worker allocation for RU that increases or stays the same (Fig. 10) when submit or remove operations are performed, even though the core usage drops on DAG removal (Fig. 8)*. This is a consequence of the tasks being paused and not terminated when a dataflow is removed, thus taking up whole workers and also partial CPU resources. This shortcoming is addressed by defrag.

When we compare the worker allocation for MR in Fig. 10 with RU, we see that the worker count grows at a slower rate. Here, we perform defrag when a DAG is submitted – stopping, merging and restarting existing DAGs that it reuses. We do not use proxy tasks and can pack tasks for a single merged DAG into workers more efficiently in Storm. *Hence, the worker growth when a DAG is submitted is slower for MR*. However, this defrag is not relevant when removing a DAG, and the worker allocation does not reduce. MPR, on the other hand, is active during both DAG submit and remove, and it avoids both proxy and paused tasks by pruning the latter from DAGs when removed. *So its benefits accrue for every operation*.

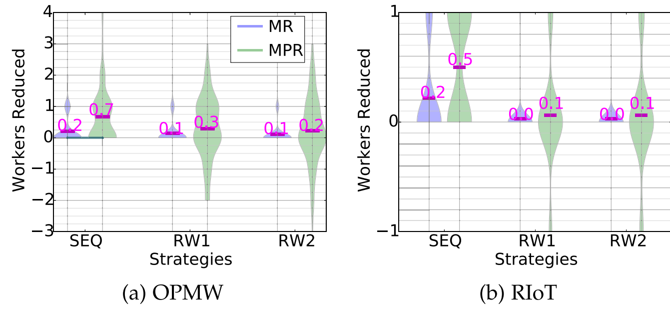
E.g., in Fig. 10a, we see that *OPMW SEQ* is allocated 61 workers using MR and MPR at its peak since both are identical during the DAG submission phase. RU takes 88 workers at the peak, and these are all lower than NRU that has 124 workers. But the worker count stays the same for RU and MR during the removal phase of SEQ while it drops to 0 for NRU and MPR. For *OPMW's RW1 trace* (Fig. 10b), the worker allocation for NRU is $82 \pm 14\%$, for RU is $83 \pm 24\%$, for MR is $56 \pm 21\%$ and for MPR is $47 \pm 13\%$. NRU and RU marginally differ in the worker allocation till step 120 beyond which RU is allocated strictly more workers. Though there is reuse of tasks in



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 10. *Cumulative Workers Allocated* across operation steps, with and without reuse and defragmentation.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.



(a) OPMW (b) RIoT

Fig. 11. Distribution of *Worker Allocation Reduced* at each step, relative to RU, when using MR and MPR strategies.

RU as reported in Fig. 7, Storm does not allow workers to be shared across dataflows. Since every reused dataflow also has at least 1 proxy task that is placed in 1 worker, its benefits are reduced. However, *MR and MPR have fewer workers than NRU during the entire random walk phase,* benefiting from the lack of proxy (MR, MPR) and paused (MPR) tasks, and better packing of tasks from DAGs merged into a single dataflow onto a common set of workers. OPMW RW2 behaves similarly.

Interestingly, for *RIoT's SEQ* in Fig. 10d, the workers allocated for both NRU and RU are the same up to the peak submission step of 21. RIoT DAGs have a maximum of 8 tasks that can all fit in a single worker ($p = 8$). So every DAG causes 1 worker to be allocated, either to run all tasks (no reuse) or for the proxy task (with reuse). *MR and MPR are alloted 12 workers at the peak step – a drop of 42 percent over NRU and RU.* In the drain phase, NRU and MPR decrease linearly whereas MR and MPR stay constant. In *RIoT RW1* (Fig. 10e), the worker allocation for NRU is $19 \pm 21\%$, RU is $21 \pm 23\%$, MR is $12 \pm 24\%$, and *MPR is the least at* $9 \pm 18\%$. RU is actually worse than NRU, but the defrag sharply drops the workers alloted to match the CPU usage reduction seen with reuse.

7.3.2 Reduction in Worker Count

In Fig. 11, we also report the *incremental number of workers reduced* by MR and MPR defrag after each operation step, relative to workers allocated for RU. This only shows the *savings per step*, rather than their cumulative benefit across time seen before in Fig. 10. The width of the violin plot shows the fraction of all operations that have the corresponding Y axis value, and the mean is marked. The Y axis values of workers reduced per step are whole numbers but the violin smooths them. For RW, only operations in the 100 RW steps are included in this and other violin plots.

MR acts only on a DAG *submission* and not *removal*. So in half of the steps, MR does not save any workers over RU and the plots are wide at $Y = 0$. For the DAG submission steps, we see that up to 2 workers are saved for OPMW, and up to 1 worker saved for RIoT – the latter at best saves the 1 worker that is always allocated to RIoT dataflows, either with or without reuse. *Considering all 6 traces, MR is able to reduce one or more workers in 13 percent of the operation steps.*

MPR acts on both DAG submit and removal, and it is able to save up to 4 workers from being allocated, relative to RU for OPMW. RIoT is limited to 1 worker due to the reasons given above for MR. We observe that *the worker allocation drops by 1 in 31 percent of the steps using MPR, across all 6 traces, and drop by more than 1 worker in 8 percent of the steps.*

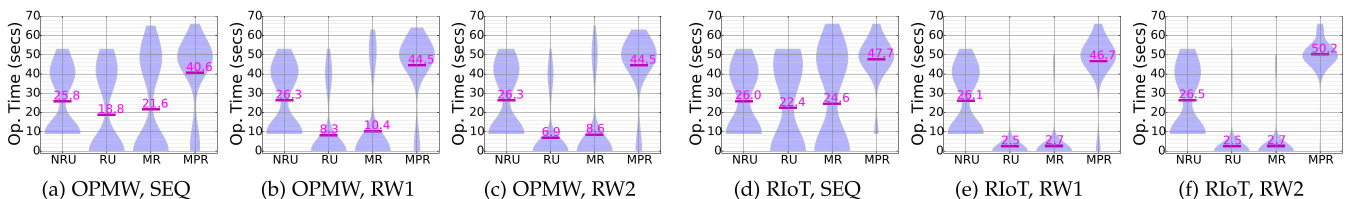
Interestingly, MPR can allocate more workers than RU as seen by the negative reduction. This occurs only when RU is able to handle a DAG submission solely by unpausing existing paused tasks that already have workers allocated, hence taking no additional workers than before, while MPR needs to deploy these tasks and allocate workers for them. This happens in 7 percent of steps across all 6 traces, and only for the RW traces and not SEQ since the latter does not resubmit running dataflows. This also gives SEQ a better reduction than RW. So, up to 3 more workers for OPMW and 1 more for RIoT may be assigned by MPR than RU.

Despite these marginal benefits per step, *MPR consistently reduces the cumulative number of workers allocated for all traces to a greater extent than MR.* For OPMW, this ranges from 29–88 workers reduced by MPR, and 11–27 reduced by MR. For RIoT, this is 6–21 workers saved by MPR and 3–9 workers by MR. The savings-potential is greater for larger DAGs like OPMW with more tasks, and workers allocated.

7.3.3 Dataflow Submission or Removal Operation Time

However, the defrag benefits come with additional overheads when a DAG submit or remove operation is done. This increases the latency time to complete the operation, and can impact time sensitive applications. When a dataflow is submitted, the *operation time* for NRU and RU, is the time to start a full or incremental DAG in Storm, while MR and MRU may also stop DAGs that are being reused, before they start a merged DAG. When a DAG is removed, NRU has to stop it in Storm, while RU and MR just pause its non-reused tasks using control messages, and MRU may stop, prune and restart the updated DAGs with Storm.

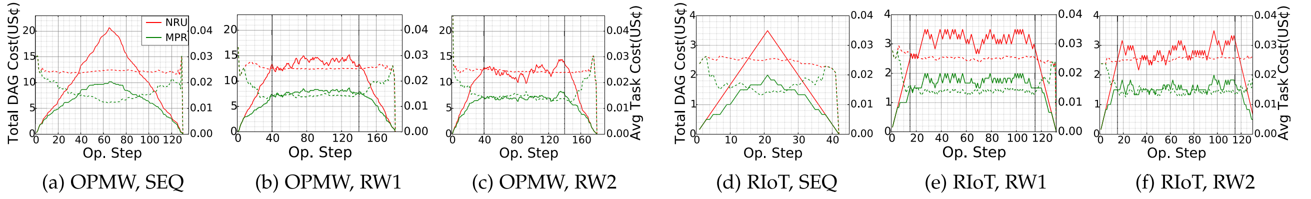
We perform micro-benchmarks in Storm to find the distributions of the time taken to submit a DAG and have it produce an output, and the time taken to stop a DAG completely, for the diverse OPMW DAGs. The time to deploy a DAG ranges from $30 - 849$ $secs$ with a mean of 64.5 $secs$ and median of 40.4 $secs$, while the time to undeploy it is much smaller at up to 12.9 $secs$, with a median of 9.4 $secs$. This time is not deterministic per DAG, does not correlate with the DAG size, and is much larger than the time to transmit the control messages, which takes ≈ 200 ms. So we sample from these distributions to estimate the *DAG operation time* to complete each step of a trace by the different strategies, and plot their distribution in Fig. 12.



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 12. Violin plot distribution of *DAG Operation Time* using different strategies. Mean values are shown by a bar.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.



(a) OPMW, SEQ (b) OPMW, RW1 (c) OPMW, RW2 (d) RIoT, SEQ (e) RIoT, RW1 (f) RIoT, RW2

Fig. 13. *Cumulative Billing cost* (solid line, left Y axis) and *Average cost per task* (dashed line, right Y axis) using a *1 core VM* billed at US¢ 10 per VM-hour at $1\ min$ granularity, with NRU and with MPR approaches.

The operation time distribution for NRU is nearly identical in all the traces as each DAG submit or remove operation step maps to the same Storm DAG operation. The time for this single operation is sampled from the same Storm add or remove distribution, and all traces have an equal number of add and remove steps – hence the wider violin at median add and remove times of $40.4\ secs$ and $9.4\ secs$. So the operation times are similar, with a mean of $\approx 26\ secs$.

The mean operation time is the least for RU in all the traces, lower than even NRU, since only DAG submit operations incur time and there is zero time for remove operations as tasks are just paused. This is visible for the two SEQ traces, Figs. 12a and 12d, where the time to add a DAG is similar to NRU while the time to remove gets shifted down to zero. Unlike SEQ where a dataflow is submitted only once, many of them are resubmissions in the RW traces, where RU just unpauses prior DAGs with zero operation time. This is more so for RIoT, which has fewer unique DAGs, than for OPMW. Hence, the time distribution skews lower, at $\approx 7.6\ secs$ mean for OPMW and $2.5\ secs$ for RIoT.

MR is similar to RU with no time taken for DAG removal, but during submission, it can take extra time to stop one or more merged dataflow(s) with overlaps before it starts a new one. Since the time to stop a Storm DAG is smaller than to start one, the operation times for RU and MR are comparable with a shift of about $9.4\ secs$ at the top of the violin. E.g., for OPMW, RU's second most frequent operation time is $\approx 40\ secs$ while it is $\approx 50\ secs$ for MR.

Though MPR reduces worker allocation and usage the most, it also takes the most operation time. Here, every DAG operation is likely to cause a (merged) DAG to be stopped, and the merged or pruned DAG to be started, depending on a submission or removal. Hence, the time taken by most traces lies close to $50\ secs$, which is the sum of the median times to stop and start DAGs in Storm. In some cases, more than 1 DAG may be stopped, and in others, an independent DAG stopped, as reflected by whiskers of the violin.

In NRU and RU, the time spent affects only the specific dataflow that is being submitted, while MR and MPR also affect other DAGs that are being reused. Similarly, when a DAG is removed, NRU affects just that DAG, RU and MR have negligible impact, while MPR impacts all DAGs that were reused by the removed DAG. However, this operation time is paid once when a DAG is submitted or removed while the resource usage accrues for the lifetime of the DAG. If dataflows are long-runing and there is limited flux, then the higher one-time costs for MPR will be acceptable in return for its significant reduction resource benefits.

7.4 Results for Billing Benefits

We quantify the *monetary costs and benefits of the strategies on Cloud VMs* by estimating their billing costs. We mimic the round-robin scheduling of tasks to workers in Storm for the simulation runs in the previous section. We then use per-minute billing, common in Cloud providers, to calculate the proportional cost per submitted dataflow for the duration of its submission in the trace using the model in Section 6, and the cumulative cost for all submitted dataflows at a step.

For each step of a trace, we plot the *cumulative billing cost* for all active DAGs and the consequent *cost per running task* that is billed. We report these for NRU and MPR, since the former is the baseline and the latter is more resource efficient than RU and MPR which suffer from proxy or paused tasks with idle worker allocation. In particular, we show the *benefits of Cloud elasticity* by adding and removing 1-core VMs with 1 worker on each, on-demand, from the Storm cluster. This allows the VM costs to match the worker allocation. For simplicity, we assume a billing cost of US¢ 0.10 per core-hour, which is US¢ 0.1667 per core-minute. The costs for different traces are shown in Fig. 13. While the absolute costs appear modest, note that each step in our trace is just $1\ min$ long. So the total costs (left Y axis) need to be scaled for the duration for which a DAG will run in the real-world, e.g., if each DAG runs for $100\ mins$ as step size, then the left Y axis values can be treated as US$ rather than US¢. Often, streaming dataflows run for days.

When using *1-core VMs* in the Storm cluster, Fig. 13 shows the total cost of all DAGs in the left Y axis, and it closely matches the worker allocation pattern in Fig. 10. Having the VM acquisition match the worker allocation ensures that billed VMs have all workers fully allocated. For *OPMW*, the SEQ trace costs US¢ 21 at the peak for all its DAGs using NRU, corresponding to the 21 workers allocated for the $1\ min$ step duration (Fig. 13a), while it costs only US¢ 10 for MRU. *This translates to peak cost reduction of 51 percent.* The area under the curve, which gives the total cost incurred for the entire trace, is US$ 13.57 for NRU and US$ 8.26 for MPR – an average reduction of 39 percent. The total DAG costs paid during the 100 steps of RW1 and RW2 are US$ 13.70 and US$ 12.20 for NRU, and US$ 7.87 and US$ 7.11 for MPR, *with a mean savings of 42 percent for both.*

The *average cost per task*, on the right Y axis of Fig. 13, divides the DAG costs equally among all tasks of the original dataflow submission. Since NRU does not reuse tasks, its least average cost per task will be when a worker has the maximum possible $p = 8$ tasks packed in it, at US¢ $\frac{0.1667}{8}$ =US¢ $0.023/min$. The average task cost at the start and end of the *OPMW* SEQ trace is higher at about US¢ 0.030 per step for NRU and also MPR. At these steps, workers have fewer than 8 tasks on each and MPR also has less chance of reuse as there are fewer tasks running. These cause the average to be higher. As more DAGs are submitted, the packing of tasks to workers improves and stabilizes to an average task cost of ≈US¢ 0.024 per step for NRU. This applies to OPMW's RW traces as well.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

The average task costs for MPR continue to drop for SEQ to reach US¢ $0.012/min$ at the peak DAG submission step, where reuse is maximum. For the RW traces, the average task cost drops to US¢ 0.014 per step for MPR, relative to NRU. RW traces have less reuse than SEQ at the peak. Also, as mentioned before, we do not allow the trace to resubmit an entire running DAG out of fairness to NRU. This avoids whole DAG reuse which can offer better benefits. We do notice more variation in the average costs for MPR than NRU since adding or removing a single DAG can impact the reuse of running tasks from multiple submitted dataflows.

The *RIoT* traces cost about a third of the OPMW traces, in keeping with their smaller DAG sizes. The total DAG costs change in discrete steps since the submission or removal of a DAG causes, at best, 1 VM to be added or removed as all tasks of any RIoT DAG will fit in 1 worker. The total cost curve for OPMW is smoother due to having DAGs with more numbers of tasks, and hence more workers.

For RIoT's SEQ, RW1 and RW2 traces, the area under the curve for *total DAG cost* for NRU is US¢ 73.5, US¢ 315 and US¢ 280, while for MPR it is US¢ 47, US¢ 174 and US¢ 157. *This puts the mean total cost reduction for MPR over NRU at 36–45 percent for each trace.* The *average task cost* using NRU stays at US¢ 0.024–0.026 per step for all traces. This is smaller for MPR in all cases, at US¢ 0.014–$0.016/min$. While the average task cost lines are smoother than total costs, the fact that a whole RIoT DAG fits in one worker means that with reuse, there are cases where no extra workers are required for submitted DAGs as their incremental tasks are repacked in existing ones. This causes the average cost to drop. The sharp increases for MPR indicate that a new worker is provisioned for a submitted DAG or the packing is less efficient after a DAG removal, increasing the average.

8 Related Work

Application sharing and reuse has been examined in the context of *distributed stream processing systems (DSPS)* and *data stream management systems (DSMS)* [29]. Repantis, et al. [24], [30] explore streaming application composition in a wide area P2P network (WAN), along with stream and task reuse. Their DAG of tasks has ontologically unique names for streams, and newly submitted DAGs have their stream names matched against existing ones. Identical streams are reused, while new tasks are collocated with upstream or downstream tasks to minimize network hops. We instead use a more rigorous graph-based approach to distinctively identify equivalent tasks and their output streams. They do not adequately examine the removal of a submitted DAG – as we saw, demerging has a cascading effect on the deployed DAGs. We limit our work to a Cloud data-center rather than WANs. But we do examine the monetary benefits of sharing dataflows within Cloud VMs. We extend the popular Storm platform for our validation, which requires us to perform defragmentation to address its limitations in dynamically changing deployed DAGs.

In DSMS, query DAGs are formed from operators with well-defined semantics to execute on tuple streams. Hirzel, et al. [31] suggest optimizations for eliminating equivalent computation and redundancy operators. Zhou, et al. [32] consider overlaps between the results of continuous queries and merges them into an equivalent query based on shared attributes, predicates and streams. We similarly merge dataflows that share equivalent streams, but adopt a DAG model for comparing equivalence that relies on typed tasks, rather than operator semantics and tuple schemas used in query reuse. So our work is generalizable to any DAG-based streaming application. Others have examined query admission control, operator allocation and reuse as an inter-related constrained optimization problem [33]. Their reuse of base (raw) and computed (derived) streams is conceptually similar to us but requires knowledge of operator behavior. They also impose resource constraints to limit the number of queries entering the system. We focus on opportunistic sharing of dataflow subsets to reduce their execution cost rather than be resource limited. We provision VMs elastically, with costs split among the shared dataflows.

Dynamic scheduling of streaming dataflows has been well-studied, to respond to changing rates, VM performance or application features [27], [34]. These can be used as alternatives to Storm's default round-robin model. But there is also similarity between our defrag approaches to consolidate DAGs into fewer ones, and task placement strategies to consolidate them into fewer VMs to reduce costs. There has been work on *billing of stream processing applications* to aggregate costs for infrastructure services in a multi-cloud environment [35]. Our billing approach is motivated more by partial sharing of applications themselves rather than just the underlying VMs on which they run.

Dataflow reuse for static rather than streaming applications has also been explored for *scientific and business workflows*. e-Science communities actively compose and publish workflows through portals for loosely-coupled collaboration [20], [21], [36], including for Big Data applications [37]. This can be adopted by the IoT domain for streaming dataflows, as they grow popular. E.g., *myExperiment* [20] offers a workflow repository, with annotations to help locate, modify and reuse workflows. However, reuse of the workflow composition is done manually, and modified workflows are published back for others to use. We instead consider automated reuse of running dataflows.

Goderis, et al. [38] identify similar worklows from existing DAGs based on their structure similarity using subgraph isomorphism, and rank the matched workflows. We also use graph structure matching for our DAGs, but require an exact match of the ancestor graphs to guarantee task equivalence. Others perform statistical analysis on workflows from myExperiment to examine recurring services (tasks) and reuse among workflows [39]. Network analytics is used to recommend services for new workflows being composed. We focus on running dataflows rather than composing them.

Reuse has also been considered in *business workflows* to efficiently manage business processes and save money or time [18], [40]. Ivan et al. [19] model business processes using π-calculus along with ontologies and annotations to allow artifact reuse. Others have also used rule-based inferencing for reuse [41]. Such semantic annotations are costly to define, and keyword matching cannot guarantee exact equivalence. There has been substantial work on efficient and cost-effective scheduling of workflows on Clouds to meet QoS needs [42], [43]. Rather than just share VMs across applications, we share running tasks themselves. These VM scheduling strategies can complement our work.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

Graph analysis over *provenance metadata* [23] has been used to compare the reproducibility of workflows, and identify divergence in the output [44]. *Prospective provenance* [22] describes the expected workflow behavior based on the DAG, and uses it for comparison and reuse. This is similar to our static DAG analysis for finding dataflow equivalence, but for streaming dataflows that are running. Data sharing among applications also raises concerns on confidentiality and privacy. Provenance has been used to verify authenticity of data processed in the Cloud to enable trustworthy (re)use [45]. Others have also proposed Information Flow Control (IFC) models to specify and enforce sharing mechanisms in the Cloud, and validated it for web service transformations [46]. These complement our work.

Unlike continuous stream processing, workflows execute in batch and generate persistent files. Hence, it is the workflow composition, or the static data that a prior execution generates, that is reused for future executions rather than the running workflow and transient streams. Provenance serves as an enabler. Our approach also automates the merging and unmerging of DAGs without user inputs, and does not have the overheads of maintaining semantic ontologies, instead relying on typed tasks that is common in DSPS.

One can find similarities between our approach and *Common Subexpression Elimination (CSE)* used in compiler optimization. Here, redundant expressions are found in code, often represented as a DAG, and substituted by an equivalent computed value. Aho, et al. [47] find the lowest common ancestors in a DAG and use a merge concept to capture ancestor information by adding parent-child edges and maintaining a forest structure. Others use an intermediate representation that combines data and control dependency in a program with DAG structure, on which they identify the lowest common ancestor for optimization [48].

9 Conclusion

In this article, we have motivated the need and opportunity for reusing partial subsets of tasks from streaming dataflows, in the emerging collaborative IoT domain where data stream and dataflow sharing is expected. We have formalized the problem definition with a tight specification of task equivalence between two dataflows, which allows them to be reused. We also define invariants that achieve output stream consistency and resource minimization. We use these specifications to design merge and unmerge algorithms for dataflows that are submitted to and removed from the DSPS, and implement them for Apache Storm. We also propose strategies to address DAG fragmentation from deploying incremental dataflows that can reduce resource efficiency. A formal model for proportional billing of Cloud resources for shared dataflows is also offered.

The algorithms are empirically validated in a Storm cluster using three traces of the OPMW scientific dataflow DAGs and real IoT streaming applications from RIoTBench. With reuse, we reduce the running task count by 34–45 percent and cumulative cores by 29–63 percent. We use defragmentation to remove stale workers from being billed after the DAGs are removed. Here, MPR defragmentation offers the better benefit with a monetary savings of 36–44 percent when simulating execution on Cloud VMs, while its operation execution time per DAG goes up by about $20\ secs$. We also see a greater and smoother impact of the larger dataflows of OPMW compared to the smaller RIoT dataflows. These results show that our algorithms are viable for deployment in collaboratory IoT environments hosted in Cloud data centers.

Such redundancy can be avoided manually for a small number of dataflows. But collaborative IoT applications, at large scales and across organizational boundaries, are emerging. The growing access to public IoT streams, the prevalence of IoT application platforms, and innovative streaming dataflows hosted on the Cloud will drive this. In that respect, our work is ahead of the curve.

In future, we propose to study the impact on DAG latency due to the broker indirection. Improved scheduling strategies can consider dataflow reuse and defragmentation for their optimization, consider locality of tasks that are part of the same merged DAG but present as separate dataflows, and prioritize shared tasks that support multiple dataflows. Validation on other DSPS platforms should also be explored. We will also consider using techniques like bi-simulation to formally prove the correctness of our merge and unmerge algorithms in ensuring ancestor graph equivalence [49].

References

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, “Sensing as a service model for smart cities supported by internet of things,” *Trans. Emerging Telecommun. Technol.*, vol. 25, no. 1, pp. 81–93, 2014.

[2] A. R. M. Forkan, I. Khalil, A. Ibaida, and Z. Tari, “BDCaM: Big data for context-aware monitoring – a personalized knowledge discovery framework for assisted healthcare,” *IEEE Trans. Cloud Comput.*, vol. 5, no. 4, pp. 628–641, Oct.-Dec. 2017.

[3] S. Aman, Y. Simmhan, and V. K. Prasanna, “Holistic measures for evaluating prediction models in smart grids,” *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 2, pp. 475–488, Feb. 2015.

[4] J. Baek, Q. H. Vu, J. K. Liu, X. Huang, and Y. Xiang, “A secure cloud computing based framework for big data information management of smart grid,” *IEEE Trans. Cloud Comput.*, vol. 3, no. 2, pp. 233–244, Apr.-Jun. 2015.

[5] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, “IBM Infosphere Streams for scalable, real-time, intelligent transportation services,” in *Proc. ACM SIGMOD Int. Conf. Manage.*, 2010, pp. 1093–1104.

[6] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., “Storm@ twitter,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *IEEE Data Eng. Bulletin*, vol. 38, pp. 28–38, 2015.

[8] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proc. Symp. Operating Syst. Principles*, 2013, pp. 423–438.

[9] M. Yannuzzi, F. van Lingen, A. Jain, O. L. Parellada, M. M. Flores, D. Carrera, J. L. Perez, D. Montero, P. Chacin, A. Corsaro, and A. Olive, “A new era for cities with fog computing,” *IEEE Internet Comput.*, vol. 21, no. 2, pp. 54–67, Mar.-Apr. 2017.

[10] Y. Simmhan, V. Prasanna, S. Aman, A. Kumbhare, R. Liu, S. Stevens, and Q. Zhao, “Cloud-based software platform for big data analytics in smart grids,” *Comput. Sci. Eng.*, vol. 15, no. 4, pp. 38–47, 2013.

[11] M. Strohbach, H. Ziekow, V. Gazis, and N. Akiva, “Towards a big data analytics framework for IoT and smart city applications,” in *Proc. Model. Proc. Next-Gen. Big-Data Tech.*, 2015, pp. 257–282.

[12] S. Hausmann, “Build a real-time stream processing pipeline with apache flink on AWS,” Apr. 2017. [Online]. Available: https://aws.amazon.com/blogs/big-data/build-a-real-time-stream-processing-pipeline-with-apache-flink-on-aws/

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.

[13] Apache, "Apache openwhisk: Open source serverless cloud platform," 2019. [Online]. Available: https://openwhisk.apache.org/
[14] Microsoft Azure, "Azure IoT solution accelerators," 2019. [Online]. Available: https://www.azureiotsolutions.com
[15] PTC, "Thingworx industrial IoT," 2019. [Online]. Available: https://www.ptc.com/en/products/iiot/
[16] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: A real-time IoT benchmark for distributed stream processing platforms," *Concurrency Comp.: Practice Exp.*, vol. 29, no. 21, 2017, Art. no. e4257.
[17] S. Nastic, S. Sehic, M. Vogler, H.-L. Truong, and S. Dustdar, "Patricia–a novel programming model for IoT applications on cloud platforms," in *Proc. Int. Conf. Service-Oriented Comput. Appl.*, 2013, pp. 53–60.
[18] M. Rosemann and J. vom Brocke, "The six core elements of business process management," in *Handbook on Business Process Management 1*, Berlin, Germany: Springer, 2015.
[19] I. Markovic and A. C. Pereira, "Towards a formal framework for reuse in business process modeling," in *Proc. Int. Conf. Bus. Process Manag.*, 2007, pp. 484–495.
[20] D. D. Roure, C. Goble, and R. Stevens, "Designing the myexperiment virtual research environment for the social sharing of workflows," in *Proc. 3rd IEEE Int. Conf. e-Sci. Grid Comput.*, 2007, pp. 603–610.
[21] D. Garijo and Y. Gil, "A new approach for publishing workflows: Abstractions, standards, and linked data," in *Proc. Workshop Workflows Support Large-scale Sci.*, 2011, pp. 47–56.
[22] S. B. Davidson and J. Freire, "Provenance and scientific workflows: Challenges and opportunities," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1345–1350.
[23] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, 2005.
[24] T. Repantis, X. Gu, and V. Kalogeraki, "QoS-aware shared component composition for distributed stream processing systems," *IEEE Trans. Parallel Distrib. Sys.*, vol. 20, no. 7, pp. 968–982, Jul. 2009.
[25] S. Chaturvedi, S. Tyagi, and Y. Simmhan, "Collaborative reuse of streaming dataflows in IoT applications," in *Proc. IEEE Itn. Conf. eSci. Conf.*, 2017, pp. 403–412.
[26] Y. Simmhan, P. Ravindra, S. Chaturvedi, M. Hegde, and R. Ballamajalu, "Towards a datadriven IoT software architecture for smart city utilities," *Softw.: Practice Experience*, vol. 48, pp. 1390–1416, 2018.
[27] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proc. ACM 16th Annu. Middleware Conf.*, 2015, pp. 149–161.
[28] A. G. Kumbhare, Y. Simmhan, M. Frincu, and V. K. Prasanna, "Reactive resource provisioning heuristics for dynamic dataflows on cloud infrastructure," *IEEE Trans. Cloud Comput.*, vol. 3, no. 2, pp. 105–118, Apr.-Jun. 2015.
[29] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surveys*, vol. 44, no. 3, 2012, Art. no. 15.
[30] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware component composition for distributed stream processing systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2006, pp. 322–341.
[31] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surveys*, vol. 46, no. 4, 2014, Art. no. 46.
[32] Y. Zhou, A. Salehi, and K. Aberer, "Scalable delivery of stream query result," *PVLDB Endowment*, vol. 2, no. 1, pp. 49–60, 2009.
[33] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch, "Sqpr: Stream query planning with reuse," in *Proc. IEEE Int. Conf. Data Eng.*, 2011, pp. 840–851.
[34] M. Yang and R. T. Ma, "Smooth task migration in apache storm," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 2067–2068.
[35] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring for heterogeneous clouds using stream processing," *Future Generation Comput. Syst.*, vol. 29, no. 8, pp. 2103–2114, 2013.
[36] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Comput. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.
[37] J. Wang, D. Crawl, and I. Altintas, "Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems," in *Proc. Workshop Workflows Support Large-Scale Sci.*, 2009, Art. no. 12.
[38] A. Goderis, P. Li, and C. Goble, "Workflow discovery: The problem, a case study from e-science and a graph-based solution," in *Proc. Int. Conf. Web Serv.*, 2006, pp. 312–319.
[39] W. Tan, J. Zhang, and I. Foster, "Network analysis of scientific workflows: A gateway to reuse," *Comput.*, vol. 43, no. 9, pp. 54–61, 2010.
[40] S. Zlatkin and R. Kaschek, "Towards amplifying business process reuse," in *Proc. Int. Conf. Perspectives Conceptual Model.*, 2005, pp. 364–374.
[41] Y. Mou, J. Cao, and S. Zhang, "A process component model for enterprise business knowledge reuse," in *Proc. IEEE Int. Conf. onServices Comput.*, 2004, pp. 409–412.
[42] A. C. Zhou, B. He, and C. Liu, "Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds," *IEEE Trans. Cloud Comput.*, vol. 4, no. 1, pp. 34–48, Jan.-Mar. 2016.
[43] J. Li, M. Woodside, J. Chinneck, and M. Litiou, "Adaptive cloud deployment using persistence strategies and application awareness," *IEEE Trans. Cloud Comput.*, vol. 5, no. 2, pp. 277–290, Apr.-Jun. 2017.
[44] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workflow reproducibility analysis," *Concurrency Comp.: Practice Exp.*, vol. 28, no. 4, pp. 995–1015, 2016.
[45] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer, "Provenance for the cloud," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, vol. 10, pp. 15–14.
[46] T. F. J. M. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 472–484, Jul.-Sep. 2017.
[47] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On finding lowest common ancestors in trees," in *Proc. ACM 5th Annu. ACM Symp. Theory Comput.*, 1973, pp. 253–265.
[48] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *Trans. Program. Languages Syst.*, vol. 9, no. 3, pp. 319–349, 1987.
[49] P. Buneman and S. Staworko, "RDF graph alignment with bisimulation," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1149–1160, 2016.

Shilpa Chaturvedi received the master's degree from the Indian Institute of Science (IISc), and this article part of her graduate studies. She is member technical staff with the Advanced Technology Group (ATG) of NetApp Inc., Bangalore. Her research interests include scheduling and sharing of streaming dataflows for IoT. She is a student member of the IEEE.

Sahil Tyagi is working toward the PhD degree at Indiana University, Bloomington, and this article was part of his work as a project staff at IISc. His research interests include on middleware for IoT applications.

Yogesh Simmhan received the PhD degree from Indiana University. He is an assistant professor with the Indian Institute of Science (IISc). His research explores abstractions, algorithms and applications on distributed systems, including Clouds and Internet of Things. He is the associate editor-in-chief for the *Journal of Parallel and Distributed Systems*, and earlier served as an associate editor of the *IEEE Transactions on Cloud Computing*. Previously, he was a research faculty with the University of Southern California (USC) and a postdoc at Microsoft Research. He is a senior member of the IEEE and ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Indiana University. Downloaded on June 05,2024 at 18:28:07 UTC from IEEE Xplore. Restrictions apply.