Towards Comprehensive Traffic Forecasting in Cloud Computing: Design and Application

Yang Peng, Kai Chen, Guohui Wang, Wei Bai, Yangming Zhao, Hao Wang, Yanhui Geng, Zhiqiang Ma, and Lin Gu

***Abstract*—In this paper, we present our effort towards comprehensive traffic forecasting for big data applications using external, light-weighted file system monitoring. Our idea is motivated by the key observations that rich traffic demand information already exists in the log and meta-data files of many big data applications, and that such information can be readily extracted through run-time file system monitoring. As the first step, we use Hadoop as a concrete example to explore our methodology and develop a system called HadoopWatch to predict traffic demands of Hadoop applications. We further implement HadoopWatch in a small-scale testbed with 10 physical servers and 30 virtual machines. Our experiments over a series of MapReduce applications demonstrate that HadoopWatch can forecast the traffic demand with almost 100% accuracy and time advance. Furthermore, it makes no modification on the Hadoop framework, and introduces little overhead to the application performance. Finally, to showcase the utility of accurate traffic prediction made by HadoopWatch, we design and implement a simple HadoopWatch-enabled network optimization module into the HadoopWatch controller, and with realistic Hadoop job benchmarks we find that even a simple algorithm can leverage the forecasting results provided by HadoopWatch to significantly improve the Hadoop job completion time by up to 14.72%.**

***Index Terms*—Cloud computing, data center networks, traffic prediction, Hadoop.**

I. Introduction

THE EXPLOSION of big data and cloud applications has imposed significant challenges on the design of network infrastructure in data center environments. Researchers have proposed various new network architectures [1], [23], [39] and traffic engineering mechanisms [2], [6], [43] to handle the rapid growth of bandwidth requirement in data center networks. Many of these proposals leverage the knowledge of application traffic demands to customize network design. For example, Hedera [2], MicroTE [6], and D3 [43] perform flow-level traffic engineering and rate control based on predicted traffic demands. Helios [17], c-Through [39], and OSA [7] rely on accurate traffic demand estimation to perform dynamic optical circuit provisioning. More recently, researchers have also looked into the tight integration of applications and network to configure the network topology and routing based on application run-time traffic demands [19], [40], [42]. All these systems require comprehensive understanding of application traffic in data center networks—the ability to forecast traffic demand before packets enter the network.

However, it is difficult to predict application traffic demand accurately. All the existing solutions focus on using heuristic algorithms based on the measurement of network-level parameters. For example, Hedera [2] and Helios [17] estimate traffic demands using flow counter measurement on switches; c-Through [39] and Mahout [12] use socket buffer occupancy at end hosts to estimate traffic demands for different destinations. However, these schemes have important drawbacks. First, most of them cannot predict the traffic demand before the traffic enters the network. Second, parameters observed on network paths cannot accurately reflect the truth of application demands due to the noise of background flows and congestion control at end hosts. Third, they fail to capture the fine-grained traffic dependencies and priorities information imposed by applications. As a result, these network-layer solutions are shown to perform poorly in predicting the real application demands [4], which could further lead to undesirable performance in network provisioning and traffic engineering mechanisms using these demands.

In this paper, we explore an alternative solution to provide comprehensive traffic forecasting at application layer. With application-specific information, application-layer traffic forecasting is expected to achieve more accurate traffic demand estimation. However, there are several design issues that make this approach challenging and interesting for exploration.

- *Ahead-of-time*: The scheme must be able to predict traffic demand before the data is sent out to the network, so that it can be most useful for network configuration and traffic engineering.
- *Transparency*: Many big data and cloud applications in data centers are complex distributed applications. Traffic forecasting should be transparent to these applications and do not modify any application codes.
- *Light-weighted*: Traffic forecasting should be light-weighted. Many data center applications are large-scale

Manuscript received November 13, 2014; revised April 12, 2015; accepted July 15, 2015; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Bremler-Barr. Date of publication August 18, 2015; date of current version August 16, 2016. This work was supported by HKRGC-ECS 26200014 and Huawei Noah's Ark Lab. Part of this work was presented at IEEE INFOCOM 2014.

Y. Peng, K. Chen, W. Bai, Z. Ma, and L. Gu are with Hong Kong University of Science and Technology, Hong Kong (e-mail: ypengab@cse.ust.hk; kaichen@cse.ust.hk; wbaiab@cse.ust.hk; zma@cse.ust.hk; lingu@cse.ust.hk).

G. Wang is with Facebook, New York, NY 10003 USA (e-mail: ricewgh@gmail.com).

Y. Zhao is with University of Electronic Science and Technology of China, Chengdu 610051, China (e-mail: zhaoyangming.uestc@gmail.com).

H. Wang is with Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: wh.sjtu@gmail.com).

Y. Geng is with Huawei Noah's Ark Lab, Hong Kong (e-mail: geng.yanhui@huawei.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2015.2458892

systems with high performance requirements. The traffic forecasting system should not introduce much overhead to degrade the application performance, and it should scale gracefully with the number of computing nodes and concurrent jobs.
- *Fine-grained*: An application-layer traffic forecaster is expected to provide more fine-grained traffic demand information than just traffic volume. For example, many distributed applications have computation barriers that cause co-dependencies among multiple flows, and co-dependent flows may need to be treated collectively in transfer to be more meaningful. Such structural information will be much useful to enable better network control and optimization.

We observe that rich traffic demand information already exists in the log and meta-data files of many big data applications, and such information can be extracted efficiently through run-time file system monitoring. We use Hadoop as a concrete example to explore the design of application-layer traffic forecasting using file system monitoring. We develop a system called HadoopWatch, which is a passive monitoring agent attached to the Hadoop framework that monitors the meta-data and logs of Hadoop jobs to forecast application traffic before the data is sent out to network. It does not require *any* modification to the Hadoop framework.

We have implemented HadoopWatch and deployed it on a real small-scale testbed with 10 physical machines and 30 virtual machines. Our experiments over a series of MapReduce applications demonstrate that HadoopWatch can forecast the application-layer traffic demand with almost 100% accuracy and strict time advance, while introducing little overhead to the application performance.

To further demonstrate the utility of accurate traffic forecasting made by HadoopWatch, we introduce a case on how the co-dependent flows predicted by HadoopWatch can be utilized to improve the data transfer efficiency of Hadoop jobs. Specifically, we design and implement a simple network optimization software module, called HadoopOptimizer, into the HadoopWatch controller, and through running Hadoop TeraSort-25G and TeraSort-50G we show that even a simple optimization algorithm can leverage the forecasting results provided by HadoopWatch to significantly improve the Hadoop job completion time by up to 14.72%. We envision that HadoopWatch will have more applications than what we discussed in this showcase example.

In a nutshell, our work is a first step towards comprehensive traffic forecasting at the application layer. Many research problems remain to be explored in future work. However, we believe our work shows early promises of performing comprehensive and light-weighted traffic forecasting through file system monitoring, which could be a useful building block for tight network and application integration in cloud and data center environments.

Roadmap: The rest of the paper is organized as follows. Section II introduces the background and key observations that enables the forecasting architecture. Section III presents the design of HadoopWatch. Section IV discusses the implementation and evaluation results of HadoopWatch. Section V shows a concrete example to apply the forecasting results by HadoopWatch to optimize Hadoop jobs. Section VI reviews the related works. We discuss the future work and conclude the paper in Section VII.

II. Traffic Forecasting via File System Monitoring

Due to a variety of data exchange requirements, there is a large amount of network traffic in the life cycle of a Hadoop job. We first briefly introduce the Hadoop architecture and its dataflow in different stages. Then, to motivate our idea of forecasting application traffic through external, light-weight file system monitoring, we introduce the observations and opportunities in file systems that provide rich-semantics enabling traffic forecasting.

A. Hadoop Background

Hadoop consists of *Hadoop MapReduce* and *Hadoop Distributed File System* (HDFS). Hadoop MapReduce is an implementation of MapReduce designed for large clusters, while HDFS is a distributed file system designed for batch-oriented workloads. Each job in MapReduce has two phases. First, users specify a *map* function that processes the input data to generate a list of intermediate key-value pairs. Second, a user-defined *reduce* function is called to merge all intermediate values associated with the same intermediate key [15]. HDFS is used to store both the input to the *map* and the output of the *reduce*, but the intermediate results, such as the output of the *map*, are stored in each node's local file system.

A Hadoop implementation contains a single *master* node and many *worker* nodes. The master node, called the JobTracker, handles job requests from user clients, divide these jobs into multiple tasks, and assign each task to a worker node for execution. Each worker node maintains a TaskTracker process that executes the tasks assigned to itself. Typically, a TaskTracker has a fixed number of slots for accepting tasks.

B. Hadoop Dataflows

Many Hadoop jobs are communication intensive, involving a large amount of data transfer during their execution. We broadly characterize Hadoop dataflows into three types.

- *Import*: *The map reads data from HDFS*. To increase overall data loading throughput of a job, multiple concurrent map tasks may be scheduled to fetch data in parallel. Specifically, for each map task, the JobTracker will specify its input split. As a map task runs, it will fetch the corresponding data (in the form of key-value pairs) from HDFS and iteratively perform the user defined map function over it. Optimized with various scheduling techniques [41], [44], most map tasks achieve data locality, while there also exist some nonlocal map tasks reading their input splits from remote DataNodes that involve network transfer.
- *Shuffle*: *Intermediate results are shuffled from the map to the reduce*. Before the reduce function is called, a reduce task requires intermediate results from multiple map tasks as its input. When a map task finishes, it will write its output to local disk, commit to the JobTracker, and reclaim the resources. Meanwhile, the reduce task will periodically query the JobTracker for any latest map completion events.

Being aware of these finished map tasks and their output locations, a reduce task initiates a few threads and randomly requests intermediate data from the TaskTracker daemons on these nodes.

- *Export*: *The reduce writes output to HDFS*. Large output data is partitioned into multiple fix-sized blocks,[1] while each of them is replicated to three DataNodes in a pipeline [35], [38]. A Hadoop job completes after the outputs of all reduce tasks are successfully stored.

C. Observations and Opportunities

Rich Traffic Information in File Systems: Most cloud big data applications, such as Hadoop, have various file system activities during the job execution. We observe that these file system activities usually contain rich information about the job run-time status and its upcoming network traffic.

First, logging is a common facility in big data applications for troubleshooting and debugging purposes. However, these log file entries can also be used to identify network traffic. For example, the source and destination of a shuffle flow in MapReduce can be determined after locating a pair of map and reduce tasks, while such task scheduling results are written in the JobTracker's log.

Moreover, intermediate computing results and meta-data are periodically spilled to disk due to limited capacity of memory allocation. The size of all the output partitions of a map task is saved in a temporary index file, from which we can compute the payload volume of all the shuffle flows from this map.

In addition, the namespace of a distributed file system may also be accessible by parsing its meta-data files on disk. For example, when the HDFS starts up, its whole namespace is updated in a file named *FsImage*. Another file called *EditLog* is used to record all namespace changes thereafter. Although Hadoop only maintains a snapshot of the namespace in memory for frequent remote queries, it can be externally reconstructed with those two files on disks. The reconstructed namespace can be used to recover the network traffic generated by HDFS read and write operations.

Light-Weighted File System Monitoring: We further observe that these file system activities can be monitored using light-weighted file monitoring facilities in modern operating systems. In recent Linux system, there is a file change notification subsystem called *inotify* [29]. The key of inotify is to perform file surveillance in the form of watch, with a pathname and an event mask specifying the monitored file and the types of file change events. A file will be monitored after it is tagged to watch, whereas all the files in a directory will be monitored if the directory is watched. Table I shows all the valid events in inotify. With inotify, we can dynamically retrieve the footprint of an application with its file system activities. Furthermore, inotify can monitor file change events efficiently in an asynchronous manner rather than polling. As a result, the application's file system operations can be executed in a nonblocking mode while inotify is running. This effectively minimizes the impact of monitoring on the Hadoop applications.

In summary, the above observations enable us explore the idea of forecasting application traffic through external, light-weight file system monitoring.

[1]The last block in a file may be smaller.

TABLE I
VALID EVENTS IN *INOTIFY*

Event	Description
IN_CREATE	File was created.
IN_ACCESS	File was read from.
IN_MODIFY	File was written to.
IN_ATTRIB	File's attribute was changed.
IN_CLOSE_WRITE	File was closed (opened for writing).
IN_CLOSE_NOWRITE	File was closed (opened not for writing).
IN_OPEN	File was opened.
IN_MOVED_FROM	File was moved away from watch.
IN_MOVED_TO	File was moved to watch.
IN_DELETE	File was deleted.
IN_DELETE_SELF	The watch itself was deleted.

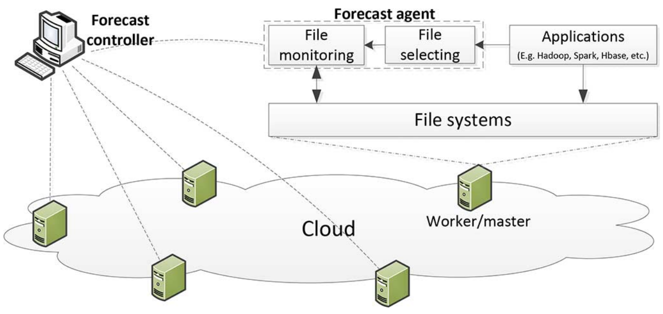


Fig. 1. Architecture of traffic forecasting.

III. HADOOPWATCH

As the first step, we use Hadoop as a concrete example to explore the detailed design of our traffic forecasting approach. We choose Hadoop because it is one of the most popular computing frameworks in today's data centers, and its execution structure represents several typical traffic patterns in data center applications. We develop a system called HadoopWatch and show how we can forecast traffic demand of Hadoop jobs by monitoring the file system activities. In the following, we first discuss the HadoopWatch architecture, then introduce the monitor file selection, and finally show how to perform traffic forecasting.

A. Architecture

Based on the above observations, we propose to forecast the traffic demand of big data and cloud applications by monitoring their file system activities. Fig. 1 shows the general architecture of this traffic forecasting framework. It is a passive monitoring engine attached to a big data cluster, which does not require any modification to the applications. It continuously monitors the file system activities of the applications and predict their traffic demands at run-time. The traffic forecasting framework has two components, *Forecast Agents* and *Forecast Controller*. We implant a Forecast Agent in each worker/master node to collect traffic information and report to the centralized Forecast Controller.

- *Forecast Agent*: Forecast Agent is a daemon implanted on each node that collecting run-time file system activities. To keep it light-weighted, we selectively monitor the specific files and activities with *inotify*. To monitor file access details, some system call parameters are also collected and encoded in the redundant space of *inotify* interface. To get the content of other bulk data and metadata on disk, the agent will read them directly. Information collected by Forecast Agents will be reported to the Forecast Controller.

TABLE II
FILES FOR TRAFFIC FORECASTING IN HADOOPWATCH

Application	Flow type	Required information	Monitoring files & events		Response action
Hadoop	Shuffle	location of a complete map task, size of each map output partition	file.out.index	IN_CLOSE_WRITE	parse the sequence file format, collect partLengths of all partitions
		location of a new reduce task	JobTracker's log	IN_MODIFY	scan next entry for the TaskTracker that launches a reduce task
		when a flow starts	file.out	IN_SEEK	determine which reduce task are fetching the intermediate data
		when a flow terminates	TaskTracker's log	IN_MODIFY	scan next entry indicating success of the shuffle flow
	Export	block allocation results	NameNode's log	IN_MODIFY	scan next entry for a new block allocation result
		where a pipeline establishes and when a flow starts	DataNode's logs	IN_MODIFY	scan next entry for remote HDFS writing request
		when a flow terminates	DataNode's log	IN_MODIFY	scan next entry indicating success of the HDFS writing flow
	Import	locality of a map task	JobTracker's log	IN_MODIFY	scan next entry for locality of a map task
		input split (blocks) and flow size	split.dta	IN_CLOSE_WRITE	parse split(HDFS path/start/offset), query for corresponding blocks
		when a flow starts	block file	IN_SEEK	match the probable map task fetching the data block
		when a flow terminates	DataNode's log	IN_MODIFY	scan next entry indicating success of the HDFS reading flow

- *Forecast Controller*: The main task of the Forecast Controller is to collect reports from all the Forecast Agents and generate comprehensive traffic demand results. The reported traffic demand information may include the basic information such as source, destination, and data volume, and more fine-grained information such as flow dependencies and priorities.

This centralized model is inspired by the success of several large-scale infrastructure deployments such as GFS [22] and MapReduce [15], which employ a central master to manage tasks at the scale of tens of thousands of worker nodes. In addition, we note that, just as Hadoop, HadoopWatch can be inherently fault-tolerant. For example, when worker node failure happens, the log/meta-data information associated with the failed node is no longer useful because this node is excluded from the computing framework, thus the Forecast Controller simply discards any partial information reported from the failed node, and instead exploits the information from the backup node where the data replica resides for traffic forecasting (by default, there are three replicas in HDFS).

B. Monitor File Selection

In Table II, we summarize all the files that should be monitored to forecast the three types of dataflows in Hadoop. The selection of these files is based on the detailed analysis of the Hadoop framework and the job execution semantics as follows.

Import: The map needs to read input from HDFS. From the JobTracker's log, we can easily identify whether a map task is data-local, rack-local, or nonlocal. For rack-local and nonlocal maps, they introduce data import from remote nodes through networks. However, the JobTracker's log does not tell from which DataNodes a map will read its input data split. To forecast this information, we pick out the *split.dta* file. This file contains a map task's input split information, i.e., the input file location, offset and length in HDFS.

Shuffle: To forecast the shuffle traffic from a map to a reduce, we observe that the output of a map task is stored in a file named *file.out* as shown in Fig. 2. To support direct access to all data partitions in it, there is an index file named *file.out.index* that maintains all their offsets and sizes. Since each partition in *file.out* corresponds to the payload of a shuffle flow sent to a

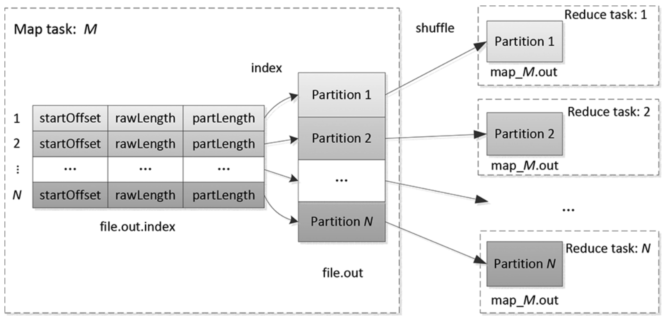


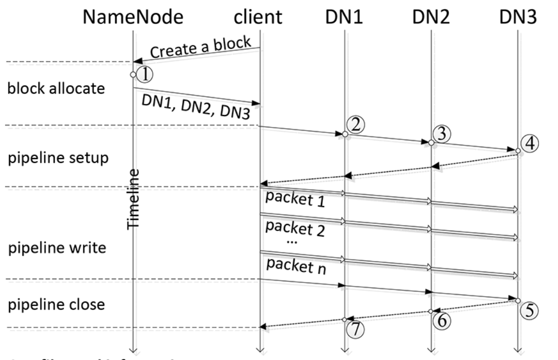
Fig. 2. Data flows in shuffle phase.

reduce, we can forecast its volume based on the size of the corresponding partition. On the other hand, to infer the source and destination of a shuffle flow, we use the scheduling result in the JobTracker's log which includes the information where a map task or a reduce task is launched.

Export: The reduce may export its output to HDFS, which entails HDFS writing. In HDFS, each data block is replicated with multiple replicas (three replicas by default). The HDFS writing is implemented using a pipeline set up among replica nodes to maintain data consistency. Fig. 3 shows the diagram of a pipelined HDFS writing. First, when a HDFS client requests to write a block, the NameNode allocates three sequential DataNodes and writes this allocation in its log (①). Then, before the block is transferred between each pair of DataNodes, the receiver side writes a log indicating the upcoming HDFS writing (②–④). Finally, when the write completes, its volume is saved in the DataNode's log (⑤–⑦). Through these log files, we can forecast the communication matrix according to the pipeline and predict the volume of upcoming flows based on the HDFS block size (e.g., 64 MB).

C. Traffic Forecasting

As above, HadoopWatch can provide accurate traffic forecasting for every data flow, including its source, destination, and volume. As shown in Table III, the source and destination are two identities, which can uniquely identify a flow in a Hadoop job. With these per-flow metrics, we can not only compose the



Log files and information:
1. { blockID, DN1, DN2, DN3, HDFS filename and offset };
2. { blockID, client, DN1}; **3**. { blockID, DN1, DN2 }; **4**. { blockID, DN2, DN3};
5. { blockID, size }; **6**. { blockID, size }; **7**. { blockID, size }

Fig. 3. Pipelined HDFS writing.

TABLE III
PER-FLOW METRICS

Dataflow	Source	Destination	Volume
Import	mapID, input blockID	mapID	blockID's size
Shuffle	mapID	reduceID	partition size
Export	reduceID	reduceID, output blockID	blockID's size

overall traffic matrix in the cluster, but also identify fine-grained flow relations such as dependency and priority.

The source and destination in Table III are logical identities. To determine the physical locations, we just map the logical source and destination to the physical nodes. The mapping requires knowledge of task scheduling results and block locations. Most of these information is plainly accessible by monitoring files listed in Table II. However, the source locations of map input blocks and the volumes of reduce export flows cannot be explicitly captured. Therefore, HadoopWatch develops the following two heuristics to infer these information:

- *Source location of a map input block*: When the JobTracker schedules a rack-local or nonlocal map task, it will independently choose the closest block replica to fetch. For these map tasks, we can translate their input splits to blockIDs through querying the NameNode. Since there are rarely two tasks processing the same input dataset simultaneously, the node where we captured such block file access event is likely to be the source of the data import flow.
- *Volume of a data export flow*: When the reduce task output is written into HDFS, the data will be divided and stored into multiple blocks (typically 64 MB) and replicated to several DataNodes. Because the block size is fixed, in most cases, the size of an export flow is fixed, i.e., 64 MB. However, the last block size is uncertain and needs estimation. Based on the definition of reduce function [15], it is likely that the output size of a user-defined reduce function is roughly proportional to its input size. Then, we maintain a selectivity, $s(r)$, for reduce tasks in a job, which is defined as the output size to its input size. For an upcoming reduce operation, we estimate its selectivity based on the selectivities of the reduce operations in recent past using exponentially weighted moving average (EWMA)

$$s_{\text{new}}\,(\mathrm{r}) = \alpha s_{\text{measured}}\,(\mathrm{r}) + (1-\alpha)s_{\text{old}}\,(\mathrm{r})$$

Algorithm 1 Traffic matrix calculation

```
Input: every $f$: (src, dest, vol)
1: for each $f$ do
2:     ip1 $=$ location($f$'s $src$)
3:     ip2 $=$ location($f$'s $dest$)
4:     $TM$(ip1, ip2) $\leftarrow TM$(ip1, ip2) $+ f$'s $vol$
5: end for
```

where α is the smoothing factor (HadoopWatch uses $\alpha = 0.25$). On the other hand, a reduce task r's input size, $I(r)$, is the sum of shuffle flow volumes from all map tasks. We get the estimated reduce output using

$$O(r) = I(r) \times s(r).$$

For the ith export flow of the reduce task r, its volume $\mathrm{vol}(i)$ is calculated by checking whether the maximum block size is enough to save the remaining bytes

$$\mathrm{vol}(i) = \begin{cases} \mathrm{BLK}_{\max}, & \text{if } O(r) > i \times \mathrm{BLK}_{\max} \\ O(r) - (i-1)\mathrm{BLK}_{\max}, & \text{otherwise.} \end{cases}$$

Here, $\mathrm{BLK}_{\max}$ stands for the maximum size of a HDFS block (e.g., 64 MB).

With the above basic data flow information inferred, we are able to compose the traffic matrix and identify the flow dependency and priority. We believe that this information is useful for many aspects, e.g., traffic engineering, network profiling, or transport protocol design. For example, traffic distribution information is critical for fine-grained flow scheduling in Hedera [2] and traffic engineering in MicroTE [6], while flow dependency and priority information can be incorporated into several recent deadline-aware transport designs like D^{2}TCP [37], D^3 [43], and MCP [8] for more intelligent congestion control.

- *Traffic matrix*: We can easily calculate the traffic matrix in a cluster with every data flow information. As shown in Algorithm 1, to calculate the traffic volume between any two physical nodes ip1 and ip2, we just need to sum up the volumes of individual flows between them.
- *Dependency*: We define two types of dependencies, i.e., causal-dependency and co-dependency. The causal-dependency $f_1 \rightarrow f_2$ means that the initiation of f_2 depends on the completion of f_1. For example, $flowA \rightarrow flowC$ and $flowC \rightarrow flowE$ in Fig. 4. The co-dependency $f_1 \leftrightarrow f_2$ specifies that f_1 and f_2 share a common barrier. The barrier cannot be passed through until the completion of both flows. One such example is the shuffle flows to the same reduce task (e.g., $flowB \leftrightarrow flowC$), and another example is the pipeline flows replicating the same block (e.g., $flowE \leftrightarrow flowF$).

  We can infer the dependency of two flows based on their logical source and destination identities in Table III using Algorithm 2. For example, we know $flowA \rightarrow flowC$, since the destination of $flowA$ and the source of $flowC$ are both Map 4. Also, $flowB \leftrightarrow flowC$, as they share a same destination, Reduce 2.

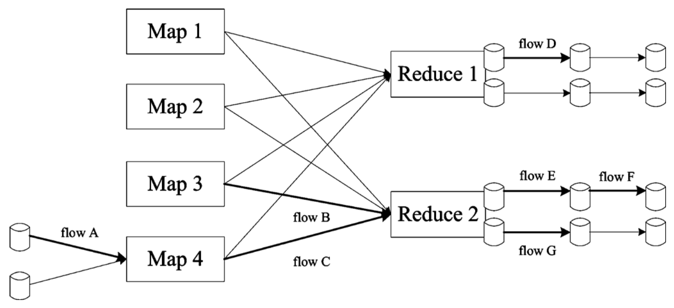


Fig. 4. Flow patterns in Hadoop.

Algorithm 2 Dependency

```
Input: $f_1$ : (src, dest) and $f_2$ : (src, dest)
Output: $1 : f_1 \leftarrow f_2, 2 : f_1 \rightarrow f_2, 3 : f_1 \leftrightarrow f_2$, 0 otherwise.
1: if $f_1$'s dest $==$ $f_2$'ssrc then
2:     return 1
3: else if $f_1$'s src $==$ $f_2$'s dest then
4:     return 2
5: else if $f_1$'s dest $==$ $f_2$'s dest then
6:     return 3
7: end if
8: return 0
```

- *Priority*: Unlike flow dependency that reflects the inherent structure of Hadoop jobs, flow priority is a metric that depends more on the optimization objectives when the forecasted traffic is used in network planning. For different optimization objectives, we can define different policies to assign flow priorities. In a normal use case that we want to boost the execution of a Hadoop job, a flow in an earlier phase should be assigned to a higher priority. Because most Hadoop jobs are composed of multiple tasks (e.g., import ⇒ map ⇒ shuffle ⇒ reduce ⇒ export), a job completes when the slowest task is finished. As shown in Fig. 4, import, shuffle, and reduce flows should be prioritized accordingly (e.g., $flowA > flowB > flowD$) to ensure that the slowest flow finishes as fast as possible. Another example of priority assignment policy is that shorter flows in larger co-dependent groups should be assigned with higher priorities. Using the "shortest job first" strategy, finishing shorter flows in a larger co-dependent group first can quickly decrease the number of blocking flows and speed up the execution of a Hadoop job.

IV. Evaluation

A. Methodology

Testbed setup: We deployed HadoopWatch on 30 virtual machines (VMs) running on 10 physical servers. All the 10 physical servers are HP PowerEdge R320 with a quad-core Intel E5-1410 2.8 GHz CPU, 8 GB memory, and a 1-Gb/s network interface. On each physical server, we set up three Xen VMs (DomU), and each of the VMs is allocated with one dedicated processor core (2 threads) and 1.5 GB memory. We run Hadoop 0.20.2 on all the 30 VMs for the traffic forecasting experiments.

Evaluation Metrics: In our evaluation, we study the accuracy of HadoopWatch in prediction flow volumes, the time advance of these predictions, and the overhead of HadoopWatch. To collect the ground truth of Hadoop traffic volume and timing, we use TCPdump to capture the actual time and volume of data flows. We extract the source and destination of each flow by parsing the application-level requests and responses in Hadoop flows. We use the absolute difference between the predicted volume and actual volume to evaluate the forecasting accuracy of HadoopWatch. We use the lead time metric to evaluate the time advance, which is defined as $(actual\_time - predicted\_time)$ of Hadoop flows.

B. Experiment Results

Accuracy: Fig. 5 shows the traffic volume and forecast accuracy for four representative Hadoop jobs: Terasort, Wordcount, Hive Join, and Hive Aggregation [27]. Overall, it can be seen that the shuffle and export phase introduced most of the traffic for these jobs,[2] and we achieve high accuracy for all types of traffic. The slight difference between the forecast results and the actual ones is mainly caused by the control signals and TCP retransmission packets. Furthermore, there are occasionally a few dead export flows since a slower reduce task will be killed between a normal task and its backup instance.

Time Advance: Figs. 6–8 show the forecasting lead time for data import, shuffle, and export flows, respectively. We use NTP [31] to synchronize the clock on these nodes. The results show that almost 100% traffic flows are successfully forecasted in advance. Most data import and export flows occur soon after the corresponding traffic forecasts ($\leq$ 100 ms), while most shuffle flows are forecasted much earlier in advance (5–20 s) because a reduce task only initiates 5 shuffle flows fetching intermediate data and other shuffle flows are pending. In a small percent ($<$3%) of flows, we do observe some forecast delays that flows are forecasted after they are actually sent out. They are either caused by deviation of clock synchronization or monitoring delay of inotify events.

Overhead: Fig. 9 compares the execution time of Hadoop jobs with and without HadoopWatch to assess the overhead introduced by HadoopWatch. For all the four jobs we tested, their execution time only increases by up to 1%–2% with HadoopWatch running in the cluster.

Dependency: Fig. 10 shows the distribution of flow's co-dependent flow numbers and casual-dependent flow numbers. We take the 40-GB Wordcount job as an example, which consists of 658 map tasks and 20 reduce tasks. Therefore, shuffle flows can be divided into 20 co-dependent groups. Each group contains 658 co-dependent flows initiated by the same reduce task. On the other hand, because of the data locality of map tasks, a large number of them are importing data from local disks. Thus, only a small number of shuffle flows are casual-dependent on import flows. Meanwhile, the output export flows of each reduce task are casual-dependent on its input shuffle flows.

Scalability: Due to the limitations of testbed size, our evaluation results of HadoopWatch are limited to tens of nodes. We use simulation analysis to understand the scalability of HadoopWatch. We first analyze the major determinants of monitoring

[2]Note that the output of Terasort is not replicated in remote DataNodes, so it does not introduce any export flows.

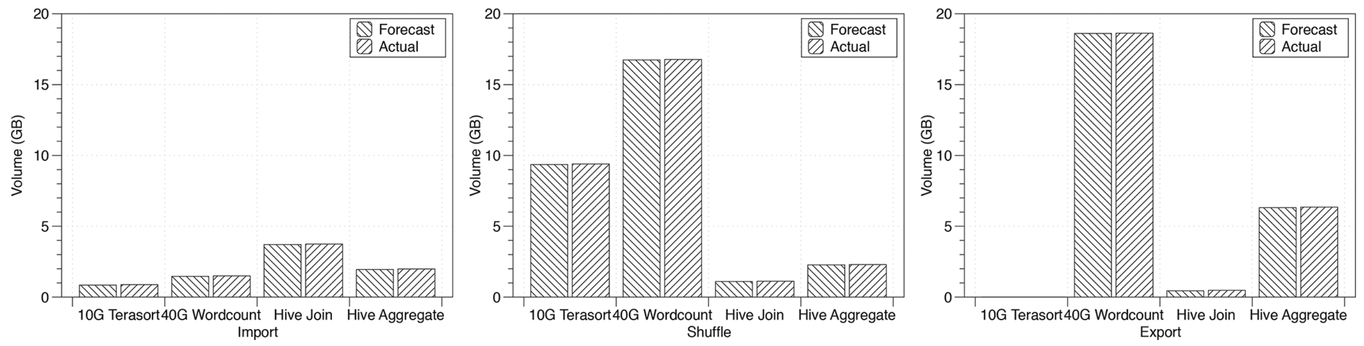


Fig. 5. Accuracy of traffic volume forecasting.

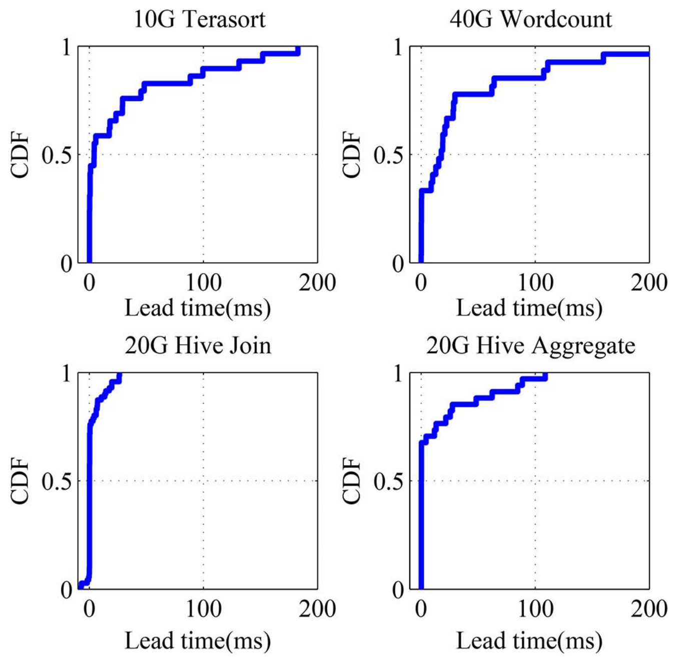


Fig. 6. Time advance in remote import flow forecasting.

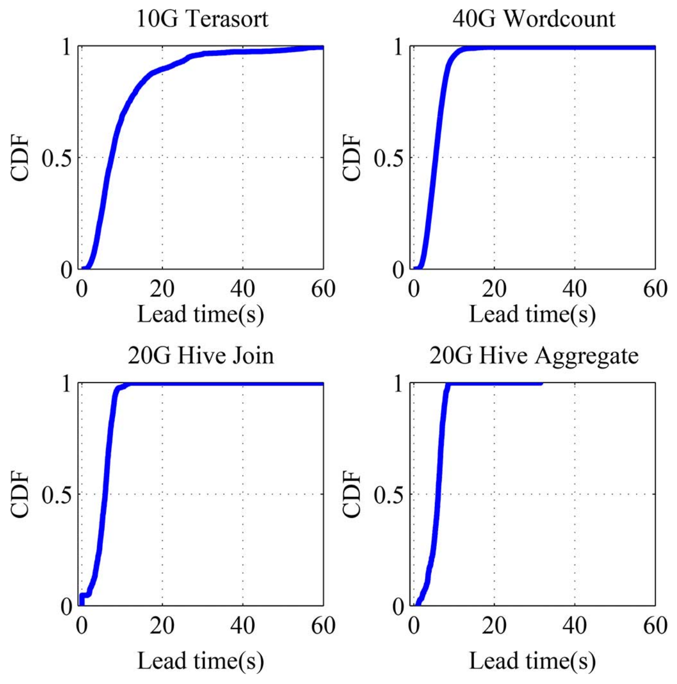


Fig. 7. Time advance in shuffle flow forecasting.

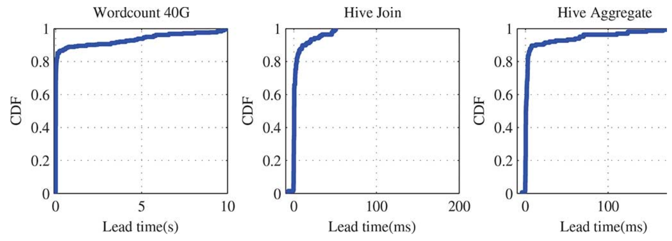


Fig. 8. Time advance in remote export flows forecasting.

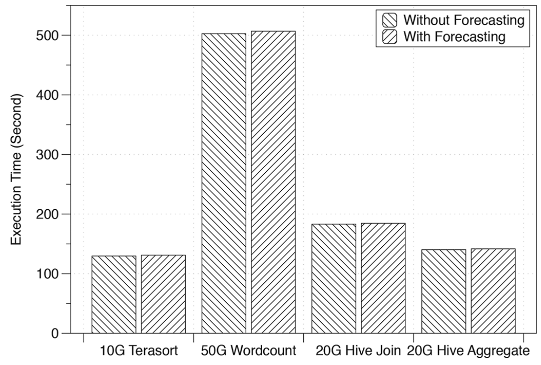


Fig. 9. Execution time (seconds).

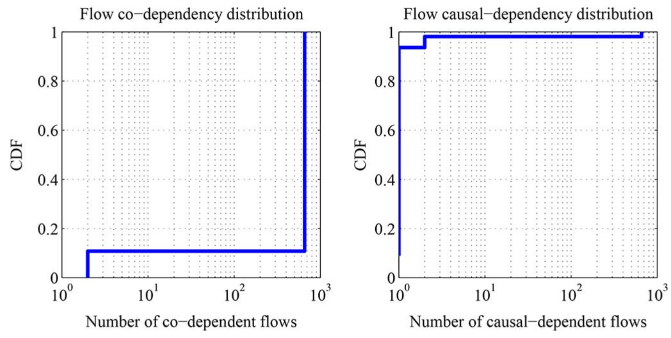


Fig. 10. Distribution of co/casual-dependent flows.

overhead in Forecast Agent and Forecast controller, then estimate the HadoopWatch overhead in large production settings.

On a worker node, our agent iteratively processes the inotify events that are caused by multiple tasks. It just parses the traffic related information and sends to the controller. The memory usage is fixed since no extra data is stored locally. In addition, the CPU usage scale linearly with the number of active tasks (n_{task}) on each node since the execution of each agent is strictly driven by these tasks' file system events. On

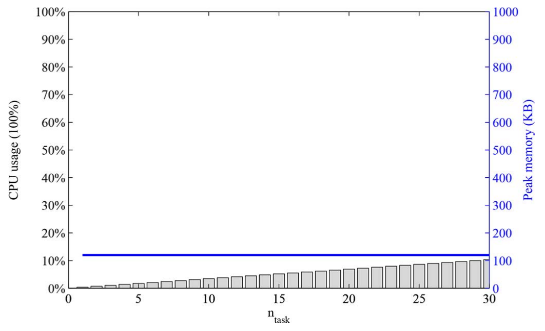


Fig. 11. Agent overhead prediction.

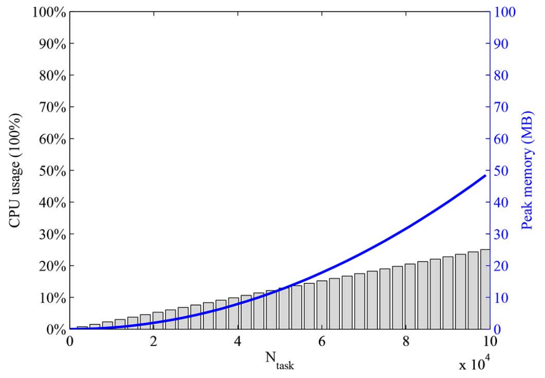


Fig. 12. Central control overhead prediction.

the controller end, the forecasting controller continuously receives event reports from multiple agents and generates traffic forecasts accordingly. As a result, its CPU usage scales linearly with the number of total active tasks (N_{task}). Meanwhile, a lot of memory is required to store the volumes of shuffle flows between these map tasks and reduce tasks. The total memory usage exhibits a quadratic growth as N_{task} increases.

Based on the performance metrics collected on our testbed, we estimate the potential overhead that HadoopWatch introduces on large production clusters. Figs. 11 and 12 show the estimated overhead on clusters with sizes released by Google and Facebook. We observed that the agent only consume 6.89‰ of a CPU core under hours of heavy workload. To support 30 concurrent tasks on each node in the Google cluster [9], [46], HadoopWatch agent may take 10% consumption of a CPU core and fixed memory. Similarly, we estimate the overhead of HadoopWatch central controller. It will only consume a CPU core's 30% resources and around 50 MB memory to support hundreds of thousands of tasks running concurrently. In summary, we believe HadoopWatch is scalable to support traffic forecasting in large MapReduce clusters.

V. Utility of HadoopWatch

Our evaluation in Section IV has shown that HadoopWatch can predict the Hadoop traffic with high accuracy and further identify the co-dependency of flows in Hadoop jobs. The predicted traffic demand can have a variety of benefits from traffic engineering, flow scheduling, to network optimization, etc. In this section, we showcase the utility of HadoopWatch through a concrete example on how co-dependent flows predicted by HadoopWatch can be utilized to improve the data transfer efficiency of Hadoop jobs. To this end, we first explain why and how to optimize co-dependent flows collectively. Then, we propose and implement a simple HadoopWatch-enabled optimization scheme into the HadoopWatch controller of our testbed. Finally, we show that, with accurate prediction from HadoopWatch, the job completion time of Hadoop benchmarks can be significantly improved even with the proposed simple algorithm.

A. How to Optimize Co-Dependent Flows

As introduced in Section III-C, co-dependent flows often share a common barrier, which cannot be passed through until the completion of the last flow among them. Thus, in order to be effective, co-dependent flows must be optimized collectively. It should be noted that our "co-dependent flows" is similar to the recently proposed "coflow" concept [10]. A group of co-dependent flows can be treated as a coflow.

We note that two recent schemes Varys [11] and Baraat [16] optimize coflows using scheduling only. Instead, we use an example in Fig. 13 to illustrate the need of optimizing coflows using both routing and scheduling. In this example, there are two coflows (i.e., two groups of co-dependent flows): Coflow a has flows f_{a1} and f_{a2} with sizes 20 and 50 Mb, respectively, and coflow b has flows f_{b1} and f_{b2} with sizes 30 and 50 Mb, respectively. Assume the link bandwidth is 100 Mb/s. As a reference point, the optimal average coflow completion time (CCT) of this example should be 0.65 s.

The first takeaway is that scheduling alone is not sufficient to optimize average CCT. If routing is fixed, good scheduling can minimize the average CCT by determining the sequence of flows to send out to network. Fig. 13(a) shows a possible routing with ECMP. With a naive scheduling such as fair sharing, both coflows are dominated by path $S \to M_d \to D$, and hence their CCTs are both 1 s. If using the optimal scheduling shown in Fig. 13(d), the CCTs for two coflows become 0.5 and 1 s, respectively; apparently, scheduling does play a critical role. However, the average CCT (in this routing) is only 0.75 s, which still has a 0.1 s gap to the real optimal value 0.65 s. It is clear that routing should also play a critical role: The loads of two paths in Fig. 13(a) are unbalanced, where path $S \to M_d \to D$ has a traffic load twice that of path $S \to M_u \to D$.

The second takeaway is that considering routing and scheduling separately cannot optimize average CCT. As an example in Fig. 13(b), a load-balanced routing results in the following: Both flows of coflow a are routed on $S \to M_u \to D$, while the flows of coflow b on $S \to M_d \to D$; now the network is more balanced. However, the optimal CCTs for coflows a and b in this case are 0.7 and 0.8 s, respectively [see Fig. 13(e)], and the average CCT is 0.75 s, which is still not the optimal. The reason is that flows of the same coflow are routed through the same path without taking the flow co-dependence into account, which leaves little space for scheduling to take effect to reduce the average CCT.

The conclusion is that both routing and scheduling must be jointly considered in order to optimize the average CCT. When performing load-balancing, co-dependent flows in a coflow should be spread disjointly across the network so that scheduling can further take effect. In our example, the minimal average CCT can be achieved by combining the load-balanced coflow-aware routing in Fig. 13(c) and the scheduling in

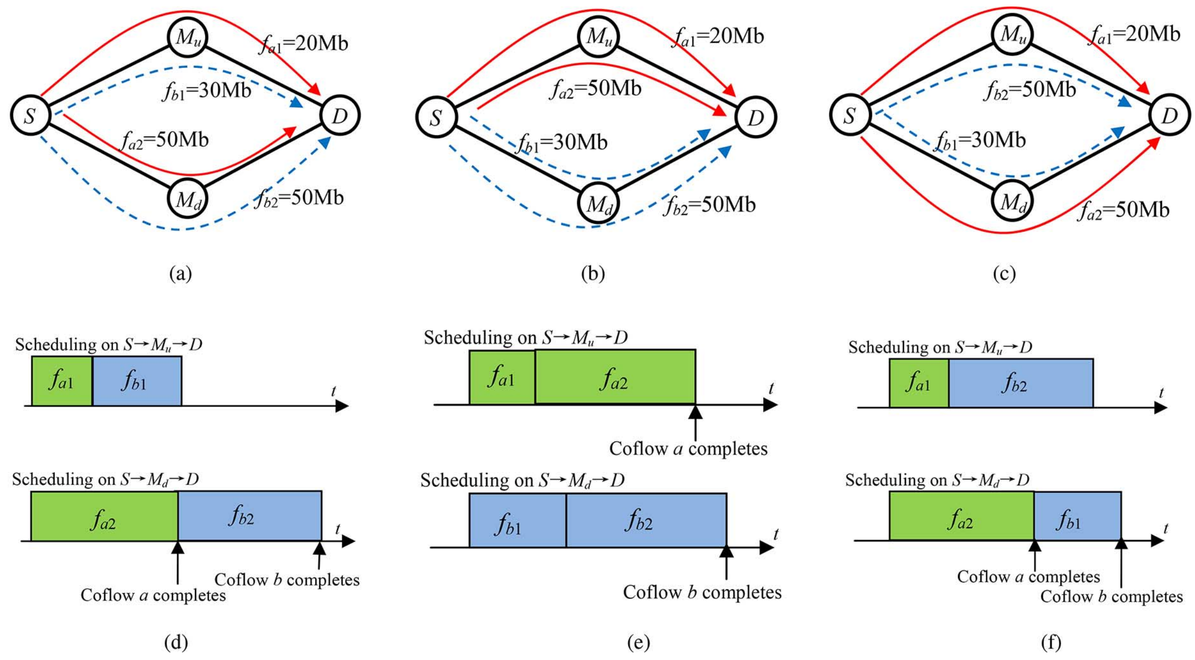


Fig. 13. Motivating example, where (a)–(c) show different routing schemes and (d)–(f) show the optimal scheduling schemes for (a)–(c). (a) One possible unbalanced routing generated by ECMP. (b) Load-balanced routing without taking coflow concept into account. (c) Optimal load-balanced routing considering coflow concept. (d) Optimal scheduling up on (a). (e) Optimal scheduling up on (b). (f) Optimal scheduling up on (c).

Fig. 13(f). In this case, the CCTs of two coflows are 0.5 and 0.8 s, respectively, and the average CCT is minimal.

B. How HadoopWatch Benefits Coflow Optimization

In light of the above motivating example, we should optimize co-dependent flows in a coflow as a whole instead of treating them individually. To this end, information such as flow co-dependence, number of flows in a coflow, flow sizes, etc., is needed. Fortunately, HadoopWatch can directly benefit such coflow optimization by providing most of such information ahead of time. For example, in some cases HadoopWatch can even forecast the shuffle flows 5–20 s before they are sent to network as we observed in our evaluation (Section IV-B). This gives us abundant time to calculate the routing and scheduling for each flow. In what follows, we introduce a simple HadoopWatch-enabled network optimizer, called HadoopOptimizer, that leverages the forecasting results provided by HadoopWatch to optimize Hadoop applications. For simplicity, our focus is to optimize the average CCT. We must acknowledge that the proposed algorithm below is by no means optimal, and our hope is just to demonstrate the utility of HadoopWatch. We show that even a simple algorithm can leverage the forecasting results provided by HadoopWatch to significantly improve the Hadoop job performance. Designing a comprehensive/optimal coflow routing/scheduling algorithm is challenging and is beyond the scope of this paper.

C. Simple HadoopWatch-Enabled Optimizer

The HadoopOptimizer framework is shown in Fig. 14. It consists of two modules: 1) the *coflow-aware routing module* that calculates the routing path for each incoming flow based on the coflow information inferred by HadoopWatch; and 2) the *coflow-aware scheduling module* that determines the scheduling priority of each flow based on its path and characteristics (e.g., flow size, whether it belongs to the last batch of flows of a

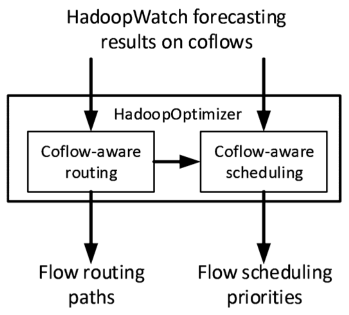


Fig. 14. HadoopOptimizer—a HadoopWatch-enabled network optimization framework.

coflow). The scheduling module updates the priorities of related flows when a flow enters the network or completes. It is worthwhile to note that in our current design HadoopOptimizer only works on large flows, while small flows route through ECMP by default and receive the highest priority as we will explain subsequently.

1) Coflow-Aware Flow Routing: The goal of coflow-aware flow routing is to fully utilize network bandwidth by performing load-balancing, subject to a soft constraint that all co-dependent flows in a coflow should use link-disjoint paths. This is because that in order to further reduce average CCT through scheduling, co-dependent flows in a coflow should be spread to different paths as discussed in Section V-A.

When computing a routing path for flow f, to approximate coflow-aware load-balanced routing, we compute the minimal-weighted shortest path and set the weight on each link l as follows:

$$w_l = \left(N_l^f + 1\right) \times L_l \qquad (1)$$

where N_l^f is the number of co-dependent flows in the same coflow as f using link l, and L_l is the relative traffic load on link l. Using w_l, the algorithm in general prefers a light-loaded (due to L_l) and link-disjoint (due to N_l) path for each incoming flow f. Here, L_l can be calculated by

$$L_l = \frac{\sum_{f \in F_l} v_f}{\mathrm{RB}_l} \qquad \forall l \in P_f \tag{2}$$

where P_f is the flow path used by flow f, F_l is all the flows on link l, v_f is the volume of flow f, and RB_l is the residual bandwidth on link l that is reserved for elephant flows. This formula also reflects the time to take to serve all the elephant flows on link l.

Note that HadoopOptimizer performs online flow routing. A key issue with such online flow routing is that it may lead race condition, i.e., the order to route flows may affect the effect of load-balancing. Recall the example in Fig. 13, if we route the flows in the order of f_{a1}, f_{a2}, f_{b1}, f_{b2} to minimize the maximal link utilization, the flows will be routed as in Fig. 13(a), which is not a good load-balanced result. To achieve a better result as in Fig. 13(c), we should route flows in a decreasing order in terms of the flow sizes. This requires prior knowledge of incoming flows. With HadoopWatch, we can predict the incoming flows and their sizes. Thus, in HadoopOptimizer, if we forecast that some larger flows will be arriving later, we will reserve paths for those larger flows proactively.

Another issue is how to handle small flows. Previous measurement study [5] shows that 80% of the flows in datacenters are less than 10 kB, while most of the bytes are in the 10% of large flows. Most of these short flows can be completed within 1 or 2 round-trip times (RTTs), and they cannot fully utilize the link capacity. Thus, explicitly assigning routing paths for short flows according to link bandwidth is not necessary. In HadoopOptimizer, we simply use the network bandwidth-delay product (BDP) to differentiate between mice and elephant flows, and forward mice flows using ECMP. To minimize the queueing delay of mice flows, we prioritize them in the network.

2) Coflow-Aware Flow Scheduling: In HadoopWatch, we observed that co-dependent flows in a coflow may enter the network asynchronously. Motivated by this, the main idea of coflow-aware flow scheduling in HadoopOptimizer is to speed up a coflow only when the last batch of flows (or tail flows) of this coflow begin to enter the network. This is because accelerating the first few flows (or head flows) in a coflow does not directly translate to the completion of the coflow, as the coflow completion time is decided by the completion of the last flow. With HadoopWatch, we know the total number of co-dependent flows in each coflow *a priori*, and based on this we can well control when to speed up.

To minimize average CCT, HadoopOptimizer tries to approximate the shortest job first (SJF) scheduling. However, the SJF schemes introduced in PDQ [24] and pFabric [3] for scheduling individual flows do not directly apply to our case of scheduling coflows. To simplify the problem, we define the size of a coflow as the average flow size of all active flows in this coflow. Once the last batch of flows of a coflow arrives, we calculate the remaining size of this coflow and assign the priority accordingly: Smaller coflows will be given higher priorities. In the network, coflows with higher priorities will preempt those with lower priorities, thus mimicking SJF. We note that this heuristic is coarse-grained and might be far from optimal, even though we see impressive performance gains in our evaluation subsequently.

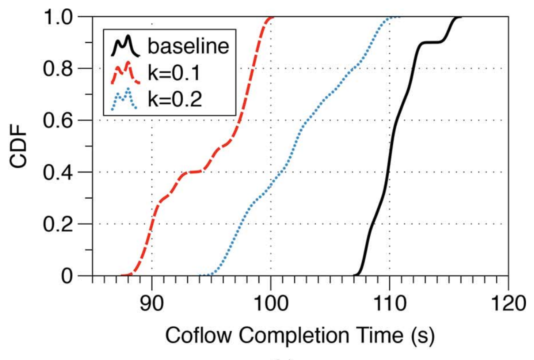
One concern with SJF is starvation. However, HadoopOptimizer can well handle such problem. For mice flows, because they are prioritized in the network, they are free of starvation. For large flows, to protect them from persistent starvation (actually temporary packet loss or TCP timeout is less harmful to large flows), we resort to aging—HadoopOptimizer periodically calculates the average transmission rate of each coflow in the network during the past period. If the average transmission rate of a coflow is less than a predefined threshold, the priority of this coflow will be increased gradually. We note that another way to avoid starvation is to use weighted fair queueing (WFQ) instead of priority queueing, however in our implementation of this paper, we do not enable WFQ.

As mentioned above, in order to transmit mice flows with minimal queueing delay, we reserve the highest priority level to them. Due to the small size of a mice flow, it cannot fully utilize the link bandwidth, even if multiple mice flows reside on the same link simultaneously, the competition among these mice flows would not introduce severe delay. In our implementation, we statically provision 5% of link bandwidth for mice flows. However, we note that dynamically predicting mice flow demand based on its average usage in the past such as [25] can be adopted in our framework to perform adaptive mice flow bandwidth provisioning.

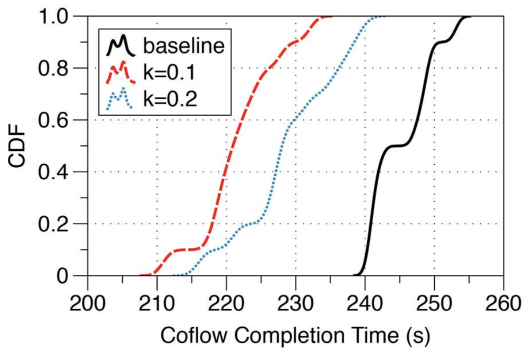
A potential problem with HadoopOptimizer is that, if we set priorities naively based on the discussion above, we may need too many priority levels that are beyond the capability of existing commodity switches. For example, the Pronto-3295 Broadcom switch on our testbed can support 8 priority queues. Consider that a flow with lower priority will be almost blocked by flows with higher priority if they share the same link, we made the following approximation. For any path, there is only one higher-priority flow; all the rest of the flows share the same low-priority level (Note that we only refer to large flows here, all the mice flows use one highest priority level as discussed above). When a new flow comes, we compare this new flow with the current higher-priority flow, if the new flow's priority is higher, then it preempts the current higher-priority flow, whose priority is demoted to be low; Otherwise, the new flow's priority is set to be low. When a current higher-priority flow completes, we promote one low-priority flow to be higher provided that its priority is the highest among all the relevant flows. In this way, we only need 3 queues for implementation. One for mice flows with the highest priority, one for higher-priority large flows, and one for low-priority large flows.

D. Performance Improvement

We implemented HadoopOptimizer algorithm as a software module into the HadoopWatch controller to navigate data shuffle in Hadoop jobs. We expand our 10-server testbed in Section IV-A to 36 physical servers connected in a 2-layer leaf-spine network (2 spine switches and 4 leaf/ToR switches), in which each ToR connects to 9 servers. As before, all the 36 servers are Dell PowerEdge R320 with a quad-core Intel Xeon E5-1410 2.80 GHz CPU and 8 GB memory and run Debian GNU/Linux 6.0. We leverage XPath [26] to explicitly control flow routing, and adopt the built-in priority queueing function in each switch for flow scheduling.



(a)



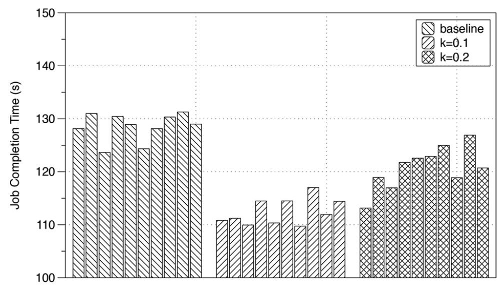
(b)

Fig. 15. CCT improvement enabled by HadoopWatch. (a) 25G TeraSort on Hadoop. (b) 50G TeraSort on Hadoop.

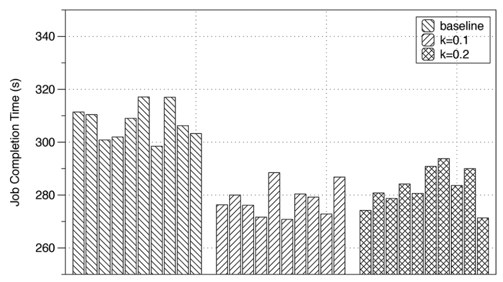
Over the Hadoop cluster, we run two Hadoop job benchmarks: TeraSort-25G and TeraSort-50G. We focus on measuring the improvement of CCT and job completion time (JCT). Our experiments take k as an input parameter, which indicates the time to speed up a coflow. For example, $k = 0.1$ means that only when the last 10% of flows in a coflow begin to enter the network, HadoopOptimizer starts to speed up this coflow. In our demonstration below, we show CCT and JCT under the baseline (i.e., without running HadoopOptimizer), $k = 0.1$, and $k = 0.2$.

Fig. 15 shows the cumulative distribution function (CDF) of CCT for TeraSort-25G and TeraSort-50G, respectively. We make two observations. First of all, enabled by HadoopWatch, HadoopOptimizer can significantly improve the completion times of coflows through coflow-aware routing and scheduling. For example, with $k = 0.1$, HadoopOptimizer reduces the average CCT by 13.64% (and 13.79% at the 99th percentile) for TeraSort-25G, and by 10.2% (and 7.84% at the 99th percentile) for TeraSort-50G respectively compared to the baseline. Second, we find that the CCT of $k = 0.1$ is better than that of $k = 0.2$. One possible reason is that, in our setting, it is still premature for us to speed up a coflow when the last 20% flows of this coflow just start to enter the network.

Fig. 16 demonstrates the CDF of JCT under the baseline, $k = 0.1$, and $k = 0.2$, respectively. In our experiments, we run both TeraSort-25G and TeraSort-50G for 10 times, and the measured JCT for each time is plotted as a bar in Fig. 16. It is evident that the improvement on CCT directly translates to job-level performance improvement; for example, the JCT can be improved by up to 14.72% in our evaluation. Overall, our simple design and evaluation already demonstrate the nontrivial benefits brought by HadoopWatch, and we expect HadoopWatch to be used in more aspects in the future.



(a)



(b)

Fig. 16. JCT improvement enabled by HadoopWatch. (a) 25G TeraSort on Hadoop. (b) 50G TeraSort on Hadoop.

VI. Related Works

Networking researchers have been exploring different ways to predict and estimate network traffic demands in different environments. For example, ISPs have employed various traffic demand matrices in their WAN traffic engineering and capacity planning [18], [34]. However, these methods required a large number of past statistics, such as server logs or link data, to provide reliable estimation of traffic demand in the next period of time. Such techniques are not suitable in data center networks, where most of the longest-lived flows last only a few seconds [13] and the traffic is elastic.

To gain more instant information for traffic demand forecasting in data center networks, many researchers proposed to estimate the traffic demand based on real-time measuring socket buffers in end-hosts [12], [39] or counters in network switches [2], [13], [17]. Such techniques are designed for general traffic load prediction at network layer, while our method targets a more accurate and fine-grained traffic forecasting with application-level semantics captured in real time.

Various tracing and profiling tools have been proposed to collect execution information of Hadoop. X-Trace [20] is integrated into Hadoop to collect cross-layer event traces for performance diagnosis. To avoid modifying Hadoop, researchers proposed to perform off-line analysis on Hadoop log files for performance tuning and anomaly detection [21], [36]. However, HadoopWatch focuses on predicting traffic demands based on real-time file system monitoring. Compared to a recent attempt that focused on predicting shuffle flows by periodically scanning Hadoop logs [14], HadoopWatch can provide more fine-grained traffic forecasting and more scalable event-driven monitoring.

VII. Conclusion and Future Work

In this paper, we have proposed to use file system monitoring to provide comprehensive traffic forecasting for big data and cloud applications. We developed HadoopWatch, a traffic forecaster for Hadoop, that can forecast Hadoop traffic accurately and efficiently without modifying the Hadoop framework. We believe application-layer traffic forecasting is a key building block to enable workload optimized networking in cloud data centers and tightly integrates network design with applications.

We have implemented HadoopWatch and deployed it on a small-scale testbed with 10 physical machines and 30 virtual machines. Our experiments over a series of Hadoop applications demonstrate that HadoopWatch can predict the application-layer traffic demand with almost 100% accuracy in advance, while introducing little overhead to the application performance.

We further demonstrated the utility of HadoopWatch by showing a case that coflows predicted by HadoopWatch can be used to improve the performance of Hadoop jobs. For this, we implemented a network optimizer, HadoopOptimizer, into the HadoopWatch controller, and through running Hadoop job benchmarks we found that even a simple optimizer can leverage HadoopWatch to significantly improve the Hadoop job completion time by up to 14.72%.

Our work is a first attempt at exploring the space of comprehensive traffic forecasting at application layer, and many follow-up problems remain to be explored in future work. For example, HadoopWatch is now a traffic forecaster particularly designed for the Hadoop framework; an important question is how to generalize it to other frameworks. On the one hand, by run-time file system monitoring, we believe the Hadoop-Watch-like idea can not only be applied to Hadoop, but also to some other frameworks such as Dryad [28], CIEL [32], Pregel [30], etc. Since most of these distributed computing frameworks need to cache data blocks in local disks when performing distributed computation, monitoring file system activities and analyzing log files can potentially provide data traffic information. On the other hand, for frameworks based on memory, such as Spark [45], a possible way for traffic forecasting might be through monitoring both memory and file system activities. However, whether this is feasible and how it can be done require further investigation.

Acknowledgment

This work was performed when Y. Zhao and H. Wang were interns with the SING Group at Hong Kong University of Science and Technology (HKUST).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM*, 2008, pp. 63–74.
[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, 2010, p. 19.
[3] M. Alizadeh *et al.*, "pFabric: Minimal near-optimal datacenter transport," in *Proc. SIGCOMM*, 2013, pp. 435–446.
[4] H. H. Bazzaz *et al.*, "Switching the optical divide: Fundamental challenges for hybrid electrical/optical datacenter networks," in *Proc. SOCC*, 2011, Art. no. 30.
[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. IMC*, 2010, pp. 267–280.
[6] T. Benson, A. Anand, A. Akella, and M. Zhang, "Micro TE: Fine grained traffic engineering for data centers," in *Proc. CoNEXT*, 2011, Art. no. 8.
[7] K. Chen *et al.*, "OSA: An optical switching architecture for data center networks with unprecedented flexibility," in *Proc. NSDI*, 2012, p. 18.
[8] L. Chen, S. Hu, K. Chen, H. Wu, and D. Tsang, "Towards minimal-delay deadline-driven data center TCP," in *Proc. HotNets*, 2013, Art. no. 21.
[9] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy efficiency for large-scale mapreduce workloads with significant interactive analysis," in *Proc. EuroSys*, 2012, pp. 43–56.
[10] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. HotNets*, 2012, pp. 31–36.
[11] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proc. SIGCOMM*, 2014, pp. 443–454.
[12] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, 2011, pp. 1629–1637.
[13] A. R. Curtis *et al.*, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, 2011, pp. 254–265.
[14] A. Das *et al.*, "Transparent and flexible network management for big data processing in the cloud," in *Proc. HotCloud*, 2013.
[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, p. 10.
[16] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. SIGCOMM*, 2014, pp. 431–442.
[17] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," in *Proc. ACM SIGCOMM*, 2010, pp. 339–350.
[18] A. Feldmann, N. Kammenhuber, O. Maennel, B. Maggs, R. De Prisco, and R. Sundaram, "A methodology for estimating interdomain web traffic demand," in *Proc. ACM IMC*, 2004, pp. 322–335.
[19] A. D. Ferguson, A. Guha, J. Place, R. Fonseca, and S. Krishnamurthi, "Participatory networking," in *Proc. Hot-ICE*, 2012, p. 2.
[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *Proc. NSDI*, 2007, p. 20.
[21] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. IEEE ICDM*, 2009, pp. 149–158.
[22] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. SOSP*, 2003, pp. 29–43.
[23] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.
[24] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. SIGCOMM*, 2012, pp. 127–138.
[25] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, and V. Gill, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.
[26] S. Hu *et al.*, "Explicit path control in commodity data centers: Design and applications," in *Proc. NSDI*, 2015, pp. 15–28.
[27] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. 26th IEEE ICDEW*, 2010, pp. 41–51.
[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. EuroSys*, 2007, pp. 59–72.
[29] R. Love, "Kernel Korner: Intro to Inotify Linux J," 2005.
[30] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. SIGMOD*, 2010, pp. 135–146.
[31] D. L. Mills, "RFC 1305. Network Time Protocol (Version 3)," 1992.
[32] D. Murray *et al.*, "CIEL: A universal execution engine for distributed data-flow computing," in *Proc. NSDI*, 2011, pp. 113–126.
[33] Y. Peng *et al.*, "Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing," in *Proc. IEEE INFOCOM*, 2014, pp. 19–27.
[34] M. Roughan, M. Thorup, and Y. Zhang, "Traffic engineering with estimated traffic matrices," in *Proc. ACM IMC*, 2003, pp. 248–258.
[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE MSST*, 2010, pp. 1–10.
[36] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: Visual log-analysis based tools for debugging Hadoop," in *Proc. HotCloud*, 2009, Art. no. 18.
[37] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *Proc. SIGCOMM*, 2012, pp. 115–126.
[38] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. OSDI*, 2004, p. 7.
[39] G. Wang *et al.*, "c-Through: Part-time optics in data centers," in *Proc. SIGCOMM*, 2010, pp. 327–338.

[40] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at runtime for big data applications," in *Proc. HotSDN*, 2012, pp. 103–108.

[41] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Map task scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality," in *Proc. IEEE INFOCOM*, 2013, pp. 1609–1617.

[42] K. C. Webb, A. C. Snoeren, and K. Yocum, "Topology switching for data center networks," in *Proc. Hot-ICE*, 2011, p. 14.

[43] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. SIGCOMM*, 2011, pp. 50–61.

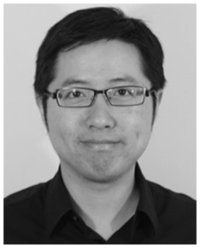
[44] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSys*, 2010, pp. 265–278.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. NSDI*, 2012, p. 2.

[46] X. Zhang *et al.*, "CPI2: CPU performance isolation for shared compute clusters," in *Proc. EuroSys*, 2013.

Yang Peng received the B.S. degree from Wuhan University, Wuhan, China, in 2010, and the M.Phil. degree from Hong Kong University of Science and Technology, Hong Kong, in 2014, both in computer science.

He is now a full-time Software Developing Engineer with the Bing search engine, Microsoft, Redmond, WA, USA. He has research experiences on system virtualization, big data applications, and data center networks.

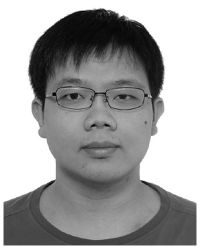


Kai Chen received the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2012.

He is an Assistant Professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interesst includes networked systems design and implementation, data center networks, and cloud computing.

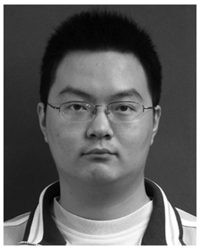
Guohui Wang received the Ph.D. degree in computer science from Rice University, Houston, TX, USA, in 2011.

He is an Engineer with Facebook, New York, NY, USA, working on network systems. Before joining Facebook, he was a Research Staff Member with the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, where his research has been focused on new network architectures, network virtualization, and management for cloud data centers.



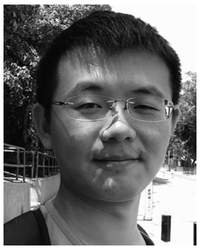
Wei Bai received the B.E. degree in information security from Shanghai Jiao Tong University, Shanghai, China, in 2013, and is currently pursuing the Ph.D. degree in computer science at Hong Kong University of Science and Technology, Hong Kong.

His current research interests are in the area of data center networks.



Yangming Zhao received the B.S. degree in communication engineering from the University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 2008, and is currently pursuing the Ph.D. degree at UESTC.

His research interests include network optimization, game theory, and data center networks.



Hao Wang received the B.S. degree in information security from Shanghai Jiao Tong University, Shanghai, China, in 2012, and is currently pursuing the Master degree in software engineering at the same university.

His research interests include load balancing schemes in DCN and distributed computing optimization.

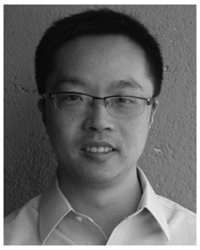
Yanhui Geng received the B.Eng. and M.Eng. degrees from the University of Science and Technology of China (USTC), Hefei, China, in 2002 and 2005, respectively, and the Ph.D. degree from the University of Hong Kong (HKU), Hong Kong, in 2009, all in computer science.

He is a Researcher and Project Manager with Huawei Noah's Ark Lab, Hong Kong. Before joining Huawei, he was a Post-Doctoral Research Fellow with HKU from 2009 to 2012, and was a Senior Engineer with the Hong Kong Science and Technology Research Institute (ASTRI), Hong Kong, from 2012 to 2013. He has over 15 technical publications in international journals and conferences. His research interests include software-defined networking (SDN), machine learning, big data analytics, cloud computing, and indoor positioning technology.

Dr. Geng received the IEEE ICC 2010 Best Paper Award.

Zhiqiang Ma received the B.S. degree from Fudan University, Shanghai, China, in 2009, and the Ph.D. degree from the Hong Kong University of Science and Technology (HKUST), Hong Kong, in 2014, both in computer science.

He is currently the co-founder and CTO of Hututa Technologies Limited, Hong Kong. His areas of interest include large-scale distributed computing and storage systems, operating systems and cloud computing.



Lin Gu received the B.S. degree from Fudan University, Shanghai, China, in 1996, the M.S. degree from Peking University, Beijing, China, in 2001, and the Ph.D. degree from the University of Virginia, Charlottesville, VA, USA, in 2006, all in computer science.

He is an Assistant Professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST), Hong Kong. His research interest includes big data systems, cloud computing, operating systems, and wireless sensor networks. He developed the Virtk (a.k.a., t-kernel) operating system, the CCMR cloud platform, and VOLUME, a datacenter-scale distributed virtual memory technology.