# A Deep Recurrent Neural Network Based Predictive Control Framework for Reliable Distributed Stream Data Processing

Jielong Xu, Jian Tang, Zhiyuan Xu, Chengxiang Yin, Kevin Kwiat and Charles Kamhoua

***Abstract*—In this paper, we present design, implementation and evaluation of a novel predictive control framework to enable reliable distributed stream data processing, which features a Deep Recurrent Neural Network (DRNN) model for performance prediction, and dynamic grouping for flexible control. Specifically, we present a novel DRNN model, which makes accurate performance prediction with careful consideration for interference of co-located worker processes, according to multi-level runtime statistics. Moreover, we design a new grouping method, dynamic grouping, which can distribute/re-distribute data tuples to downstream tasks according to any given split ratio on the fly. So it can be used to re-direct data tuples to bypass misbehaving workers. We implemented the proposed framework based on a widely used Distributed Stream Data Processing System (DSDPS), Storm. For validation and performance evaluation, we developed two representative stream data processing applications: Windowed URL Count and Continuous Queries. Extensive experimental results show: 1) The proposed DRNN model outperforms widely used baseline solutions, ARIMA and SVR, in terms of prediction accuracy; 2) dynamic grouping works as expected; and 3) the proposed framework enhances reliability by offering minor performance degradation with misbehaving workers.**

***Index Terms*—Deep Learning, Recurrent Neural Network, Distributed Stream Data Processing, Storm, Prediction.**

## I. Introduction

A Distributed Steam Data Processing System (DSDPS) handles unbounded streams of data tuples with many (physical or virtual) machines and worker processes (simply called workers in the following) in a distributed manner. Reliability and fault tolerance are critical for Distributed Stream Data Processing (DSDP). Currently, most DSDPSs (such as Storm [7]) handle anomalies or failures in a reactive manner. Specifically, they track acknowledgment of each tuple to detect failed tuples and re-process the associated source tuples or recover processing from the checkpoints to fulfill the at-least-once processing guarantee; moreover, workload on misbehaving or failed workers will be rescheduled and then related tuple failures will be re-processed accordingly. However, this simple reactive approach is not suitable for those Stream Data Processing (SDP) applications that demand data tuples to be processed in real or near real time due to the following reasons: 1) Data processing and worker problems are commonly detected by a timeout mechanism (e.g., 30s for tuple failure detection in Storm), which may significantly slow down data processing. 2) As observed in the previous work [42], rescheduling at runtime introduces a large number of failed tuples and longer tuple processing time. Hence, we propose a predictive approach to enhance reliability for DSDPSs by predicting performance of workers and re-directing data tuples to bypass those whose performance deviates from the expected, which we call *misbehaving workers* in the following. In this way, after a failure, a DSDPS only experiences very minor performance degradation, specifically, a minor increase on average tuple processing time and a small number of failed tuples, which have been confirmed by our experimental results (Section IV).

Accurate performance prediction is obviously the key to the success of a predictive approach. Here, we need to deal with high-dimensional time series data, which are basically runtime statistics of workers and machines. Time series prediction for a target is usually done based only on its own historical data. However, this may not work well in a DSDPS, in which a worker may share a common machine with many other workers, and those co-located workers may cause interference and affect its performance due to resource competitions. To improve prediction accuracy, such co-location interference needs to be well addressed. Recurrent Neural Networks (RNN), especially gated RNNs, have been reported to deliver the state of the art performance on a few sequence learning tasks (such as speech recognition [15] and text generation [16]). In this paper, we, for the first time, leverage Deep Recurrent Neural Networks (DRNNs) for performance modeling in DSDPSs, with consideration for co-location interference.

In additions, quick and effective actions need to be taken to minimize performance degradation that may be caused by misbehaving workers. In case of a failure, most DSDPSs re-schedule/re-assign tasks for recovery, which may lead to noticeable or even serious performance degradation [42] because it may take a few seconds to several minutes for a new task assignment to be deployed and may cause tuple failures during new deployment. This may counteract benefits brought by accurate prediction. We aim to come up with a quick, effective and smooth control mechanism that can prevent system performance from being affected by misbehaving workers or failures. We achieve this goal by using a quite different approach, i.e., designing a new grouping mechanism, to re-distribute and re-route tuples if a failure is predicted to occur.

In this paper, we present a novel predictive control framework to enable reliable DSDP. Specifically, we make the following contributions:

Jielong Xu, Jian Tang, Zhiyuan Xu and Chengxiang Yin are with Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, 13244. Email: {jxu21, jtang02, zxu105, cyin02}@syr.edu. Kevin Kwiat is with US Air Force Research Lab (AFRL), Email: kwiatk@gmail.com. Charles Kamhoua is with US Army Research Lab (ARL), Email: kkcharlesa@yahoo.fr. This research was supported in part by AFOSR grant FA9550-16-1-0077 and NSF grants 1525920 and 1704662. The paper was Approved for Public Release; Distribution Unlimited: 88ABW-2016-5163, Dated 17 Oct 2016. The information reported here does not reflect the position or the policy of the federal government.

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00036

- For prediction, we develop a novel DRNN-based interference-aware model to accurately predict performance of workers, which takes into account co-location interference.
- For control, we design and implement a new grouping method, dynamic grouping, which can distribute/redistribute data tuples to downstream tasks according to any given split ratio on the fly. So it can be used to redirect data tuples to bypass misbehaving workers.
- We implemented the proposed framework based on a widely-used DSDPS, Apache Storm [7], and validated and evaluated its performance with two representative SDP applications: Windowed URL Count and Continuous Queries.
- Extensive experimental results well justify accuracy of the proposed prediction model, demonstrate that dynamic grouping works as expected and show the proposed framework enhances reliability by offering minor performance degradation with misbehaving workers.

Note that we implemented and evaluated the proposed framework based on Storm. However, our design, especially the proposed DRNN model and dynamic grouping method, are quite general, which can be applied to other DSDPSs that has a similar programming model and architecture with minor modifications. More importantly, the proposed DRNN model can be used independently for time series analysis and prediction; and the proposed dynamic grouping method can also be used to serve other purposes, such as load balancing and power-efficient resource allocation.

## II. Distributed Stream Data Processing and Storm

In a DSDPS, a stream is basically an unbounded sequence of tuples. A data source (known as spout in Storm) reads data from external source(s) and emits streams. A Processing Unit (PU, known as bolt in Storm) consumes tuples from data sources or other PUs, and processes them using code provided by a user. It can either store data to a database, or pass it to other PUs for further processing. An application is usually modeled as a directed graph (known as topology in Storm), in which each vertex corresponds to a data source or a PU, and direct edges indicate how data streams are routed. A task is an instance of a data source or PU, and each data source or PU can be executed as many parallel tasks on multiple machines.

A DSDPS usually uses two levels of abstractions (logical and physical) to express parallelism. In the physical layer, it usually includes a master (known as Nimbus in Storm) that serves as the central control responsible for distributing user code around the cluster, scheduling tasks, and monitoring them for failures, and a set of virtual or physical machines that actually process incoming data. An application graph (topology) is executed on multiple worker processes (called workers in Storm) running on one or multiple machines. Slots are configured on each machine. The number of slots indicates the number of workers that can be run on this machine, and is usually pre-configured by the cluster operator based on hardware constraints such as the number of CPU cores. Each worker uses multiple threads (known as executors in Storm) to actually process data using user code. Each machine runs a daemon called supervisor that listens for any work assigned to it by the master.

A DSDPS usually supports 5 ways for grouping, which define how to distribute tuples among tasks: 1) Shuffle grouping: Tuples are randomly distributed across the downstream PU's tasks and each task is guaranteed to receive an equal number of tuples. 2) Fields grouping: A field of a tuple is used as the key to partition the stream. Tuples with the same key will be mapped to the same task. 3) All grouping: Each tuple is broadcasted to all tasks of the downstream PU. 4) Global grouping: The entire stream is distributed to one of the downstream PU's tasks, usually the task with the lowest ID. 5) Direct grouping: The producer of the stream decides which task of the downstream PU will receive each tuple.

Apache Storm [7] is an open-source DSDPS, which has a architecture and programming model very similar to what described above. Storm uses *ZooKeeper* [9] as a coordination service to maintain it's own mutable configuration (such as task schedule), naming, and distributed synchronization among machines. Note that all configurations stored in ZooKeeper are organized in a tree structure. Nimbus (i.e., master) provides interfaces to fetch or update Storm's mutable configurations. Nimbus, or each supervisor/worker in Storm is a Java Virtual Machine(JVM). A Storm topology contains a topology specific configuration, which is loaded before the topology starts and does not change during runtime. Each Storm executor has a grouper, which distributes tuples according to the installed grouping configuration.

To ensure reliability, when the message ID of a tuple coming out of a data source successfully traverses the whole topology, a special acker is called to inform the originating data source that message processing is complete. If a message ID is marked failure due to acknowledgment timeout, data processing will be recovered by replaying the corresponding data source tuple. Nimbus monitors heartbeat signals from all workers. It reschedules workers only when it discovers a failure.

## III. Design and Implementation of the Proposed Framework

### A. Overview

We illustrate the proposed framework in Figure 1, which can be viewed to have three planes: SDP with Dynamic Grouping, Data Collection and Predictive Control. It consists of the following components, whose functionalities are summarized as follows:

1) *Dynamic Grouping (Section III-C):* It allows an application to change how data tuples are distributed among tasks on the fly according to a given grouping configuration. This is implemented and embedded in the original DSDPS.
2) *Monitor (Section III-D):* It collects runtime statistics of workers and machines (such as CPU usages, workload, etc), and reports them to the controller.
3) *Hook (Section III-D):* It collects DSDP-specific runtime statistics (such as tuple execution time, tuple queueing time, etc), and send them to the co-located monitor.

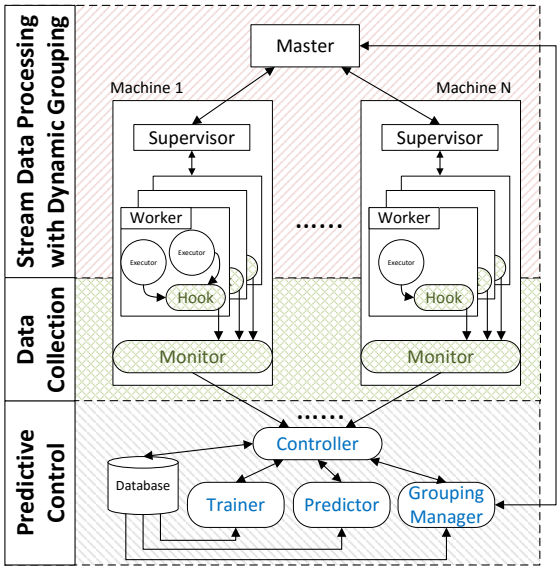



Fig. 1: The proposed predictive control framework

4) *Controller:* It obtains runtime data from all the monitors and coordinates activities of the trainer, predictor and group manager.
5) *Trainer:* It pre-processes collected data and trains the proposed DRNN model (Section III-B).
6) *Predictor:* It makes performance prediction based on runtime data using the trained DRNN model (Section III-B).
7) *Grouping Manager:* It calculates a grouping configuration (Section III-C) based on prediction results and sends it to the master.

The workflow of the proposed framework is described in the following:

1) The hooks and monitors periodically collect runtime statistics of workers and machines, and send them to the controller.
2) The controller stores runtime data from monitors to the SQLite [36] database, calls the predictor for prediction, and invokes trainer to update the DRNN model periodically.
3) The predictor makes performance prediction based on runtime data using the trained DRNN model and reports the results to the controller.
4) The controller calls the grouping manager to calculate a grouping configuration based on prediction results and send it to the master.
5) The master then uses the new grouping configuration along with dynamic grouping to re-direct data tuples to bypass misbehaving workers and/or machines.

Note that the proposed framework is general and flexible enough such that any control policy can be applied here. For example, a simple and conservative control policy could be: If the prediction error of a target feature of a worker exceeds a given threshold, stop sending tuples to the machine hosting that worker for a certain amount of time. How to find the best control policy and how to determine the best split ratio are application dependent and are out of scope of this paper since we aim to design a general predictive control framework here. But we will show that how the proposed framework along with the simple policy described above can improve reliability using experimental results in Section IV-C.

In summary, the proposed framework has the following desirable features:

1) *Interference-aware Performance Prediction with a novel DRNN Model (Section III-B):* It makes accurate performance prediction with careful consideration for co-location interference using a two-tiered DRNN model.
2) *Flexible Control with Dynamic Grouping for Enhanced Reliability (Section III-C):* It employs a new grouping method, dynamic grouping, which can distribute/re-distribute data tuples to downstream tasks according to any given split ratio and can re-direct data tuples to bypass misbehaving workers and/or machines according to prediction.
3) *Multi-level Data Collection (Section III-D):* It collects both process and machine level runtime statistics with over 70 features to enable accurate prediction.

### B. Interference-aware DRNN Model for Prediction

In most DSDPSs, multiple workers may run on a common virtual or physical machine, and multiple tasks are assigned to these workers. Runtime statistics collected from these tasks, workers and machines are basically time series data. We consider a computer cluster (hosting the DSDPS) with $N$ machines, and on each machine, there are at most $K$ workers. The values of the $M_m$ features (See Section III-D) of machine $j$ at timeslot $t$ is given by a vector $\mathbf{v}_j^t$; and the values of the $M_w$ features of worker $i$ on machine $j$ at timeslot $t$ is given by a vector $\mathbf{x}_{ij}^t$. Note that these feature values are min-max normalized. Specifically, DSDPS specific features (Section III-D) are normalized based on all samples from each worker, while others are normalized based on all samples from each machine. In addition, we denote the prediction result of worker $i$ on machine $j$ at timeslot $t$ as $\mathbf{y}_{ij}^t$, which may include multiple features.

Our problem here is to make one step ahead prediction for values of a set of important features (such as CPU usage) of every worker $i$ on machine $j$, given runtime data at worker and machine levels from machine $j$ in the past $T$ time slots.

There are a few approaches for modeling and prediction with time series data. We choose an RNN, particularly a gated RNN, as the starting point for our design due to the following reasons: 1) As regular feedforward neural networks, RNNs can be used to accommodate high-dimensional data and can be stacked together to form a deep model to handle complicated non-linear cases. 2) Gated RNNs, such as Long Short-Term Memory (LSTM) [18] and Gated Recurrent Unit (GRU) [13], use gates to control how to update hidden states, which have been shown to be effective on modeling long-term dependencies. 3) Gated RNNs have been successfully applied

in time series modeling and have been shown to offer record-setting performance for complicated problems such as speech recognition [15] and text generation [16].

As mentioned above, time series prediction for a target is usually done based only on its own historical data. However, this may not work well in a complicated DSDPS since the performance of a worker may be affected by co-located workers due to resource competitions. To tackle this time series prediction problem with consideration for co-location interference, we can feed runtime data of all co-located workers and the corresponding machine to an RNN when predicting performance of a worker.

As shown in Figure 2, we start with a gated DRNN constructed by stacking 2 layers of GRUs [31] and a sigmoid output layer, and feed all data related to co-located workers and the corresponding machine to it.

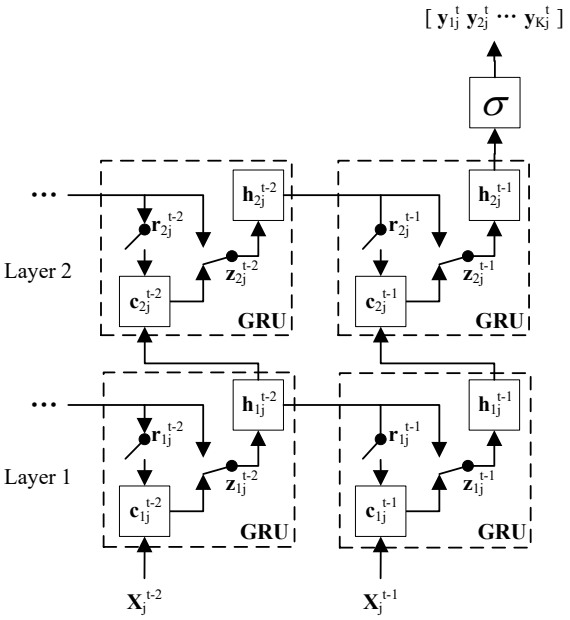



Fig. 2: A DRNN model with GRUs for machine $j$(unfold through time)

GRU is one of the most widely used RNN extensions. It uses gates to control the flow of past information and current input and specify how much past information should be let through. GRU (in layer 1) works according to the following equations (GRU in layer 2 works very similarly):

$$\mathbf{r}_{1j}^t = \sigma(W_{1r}\mathbf{X}_j^{t\prime} + U_{1r}\mathbf{h}_{1j}^{t-1}); \quad (1)$$

$$\mathbf{c}_{1j}^t = \tanh(W_{1c}\mathbf{X}_j^{t\prime} + U_{1c}(\mathbf{r}_t \odot \mathbf{h}_{1j}^{t-1})); \quad (2)$$

$$\mathbf{z}_{1j}^t = \sigma(W_{1z}\mathbf{X}_j^{t\prime} + U_{1z}\mathbf{h}_{1j}^{t-1}); \quad (3)$$

$$\mathbf{h}_{1j}^t = (\mathbf{1} - \mathbf{z}_{1j}^t) \odot \mathbf{h}_{1j}^{t-1} + \mathbf{z}_{1j}^t \odot \mathbf{c}_{1j}^t. \quad (4)$$

In these equations, $\sigma(\cdot)$ is the sigmoid function, $\tanh(\cdot)$ is the hyperbolic tangent function, and $\odot$ is element-wise multiplication. Generally, $W$ and $U$ are the weight matrices, e.g. $W_{1r}$ is the weight matrix for the reset gate. $\mathbf{X}_j^t = \left[\mathbf{v}_j^t \mathbf{x}_{1j}^t \cdots \mathbf{x}_{Kj}^t\right]$ is a row vector including the input feature values of both machine $j$ and its workers. The GRU keeps track of temporal dependencies by removing and adding information to the previous value using the reset gate and update gate. Every box in Figure 2 represents a GRU. In this model, for the $l$th layer GRU, the output $\mathbf{h}_{1j}^t$ given data from time $t$ on machine $j$ (i.e. $\mathbf{X}_j^t$) is calculated as follows:

1) Calculate the reset gate $\mathbf{r}_{1j}^t$ (Equation (1)) given all input from the lower layer and previous output $\mathbf{h}_{1j}^{t-1}$. This controls how much previous information will be forgotten while calculating the new input candidate $\mathbf{c}_{1j}^t$.
2) Calculate $\mathbf{c}_{1j}^t$ (Equation (2)) using the hyperbolic tangent function with new input and the previous output gated by $\mathbf{r}_{1j}^t$.
3) Compute the update gate $\mathbf{z}_{1j}^t$ (Equation (3)) similar as the reset gate. This gate controls the information flow from previous time step to the final output and helps the GRU memorize long-term information.
4) Compute the output $\mathbf{h}_{1j}^t$ with previous unit output $\mathbf{h}_{1j}^{t-1}$ and new output candidate $\mathbf{c}_{1j}^t$ weighted by $\mathbf{z}_{1j}^t$ (Equation (4)).

The above flat DRNN model captures co-location interference by weighting data from co-located workers equally, i.e., $\mathbf{X}_j^t$, without differentiating their impacts to the prediction, which may not be effective. When we try to predict performance of a worker $i$ on machine $j$, its own target feature data should play a big role on the prediction result. In order to address co-location interference, and in the meanwhile, emphasize the impact of historical data related to the target worker and the target feature, we come up with a novel two-tiered Interference-aware DRNN (I-DRNN) model, which is shown in Figure 3.

In this model, we harness the power of representation learning [11] to address co-location interference. Specifically, the DRNN in tier 2 uses the output of the DRNN in tier 1 as input for performance prediction, which corresponds to a representation of co-location interference. It is trained with data of all co-located workers and the corresponding machine. So the input of the DRNN in tier 2 consists of two parts: 1) The above representation of co-location interference; and 2) the target feature (such as CPU usage) of the target worker. This model well reflects the importance of the target worker's own data on the target feature.

In addition, we introduce weight sharing among multiple (up to $K$) tier-2 networks. Weight sharing has been successfully applied to convolutional neural networks and image processing [24]. We use it in our model due to following reasons: 1) With weight sharing, more data (i.e., data from up to $K$ workers) can be used to train the DRNN in tier 2, compared to the case without weight sharing. This usually leads to a better model. 2) Weight sharing reduces the total number of parameters and the training time.

Due to the parameter sharing and relatively shallow (4 layers in total) network structure, online inference can be done in real time (only about 5ms). Intuitively, the deeper the model, the more the amount of the RNN memorization, the higher

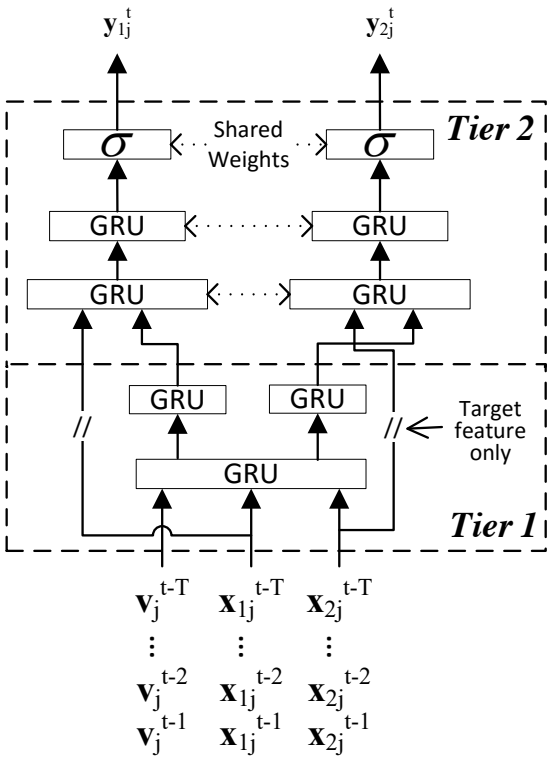



Fig. 3: The proposed Interference-aware DRNN (I-DRNN) for 2 co-located workers

the prediction accuracy, but the longer the inference delay. During our tests, we found that further increasing the number of layers increased the inference delay but brought very minor or even no gain on prediction accuracy. This may because the spatial and temporal correlations hidden in this scenarios are not as complicated or strong as those in video data, thus a very deep model is not necessary. We made similar observations and conclusions in our recent work on user interest data analysis [26]. Therefore, we used a 2-layer GRU network for each tier of the proposed model. Another important hyper-parameter is the number of neurons of the output of Tier 1. We set it to three to make it consistent with the dimension of the output of Tier 2 (even they do not have to be the same) in our implementation. We tired different settings in our tests but found it insensitive to prediction accuracy.

### C. Dynamic Grouping

To enable predictive control in a DSDPS, a method is needed to dynamically re-distribute tuples according to prediction on the fly to bypass those misbehaving workers/machines. In this section, we present design and implementation of a new grouping method, dynamic grouping, to fulfill this need. Note that our design is general, which may be used in any DSDPS that allows custom grouping methods; while our implementation is based on Storm.

*1) Design:* Shuffle grouping introduced above can be seen as a way of randomly assigning tuples to downstream tasks,

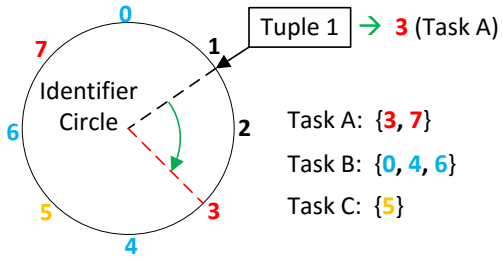



Fig. 4: Consistent hashing for dynamic grouping

which achieves an even distribution of tuples. In the proposed framework, we design a new grouping method, *dynamic grouping*, based on consistent hashing. With dynamic grouping, tuples can be dispatched to downstream tasks with any given split ratio, which enables flexible control on workload distribution in a DSDPS.

Consistent hashing [22] is a technique that can provide flexible tuple-task assignments. To use consistent hashing here, we first assign each downstream task one or more identifiers selected from an identifier circle with a modulo of $2^m$. Each tuple $d$ has its own ID $k_d$ and a modulo operation will be performed to obtain a new ID if it is larger than $2^m - 1$. The tuple will be sent to the task corresponding to the first assigned identifier on the circle (starting from its own ID and going clockwise). In this way, the tuple distribution over downstream tasks can be controlled by changing the assignment of task identifiers.

In the example shown by Figure 4, we have an identifier circle with $m = 3$, i.e., a total of 8 possible identifiers; and 3 downstream tasks. We assign identifiers 3, 7 to task A; 0, 4, 6 to task B and 5 to task C. In the following, we call such an identifier-task assignment a grouping configuration. When a tuple (with an ID of 1) arrives, it will be dispatched to task A. Because identifiers 1 and 2 are unassigned, and the first assigned identifier it meets is then 3. As mentioned above, identifer 3 is assigned to task A. Here, we can achieve a split ratio of 4:3:1 because identifiers 1 and 2 are unassigned, and tuples with an ID of 1 or 2 (after the modulo operation if needed) will be dispatched to task A too.

Note that consistent hashing may not be a unique solution to this problem. We made this design choice also because it is easy to implement and change the configuration. For example, to disable a downstream task, we can simply assign the identifier range corresponding to this task to other tasks, and then a tuple that goes to any remaining task previously will still be sent to the same task as before. Moreover, consistent hashing can be easily used to realize shuffle grouping by assigning identifiers to downstream tasks randomly with an equal probability.

*2) Implementation in Storm:* It is not trivial to implement dynamic grouping in Storm because unlike other grouping methods, it needs to be updated at runtime to enable predictive control. As mentioned above, Storm's topology specific configuration is loaded before a topology starts and does not change during runtime. However, to enable dynamic grouping

in Storm, we come up with a mutable configuration, *Topology Specific Dynamic Configuration (TSDC)*, which can be changed during runtime. Each topology has a TSDC, which specifies the identifier-task assignment (described in the last section) for every bolt-bolt or spout-bolt pair.

We implement dynamic grouping as a custom grouping policy in Storm. We store all TSDCs in ZooKeeper and monitors if there is any change with the ZooKeeper watcher. It is natural to store and manage all TSDCs in ZooKeeper because Storm stores all its configurations in ZooKeeper, and doing so can ensure Storm's deployment procedure remains untouched. Specifically, to manage TSDCs in ZooKeeper, we made following modifications to Storm:

- Add a TSDC map to the topology class.
- Add an interface *DynamicConfigurable*, which contains a callback method *void processUpdate(String)* for handling the notification of a TSDC update, a method *void initDynamicConfig(String)* for initialization, and a method *String getConfigPath()* for obtaining configuration ID, which is used to differentiate multiple TSDCs.
- Create a subtree for storing TSDCs in ZooKeeper.
- Set up ZooKeeper watchers for notifications of TSDC updates in *getConfigPath(). processUpdate(String)* is called when a watcher is triggered.
- Add an interface on Nimbus to manage TSDCs.

Next, we give an example to demonstrate how a Storm user can use dynamic grouping for his/her topology. In this example, we show how to define an *identity* bolt with 3 tasks, which receive tuples from *source* component using dynamic grouping, and how to initialize a dynamic grouping configuration using a TSDC. Here, we assign 20 identifiers for each of three tasks of the identity bolt.

```
builder.setBolt("identity", identityBolt, 3)
  .customGrouping("source",
    new DynamicGrouping("__identity"));
...
builder.setDynamicConfig("__identity",
  DynamicGrouping.initShards(3, 20));
```

At runtime, Storm stores the initial configuration as a TSDC in ZooKeeper when a topology using dynamic grouping is submitted. All the corresponding groupers fetch the TSDC from ZooKeeper. Once Nimbus receives a new grouping configuration, it updates the corresponding TSDC in ZooKeeper, which triggers ZooKeeper watchers in the corresponding workers. ZooKeeper watchers notify groupers to use the new configuration, which will re-distribute data tuples to downstream tasks accordingly.

### D. Multi-level Data Collection

A multi-level data collector is designed to collect runtime statics at both worker and machine levels.

We embed a hook (that runs as a thread) in each worker to collect information related to DSDPS specific features. In our implementation, the hook collects readings of the features in Table I from each worker every 5s. In Storm, when a tuple arrivals at a worker, it is first queued based on the downstream executor, so we monitor the queuing time for inbound tuples.

| Group | Features(Unit) |
|---|---|
| DSDPS | Average tuple execution time(ms), number of inbound tuples per second, average inbound tuple queuing time(ms), average outbound tuple batch sending time(ms), number of outbound tuple batches per second |
| JVM | Heap/non-heap memory usage(KB), process ID |

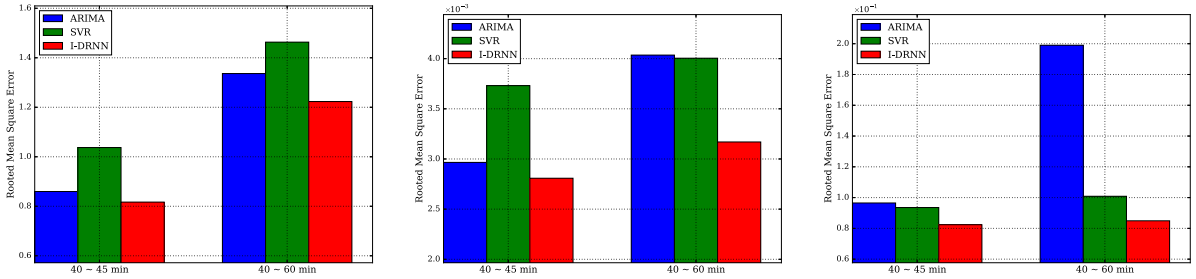
TABLE I: Features collected by a hook

Then each executor executes user code to process tuples from its own queue one by one. We collect the execution time for each tuple. For emitted outbound tuples, those tuples with the same downstream worker are batched first, and then batches are sent out. So we also collect the sending time for each outbound tuple batch. In addition, we collect memory usage information using Java Management Extensions [21].

Note that Storm comes with APIs for monitoring, but we find it not very efficient for data collection. Specifically, Storm provides *ITaskHook* interface and a *BaseTaskHook* class for user to create a monitor on each task and record runtime statistics for each tuple. Using such APIs introduces an inter-process connection for each task thread to pass out data, which leads to higher overhead compared to using inter-thread communications to collect data with an embedded hook, and possibly blocks the execution of task threads. Moreover, there is no existing API for collecting queuing and sending time. Therefore we implement our own hook for Storm-specific data collection. In order to collect tuple-level statistics efficiently without blocking data processing, we use a high performance inter-thread messaging library, *LMAX Disruptor* [27], to efficiently queue all collected data. Queued data will then be averaged and transmitted periodically by a single consumer thread.

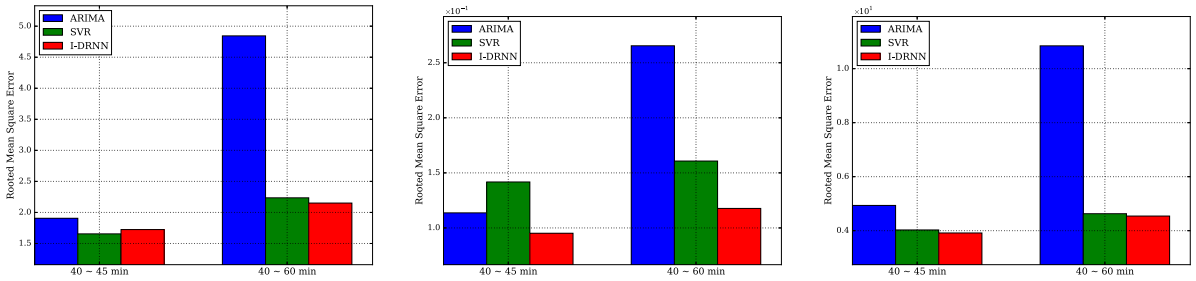
A monitor runs in the background on each machine for gathering data from hooks and collecting runtime machine and worker related statistics using the *Sysstat* [38] tool set. In the monitor, the *pidstat* program in the tool set is used to collect statistics of a large number of process (worker) level features related to CPU, memory, disk and context, such as CPU usage, physical memory usage, number of minor/major faults per second, size of data read from/written to disk per second, number of voluntary/involuntary context switch per second, etc, for all hosted workers every 15s. It also uses the *sar* program to collect machine level features related to CPU, context, memory page, data transfer, workload and networks, every 30 seconds. Due to space limitation, we omit the complete list of these process and machine level features. Each monitor sends averaged runtime statistics for each worker and for each machine to the controller every 30s. To efficiently transmit data among hooks, monitors and the controller, we utilize Apache Thrift [8] to define data structures and remote process calls.

Even though we collect statistics data with over 70 features at runtime, the overhead is negligible. This is because all these features take numerical values in the 32-bit floating point format and transmitting them from all VMs to Nimbus only take several microseconds in a 1Gbps network, which is much shorter than our collection periods mentioned above, i.e., 5s/15s/30s.




(a) CPU usage (b) Outbound sending time (c) Inbound queuing time

Fig. 5: Prediction RMSE of Windowed URL Count




(a) CPU usage (b) Outbound sending time (c) Inbound queuing time

Fig. 6: Prediction RMSE of Continuous Queries

## IV. Performance Evaluation

We implemented the proposed I-DRNN model using Lasagne 0.2 [23] and Theano 0.7 [39]. We implemented the dynamic grouping based on Storm 0.10.0 [7], and installed Storm on top of Ubuntu Linux 14.04. We performed real experiments on a virtualized cluster with 6 blade servers (each with dual Quad-core Xeon E5506 CPUs and 18GB RAM) connected by a 1Gbps network. The cluster has 11 Virtual Machines(VMs) (each with 2 vCPUs and 2GB Memory) for data processing. One of them was used to run Nimbus, while the other 10 were used to run workers with supervisors and our monitors. In addition, we also ran the Zookeeper and Apache Kafka [3] with 3 and 2 separate VMs respectively. The controller was run on a VM with 8 vCPUs and 12GB memory.

We conducted our experiments using two representative SDP applications (topologies): *Windowed URL Count* [37] and *Continuous Queries* [10], [12].

**Windowed URL Count:** Windowed word count is a well-known SDP application, which counts words from a data stream within a given time window. We modified this topology to count the number of accesses per URL in the past 2 minutes from web server request traces using dynamic grouping. It has a chain-like topology with one spout and three bolts. We used Wikipedia request traces in September 2007 [40] as the input data stream and loaded all the traces to Apache Kafka first for data fetching before experiments.

The spout is a reader which reads a batch of request traces at a time from Kafka, and feeds every single line separately to the ExtractURL bolt using dynamic grouping. The ExtractURL bolt extracts the timestamp and URL from each request trace line and feeds them to the PartialCount bolt using dynamic grouping. The PartialCount bolt performs a windowed count and sends its counts to MergeCount bolt every 500ms using global grouping. The last stage of the topology, MergeCount, aggregates latest partial counts from all tasks of PartialCount to obtain the windowed URL count results.

**Continuous Queries:** It consists of a spout and two bolts. The spout continuously emits randomly generated queries, each with a vehicle plate number and its speed, to the Query bolt using dynamic grouping. The Query bolt randomly generates a table with vehicle plates and information (such as names and driver license IDs) of their owners in the beginning, and takes a query from the spout and then iterates over the table to find if there is a matching entry when a given vehicle is speeding. If there is a match, the Query bolt emits the corresponding tuple using global grouping to the Logger bolt which simply writes what it receives to a file.

We tested the Windowed URL Count topology on 10 VMs with 14 workers, 2 spout executors, 5 and 6 executors for ExtractURL and PartialCount bolt respectively, and 1 executor for the MergeCount bolt in all experiments. We ran the Continuous Queries topology using 5 workers, with 1 spout executors, 3 Query bolt executors and 1 Logger bolt executors for the experiments related to dynamic grouping validation, and used 15 workers with 4 spout executors, 10 Query bolt executors and 1 Logger bolt executors in the other experiments.

### A. Performance Prediction

We used the proposed I-DRNN model as well as the widely used AutoRegressive Integrated Moving Average(ARIMA) [29] and Support Vector Regression (SVR) [35] for one-step-ahead prediction of CPU usage, inbound tuple queuing time and outbound tuple sending time of every worker; and compared their prediction accuracies in terms of Root Mean Squared Error (RMSE) [19].

*1) Training:* We ran each of two applications 5 times in Storm for one hour and collected the datasets for training. From each of 10 datasets, we used all data from 10 to 40min for training, and then used the trained model to predict the CPU usage, the outbound sending time (tuple batch) and the inbound queuing time (tuple) for each worker between 40 and 60min. We examined the prediction performance in two time periods: 1) between 40 and 45min, and 2) between 40 and 60min. For each time period, we computed RMSE for every target feature of each worker first, and calculated the average for each feature. Note that we didn't use data collected from first 10min since it has been shown [42] that Storm usually stabilizes after 10min. Each of the worker and machine runtime statistics is a sequence of 24- and 57-dimensional real vector respectively. Each data point is normalized to a value between $[0, 1]$ using the min-max normalization as mentioned above. We trained univariate time series models for each of the three target features of each worker, using the forecast package in R [20] for ARIMA and scikit-learn [32] for SVR respectively.

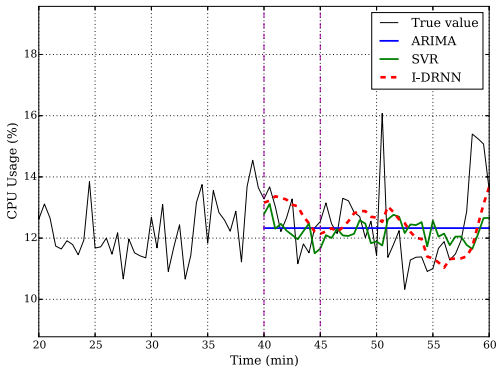
We trained the proposed I-DRNN model for each of the three target features using 20 different seeds to randomly generate initial weights for each training dataset. We used 10% data randomly selected from the collected data for validation and the rest 90% for training, trained each model for 1000 epochs, and selected the model with the lowest validation error for performance prediction.

*2) Results and Analysis:* We list all results from our experiments in Figures 5, 6 and 7.

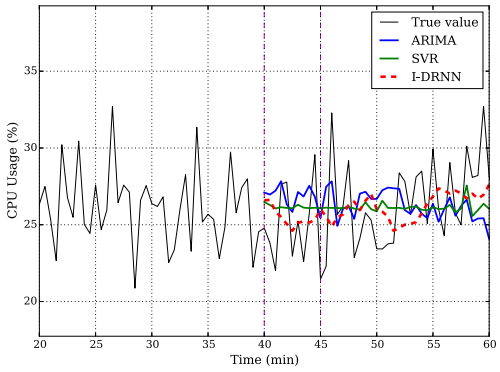
From Figure 5, we can see that for Windowed URL Count, the average RMSE of prediction of the proposed I-DRNN model between 40 and 45min are as low as 0.8169, 0.0028 and 0.0823 for three features, CPU usage, outbound sending time and inbound queuing time, respectively, which are clearly lower than those given by ARIMA and SVR. On average, I-DRNN's RMSE is 8.31% lower than ARIMA's, and 19.30% lower than SVR's for prediction between 40 and 45min; and the improvements become more significant, specifically, 29.08% and 17.69%, for prediction between 40 and 60min. It is also interesting to see SVR maintains relatively stable RMSEs when the end of the prediction period changes from 45min to 60min, while ARIMA's RMSEs go much higher.

From Fig. 6, we can observe that for Continuous Queries, I-DRNN model offers noticeable improvement over ARIMA and SVR. For ARIMA, the I-DRNN gives 15.50% and 56.46% lower RMSE for the two prediction periods respectively. Compared to SVR, the improvements become 10.50% and 10.81% respectively. The results in these two figures show that the proposed I-DRNN model outperforms both ARIMA and SVR in terms of prediction accuracy.

Fig. 7 shows prediction details of CPU usage for both SDP applications. These two figures show that 1) The proposed I-DRNN model can well catch the changes of the target feature in trend, even for a long prediction range. 2) The SVR delivers less accurate results, compared with the proposed model. 3) ARIMA fails to provide accurate prediction when the prediction range is long.




(a) Windowed URL Count




(b) Continuous Queries

Fig. 7: Prediction for CPU usage

### B. Dynamic Grouping

We validated our design and implementation of dynamic grouping with experiments using a Continuous Queries topology with 3 Query bolt executors. There were 3 tasks of Query bolt, which were assigned to 3 workers on different VMs. The corresponding results are presented in Fig. 8, which show how the tuple processing rate (i.e., the number of processed tuple per second) changes over time. We ran the Continuous Queries topology for 900s and evenly split queries to all Query bolt executors in the beginning. From time 150s to 330s (the first and second red vertical lines from the left respectively), we changed the split ratio to $7 : 3 : 2$. We can see that the tuple processing rate of these bolts change accordingly. Note that the figures shows that the new configuration starts to take effect with a very little delay. This experiment shows that our dynamic grouping method can be used to control workload distribution according to any given split ratio.

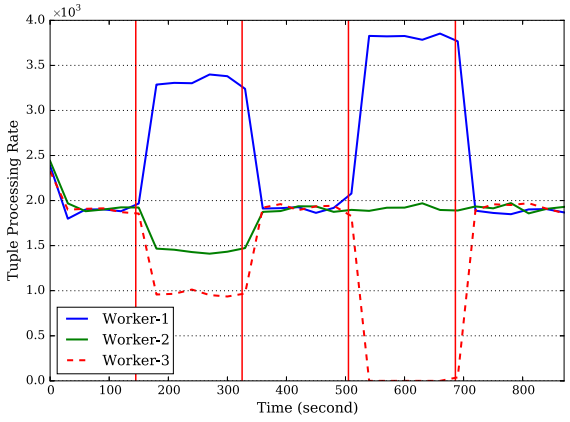



Fig. 8: Validation for dynamic grouping

We also show that dynamic grouping can be used to bypass a worker temporarily. From 500 to 690s, we set up a configuration which moves workload of worker 3 completely to worker 1. As shown in this figure (last two vertical red lines), worker 3's tuple processing rate quickly drops to 0 while worker 1's rate is doubled. These experiments validate the design and implementation of the proposed dynamic grouping method.

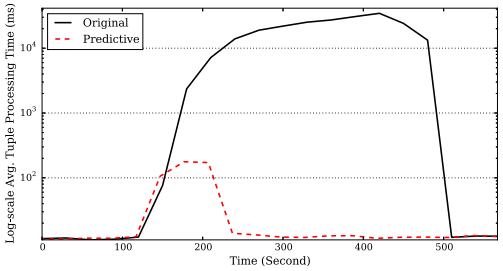
### C. Reliable Distributed Stream Data Processing

We justify effectiveness of the proposed control framework on enhancing reliability by showing how well it can deal with misbehaving workers/machines at runtime. We compared Storm with the proposed predictive control framework (labeled as "Predictive") against the original Storm (labeled as "Original") in terms of the average tuple processing time (over a topology) and failed tuple rate (the number of failed tuples per second) on affected workers.

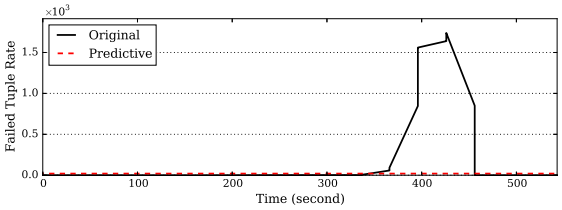
In the corresponding experiment, we injected additional workload on a VM to make the corresponding workers misbehave. Specifically, we ran four CPU hogger threads, each of which kept calculating square roots of random numbers, for every running worker on the VM for 5 minutes.

Because of high CPU consumption caused by these CPU hogger threads, the processing speed on the affected VM was unable to catch up with the tuple arrival rate, which led to slow workers and significantly increase on tuple failures. It then resulted in much longer tuple processing time. In Figure 9a, in the original Storm, the average tuple processing time skyrockets from approximately 11ms to over 34000ms due to misbehaving workers, and starts to have failed tuples. The failed tuple rate goes all the way up to 1739 tuples/second. However, with the proposed framework (in which the I-DRNN model and dymanic grouping works together), the average tuple processing time reaches its peak at 177ms at about 60sec after the workload injection starts, and then quickly drops to 13ms (normal) with no failed tuples. Similarly, for Continuous Queries, after the workload injection, we observe that the average tuple processing time rises from 90ms to more than 214ms and the failed tuple rate goes all the way up to 157 tuples/second when the original Storm is used, compared with the peak average processing time at 102ms and no failed tuples when the proposed framework is applied.

These results justify our claim that the proposed control framework can achieve minor performance degradation in terms of average tuple processing time and failed tuples.

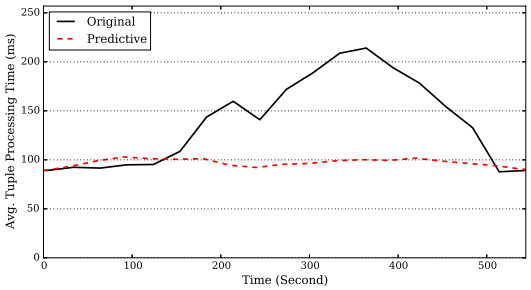



(a) Average tuple processing time

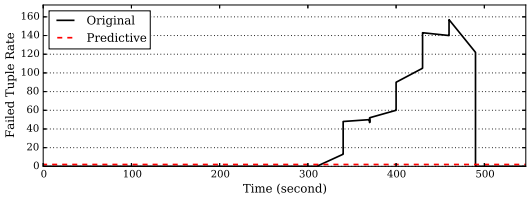



(b) Failed tuple rate

Fig. 9: Windowed URL Count




(a) Average tuple processing time




(b) Failed tuple rate

Fig. 10: Continuous Queries

## V. Related Work

**Distributed/Parallel Stream Data Processing and Reliability:** As introduced above, Storm [7] is a distributed system that is designed particularly for reliable processing of unbounded stream data. It uses a grouping policy to route tuples

between different tasks. Other similar DSDPS include: Apache S4 [4], Apache Flink [2], Microsoft's Time-Stream [33] and Google's Millwheel [1]. Currently, Apache Spark [5] may be the most popular distributed data processing platform, which has an stream extension called Spark Streaming [6]. It is an implementation of Discretized Streams (D-Streams) [43], which slices streams into small batches of time intervals before processing. In [28], Loesing *et al.* designed a DSDPS called Stormy. Unlike Storm where the user can supply their own bolts, Stormy only allows predefined processes. It provides reliability by duplicating, enforcing strong order guarantee and acknowledging events (similar to tuples in Storm). Meteor Shower [41] is a DSDPS proposed by Wang *et al.* for handling large-scale failures. Its reliability is based on parallel, asynchronous and application-aware checkpointing. In [30], the authors implemented a partial key grouping policy which provides load balancing for fields grouping using key splitting and local load estimation. This policy enables automatic tuple route adjustment by sending tuple to one of two destination tasks based on load estimation. In [17], Gu *et al.* proposed a predictive failure management approach that employs online failure prediction to achieve more efficient failure management in DSDPSs. They tested the proposed approach over IBM System S. In a recent work [25], Li *et al.* proposed a predictive scheduling framework for DSDPSs, which leverages SVR to predict the average tuple processing time for a given scheduling solution.

Unlike most of above works that use either a reactive or proactive method for supporting reliability, the proposed control framework uses a predictive approach, which can achieve minor performance degradation without any reserved resources (needed by a proactive approach). Our prediction model here is different from those presented in closely related works [17], [25]. Moreover, our objective is to design a general and flexible control framework based on performance prediction, which may be used for various purposes; so this work is different from [17] targeting particularly at failure management or [25] aiming particularly at minimizing average tuple processing time.

**Time Series Modeling and Prediction:** ARIMA [29] is one of the most popular linear model for time series prediction. In [44], Zhang *et al.* employed ARIMA to predict resource usages of VMs in a cloud computing environment and dynamically provision resources based on the prediction. Zhang [45] used a hybrid model of ARIMA and Artificial Neural Network (ANN) to improve prediction accuracy on complex problems with both linear and nonlinear correlation structures. SVR is a non-parametric regression algorithm with good scalability for high-dimensional data. In [34], Sapankevych and Sankar provided a survey of SVR on time series modeling and prediction. Connor *et al.* proposed a class of RNNs, namely NARMA in [14], which show robustness towards outlier detection with time series data. Different RNNs, especially gated RNNs, have been proposed to solve a large variety of complicated problems related to time series data such as machine translation [13], speech recognition [15] and text generation [16]. A comprehensive introduction to methods proposed for time series analysis in the literature can be found in the textbook [29].

**Memory Modeling and Management for Co-location Interference:** Methods have been proposed to model and manage memory with co-location interference in the literature. In [48], Eklov *et al.* presented a low-overhead method for accurately measuring application performance (CPI) and off-chip bandwidth (GBps) as a function of available shared cache capacity. In a later work [49], they proposed the bandwidth bandit, a general, quantitative and profiling method for analyzing the performance impact of contention for memory bandwidth on multicore machines. In [46], Casas *et al.* proposed a method for measuring and modeling the performance of hierarchical memories in terms of the application's utilization of the key memory resources: capacity of a given memory level and bandwidth between two levels. They also presented a performance measurement and analysis method for network behavior based on empirical measurements in a concurrent work [46]. Recently, Xu *et al.* [50] proposed DR-BW, a new tool based on supervised learning to identify bandwidth contention in Non-Uniform Memory Access (NUMA) architectures and provided optimization guidance. In [51], Zasadzinski *et al.* leveraged a neural network to predict job evolution based on power time series of nodes and used it to guide job termination policies.

We target at modeling co-location interference in a DSDPS, whose workload and traffic load patterns are different from those in a general distributed computing environment or in a specific environment (such as NUMA) considered in these related works. Hence, methods described above cannot be applied here. Moreover, we leverage the emerging DRNN model for our prediction task, which has not been used or considered in these related works.

## VI. Conclusions

In this paper, we presented design, implementation and evaluation of a novel predictive control framework to enable reliable DSDP. It has the following desirable features: 1) It makes accurate performance prediction with careful consideration for co-location interference using a two-tiered DRNN model. 2) It employs a new grouping method, dynamic grouping, which can distribute/re-distribute data tuples to downstream tasks according to any given split ratio and can re-direct data tuples to bypass misbehaving workers and/or machines according to prediction. 3) It collects both process and machine level runtime statistics with over 70 features to enable accurate prediction. We implemented the proposed framework based on Storm. We built two representative SDP applications for performance evaluation: Windowed URL Count and Continuous Queries. Extensive experimental results show 1) the proposed DRNN model outperforms ARIMA and SVR in terms of prediction accuracy; 2) dynamic grouping works as expected; and 3) the proposed framework enhances reliability by offering minor performance degradation with misbehaving workers.

## References


[1] T. Akidau, *et al.* , MillWheel: fault-tolerant stream processing at internet scale. *Proc. of VLDB Endowment*, Vol. 6, No. 11, pp. 1033-1044, 2013.

[2] Apache Flink, *https://flink.apache.org/*.

[3] Apache Kafka, *http://kafka.apache.org/*.
[4] Apache S4, *http://incubator.apache.org/s4/*.
[5] Apache Spark, *http://spark.apache.org/*.
[6] Apache Spark Streaming, *http://spark.apache.org/streaming/*.
[7] Apache Storm, *http://storm.apache.org/*.
[8] Apache Thrift, *https://thrift.apache.org/*.
[9] Apache Zookeeper, *https://zookeeper.apache.org/*.
[10] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. *Proc. of ACM GPGPU Workshop*, pp. 94-103, 2010.
[11] Y. Bengio, A. Courville and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 35, No. 8, pp. 1798-1828, 2013.
[12] Z. Chen, J. Xu, J. Tang, K. Kwiat and C. Kamhoua. G-Storm: GPU-enabled high-throughput online data processing in Storm. *Proc. of IEEE BigData'2015*, pp. 307-312, 2015.
[13] K. Cho, B. Van Merrinboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proc. of EMNLP'2014*, pp. 1724-1734, 2014.
[14] J. T. Connor, R. D. Martin and L. E. Atlas. Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, Vol. 5, No. 2, pp. 240-254, 1994.
[15] A. Graves, A. R. Mohamed and G. Hinton. Speech recognition with deep recurrent neural networks. In *Proc. of IEEE ICASSP'2013*, pp. 6645-6649, 2013.
[16] A. Graves, Generating sequences with recurrent neural networks, arXiv:1308.0850, 2013.
[17] X. Gu, S. Papadimitriou, P. S. Yu, S-P. Chang. Toward predictive failure management for distributed stream processing systems, *Proc. of IEEE ICDCS'2008*, pp. 825-832, 2008.
[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, Vol. 9, No. 8, pp. 1735-1780, 1997.
[19] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, Vol. 22, No. 4, pp. 679-688, 2006.
[20] R. J. Hyndman and Y. Khandakar. Automatic time series for forecasting: the forecast package for R. *Journal of Statistical Software*, Vol. 27, No. 1, pp. 1-22, 2008.
[21] Java Management Extensions, *http://openjdk.java.net/groups/jmx/*.
[22] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. *Proc. of ACM STOC'1997*, pp. 654-663, 1997.
[23] Lasagne, *https://github.com/Lasagne/Lasagne*.
[24] Y. LeCun and Bengio, Y. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, Vol. 3361, No. 10, 1995.
[25] T. Li, J. Tang and J. Xu. A predictive scheduling framework for fast and distributed stream data processing. *Proc. of IEEE BigData'2015*, pp. 333-338, 2015.
[26] C. H. Liu, J. Xu, J. Tang and J. Crowcroft, Social-aware sequential modeling of user interests: a deep learning approach, *IEEE Transactions on Knowledge and Data Engineering*, In press.
[27] LMAX disruptor, *https://lmax-exchange.github.io/disruptor/*.
[28] S. Loesing, M. Hentschel, T. Kraska and D. Kossmann. Stormy: an elastic and highly available streaming service in the cloud. *Proc. of the 2012 ACM Joint EDBT/ICDT Workshops*, pp. 55-60, 2012.
[29] D. C. Montgomery, C. L. Jennings and M. Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015.
[30] M. A. U. Nasir, G. De Francisci Morales, D. Garca-Soriano, N. Kourtellis and M. Serafini, The power of both choices: practical load balancing for distributed stream processing engines, *Proc. of IEEE ICDE'2015*, pp. 137-148, 2015.
[31] R. Pascanu, C. Gulcehre, K. Cho and Y. Bengio. How to construct deep recurrent neural networks, arXiv:1312.6026, 2013.
[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, ... J. Vanderplas. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, Vol. 12, No. 10, pp. 2825-2830, 2011.
[33] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang and Z. Zhang. Timestream: reliable stream computation in the cloud. *Proc. of ACM EuroSys'2013*, pp. 1-14, 2013.
[34] N. I. Sapankevych and R. Sankar. Time series prediction using support vector machines: a survey. *IEEE Computational Intelligence Magazine*, Vol. 4, No. 2, pp. 24-38, 2009.
[35] A. J. Smola and B. Schlkopf. A tutorial on support vector regression. *Statistics and computing*, Vol. 14, No. 3, pp. 199-222, 2004.
[36] SQLite, *https://www.sqlite.org/*.
[37] Streaming windowed word count, *https://cloud.google.com/dataflow/examples/wordcount-example*.
[38] Sysstat, *http://sebastien.godard.pagesperso-orange.fr/*.
[39] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, ... A. Belopolsky. Theano: A Python framework for fast computation of mathematical expressions, arXiv:1605.02688, 2016.
[40] G. Urdaneta, G. Pierre and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, Vol. 53, No. 11, pp. 1830-1845, 2009.
[41] H. Wang, L. S. Peh, E. Koukoumidis, S. Tao and M. C. Chan. Meteor shower: a reliable stream processing system for commodity data centers. *Proc. of IEEE IPDPS'2012*, pp. 1180-1191, 2012.
[42] J. Xu, Z. Chen, J. Tang and S Su. T-Storm: traffic-aware online scheduling in storm. *Proc. of IEEE ICDCS'2014*, pp. 535-544, 2014.
[43] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. *Proc. of ACM SOSP'2013*, pp. 423-438, 2013.
[44] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. *Proc. of ACM ICAC'2012*, pp. 145-154, 2012.
[45] G. P. Zhang, Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing*, Vol. 50, pp. 159-175, 2003.
[46] M. Casas and G. Bronevetsky, Active measurement of memory resource consumption, *Proc. of IEEE IPDPS'2014*, pp. 995–1004.
[47] M. Casas and G. Bronevetsky, Active measurement of the impact of network switch utilization on application performance, *Proc. of IEEE IPDPS'2014*, pp. 165–174.
[48] D. Eklov, N. Nikoleris, D. Black-Schaffer and E. Hagersten, Cache pirating: measuring the curse of the shared cache, *Proc. of ICPP'2011*, pp. 165–175.
[49] D. Eklov, N. Nikoleris, D. Black-Schaffer and E. Hagersten, Bandwidth bandit: quantitative characterization of memory contention, *Proc. of IEEE/ACM CGO'2013*.
[50] H. Xu, S. Wen, A. Gimenez, T. Gamblin, X. Liu, DR-BW: identifying bandwidth contention in NUMA architectures with supervised learning, *Proc. of IEEE IPDPS'2017*, pp. 367–376.
[51] Michal Zasadzinski, Victor Munts-Mulero, Marc Sol, David Carrera, Thomas Ludwig, Early termination of failed HPC jobs through machine and deep learning, *Proc. of Euro-Par'2018*.