FLOWPROPHET: Generic and Accurate Traffic Prediction for Data-parallel Cluster Computing

*Hao Wang
SJTU and HKUST

Li Chen
HKUST

Kai Chen
HKUST

*Ziyang Li
NUDT and HKUST

Yiming Zhang
NUDT

Haibing Guan
SJTU

Zhengwei Qi
SJTU

Dongsheng Li
NUDT

Yanhui Geng
Huawei

***Abstract*—Data-parallel computing frameworks (DCF) such as MapReduce, Spark, and Dryad *etc*. have tremendous applications in big data and cloud computing, and throw tons of flows into data center networks. In this paper, we design and implement FLOWPROPHET, a general framework to predict traffic flows for DCFs. To this end, we analyze and summarize the common features of popular DCFs, and gain a key insight: since application logic in DCFs is naturally expressed by directed acyclic graphs (DAG), DAG contains necessary time and data dependencies for accurate flow prediction. Based on the insight, FLOWPROPHET extracts DAGs from user applications, and uses the time and data dependencies to calculate flow information 4-tuple, (`source`, `destination`, `flow_size`, `establish_time`), ahead-of-time for all flows. We also provide generic programming interface to FLOWPROPHET, so that current and future DCFs can deploy FLOWPROPHET readily. We implement FLOWPROPHET on both Spark and Hadoop, and perform extensive evaluations on a testbed with 37 physical servers. Our implementation and experiments demonstrate that, with time in advance and minimal cost, FLOWPROPHET can achieve almost 100% accuracy in source, destination, and flow size predictions. With accurate prediction from FLOWPROPHET, the job completion time of a Hadoop TeraSort benchmark is reduced by 12.52% on our cluster with a simple network scheduler.**

I. INTRODUCTION

Data-parallel computing frameworks (DCFs) such as MapReduce [1], Dryad [2], Spark [3], *etc.* have tremendous applications, especially in big data and cloud computing. DCFs greatly enhance programmers' productivity by abstracting away implementation details, so that the programmers can focus on the application logic without worrying about resource contention, task distribution, and so on. They only need to apply the APIs (*e.g.*, `filter()`, `map()`, `reduce()`) to express their logic and manipulate their dataset as if on a single machine.

DCFs effectively decouple the detailed distributed computing implementation from the user programs. However, lower level implementation details hold the key to better application performance, and lots of research efforts have been spent along this direction recently. On the micro level, flow-based optimization mechanisms (*e.g.*, [4]–[8]) attempt to minimize average completion time of flows or groups of flows by exploiting flow size provided by the applications. On the macro level, architectural bandwidth provisioning (*e.g.*, [9]–[12]) and traffic engineering (*e.g.*, [13]–[15]) solutions try to estimate aggregate application traffic demands to enable dynamic network resource allocation. Note that both approaches depend on predicting the future: the traffic and flow information has to be known *ahead-of-time*.

Predicting the future is inherently difficult, and most existing solutions settle on using heuristic algorithms or measuring network level parameters, such as flow counters [9, 13] and socket buffer occupancy [10, 16]. However, these methods are in essence reacting to traffic, rather than predicting, and therefore result in poor performance [17].

More recently, an application level traffic forecasting solution, HadoopWatch [18], derives traffic through measuring task assignments and data size indications on file systems at the master and worker nodes in Hadoop. However, this method is customized for Hadoop, and only works when the underlying application logic is as simple as Hadoop, which can be described in 2 stages: map and reduce. When the application logic becomes more complex, this method is uncontrollable and inaccurate (or incorrect) because it does not know *where*, *what* and *when* to collect useful information. For example, in Spark [3], there are multiple stages, and stages that are consecutive in time may or may not have data dependencies when doing lazy evaluation [3]. In fact, accurate traffic prediction requires the knowledge of time and data dependencies, which are closely related to the applications logic and the corresponding representations of DCFs.

In this paper, we seek a generic and accurate method to predict flow information for data-parallel cluster computing frameworks. We specifically set our design goals as follows:

- **Generic:** We should devise a general interface for traffic prediction that works for all current and future DCFs. To this end, we should have a general description of application execution patterns in order to express complex application logic.
- **Accurate and fined-grained:** The method must be able to provide accurate flow level information, rather than coarse aggregated traffic demand, to enable fine-grained network control and optimization. The method should also provide detailed inter-flow dependency information to feed recent coflow optimizations [7, 8].
- **Ahead-of-time:** The method must be able to predict the flows before they enter the network; and ideally it should also estimate the flow `establish_time` accurately.
- **Scalable and low-overhead:** The method should be able to work at large scale and introduce as little overhead to the DCFs as possible.

*This work was performed when Hao Wang and Ziyang Li were intern students at SING Group @ HKUST.

In essence, we aim to calculate the 4-tuple (`source`, `destination`, `flow_size`, `establish_time`) for each flow. The intention of the first three elements is straightforward. The `establish_time` is used to determine the exact time when a flow will establish (*e.g.*, a network scheduler will need to make a scheduling decision before the flow establishes). Thus, we need to know both the logical order of data processing and the locations and sizes of data partitions. To this end, we examine prevalent DCFs and identify the key observation (details in Section II): since application logic is naturally represented by directed acyclic graphs (DAG) in all DCFs, DAG contains necessary time and data dependencies for accurate flow prediction. With DAG, we can explicitly know *where*, *what*, and *when* information to measure in order to accurately calculate flow information for complex parallel computing applications.

Based on the insights, we present FLOWPROPHET, a general framework to predict flow information for all DCFs. FLOWPROPHET extracts DAG from data-parallel applications, then uses the DAG to guide the measurement and prediction. In the course of design and implementation of FLOWPROPHET, we make the following contributions:

- We analyze and summarize the common execution patterns of popular computing frameworks, and extract DAG to obtain time and data dependencies from applications using these frameworks to guide the flow prediction.
- We design FLOWPROPHET, a lightweight, generic, and accurate flow information prediction framework for DCFs. The application programming interface (API) of FLOWPROPHET is general, so that existing and future computing frameworks can readily use FLOWPROPHET to generate accurate flow information.
- We have implemented FLOWPROPHET on the most popular frameworks such as Hadoop and Spark, and build a real testbed with 37 servers to evaluate it. Our experiments show that with time in advance and negligible overhead to application performance, FLOWPROPHET can achieve almost 100% accuracy in source, destination, and flow size predictions.
- Using accurate prediction from FLOWPROPHET, we show that even a simple network level optimization can greatly improve application performance. In our experiment, the job completion time of a Hadoop TeraSort-25G benchmark is reduced by 12.52% on our 37-server cluster.

The rest of this paper is organized as follows. Section II introduces the key observation that motivates us to leverage DAG to predict flow information. Section III presents the design and implementation of FLOWPROPHET. Section IV discusses the evaluation benchmarks and results of FLOWPROPHET. Section V reviews the related works. Section VI concludes the paper.

II. DAG-ASSISTED FLOW PREDICTION

In this section, we examine how DAG assists the calculation of flow information (summarized in Figure 1). We first delve into the typical application life-cycle in popular DCFs, and then establish the relationships between application logic, execution sequence, DAG, and data movement. Finally, we demonstrate the practical calculation steps of flow information prediction using DAG.

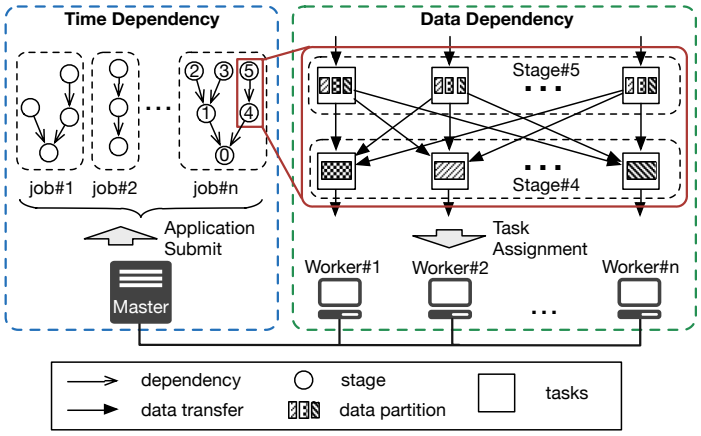


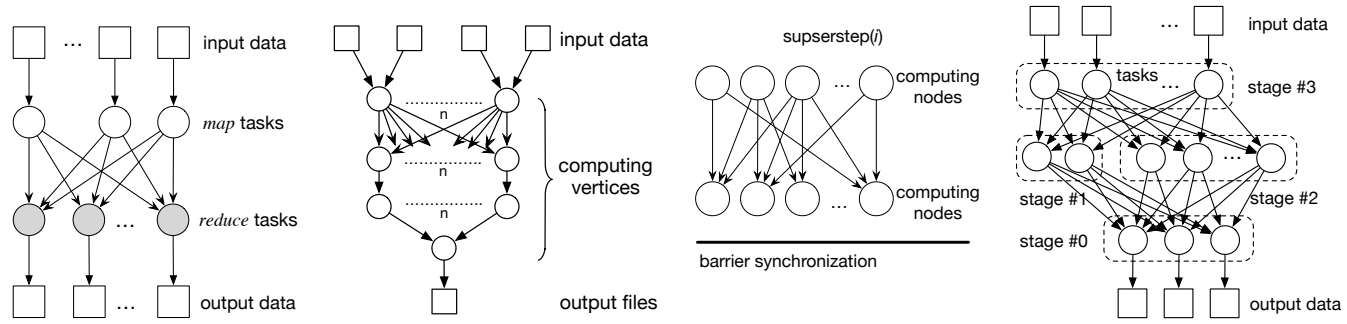
Fig. 1: Data-parallel computing framework: application logic and data movement.

Application Life-cycle: In DCFs, there is a gap between the application logic and the actual operations in the backend cluster, which may contain thousands of CPU cores, because user application only concerns with a single machine during development to lower complexity. To achieve scalable performance, DCFs automatically discover and exploit parallelism from user's application logic, and distribute parallel computational tasks to every computing node.

The life-cycle of a user application is described in Figure 1. At the start, user application is resolved into *jobs*[1]. For each job, DCFs calculate the order of executions and data dependency, which can be described by a DAG, as shown in Figure 1. Specifically, DCFs identify which tasks have dependency on which data partition, and plan the parallel executions of the application. These tasks are aggregated into a *stage*. Then, the *tasks* in a stage are assigned to workers, and the parallel operations on the dataset are launched. The nodes in DAG are stages, and the arcs represent dependency between stages. Data transfer occur only during stage transitions.

Almost all popular DCFs describe their operations in DAGs. For example, Dryad's [2] execution engine is driven by a graph description language, which empowers the developer with explicit graph construction. Pregel [20], which is based on Bulk Synchronous Parallel (BSP), adopts a sequence of supersteps to construct user application. Every superstep contains a data communication phase and a barrier synchronization phase, which is essentially a DAG with two vertices and one edge. Spark [3] defines a novel structure named Resilient Distributed Dataset (RDD) which expresses DAG with RDD lineage. Spark provides transformations, and actions (*e.g.*, `union()`, `join()`, `filter()`, `map()`, `take()`, *etc.*) to build RDD lineage and explicitly express algorithm logic. Compared with previous framework, MapReduce [1] (or Hadoop [19]) is much simpler. Its two primitive semantics: `map` and `reduce` can also be regarded as a DAG contains only two vertices and one edge. CIEL [21] develops a language named Skywriting [22] and a series of operators (*e.g.*, `exec()`, `spawn()`, `map()`, *etc.*) to express task-level parallelism in DAG.

[1]Iterative applications with termination criterions will be divided into dependent jobs: each will check the termination criterion to decide whether to move on to the next.



(a) Data shuffle between mappers and reduces in Hadoop [19]

(b) Data channels between computing vertices in Dryad [2]

(c) Data communication in one superstep of Bulk Synchronous Parallel (BSP) in Pregel [20]

(d) Data shuffle between stages in Spark [3]

Fig. 2: Data movement patterns.

Observation: DAG contains necessary time, data, and flow dependencies for accurate flow prediction.

Time dependency: Time-dependency refers to the execution order of stages. DCFs process the DAG one node (stage) at a time in a depth-first-traversal order [3], and generate this order. Stages may execute parallel in time, while others have to wait for completion of parent stages. Traffic is only generated between parent and child stages, and with DAG, we know *when* the flow transmission will occur.

Data dependency: DCFs maintain the life cycle of data: import, transfer, storage and export. First, data imported into the cluster will be split and distributed to the entire cluster. Then, DCFs assign computation tasks to each node based on data locality and resource scheduling scheme. Along with the execution of computation tasks, intermediate data is generated and cached locally. In Hadoop (Figure 2(a)), a JobTracker informs reducers when and where (*i.e.* which mapper node) to fetch data to perform `reduce` tasks. In Dryad, data channels are maintained between computing vertices (Figure 2(b)), and data flows along these channels. For Pregel, a superstep requires all the computing node to exchange data by barrier synchronization before the next superstep (Figure 2(c)). For Spark, data shuffle takes place between specific stages based on the dependency recorded in RDD lineage (Figure 2(d)). In summary, since every process of the data life cycle is conducted by DCFs, DCFs are capable of exporting location and size of every piece of intermediate data and final results.

Since traffic is essentially data movement, flow prediction requires knowing *where*, *what* and *when* the data is moved, and such information can be retrieved from the DAG. When a stage (a node in DAG) relies on the output of a group of stages (every stage in this group is called the stage's parent), it has to wait until all the parents are finished. Concurrently running stages do not have data dependency on each other. Thus, we can infer from the DAG the source (parent stages), destination (child stage), size (amount of data required), and time (upon completion of all parent stages) of the transmission of data between stage transitions.

Flow dependency: The data flows generated between consecutive stages are inter-dependent, because they usually share common communication requirements and objectives (Figure 2). Flow dependency refers to an important concept of coflow [23], which defines a semantically related collection of flows. We observe that edges in DAG can be naturally used to identify coflows in DCFs, which provides valuable information for coflow-based optimization mechanisms such as [7, 8].

Calculating flow information with DAG: Inspired by our observations, we can design a general method to calculate flow information 4-tuple, (`source`, `destination`, `flow_size`, `establish_time`) by developing a set of interfaces to: 1) output stage context[2], and to 2) extract locations and sizes of data partitions.

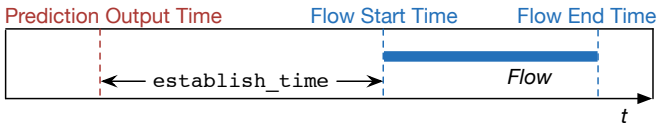


Fig. 3: An example of `establish_time`

At the high level, the 4-tuple is calculated as follows (detailed design and implementation in Section III):

- `source`: we look for the current stages in DAG, and identify the data partitions that need to be transferred. The worker node containing the data is the source.
- `destination`: we look for next stages in DAG, and identify which worker node will work on which piece of data. Thus, the destinations of the data can be identified.
- `flow_size`: we use the interface to look up sizes of data partitions to be transmitted.
- `establish_time`: as depicted in Figure 3, FLOWPROPHET outputs prediction information of a flow at the Prediction Output Time, and the flow begins at the Flow Start Time. The `establish_time` is defined as the time period between the Prediction Output Time and the Flow Start Time. We develop a heuristic algorithm to estimate the expected establishing time intervals for subsequent flows. This algorithm is adaptive to the application and the DCF.

III. FLOWPROPHET DESIGN AND IMPLEMENTATION

We introduce the design and implementation of FLOWPROPHET in this section. First, we dissect the flow information prediction in DCFs into several sub-problems, and describe our solutions (§ III-A). Then, we present the workflow of FLOWPROPHET to show how different components work together

[2]Stage context includes current stage, next stage, and the dependency between them.

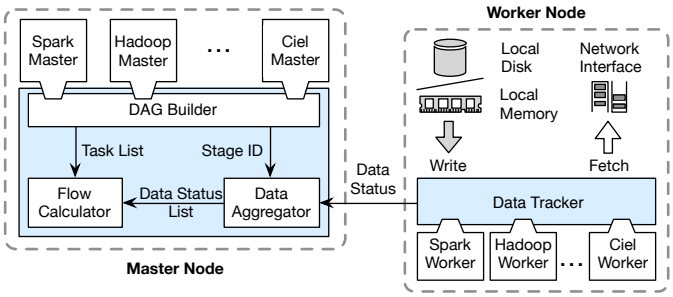


Fig. 4: The architecture of FLOWPROPHET.

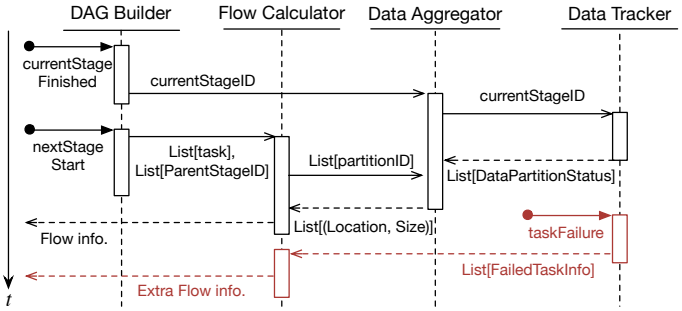


Fig. 5: Sequence diagram begins with an event that current stage is finished.

(§ III-B). Finally, we go through the implementation details of each component of FLOWPROPHET in § III-C.

A. FLOWPROPHET *Overview*

Figure 4 depicts the architecture of FLOWPROPHET, which contains 4 modules: DAG Builder, Data Tracker, Data Aggregator, and Flow Calculator (functions explained below). FLOWPROPHET is attached to DCFs to enable flow prediction. When implementing a general framework to predict the 4-tuple (`source`, `destination`, `flow_size`, `establish_time`) for every upcoming flow in DCFs, we are essentially solving the following sub-problems:

How to extract the full DAG? The DAG is the pivot for predicting flow information for DCFs. On the master node of DCFs, the DAG Builder builds a full DAG by parsing event messages from the DCF master interfaces.

How to collect data partition status? When a stage is completed, the computation result is kept as a data partition in local disk or local memory of each worker node separately. A data partition status contains the `stage_ID`, `partition_ID` and `size`. The Data Tracker receives event messages from DCF worker interfaces and maintains a data structure to record all data partition status. The Data Aggregator requests the status of each data partition from the Data Tracker on each worker.

How to be scalable and lightweight? We pursue scalability and low-overhead in the design of FLOWPROPHET. All modules in FLOWPROPHET follow the principles of Actor Model to exchange messages. The Actor Model is an asynchronous programming model for distributed applications [24]. The actors are fairly lightweight concurrent entities. They process messages asynchronously using an event-driven receive loop. The Actor Model is capable of offering a high level of abstraction for achieving high concurrency and parallelism.

B. FLOWPROPHET *Workflow*

Figure 5 depicts how modules of FLOWPROPHET cooperate to predict upcoming flows when a stage is finished. When the DAG Builder receives a message that current stage is finished, the DAG Builder checks whether there will be traffic between the current stage and the next stage. If yes, the DAG Builder will send the current stage ID to ask the Data Aggregator to collect data partition status from each Data Tracker. After the Data Aggregator finishes the collection, FLOWPROPHET knows the locations and sizes of all data partitions. Then, when DAG Builder is notified that a new stage is beginning, it will send the stage context to the Flow Calculator. The stage context contains the tasks and parent stage IDs of the next stage. Each task is identified by (`partition_ID`, `executor_ID`, `func`). The Flow Calculator then combines and matches the task list and stage list with data partition status list to output the (`source`, `destination`, `flow_size`) for each flow. Note that task failures will cause corresponding data partitions to be transmitted again. FLOWPROPHET handles task failures as follows: Data Trackers receives task failure events from the DCF worker and notify Flow Calculator of the extra flow information. Further, the Flow Calculator obtains the `establish_time` by a heuristic algorithm.

C. FLOWPROPHET *Implementation*

We now describe the implementation of the 4 modules of FlowProphet in detail. We implement FLOWPROPHET with Scala 2.10.4. We apply the actor model based on Akka 2.3.4 framework [25], which enables each FLOWPROPHET module to communicate asynchronously and concurrently at low overhead. Besides, to export DCF intrinsic information, we have also implemented the APIs for the master and workers of Spark 1.0.0 and Hadoop 0.20.2.

Event Definition	Trigger Condition
newStageEvent(stageID, childStageID)	a new stage is created
stageStartEvent(List[task], stageID)	a stage is beginning
stageFinishedEvent(stageID)	a stage is finished

TABLE I: The required APIs for DCF master.

DAG Builder: The DAG Builder relies on the information provided by DCFs to build a full DAG. DCF developers only need to develop a set of simple interfaces providing primitive events, which are outlined in Table I. Similar to the DAG Builder, the Data Tracker also calls for notification of events from the DCF worker.

`DAGBuilder` Handlers
newStageHandler(newStageEvent) ⇒ (currentStage, childStage)
stageStartHandler(stageStartEvent) ⇒ Event(List[task], List[stageID])
stageFinishedHandler(stageFinishedEvent) ⇒ Event(stageID)

TABLE II: The DAG Builder event handlers.

When a new stage is created in DCF, a `newStageEvent` will be raised. The DAG Builder obtains the new stage ID and its child stage ID. By handlers defined in Table II, the DAG Builder constructs a full DAG from all the collected pairs of parent and child stages.

DCFs process stages in a depth-first-traversal order, and traffic does not always take place between two consecutive stages. To provide accurate prediction, it is necessary to check the data dependency between current stage and next stage. For example, in Figure 1 job #n, traffic only happens at following three moments: after stage 2 and stage 3 both complete, after stage 5 completes, and after stage 1 and stage 4 both complete.

Furthermore, the `stageStartEvent` contains a list of tasks and the stage ID. In each task, the `executor_ID` is where the task to be executed; the `partition_ID` indicates the data partition that the task will fetch; the `func` is a set of nested procedures, which could be executed independently.

Data Aggregator: To manage all the Data Trackers, we place a Data Aggregator on the master, which organizes partition status from Data Trackers and exports a query interface for the Flow Calculator (Table III).

`DataAggregator` Methods	Caller
query(List[partitionID, stageID]) ⇒ List[(location, size)]	`FlowCalculator`

TABLE III: The Data Aggregator API.

When the Data Aggregator receives a stage ID from the DAG Builder, it will broadcast the stage ID to all the Data Trackers. Each Data Tracker then replies with a list of data partition status for the stage ID. Then the Data Aggregator will build a HashMap to cache these data partition status with the stage ID as the key. Besides, the Data Aggregator will append each data partition status with a location field, which is the IP address or hostname of the worker that keeps the data partition.

In DCFs, there could be thousands of workers or more, which means that there are the same number of Data Trackers. Leveraging the Actor Model, all the messages sent from the Data Trackers actors are placed in the mailbox of the Data Aggregator actor. Then the Data Aggregator processes messages in an asynchronous, non-blocking way.

Once the Data Aggregator receives a query request from the Flow Calculator, it will reply with a list of location and size for each data partition matching the stage ID.

Data Tracker: Similar with the DAG Builder relying on primitive information from the DCF master, a Data Tracker receives and records event messages from the DCF worker. The event message is defined in Table IV.

Event Definition	Trigger Condition
taskFailureEvent(taskID, stageID, partitionID)	a task is failed
taskFinishedEvent(stageID, partitionID, size)	a task is finished

TABLE IV: The required APIs for DCF worker.

The computation takes place on each worker in DCFs, *i.e.*, the `func` encapsulated by each task will be extracted and executed by executors. In general, the computation results will be written back to the local disk (*e.g.*, Hadoop), or for high performance, in local memory (*e.g.*, Spark). Besides, most DCFs designed to be fault-tolerant, and they only attempt re-execution of failed tasks for limited times. To predict extra flows generated by tasks re-execution, Data Tracker needs to be notified of failed tasks. It is simple to implement the required APIs by adding less than 50 lines of code in DCF task life-cycle context.

The Data Tracker constructs a HashMap with the stage ID as the key, and a list of partition IDs and sizes as the value. The Data Tracker will update the HashMap when the `taskFinishedEvent` is raised by the DCF interface. Then, when the Data Aggregator requests status of data partitions of a stage ID, the Data Tracker then replies with a list, in which each piece of data partition is recorded as `stage_ID`, `partition_ID` and `size`.

`DataTracker` Methods	Caller
query(stageID) ⇒ List[(stageID, partitionID, size)]	`DataAggregator`

TABLE V: The Data Tracker API.

Flow Calculator: The Flow Calculator is the converging point of knowledge on time dependency and data dependency, and it calculates the flow information (`source`, `destination`, `flow_size`), and estimates the flow `establish_time`.

Flow information: Once the DAG Builder captures the `stageStartEvent`, it will deliver two lists to the Flow Calculator. One list contains the tasks that are just starting, the other contains all the parent stage IDs. By traversing the list of tasks, the Flow Calculator queries the Data Aggregator for the location and size related to a data partition that each task will fetch. Thus, the location of data partition is the `source`, the `executor_ID` indicates the `destination`, and the size of data partition is the traffic volume `flow_size`. Since the predicted flows will not take place until all the tasks on the master are delivered to the designated workers, the Flow Calculator will most likely export flow information in advance. As is shown in our experiments in Section IV, FLOWPROPHET can predict flow information strictly ahead of time.

Flow establish time: FLOWPROPHET is able to calculate flow information of the next stage ahead of time. After the current stage is completed, DCFs usually do a relatively fixed number of operations to start the next stage, and we refer this period of time as flow `establish_time`. For a specific application, the `establish_time` is likely to fall within a range. This is confirmed by our experiments (Figure 7), which `establish_times` *all* exhibit heavy-tailed distribution in different DCFs. The majority of `establish_times` concentrate in the small range with some occasional outliers (*e.g.* network congestion).

However, different configurations of DCFs and applications may result in different the `establish_times`, and it is difficult to accurately predict for all DCFs and all applications. Therefore, we introduce an adaptive algorithm to infer `establish_time` of flows of different applications.

For an application, the algorithm tracks the average and variance of `establish_time` of the previous flows via the exponentially weighted moving average (EWMA) method [26]. EWMA has less lag than naive moving average method, and is more sensitive to recent `establish_times`, which fits our goal of tracking current applications. We describe the estimation method as follows:

Let t_i be the expected establish time in the ith stage and σ the standard deviation. It follows that the `establish_time`

of the next flow will most likely[3] fall between $t_i \pm 2\sigma$. FLOWPROPHET uses the well-known set of formulas [27] to perform online update of mean (t_i) and standard variance (σ):

$$t_i = \alpha\tau + (1-\alpha)t_{i-1}$$
$$\sigma_n^2 = (1-\alpha)(S_{n-1} + \alpha(\tau - t_{i-1})^2)$$

where τ is the latest measured establish time, and S_n is an accumulated variable

$$S_n = (1-\alpha)S_{n-1} + \alpha(\tau - t_i)(\tau - t_{i-1})$$

α is the smoothing factor (FLOWPROPHET uses $\alpha = 0.15$ for Spark and $\alpha = 0.5$ for Hadoop).

D. Discussion

FLOWPROPHET offers simple, flexible, and fine grained interfaces to predict flow information, and they are able to adapt to a wide range of scenarios. Here we remark on some specific cases:

Ambiguous DAG : Under some configurations of DCFs, there might be no clear boundary between stages. For example, Hadoop reducers can be configured to start shuffling before all mappers finish. Such configuration will not affect the predictions of FLOWPROPHET, because the Data Tracker reports to Data Aggregator whenever a task finishes. FLOWPROPHET is able to predict upcoming flows when the requested data partitions of new-launched tasks are ready.

Speculative task assignment : To address the heterogeneity of data center and data skewness, DCFs could be configured to execute copies of a task on multi-node and accept the earliest output, which introduces uncertainty to identify source or destination of flows. We note that redundant task copies will not interfere with FLOWPROPHET's prediction since the Data Aggregator only records the first arrived report for the same data partition.

Multi-tenancy : As a prototype, FLOWPROPHET currently does not support multi-tenancy, and it is part of our on-going effort. To enable FLOWPROPHET in a multi-tenant cluster, we plan to extend the argument lists of FLOWPROPHET APIs with user IDs (stages, jobs, tasks, and flows will be tagged with a user ID). Supporting multi-tenancy does not change the flow prediction mechanisms of FLOWPROPHET.

IV. EVALUATION

Testbed: We have implemented FLOWPROPHET and deployed it on our 37-server testbed (master node $\times$ 1, worker node $\times$ 36). The 37 physical servers are Dell PowerEdge R320 with a quad core Intel Xeons E5-1410 2.8GHz CPU, 24GB DDR3 memory, 500GB hard disk and one Broadcom NetXtreme Gigabit Ethernet NIC. The OS is Debian 6.0 64-bit version with kernel 2.6.32-5. We deployed FLOWPROPHET on Spark 1.0.0 and Hadoop 0.20.2 with Oracle JDK 1.7.0_25.

For accurate time measurement in a distributed setting, we deploy NTP [28] on the master node and worker nodes to synchronize system clock. We run TShark 1.2.11 on each node to capture all TCP packets, and save the captured files for further analysis.

[3]with probability of 95% assuming normal distribution of `establish_time`

Benchmarks: In our experiments, we use following benchmark applications and configuration to evaluate FLOWPROPHET:

- *WikipediaPageRank*, is a PageRank algorithm instance using Wikipedia entries as input. We process 13G and 26G Freebase-wiki-articles datasets [29] separately on our Spark testbed.
- *SparkPageRank*, is also a standard PageRank algorithm in Spark. Its input dataset is Freebase Triples [30]. The Resource Description Framework (RDF) data is serialized using the N-Triples format, compressed with Gzip and encoded in UTF-8. We prepare a 40G RDF dataset for SparkPageRank.
- *Spark K-means*, K-Means clustering is a popular clustering algorithm that can be used to partition a dataset into K clusters. The input dataset is Wikipedia Page Traffic Statistics [31].
- *Hadoop TeraSort*, is a common Hadoop benchmark which aims at testing the CPU/memory power of the cluster. It is a standard Map/Reduce sort, except for a custom partitioner that uses a sorted list of $N-1$ sampled keys that define the key range for each reduce.
- *Pi*, is a Monte Carlo Method to calculate an approximation to π. A parameter is used to setup the number of random points. We run π calculation on both Spark and Hadoop.
- *WordCount*, this benchmark reads text files and counts how often words occur. We prepared a 20G and a 40G dataset for Spark and Hadoop, which contain 3,560,179,980 words and 7,153,321,364 words respectively.

Summary of results: The main highlights of our results are as follows:

- **Time advance**: FLOWPROPHET can predict flows strictly ahead of time. In our experiments of WikipediaPageRank with Spark, FLOWPROPHET achieves an average lead time of $414.1ms$ on 13G dataset and $478ms$ on 26G dataset. The lead time is higher for Hadoop applications: $12.3123s$ for Hadoop TeraSort 10G dataset, and $7.7348s$ on Hadoop WordCount 20G.
- **Prediction accuracy**: We conduct a set of experiments to evaluate the accuracy on traffic volume. In all the experiments such as Spark WikipediaPageRank, Spark PageRank, Hadoop WordCount *etc.*, the prediction accuracy of FLOWPROPHET on source, destination and flow size reaches almost 100%. For majority (85%-90%) of the flows in our experiment, we can well predict their `establish_time` intervals.
- **Overhead and scalability**: We measure the difference of job completion times when FLOWPROPHET is enabled and disabled. The experiments are conducted on different scales of Spark and Hadoop clusters. We find the overhead FLOWPROPHET introduces is negligible—stably around 0.64%. Furthermore, this small overhead maintains when the cluster scale becomes larger.
- **Benefits of FLOWPROPHET**: In cooperation with a simple network scheduler, FLOWPROPHET's accurate flow prediction can directly improve the average job completion time by 12.52% for a Hadoop TeraSort 25G dataset on our 37-server cluster.

Next we describe and explain the experiment details.

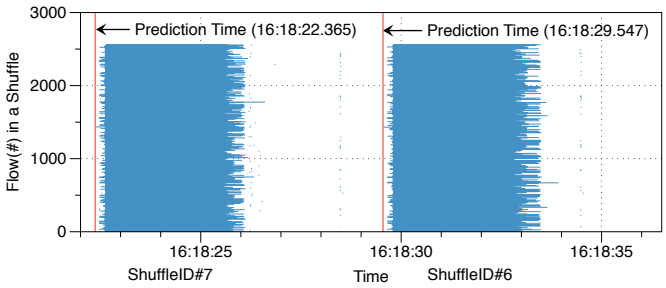
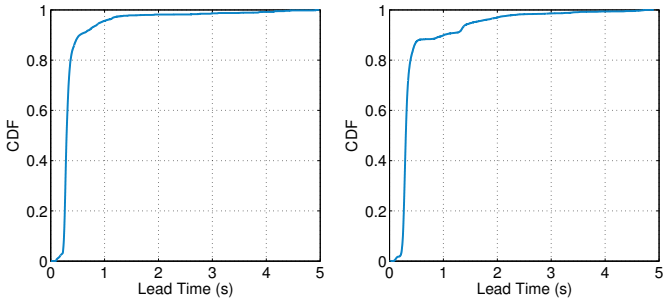


Fig. 6: Time Advance of WikipediaPageRank-13G (Spark).



(a) Spark WikipediaPageRank-13G (b) Spark WikipediaPageRank-26G

(c) Hadoop TeraSort-10G (d) Hadoop WordCount-20G

Fig. 7: CDF of lead time.

Time advance: To calculate the lead time, we record the time when FLOWPROPHET predicts flow information while running the applications. After completion, we extract traffic timing from TShark captured files and compare with the prediction time.

Figure 6 presents the prediction time of two data shuffles in the Spark WikipediaPageRank experiment on 13G dataset. The red vertical line marks the time when FLOWPROPHET outputs flow prediction. The blue horizontal lines represent the actual transmission of flows in the shuffle. The start point and end point are the start time and end time of the flow. It demonstrates that our prediction is strictly ahead-of-time for all the flows.

We plot the Cumulative Distribution Function (CDF) of lead time in Figure 7. The lead time is defined as $(actual\_time - predict\_time)$. FLOWPROPHET predicts almost 100% flows ahead of time on Spark and Hadoop. For Spark benchmark in Figure 7(a) and Figure 7(b), the lead time range of 90% flows is from $200ms$ to $500ms$. For Hadoop benchmark in Figure 7(c) and Figure 7(d), the lead time range is relatively loose compared to Spark, since the slow start mechanism [19] may force reducers to fetch data before all mappers finish. In addition, as Hadoop spends much more time

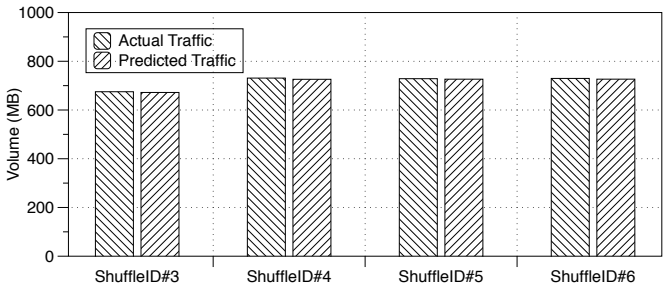


Fig. 8: Actual traffic $vs.$ predicted traffic in WikipediaPageRank-13G (Spark).

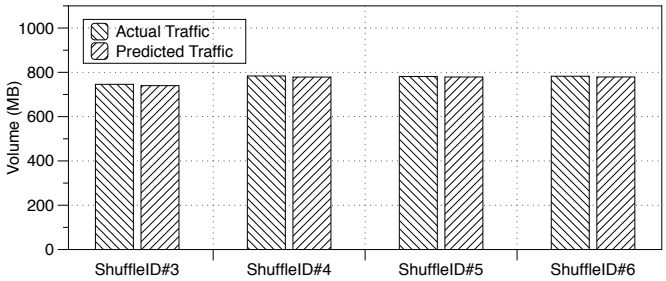


Fig. 9: Actual traffic $vs.$ predicted traffic in WikipediaPageRank-26G (Spark).

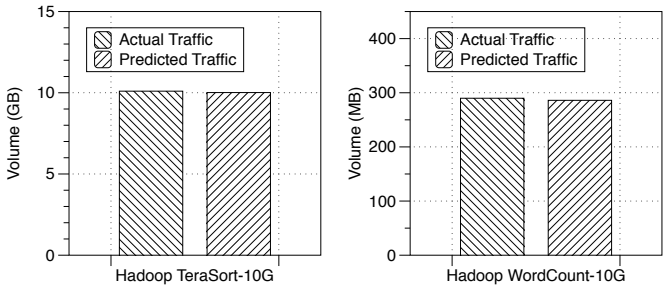


Fig. 10: Actual traffic $vs.$ predicted traffic (Hadoop).

to read and write data from disk while Spark visits data in memory directly, FLOWPROPHET manages to achieve larger lead time on Hadoop. The average lead time is $12.3123s$ and $7.7348s$ respectively for Hadoop TeraSort 10G dataset and WordCount-20G dataset.

Prediction accuracy: We evaluate the prediction accuracy of FLOWPROPHET by comparing the actual and predicted traffic volumes. In the captured traces, we filter out control messages from the DCF master to each worker and calculate the actual traffic volume. Figure 8 and Figure 9 show traffic volumes of four shuffles on the Spark cluster. In Figure 10, the predicted and the actual traffic volumes are also very close on the Hadoop cluster. We conclude that FLOWPROPHET achieves high (almost 100%) accuracy in source, destination and flow size predictions for both Spark and Hadoop.

Note that there might be very slight difference between the actual traffic volume and predicted traffic volume, which can be introduced by network control signaling, packet headers and TCP retransmission. In general, tasks failures or data partition lost will also cause a worker repetitively launches traffic, in which case the actual traffic volume is slightly larger than the predicted one.

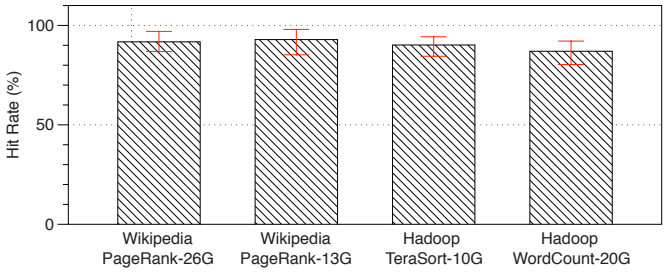


Fig. 11: Establish_time prediction hit rate.

As shown above, FLOWPROPHET can accurately predict flow sizes strictly beforehand. It will be more ideal if we can estimate the time to the future when these flows will start. We applied the adaptive algorithm in Section III-C to infer the `establish_time`. We used $\alpha = 0.15$ for Spark and $\alpha = 0.5$ for Hadoop. The hit rate of this estimation ranges from 85% to 90% as shown in Figure 11, which means that the majority of the flows come up in the time interval we predicted. With such `establish_time` estimation, one can further improve network provisioning.

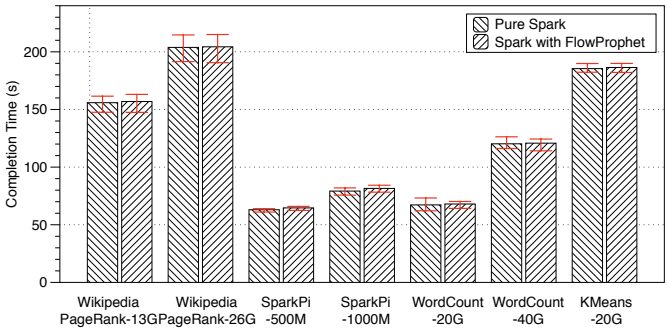


Fig. 12: Job completion time on Spark (overhead test).

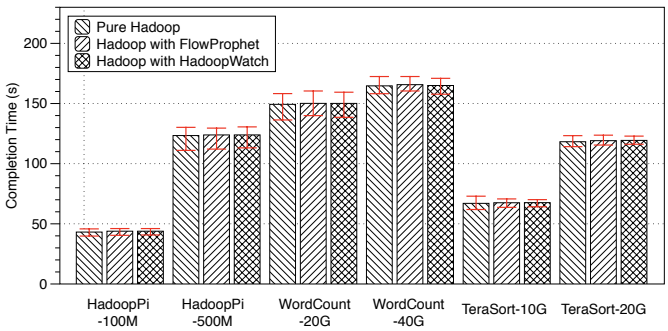


Fig. 13: Job completion time on Hadoop (overhead test).

System overhead: We measure the job completion time with and without FLOWPROPHET to illustrate the overhead introduced by FLOWPROPHET. Under the two conditions, we run each benchmark 10 times with different datasets. GNU Time 1.7 is used to measure the job completion time. Figure 12 and Figure 13 showcase the maximum, mean and minimum job completion times for Spark and Hadoop respectively. We find that the extra time introduced by FLOWPROPHET is negligible, because the implementation of FLOWPROPHET is based on an event-driven and lightweight framework (*i.e.* Actor Model). Furthermore, FLOWPROPHET usually calculates flow

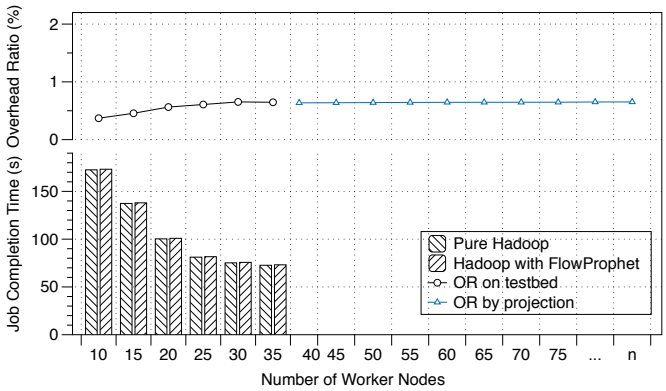


Fig. 14: FLOWPROPHET scalability evalution on TeraSort-10G (Hadoop).

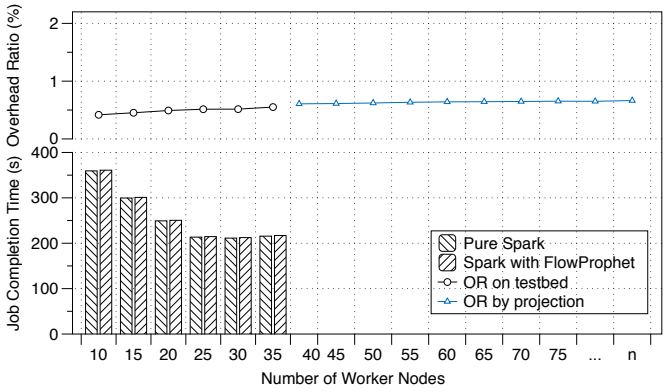


Fig. 15: FLOWPROPHET scalability evaluation on WikipediaPageRank-26G (Spark).

information after a stage is finished and before all tasks are started, at which time there is no task running in the cluster. Thus, FLOWPROPHET mostly exploits the idle computation resource of the cluster and dose not directly compete with computing jobs for resources.

In Figure 13, we included HapdoopWatch [18] as well, which also introduces little overhead as FLOWPROPHET. However, HapdoopWatch is a customized system to Hapdoop, while FLOWPROPHET is general to all frameworks.

Because of the complexity of distributed computing environment, single-server performance metrics like CPU consumption and memory footprint are volatile and inconsistent over the entire cluster. Therefore, we instead use job completion time as a collective metric over the entire cluster to reflect the overhead introduced by FLOWPROPHET.

Scalability: We run WikipediaPageRank-26G on the Spark cluster and TeraSort-10G on the Hadoop cluster when the cluster scales out (from 10 nodes to 35 nodes). We run 5 times for each condition (with or without FLOWPROPHET) and calculate the average job completion time. In Figure 14 and Figure 15, as the number of workers increases, more parallel computing resources are utilized and therefore the job completion time decreases gradually.

The main overhead of FLOWPROPHET is introduced by API calls during shifting stages. The overhead of API calls increases by the number of workers and tasks. Since the

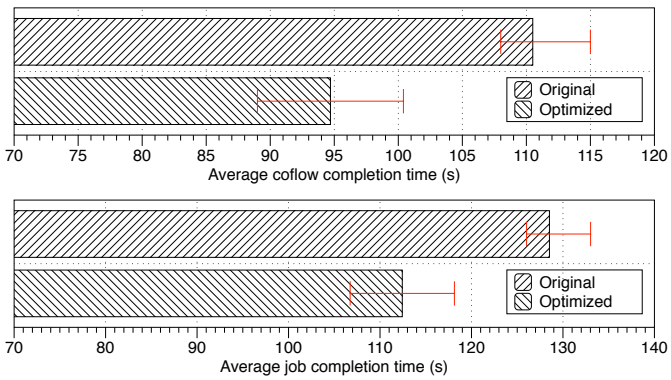


Fig. 16: Benefit with a simple network scheduler.

APIs are called discretely, it is hard to measure directly, so we define a new metric, Overhead Ratio (OR), to evaluate the overhead of FLOWPROPHET. OR is defined as: $(Time_{enabled} - Time_{disabled})/Time_{disabled}$. We then plot ORs as a function of the number of worker nodes to form a polyline. We find that OR increases very slightly when the cluster scales out. This trend showcases the scalability of FLOWPROPHET, since the overhead of FLOWPROPHET is inherently small and is insensitive to the growth of cluster scale. As our testbed is small and only has 37 nodes, in Figure 14 and Figure 15, we intentionally project the curves to assess the overhead at larger scale.

Benefits of FLOWPROPHET to applications: To demonstrate the utility of the predictions made by FLOWPROPHET, we export the predicted flow information to a simple network scheduler we implemented for Hadoop cluster running a TeraSort-25G benchmark.

Specifically, in Hadoop, a reducer will not begin the reduce task until all the dependent data on remote mappers has been fetched. Thus, the flows fetching data for one task on a reducer are barrier-synchronized. As defined in [23], they form a *coflow*. By parsing the coflow information from FLOWPROPHET, the scheduler obtains the flow dependency, and with this information, the scheduler can help a reducer shorten waiting time and start the task earlier. Basically, the scheduler can infer which dependent flow(s) of a coflow will finish late from the paths and the volumes of incoming traffic, and assign higher priority to these flows. With higher priority, the network fabric will allocate more network resource (*e.g.* bandwidth, priority queue, *etc.*) to these flows, so that they can finish and the reducers can begin their computations earlier. In this way, the coflow completion time and the job completion time will be improved.

Even with such a simple scheduler, we have seen remarkable improvement on both coflow completion time and job completion time. As shown in Figure 16, with the information predicted by FLOWPROPHET, the coflow completion time reduces by 14.28% and the job completion reduces by 12.52% on the evaluated Hadoop TeraSort-25G benchmark.

We note that the simple scheduler we implemented here is far from optimal to fully exploit the benefits brought by FLOWPROPHET. We also note that the latest work such as Varys [7] and Baraat [8] require coflow (or task-aware) information as input and FLOWPROPHET can feed accurate input to them. Our next step is to incorporate FLOWPROPHET to more advanced schedulers to benefit them.

V. RELATED WORK

Recently, data-parallel computing is popularized by various frameworks (*e.g.*, Hadoop [19], Dryad [2], Spark [3], and CIEL [21] *etc.*), which provide simple and elegant interfaces for programmers to process large dataset on commodity cluster. Heavy load is thrown upon network infrastructures as a consequence, and networking researchers have been exploring methods to schedule and optimize network resources, so as to accelerate job completion and enhance performance of DCFs. D2TCP [32], pFabric [4], MCP [33], Baraat [8] and Varys [7] leverage flow-based mechanisms and attempt to minimize average completion time of flows or groups of flows by exploiting flow size and deadline information provided by the applications. Helios [9], c-Through [10] and OSA [11] try to estimate aggregate application demands to enable dynamic network resource allocation in terms of architectural bandwidth provisioning. In addition, Hedera [13], MicroTE [14] and D^3 [15] make an effort to address the problem by traffic engineering. Note that above approaches rely on obtaining the traffic and flow information at application-level ahead of time.

To extract traffic information, traffic engineering solutions [34, 35] analyze past statistics (*e.g.*, end-host logs or link traces), and estimate the traffic demand for the next period. There are also proposals trying to monitor socket buffers [10, 16] or counters in switches [9, 13] to gather traffic measurement. However, these approaches are reactive, and the accuracy of such solutions is an issue. FLOWPROPHET proactively collects and utilizes application-level information, and can provide accurate flow information ahead-of-time.

Various tracing and profiling toolkits have been proposed towards extracting traffic information of Hadoop. X-Trace [36] collects Hadoop cross-layer event traces for performance diagnosis. HadoopWatch [18] exploits run-time file system monitoring and parses Hadoop logs and meta-data to forecast Hadoop traffic precisely. But HadoopWatch cannot cover recent DCFs, because monitoring disk operations in file system has become obsolete when more and more DCFs take advantage of memory to cache data. FLOWPROPHET observes and exploits a common theme in popular DCFs: DAG expression of application logic is widely adopted by DCFs. With time and data dependency from DAG, FLOWPROPHET is a general framework to predict flow information for all DCFs.

VI. CONCLUSION

In this paper, we present FLOWPROPHET, a generic and accurate method to predict flow information for large scale DCFs. For this purpose, we summarize the common execution patterns of popular computing frameworks, and extract DAG to obtain time and data dependencies from applications using these frameworks. With this guidance, we design FLOWPROPHET, a flow information prediction framework for DCFs. We make sure that the application programming interfaces (APIs) of FLOWPROPHET is general, so that existing and future computing frameworks can readily deploy FLOWPROPHET to generate accurate flow predictions. We implemented FLOWPROPHET on both Hadoop and Spark,

and achieved almost 100% prediction accuracy in source, destination and flow size, with time advance and minimal cost. We also show that simple network optimizations with ahead-of-time flow predictions can provide substantial improvement in application performance. The job completion time of a Hadoop TeraSort-25G benchmark is reduced by 12.52% on our 37-server cluster with a simple scheduler cooperating with FLOWPROPHET.

VII. ACKNOWLEDGMENTS

This work was supported by HKRGC-ECS 26200014, National Basic Research Program of China (973) under 2011CB302601, National R&D Infrastructure and Facility Development Program (No. 2013FY111900), NRF Singapore CREATE Program E2S2, Huawei Noah's Ark Lab and Shanghai Key Laboratory of Scalable Computing and Systems.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 435–446.

[5] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[6] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 491–502.

[7] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the ACM SIGCOMM 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. ACM SIGCOMM'14, 2014.

[8] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 431–442.

[9] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 339–350, 2011.

[10] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan, "c-through: Part-time optics in data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 327–338.

[11] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "Osa: An optical switching architecture for data center networks with unprecedented flexibility," *Networking, IEEE/ACM Transactions on*, vol. 22, no. 2, pp. 498–511, April 2014.

[12] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating microsecond circuit switching into the data center," in *Proceedings of the ACM SIGCOMM 2013 Conference*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 447–458.

[13] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[15] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 50–61.

[16] G. Ingersoll, "Introducing apache mahout," *Scalable, commercial-friendly machine learning for building intelligent applications. IBM*, 2009.

[17] H. H. Bazzaz, M. Tewari, G. Wang, G. Porter, T. S. E. Ng, D. G. Andersen, M. Kaminsky, M. A. Kozuch, and A. Vahdat, "Switching the optical divide: Fundamental challenges for hybrid electrical/optical datacenter networks," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011.

[18] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing," in *INFOCOM, 2014 Proceedings IEEE*, April 2014.

[19] A. Hadoop, "Hadoop," http://hadoop.apache.org, 2009.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[21] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing." in *NSDI*, vol. 11, 2011, pp. 9–9.

[22] D. G. Murray and S. Hand, "Scripting the cloud with skywriting," *Proceedings of HotCloud*, no. 3, 2010.

[23] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012, pp. 31–36.

[24] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[25] Akka, "Akka framework," http://akka.io, 2014.

[26] J. S. Hunter, "The exponentially weighted moving average." *J. QUALITY TECHNOL.*, vol. 18, no. 4, pp. 203–210, 1986.

[27] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[28] D. L. Mills, "Network time protocol (ntp)," *Network*, 1985.

[29] M. Technologies, "Freebase wikipedia extraction (wex)," http://download.freebase.com/wex/, 2010.

[30] Google, "Freebase data dumps," https://developers.google.com/freebase/data, 2014.

[31] P. Skomoroch, "Wikipedia traffic statistics dataset," http://aws.amazon.com/datasets/2596, 2009.

[32] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 115–126.

[33] L. Chen, S. Hu, K. Chen, H. Wu, and D. H. K. Tsang, "Towards minimal-delay deadline-driven data center tcp," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. New York, NY, USA: ACM, 2013, pp. 21:1–21:7.

[34] A. Feldmann, N. Kammenhuber, O. Maennel, B. Maggs, R. De Prisco, and R. Sundaram, "A methodology for estimating interdomain web traffic demand," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 322–335.

[35] M. Roughan, M. Thorup, and Y. Zhang, "Traffic engineering with estimated traffic matrices," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 248–258.

[36] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 2007, pp. 20–20.