HiTune: Dataflow-Based Performance Analysis for Big Data Cloud

Jinquan Dai, Jie Huang, Shengsheng Huang, Bo Huang, Yan Liu
Intel Asia-Pacific Research and Development Ltd
Shanghai, P.R.China, 200241
{jason.dai, jie.huang, shengsheng.huang, bo.huang, yan.b.liu}@intel.com

Abstract

Although Big Data Cloud (e.g., MapReduce, Hadoop and Dryad) makes it easy to develop and run highly scalable applications, efficient provisioning and fine-tuning of these massively distributed systems remain a major challenge. In this paper, we describe a general approach to help address this challenge, based on distributed instrumentations and dataflow-driven performance analysis. Based on this approach, we have implemented HiTune, a scalable, lightweight and extensible performance analyzer for Hadoop. We report our experience on how HiTune helps users to efficiently conduct Hadoop performance analysis and tuning, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling).

1. Introduction

There are dramatic differences between delivering software as a service in the cloud for millions to use, versus distributing software as bits for millions to run on their PCs. First and foremost, services must be highly scalable, storing and processing an enormous amount of data. For instance, in June 2010, Facebook reported 21PB raw storage capacity in their internal data warehouse, with 12TB compressed new data added every day and 800TB compressed data scanned daily [1]. This type of "Big Data" phenomenon has led to the emergence of several new cloud infrastructures (e.g., MapReduce [2], Hadoop [2], Dryad [4], Pig [5] and Hive [6]), characterized by the ability to scale to thousands of nodes, fault tolerance and relaxed consistency. In these systems, the users can develop their applications according to a dataflow graph (either implicitly dictated by the programming/query model or explicitly specified by the users). Once an application is cast into the system, the cloud runtime is responsible for dynamically mapping the logical dataflow graph to the underlying cluster for distributed executions.

With these Big Data cloud infrastructures, the users are required to exploit the inherent data parallelism exposed by the dataflow graph when developing the applications; on the other hand, they are abstracted away from the messy details of data partitioning, task distribution, load balancing, fault tolerance and node communications. Unfortunately, this abstraction makes it very difficult, if not impossible, for the users to understand the cloud runtime behaviors. Consequently, although Big Data Cloud makes it easy to develop and run highly scalable applications, efficient provisioning and fine-tuning of these massively distributed systems remain a major challenge. To help address this challenge, we attempt to design tools that allow users to understand the runtime behaviors of Big Data Cloud, so that they can make educated decisions regarding how to improve the efficiency of these massively distributed systems – just as what traditional performance analyzers do for a single execution of a single program.

Unfortunately, performance analysis for Big Data Cloud is particularly challenging, because these applications can potentially comprise several thousands of programs running on thousands of machines, and the low level performance details are hidden from the users by using a high level dataflow model. In this paper, we describe a specific solution to this problem based on distributed instrumentations and dataflow-driven performance analysis, which correlates concurrent performance activities across different programs and machines, reconstructs the dataflow-based, distributed execution process of the Big Data application, and relates the low level performance activities to the high level dataflow model.

Based on this approach, we have implemented *HiTune*, a scalable, lightweight and extensible performance analyzer for Hadoop. We report our experience on how HiTune helps users to efficiently conduct Hadoop performance analysis and tuning, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling). For instance, reconstructing the dataflow execution process of a Hadoop job allows users to understand the dynamic interactions between different tasks and stages (e.g., task scheduling and data shuffle; see sections 7.1 and 7.2). In addition, relating performance activities to the dataflow model allows users to conduct fine-grained,

dataflow-based hotspot breakdown (e.g., for identifying application hotspots and hardware problems; see sections 7.2 and 7.3).

The rest of the paper is organized as follows. In section 2, we introduce the motivations and objectives of our work. We give an overview of our approach in section 3, and present the dataflow-based performance analysis in section 4. In section 5, we describe the implementation of HiTune, a performance analyzer for Hadoop. We experimentally evaluate HiTune in section 6, and report our experience in section 7. We discuss the related work in section 8, and finally conclude the paper in section 9.

2. Problem Statement

In this section, we describe the motivations, challenges, goals and non-goals of our work.

2.1 Big Data Cloud

In Big Data Cloud, the input applications are modeled as directed acyclic dataflow graphs to the users, where graph vertices represent processing stages and graph edges represent communication channels. All the data parallelisms of the computation and the data dependencies between processing stages are explicitly encoded in the dataflow graph. The users can develop their applications by simply supplying programs that run on the vertices to these systems; on the other hand, they are abstracted away from the low level details of the distributed executions of their applications. The cloud runtime is responsible for dynamically mapping the logical dataflow graph to the underlying cluster, including generating the optimized dataflow graph of execution plans, assigning the vertices and edges to physical resources, scheduling and executing each vertex (usually using multiple instances and possibly multiple times due to failures).

For instance, the MapReduce model dictates a two-stage group-by-aggregation dataflow graph to the users, as shown in Figure 1. A MapReduce application has one input that can be trivially partitioned. In the first stage a *Map* function, which specifies how the grouping is performed, is applied to each partition of input data. In the second stage a *Reduce* function, which performs the aggregation, is applied to each group produced by the first stage. The MapReduce framework is then responsible for mapping this logical dataflow graph to the physical resources. For instance, the Hadoop framework automatically executes the input MapReduce application using an internal dataflow graph of execution plan, as shown in Figure 2. The input data is divided into *splits*, and a distinct Map task is launched for each split. Inside each Map task, the map stage applies the *Map* function to the input data, and the spill stage stores the map output on local disks. In addition, a distinct Reduce task is launched for each partition of the map outputs. Inside each Reduce task, the copier and merge stages run in a pipelined fashion, fetching the relevant partition over the network and merging the fetched data respectively; after that, the sort and reduce stages merge the reduce inputs and apply the *Reduce* function respectively.

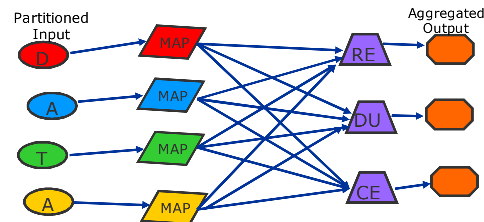

Figure 1. Dataflow graph of a MapReduce application

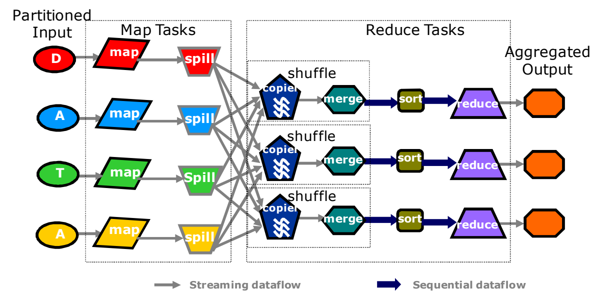

Figure 2. Dataflow graph of the Hadoop execution plan

Pig Script

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>1000000;
output = FOREACH big_groups GENERATE category,
          AVG(good_urls.pagerank);
```

Hive Query

```
SELECT category, AVG(pagerank)
FROM (SELECT category, pagerank, count(1) AS recordnum
         FROM urls WHERE pagerank > 0.2
         GROUP BY category) big_groups
WHERE big_groups.recordnum > 1000000
```

Figure 3. The Pig program [5] and Hive query example

In addition, the Pig and Hive systems allow the users to perform ad-hoc analysis of Big Data on top of Hadoop, using dataflow-style scripts and SQL-like queries respectively. For instance, Figure 3 shows the Pig program (an example in the original Pig paper [5]) and Hive query for the same operation (i.e., finding, for each sufficiently large category, the average pagerank of high-pagerank urls in that category). In these two systems, the logical dataflow graph of the operation is implicitly dictated by the language or query model, and

is automatically compiled into the physical execution plan (another dataflow graph) that is executed on the underlying Hadoop system.

Unlike the aforementioned systems that restrict their applications' dataflow graph, the Dryad system allows the users to specify an arbitrary directed acyclic graph to describe the application, as illustrated in Figure 4 (an example in the Dryad website [7]). The cloud runtime then refines the input dataflow graph and executes the optimized execution plan on the underlying cluster.

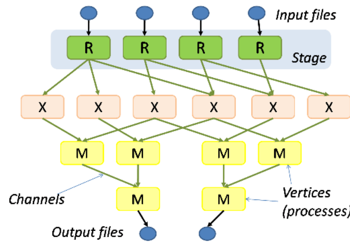

Figure 4. Dataflow graph of a Dryad application [7]

2.2 Motivations and Challenges

By exposing data parallelisms through the dataflow model and hiding the low level details of the underlying cluster, Big Data Cloud allows users to work at the appropriate level of abstraction, which makes it easy to develop and run highly scalable applications. Unfortunately, this abstraction makes it very difficult, if not impossible, for users to understand the cloud runtime behaviors. Consequently, it remains as a big challenge to efficiently provision and tune these massively distributed systems, which entails requesting and allocating the optimal number of (physical or virtual) resources, and optimizing the system and applications for better resource utilizations.

As Big Data Cloud grows in pervasiveness and scale, addressing this challenge becomes critically important (for instance, tuning Hadoop jobs is considered as a very difficult problem and requires a lot of efforts on understanding Hadoop internals in Hadoop community [8]; in addition, lack of tuning tools for Hadoop often forces users to resort to trial and error tuning [9]). This motivates our work to design tools that allows users to understand the runtime behaviors of Big Data applications, so that they can make educated decisions regarding how to improve the efficiency of these massively distributed systems – just as what traditional performance analyzers (e.g., gprof [10] and Intel VTune [11]) do for a single execution of a single program. Unfortunately, performance analysis for Big Data Cloud is particularly challenging due to its unique properties.

- *Massively distributed systems*: Each Big Data application is a complex distributed application, which may comprise tens of thousands of processes and threads running on thousands of machines. Understanding system behaviors in this context would require correlating concurrent performance activities (e.g., CPU cycles, retired instructions, lock contentions, etc.) across many programs and machines with each other.
- *High level abstractions*: Big Data Cloud allows users to work at an appropriately high level of abstraction, by hiding the messy details of parallelisms behind the dataflow model and dynamically instantiating the dataflow graph (including resource allocations, task scheduling, fault tolerance, etc.). Consequently, it is very difficult, if not impossible, for users to understand how the low level performance activities can be related to the high level abstraction (which they have used to develop and run their applications).

In this paper, we address these technical challenges through distributed instrumentations and dataflow-driven performance analysis. Our approach allows users to easily associate different low level performance activities with the high level dataflow model, and provide valuable insights into the runtime behaviors of Big Data Cloud and applications.

2.3 Goals and Non-Goals

Our goal is to design tools that help users to efficiently conduct performance analysis for Big Data Cloud. In particular, we want our tools to be broadly applicable to many different applications and systems, and to be applicable to even production systems (because it is often impossible to reproduce the cloud behaviors given the scale of Big Data Cloud). Several concrete design goals result from these requirements.

- *Low overhead*: It is critical that our tools have negligible (e.g., less than a few percent) performance impacts on the running applications.
- *No source code modifications*: Our tools should not require any modifications to the cloud runtime, middleware, messages, or applications.
- *Scalability*: Our tools need to handle applications that potentially comprise tens of thousands of processes/threads running on thousands of servers.
- *Extensibility*: We would like our tools to be flexible enough so that it can be easily extended to support different cloud systems.

We also have several non-goals.

- We are not developing tools that can replace the need for developers (e.g., by automatically

allocating the right amount of resources). Performance analysis for distributed systems is hard, and our goal is to make it easier for users, not to automate it.

- Our tools are not meant to verify the correct system behaviors, or diagnose the cause of faulty behaviors.

3. Overview of Our Approach

Our approach relies on distributed instrumentations on each node in the cloud, and then aggregating all the instrumentation results for dataflow-based analysis. The performance analysis framework consists of three major components, namely *tracker*, *aggregation engine* and *analysis engine*, as illustrated in Figure 5.

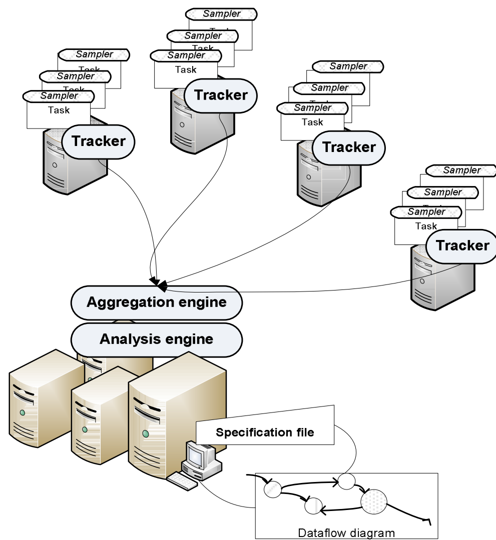

Figure 5. Performance analysis framework

Timestamp	Type	Target	Value

Figure 6. Format of the sampling record

The tracker is a lightweight agent running on every node. Each tracker has several *samplers*, which inspect the runtime information of the programs and system running on the local node (either periodically or based on specific events), and sends the sampling records to the aggregation engine. Each sampling record is of the format shown in figure 6.

- *Timestamp* is the sampling time for each record. Since the nodes in the cloud are usually in the same administrative domain and highly connected, it is easy to have all the nodes time-synchronized (e.g., in Hadoop all the slaves exchange *heartbeat* messages with the master periodically and the time synchronization information can be easily piggybacked); consequently the sampler can directly record its sampling time. Alternatively, the sampler can send the sampling record to the aggregation engine in real-time, which can then record the receiving time.
- *Type* specifies the type of the sampling record (e.g., CPU cycles, disk bandwidth, log files, etc.).
- *Target* specifies the source of the sampling record. It contains the name of the local node, as well as other sampler-specific information (e.g., CPUID, network interface name or log file name).
- *Value* contains the detailed sampling information of this record (e.g., CPU load, network bandwidth utilization, or a line/record in the log file).

The aggregation engine is responsible for collecting the sampling information from all the trackers in a distributed fashion, and storing the sampling information in a separate monitoring cluster for analysis. Any distributed log collection tools (e.g., Chukwa [12], Scribe [13] and Flume [14]) can be used as the aggregation engine. In addition, the analysis engine runs on the monitoring cluster, and is responsible for conducting the performance analysis and generating the analysis report, using the collected sampling information and a specification file describing the logical dataflow model of the specific Big Data cloud.

4. Dataflow-Based Performance Analysis

In order to help users to understand the runtime behaviors of Big Data Cloud, our framework presents the performance analysis results in the same dataflow model that is used in developing and running the applications. The key technical challenge is to re-construct the high level, dataflow-based, distributed and dynamic execution process for each Big Data application, based on the low level sampling records collected across different programs and machines. We address this challenge by:

1) Running a *task execution sampler* on every node to collect the execution information of each task in the application.
2) Describing the high level dataflow model of Big Data Cloud in a specification file provided to the analysis engine.
3) Constructing the dataflow execution process for the application based the dataflow specification and the program execution information.

4.1 Task Execution Sampler

To collect the program execution information, the task execution sampler instruments the cloud runtime and tasks running on the local node, and stores associated information into its sampling records at fixed time

intervals as follows.

- The *Target* field of the sampling record needs to store the identifier (e.g., application name, task name, process ID and/or thread ID) of the program that is instrumented to collect this piece of sampling information.
- The *Value* field of the sampling record must contain the *execution position* of the program (e.g., thread name, stack trace, basic-block ID and/or instruction address) at which the program is running when it is instrumented to collect this piece of sampling information.

In practice, the task execution sampler can be implemented using any traditional instrumentation tool (which runs on a single machine), such as Intel VTune.

4.2 Dataflow Specification

In order for the analysis engine to efficiently conduct the dataflow-based analysis, the dataflow specification needs to describe not only the dataflow graph (e.g., vertices and edges), but also the high level resource mappings of the dataflow model (e.g., physical implementations of vertices/edges, parallelisms between different phases/stages, and communication patterns between different stages). Consequently, the dataflow specification does require *a priori* knowledge of the cloud system. On the other hand, the users are not required to write the specification; instead, the dataflow specification is provided by the cloud system or the performance analyzer, and is either written by the developers of the cloud system (and/or the performance analyzer), or dynamically generated by the cloud runtime (e.g., by the Hive query compiler). The format of the dataflow specification is illustrated in Figure 7 and described in detail below.

- The *Input* (*Output*) section contains a list of *<inputId: storage location>* (*<outputId: storage location>*), where the storage location specifics which storage system (e.g., HDFS or MySQL) is used to store the input (output) data.

- The *Vertices* section contains a list of *<vertexId: program location>*, where the *program location* specifies the portion of program (e.g., the corresponding thread or function) running on this graph vertex (i.e., processing stage). It is required that each *execution position* collected by the task execution sampler can be mapped to a unique program location in the specification, so that the analysis engine can determine which vertex each task execution sampling record belongs to.

```
//dataflow graph
Input { //list of <inputId: storage location>
  In1: storage location
  …
}
Output { //list of <outputId: storage location>
  Out1: storage location
  …
}
Vertices { //list of <vertexId: program location>
  V1: program location
  …
}
Edges { //list of <edgeId: inputId/vertexId→vertexId/outputId>
  E1: In1→V1
  E2: V1→V2
  …
}
//resource mapping
Vertex Mapping { //list of Task Pool
  Task Pool [(name)] <(cardinality)> { //ordered list of Phase
    Phase [(name)] { //list of Thread Pool or Thread Group Pool
      Thread Pool [(name)] <(cardinality)> {
        //ordered list of vertexId
        V1, V2, …
      } //end of Thread Pool
      Thread Group Pool [(name)] <(cardinality, group size)> {
        //a single vertexId
        V3
      } //end of Thread Group Pool
      ..
    } //end of Phase
    Phase [(name)] { … }
    …
  } //end of Task Pool
  Task Pool [(name)] <(cardinality)> { … }
  …
}
Edge Mapping { //list of <edgeId: edge type, endpoint location>
  E1: edge type, endpoint location
  …
}
```

Figure 7. Dataflow specification of Big Data Cloud

- The *Edges* section contains a list of *<edgeId: inputId/vertexId→vertexId/outputId>*, which defines all the graph edges (i.e., communication channels).

- The *Vertex Mapping* section describes the high level resource mappings and parallelisms of the graph vertices. This section contains a list of *Task Pool* subsections; for each *Task Pool* subsection, the cloud runtime will launch several tasks (or processes) that can potentially run on the different nodes in parallel. The *Task Pool* subsection contains an ordered list of *Phase* subsections, and each task belonging to this task pool will sequentially execute these phases in the specified order.

- The *Phase* subsection contains a list of *Thread Pool* or *Thread Group Pool* subsections; for each of these subsections, the associated task will spawn several

threads or thread groups in parallel. The *Thread Pool* subsection contains an ordered list of *vertexId*, and each thread belonging to this thread pool will sequentially execute these vertices in the specified order. On the other hand, a number of threads (as determined by *group size*) in the thread group will run in concert with each other, executing the vertex specified in the *Thread Group Pool* subsection.

- The *cardinality* of the *Task Pool*, *Thread Pool* or *Thread Group Pool* subsections determines the numbers of instances (i.e., processes, threads or thread groups) to be launched. It can have several values as follows.
  (1) *N* – exactly *N* instances will be launched.
  (2) *1~N* – up to *N* instances will be launched.
  (3) *1~∞* – the number of instances to be launched is dynamically determined by the cloud runtime.

- The *Edge Mapping* section contains a list of *<edgeId: edge type, endpoint location>*. The edge type specifies the physical implementation of the edge, such as network connection, local file or memory buffer. The endpoint location specifies the communication patterns between the vertices, which can be *intra-thread/intra-task/intra-node* (i.e., data transfer exists only between vertex instances running in the same thread, the same task and the same node respectively), or *unconstrained*.

It is possible to extend the specification to support even more complex dataflow model and resource mappings (e.g., *Process Group Pool*); however, the current model is sufficient for all the Big Data cloud infrastructures that we have considered. For instance, the dataflow specification for our Hadoop cluster is shown in Figure 8.

4.3 Dataflow-Based Analysis

As described in the previous sections, the program execution information collected by task execution samplers is generic in nature, and the dataflow model of the specific Big Data cloud is defined in a specification file. Based on these data, the analysis engine can re-construct the dataflow execution process for the Big Data applications, and associate different performance activities with the high level dataflow model. In this way, our framework can be easily applied to different cloud systems by simply changing the specification file. We defer the detailed description of the dataflow construction algorithm to section 5.3.

```
//Hadoop dataflow graph
Input { //list of <inputId: storage location>
   Input:HDFS
}
Output { //list of <outputId: storage location>
   Output:HDFS
}
Vertices { //list of <vertexId: program location>
   map:      MapTask.run
   spill:    SpillThread.run
   copier:   MapOutputCopier.run
   merge:    InMemFSMergeThread.run or
             LocalFSMerger.run
   sort:     ReduceCopier.createKVIterator#ReduceCopier.access
   reduce:   runNewReducer or runOldReducer
}
Edges { //list of <edgeId: inputId/vertexId→vertexId/outputId>
   E1: Input→map
   E2: map→spill
   E3: spill→copier
   E4: copier→merge
   E5: merge→sort
   E6: sort→reduce
   E7: reduce→Output
}
Vertex Mapping { //list of Task Pool
   Task pool (Map) (1~∞) { //ordered list of Phase
      Phase { //list of Thread Pool or Thread Group Pool
         Thread Pool (1) {map}
         Thread Pool (1) {spill}
      }
   }
   Task Pool (Reduce) (1~∞) {
      Phase (shuffle) {
         Thread Group Pool (copy) (1, 20) {copier}
         Thread Group Pool (merge) (1, 2) {merge}
      }
      Phase { Thread Pool (1) {sort, reduce} }
   }
}
Edge Mapping { //list of <edgeId: edge type, endpoint location>
   E1: HDFS, unconstrained
   E2: memory buffer, intra-task
   E3: HTTP, unconstrained
   E4: memory buffer, intra-task
   E5: memory buffer or local file, intra-task
   E6: memory buffer or local file, intra-thread
   E7: HDFS, unconstrained
}
```

Figure 8. Hadoop dataflow specification

5. HiTune: A Dataflow-Based Hadoop Performance Analyzer

Based on our general performance analysis framework, we have implemented HiTune, a scalable, lightweight and extensible performance analyzer for Hadoop. In this section, we describe the implementation of HiTune, and in particular, how it is carefully engineered to meet our design goals that are described in section 2.3.

5.1 Implementation of Tracker

In our current implementation, all the nodes in the

Hadoop cluster are time synchronized (e.g., using a time service), and a tracker runs on each node in the cluster. Currently, the tracker consists of three samplers (i.e., *task execution sampler*, *system sampler* and *log file sampler*), and each sampler directly stores the sampling time in the *Timestamp* field of its sampling record.

We have chosen to implement the task execution sampler using binary instrumentation techniques, so that it can instrument and collect the program execution information without any source code modifications. Specifically, the task execution sampler runs as a Java programming language agent [15]; whenever the Hadoop framework launches a JVM to be instrumented, it dynamically attaches to the JVM the sampler agent, which samples the Java thread stack trace and state for all the threads in the JVM at specified intervals (during the entire lifetime of the JVM).

For each sampling record generated by the task execution sampler, its *Target* field contains the node name, the task name and the Java thread ID; and its *value* field contains the current execution position (i.e., the Java thread name and thread stack trace) as well the as the Java thread state. That is, the *Target* field is specified using the identifier of the runtime program instance (i.e., the thread ID), which allows the analysis engine to construct the entire sampling trace of each thread; in addition, the execution position is specified using the static program names (i.e., the Java thread name and method names), which allows the dataflow model and resource mappings to be easily described in the specification file.

In addition, the system sampler simply reports the system statistics (e.g., CPU load, disk I/O, network bandwidth, etc.) periodically using the *sysstat* package, and the log sampler reports the Hadoop log information (including Hadoop metrics and job history files) whenever there is new log information.

Since the tracker (especially the task execution sampler) needs to instrument the Hadoop tasks running on each node, it is the major source of performance overheads in HiTune. We have carefully designed and implemented the tracker (e.g., the task execution sampler caches the stack traces in memory and batches the write-out to the aggregation engine), so that it has very low (less than 2% according to our measurement) performance impacts on Hadoop applications.

5.2 Implementation of Aggregation Engine

To ensure its scalability, the aggregation engine is implemented as a distributed data collection system, which can collect the sampling information from potentially thousands of nodes in the Hadoop cluster. In the current implementation, we have chosen to use Chukwa (a distributed log collection framework) as our aggregation engine. Every sampler in HiTune directly sends its sampling records to the *Chukwa agent* running on the local node, which in turn sends data to the *Chukwa collectors* over the network; the collector is responsible for storing the sampling data in a (separate) monitoring Hadoop/HDFS cluster.

5.3 Implementation of Analysis Engine

The sampling data for a Hadoop job can be potentially very large in size (e.g., about 100GB for TeraSort [16][17] in our cluster). We address the scalability challenge by first storing the sampling data in HDFS (as described in section 5.2), and then running the analysis engine as a Hadoop application on the monitoring Hadoop cluster in an offline fashion.

Based on the *Target* field (i.e., the node name, the task name and the Java thread ID) of every task execution sampling record, the analysis engine first constructs a sampling trace for each thread (i.e., the sequence of all task execution sampling records belonging to that thread, ordered by the record timestamps) in the Hadoop job.

The program location (used in the dataflow specification) can therefore be defined as a range of consecutive sampling records in one thread trace (or, in the case of thread group, multiple ranges each in a different thread). Each record range is identified by the starting and ending records, which are specified using their execution positions (i.e., partial stack traces). For instance, all the records between the first appearance and the last appearance of *MapTask.run* (or simply the *MapTask.run* method) constitute one instance of the *map* vertex. See Figure 8 for the detailed dataflow specification of our Hadoop cluster.

Based on the *Target* and *Timestamp* fields of the two boundary records of corresponding program locations, the analysis engine then determines which machine each instance of a vertex runs on, when it starts and when it ends. Finally, it associates all the system and log file sampling records to each instance of the dataflow graph vertex (i.e., the processing stage), again using the *Target* and *Timestamp* information of the records.

With the algorithm and dataflow specification described above, the analysis engine can easily reconstruct the

dataflow execution for the Hadoop job and associates different sampling records with the dataflow graph. In addition, the performance analysis algorithm is itself implemented as a Hadoop application, which processes the sampling records for each JVM simultaneously.

Consequently, we can generate various analysis reports that provide valuable insights into the Hadoop runtime behaviors (presented in the same dataflow model used in developing and running the Hadoop job). For instance, a timeline based execution chart for all task instances, similar to the pipeline space-time diagram [18], can be generated so that users can get a complete picture about the dataflow-based execution process of the Hadoop job. It is also possible to generate the hotspot breakdown (e.g., disk I/O vs. network transfer vs. computations) for each vertex in the dataflow, so that users can identify the possible bottlenecks in the Hadoop cluster. We show some analysis reports and how they are used to help our Hadoop performance analysis and tuning in section 7.

6. Evaluations

In this section, we experimentally evaluate the runtime overheads and scalability of HiTune, using three benchmarks (namely, Sort, WordCount and Nutch indexing) in the *HiBench* Hadoop benchmark suite [17], as shown in Table 1. The Hadoop cluster used in our experiment consists of one master (running JobTracker and NameNode) and up to 20 slaves (running TaskTracker and DataNode); the detailed server configurations are shown in Table 2. Every server has two GbE NICs, each of which is connected to a different gigabit Ethernet switch, forming two different networks; one network is used for the Hadoop jobs, and the other is used for administration and monitoring tasks (e.g., the Chukwa aggregation engine in HiTune).

Table 1. Hadoop benchmarks

Benchmark	Input Data
Sort	60GB generated by *RandomWriter* example.
WordCount	60GB generated by *RandomTextWriter* example
Nutch indexing	19GB data generated by crawling an in-house Wikipedia mirror

Table 2. Server configurations

Processor	Dual-socket quad-core Intel Xeon processor
Disk	4 SATA 7200RPM HDDs
Memory	24 GB ECC DRAM
Network	2 Gigabit Ethernet NICs
OS	Redhat Enterprise Linux 5.4

We evaluate the runtime overheads of HiTune by measuring the Hadoop performance (speed and throughput) as well as the system resource (e.g., CPU and memory) utilizations of the Hadoop cluster. The speed is measured using the job running time, and the throughput is defined as the number of tasks completed per minute when the Hadoop cluster is at full utilization (by continuously submitting multiple jobs to the cluster). In addition, we evaluate the scalability of HiTune by analyzing that, when there are more nodes in the Hadoop cluster, whether the runtime overheads increase and whether it becomes more complex for HiTune to conduct the dataflow-based performance analysis.

6.1 Runtime Overheads

As mentioned in section 5.1, the tracker (especially the task execution sampler) is the major source of runtime overheads in HiTune. This is because the task execution sampler needs to instrument the Hadoop tasks running on each node, while the aggregation and analysis of sampling data are performed on a separate monitoring cluster in an offline fashion.

We first compare the instrumented Hadoop performance (measured when the tracker is running) and the baseline performance (measured when the tracker is completely turned off). In our experiment, when the tracker is running, the task execution sampler dumps the Java thread stack trace every 20 milliseconds, the system sampler reports the system statistics every 5 seconds, and the Hadoop cluster is configured to output its metrics to the log file every 10 seconds. Figures 9 and 10 show the ratio of the instrumented performance over the baseline performance for job running time (lower is better) and throughput (higher is better) respectively. It is clear that the overhead of running the tracker is very low in terms of performance – the instrumented job running time is longer than the baseline by less than 2%, and the instrumented throughput is lower than the baseline by less than 2%.

In addition, we also compare the *instrumented system resource utilizations* (measured when the tracker is running) and the *baseline utilizations* (measured when only the system sampler is running, which is needed to report the system resource utilizations periodically). Since the sampling records are aggregated using a separate network, we only present the CPU and memory utilization results of the Hadoop cluster in this paper. Figures 11 and 12 show the ratio of the instrumented CPU and memory utilizations over the baseline utilizations respectively. It is clear that the overhead of running the tracker is also very low in terms of resource utilizations – either the instrumented CPU or memory utilization is higher than the baseline by less than 2%.

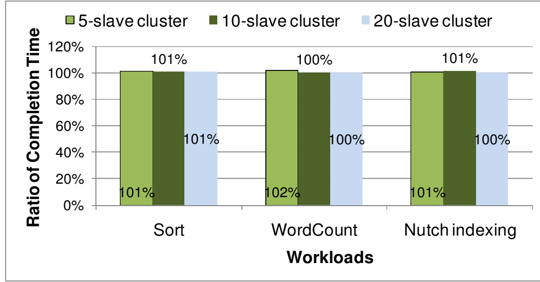

Figure 9. Ratio of instrumented job running time over baseline job running time

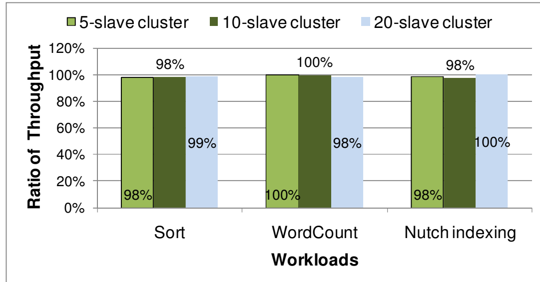

Figure 10. Ratio of instrumented cluster throughput over baseline cluster throughput

Figure 11. Ratio of instrumented cluster CPU utilization over baseline cluster CPU utilization

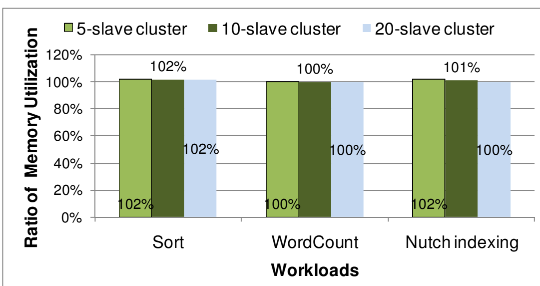

Figure 12. Ratio of instrumented cluster memory utilization over baseline cluster memory utilization

In summary, HiTune is a very lightweight performance analyzer for Hadoop, with very low (less than 2%) runtime overheads in terms of speed, throughput and system resource utilizations. In addition, HiTune scales very well in terms of the runtime overheads, because it instruments each node in the cluster independently and consequently the runtime overheads remain the same even when there are more nodes in the cluster (as confirmed by the experimental results).

6.2 Complexity of Performance Analysis

Since the analysis engine needs to re-construct the dataflow execution of a Hadoop job and associate the sampling records to each vertex instance in the dataflow, the complexity of analysis can be evaluated by comparing the sizes of sampling data and the numbers of vertex instances between different sized clusters.

Figure 13 shows the sampling data sizes for the 5-, 10- and 20-slave clusters. It is clear that the sampling data sizes remain about the same (or increase very slowly) for different sized clusters (e.g., only less than 18% increase in the sample data size even when the cluster size is increased by 4x). Intuitively, since HiTune samples each instance of the processing stages at fixed time intervals, the sampling data size is proportional to the sum of the running time of all vertex instances. As long as the underlying Hadoop framework scales well with the cluster sizes, the sum of the vertex instance running time will remain about the same (or increase very slowly), and so does the sampling data size. In practice, even with very large (1000s of nodes) clusters, a MapReduce job usually runs on about 100s of worker machines [19], and the Hadoop framework scales reasonably well with that number (100s) of machines.

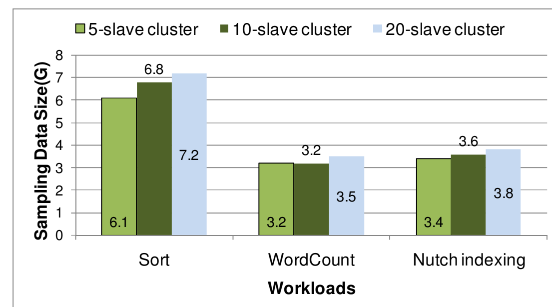

Figure 13. Comparison of sampling data sizes

In addition, assume M and R are the total numbers of the map and reduce tasks of a Hadoop job respectively. Since in the Hadoop dataflow model (as shown in Figure 8) each map task contains two stages and each reduce task contains four stages, the total number of vertex instances can be computed as $2*M+4*R$. In practice, the number of map tasks is about 26x of that of reduce tasks in average for each MapReduce job [20], and therefore the vertex instance count is about $2.15*M$. Since the number of map tasks (M) of a Hadoop job is typically determined by its input data size (e.g., by the number of HDFS file blocks), the number of vertex instances will also remain about the same for different sized clusters in practice.

In summary, the complexity for HiTune to conduct the dataflow-based performance analysis will remain about the same even when there are more nodes in the cluster (or, more precisely, the dataflow-based performance analysis in HiTune scales as well as Hadoop does with the cluster sizes), because the sampling data sizes and the vertex instance counts will remain about the same even when there are more nodes in the cluster. In addition, we have implemented the analysis engine as a Hadoop application, so that the dataflow-based performance analysis can be parallelized using another monitoring Hadoop cluster. For instance, to process the 100GB sampling data generated when running TeraSort in our cluster, it takes about 16 minutes on a single-slave monitoring cluster, and about 5 minutes on a 4-slave monitoring cluster.

7. Experience

HiTune has been used intensively inside Intel for Hadoop performance analysis and tuning (e.g., see [17]). In this section, we share our experience on how we use HiTune to efficiently conduct performance analysis and tuning for Hadoop, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling).

7.1 Tuning Hadoop Framework

One performance issue we encountered is extremely low system utilizations when sorting many small files (3200 500KB-sized files) using Hadoop 0.20.1 – system statistics collected by the cluster monitoring tools (e.g., Ganglia [21]) show that the CPU, disk I/O and network bandwidth utilizations are all below 5%. That is, there are no obvious bottlenecks or hotspots in our cluster; consequently, traditional tools (e.g., system monitors and program profilers) fail to reveal the root cause.

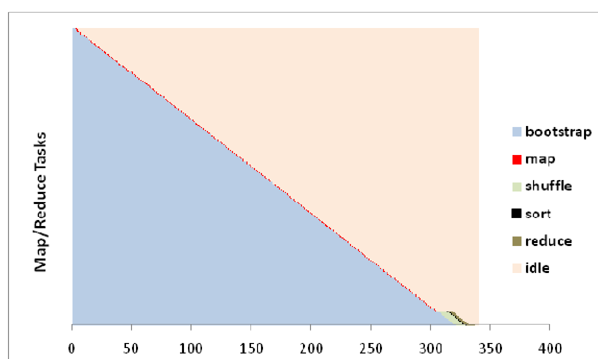

Figure 14. Dataflow execution for sorting many small files with Hadoop 0.20.1

To address this performance issue, we used HiTune to reconstruct the dataflow execution process of this Hadoop job, as illustrated in Figure 14. The x-axis represents the elapse of wall clock time, and each horizontal line in the chart represents a map or reduce task. Within each line, *bootstrap* represents the period before the task is launched, *idle* represents the period after the task is complete, *map* represents the period when the map task is running, and *shuffle*, *sort* and *reduce* represent the periods when (the instances of) the corresponding stages are running respectively.

As is obvious in the dataflow execution, there are few parallelisms between the Map tasks, or between the Map tasks and Reduce tasks in this job. Clearly, the task scheduler in Hadoop 0.20.1 (*Fair Scheduler* [22] is used in our cluster) fails to launch all the tasks as soon as possible in this case. Once the problem is isolated, we quickly identified the root cause – by default, the Fair Scheduler in Hadoop 0.20.1 only assigns one task to a slave at each heartbeat (i.e., the periodical keep-alive message between the master and slaves), and it schedules map tasks first whenever possible; in our job, each map task processes a small file and completes very fast (faster than the heartbeat interval), and consequently each slave runs the map tasks sequentially and the reduce tasks are scheduled after all the map tasks are done.

To fix this performance issue, we upgraded the cluster to *Fair Scheduler 2.0* [23][24], which by default schedules multiple tasks (including reduce tasks) in each heartbeat; consequently the job runs about 6x faster (as shown in Figure 15) and the cluster utilization is greatly improved.

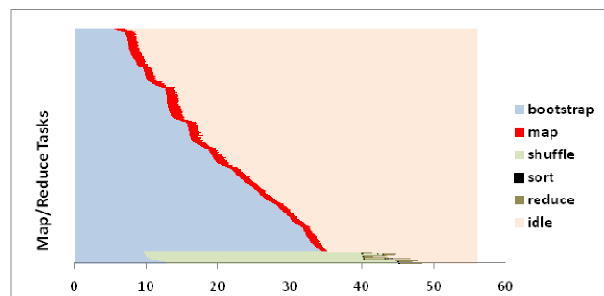

Figure 15. Dataflow execution for sorting many small files with Fair Scheduler 2.0

7.2 Analyzing Application Hotspots

In the previous section, we demonstrate that the high level dataflow execution process of a Hadoop job helps users to understand the dynamic task scheduling and assignment of the Hadoop framework. In this section,

we show that the dataflow execution process helps users to identify the data shuffle gaps between map and reduce, and that relating the low level performance activities to the high level dataflow model allows users to conduct fine-grained, dataflow-based hotspot breakdown (so as to understand the hotspots of the massively distributed applications).

Figure 16 shows the dataflow execution, as well as the timeline based CPU, disk and network bandwidth utilizations of TeraSort [16][17] (sorting 10 billion 100-byte records). It has high CPU utilizations during the map tasks, because the map output data are compressed (using the default codec in Hadoop) to reduce the disk and network I/O. (Compressing the input or output of TeraSort is not allowed in the benchmark specs).

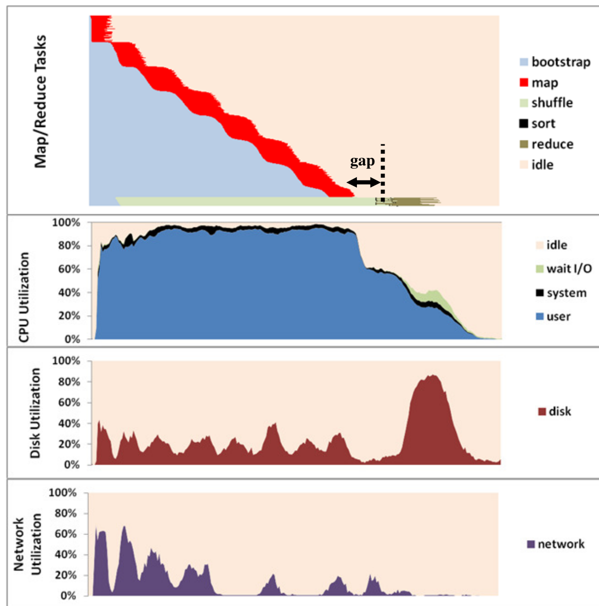

Figure 16. TeraSort (using default compression codec)

However, the dataflow execution process of TeraSort shows that there is a large gap (about 15% of the total job running time) between the end of map tasks and the end of shuffle phases. According to the communication patterns specified in the Hadoop dataflow model (see Figure 2 and Figure 8), shuffle phases need to fetch the output from all the map tasks in the copier stages, and ideally should complete as soon as all the map tasks complete. Unfortunately, traditional tools or Hadoop logs fail to reveal the root cause of the large gap, because during that period, none of the CPU, disk I/O and network bandwidth are bottlenecked, the "*Shuffle Fetchers Busy Percent*" metric reported by the Hadoop framework is always 100%, while increasing the number of *copier* threads does not improve the utilization or performance.

To address this issue, we used HiTune to conduct hotspot breakdown of the shuffle phases, which is possible because HiTune has associated all the low level sampling records with the high level dataflow execution of the Hadoop job. The dataflow-based hotspot breakdown (see Figure 17) shows that, in the shuffle stages, the *copier* threads are actually idle 80% of the time, waiting (in the *ShuffleRamManager. reserve* method) for the occupied memory buffers to be freed by the *memory merge* threads. (The idle vs. busy breakdown and the method hotspot are determined using the Java thread state and stack trace in the task execution sampling records respectively). On the other hand, most of the busy time of the *memory merge* thread is due to the compression, which is the root cause of the large gap between map and shuffle. To fix this issue and reduce the compression hotspots, we changed the compression codec to *LZO* [25], which improves the TeraSort performance by more than 2x and completely eliminates the gap (see Figure 18).

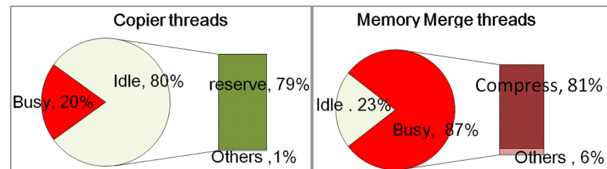

Figure 17. Copier and Memory Merge threads breakdown (using default compression codec)

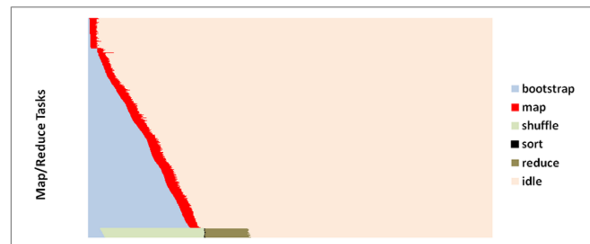

Figure 18. TeraSort (using *LZO* compression)

7.3 Diagnosing Hardware Problems

By examining Figure 18 in more detail, we also found that the reduce stage running time is significantly skewed among different reduce tasks – a small number of stages are much slower than the others, as shown in Figure 19.

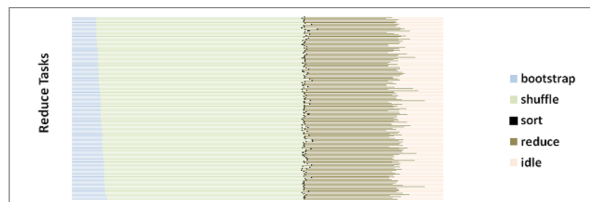

Figure 19. Reduce tasks of TeraSort (using *LZO* compression)

Based on the association of the low level sampling records and the high level dataflow model, we use HiTune to generate the normalized average running time and the idle vs. busy breakdown of the reduce stages (grouped by the Tasktrackers that the stages run on) in Figure 20. It is clear that reduce stages running on the 3rd and 7th TaskTrackers are much slower (about 20% and 14% slower than the average respectively). In addition, while all the reduce stages have about the same busy time, the reduce stages running on these two TaskTrackers have more idle time, waiting in the *DFSOutputStream.writeChunk* method (i.e., writing data to HDFS). Since the data replication factor in TeraSort is set to 1 (as required by the benchmark specs), the HDFS write operations in the reduce stage only writes to the local disks. By examining the average write bandwidth of the disks on these two TaskTrackers, we finally identified the root cause of this problem – there is one disk on each of these two nodes that is much slower than other disks in the cluster (about 44% and 30% slower than the average respectively), which is later confirmed to have bad sectors through a very expensive *fsck* process.

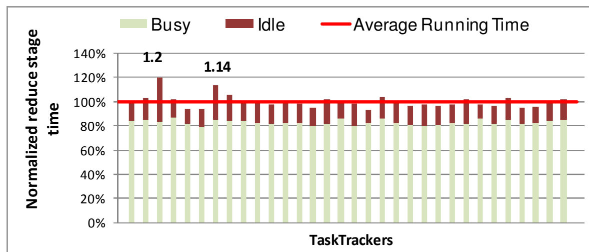

Figure 20. Normalized average running time and busy vs. idle breakdown of reduce stages

7.4 Extending HiTune to Other Systems

Since the initial release of HiTune inside Intel, it has been extended by the users in different ways to meet their requirements. For instance, new samplers are added so that processor microarchitecture events and power state behaviors of Hadoop jobs can be analyzed using the dataflow model.

In addition, HiTune has also been applied to Hive (an open source data warehouse built on top of Hadoop), by extending the original Hadoop dataflow model to include additional phases and stages, as illustrated in Figure 21. The map stage is divided into 5 smaller stages – namely, *Stage Init*, *Hive Init*, *Hive Active*, *Hive Close* and *Stage close*; in addition, the reduce stage is divided into 4 smaller stages – namely, *Hive Iinit*, *Hive Active*, *Hive Close* and *Stage Close*. This is accomplished by providing to the analysis engine a new specification file that describes the dataflow model and resource mappings in Hive.

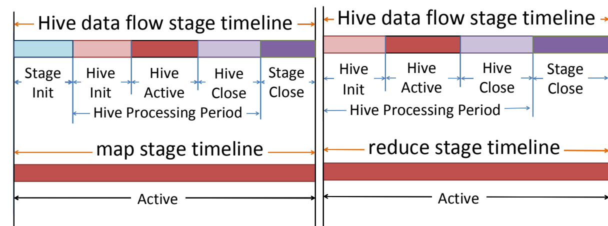

Figure 21. Extended dataflow model for Hive

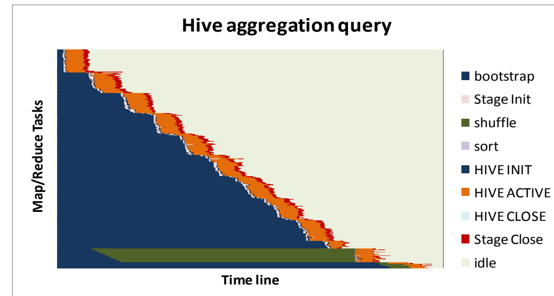

Figure 22. Dataflow execution of the Hive query

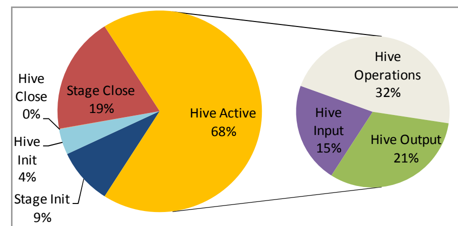

Figure 23. Map stage breakdown

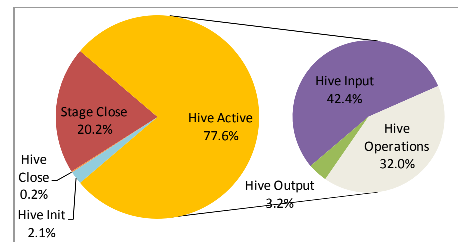

Figure 24. Reduce stage breakdown

Figure 22 shows the dataflow execution process for the aggregation query in Hive performance benchmarks [26][9]. In addition, Figures 23 and 24 show the dataflow-based breakdown of the map/reduce stages for the aggregation query (both the map and reduce Hive active stages are further broken into 3 portions: *Hive Input*, *Hive Operation* and *Hive Output* based on the Java methods). As shown in Figures 23 and 24, the query spends only about 32% of its time performing the Hive Operations; on the other hand, it spends about 68% of its time on the data input/output, as well as the initialization and cleanup of the Hadoop/Hive frameworks. Therefore, to optimize this Hive query, it is more critical to reduce the size of intermediate results,

to improve the efficiency of data input/output, and to reduce the overheads of the Hadoop/Hive frameworks.

8. Related Work

There are several distributed system tracing tools (e.g., Magpie [27], X-Trace [28] and Dapper [29]), which associates and propagates the tracing metadata as the request passes through the system. With this type of path information, the tracing tools can easily construct an event graph capturing the causality of events across the system, which can be then queried for various analyses [30]. Unfortunately, these tools would require changes not only to source codes but also to message schemas, and are usually restricted to a small portion of the system in practice (e.g., Dapper only instruments the threading, callback and RPC libraries in Google [29]). In contrast, our approach uses binary instrumentations to sample the tasks in a distributed and independent fashion at each node, and reconstructs the dataflow execution process of the application based on *a priori* knowledge of Big Data Cloud. Consequently, it requires no modifications to the system, and therefore can be applied more extensively to obtain richer information (e.g., the hottest function) than these tracing tools.

Our distributed instrumentations are similar to Google-Wide Profiling (GWP) [31], which samples across machines in multiple data centers for production applications. In addition, the current Hadoop framework can profile specific map/reduce tasks using traditional Java profilers (e.g., HPROF [32]), which however have very high overheads and are usually applied to a small (2 or 3) number of tasks. More importantly, both GWP and the existing profiling support in Hadoop focus on providing traditional performance analysis to the distributed systems (e.g., by allowing the users to directly query the low level sampling data). In contrast, our key technical challenge is to reconstruct the high level dataflow execution of the application based on the low level sampling data, so that users can work on the same dataflow model used in developing and running their Big Data applications.

In the industry, traditional cluster monitoring tools (e.g., Ganglia [21], Nagios [33] and Cacti [34]) have been widely used to collect system statistics (e.g., CPU load) from all the machines in the cluster; in addition, several large-scale log collection systems (e.g., Chukwa [12], Scribe [13] and Flume [14]) have been recently developed to aggregate log data from a large number of servers. All of these tools focus on providing a distributed framework to collect statistics and logs, and are orthogonal to our work (e.g., we have actually used Chukwa as the aggregation engine in the current HiTune implementation).

Existing diagnostic tools for Hadoop and Dryad (e.g., Vaidya [35], Kahuna [36] and Artemis [37]) focus on mining the system logs to detect performance problems. For instance, it is possible to construct the task execution chart (as shown in section 7.1) using the Hadoop job history files. Compared to these tools, our approach (based on distributed instrumentation and dataflow-driven performance analysis) has many advantages. First, it can provide much more insights, such as dataflow-based hotspot breakdown (see sections 7.2 and 7.3), into the cloud runtime behaviors. More importantly, performance problems of massively distributed systems are very complex, and are often due to issues that the developers are completely unaware of (and therefore are not exposed by the existing codes or logs). For instance, in section 7.2, the Hadoop framework shows that the shuffle fetchers are always busy, while detailed breakdown provided by HiTune reveals that copiers are actually idle most of the time. Finally, our approach is much more general, and consequently can be easily extended to support other systems such as Hive (see section 7.4).

9. Conclusions

In this paper, we propose a general approach of performance analysis for Big Data Cloud, based on distributed instrumentations and dataflow-driven performance analysis. Based on this approach, we have implemented HiTune (a Hadoop performance analyzer), which provide valuable insights into the Hadoop runtime behaviors with every low overhead, no source code changes, very good scalability and extensibility. We also report our experience on how to use HiTune to efficiently conduct performance analysis and tuning for Hadoop, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches.

Reference

[1] D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, H. Liu. “Data warehousing and analytics infrastructure at facebook”. The 36th ACM SIGMOD International Conference on Management of Data, 2010.

[2] J. Dean, S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. The 6th Symposium on Operating Systems Design and Implementation, 2004.

[3] Hadoop. http://hadoop.apache.org/

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. “Dryad: Distributed Data-Parallel Programs from

Sequential Building Blocks". The 2nd European Conference on Computer Systems, 2007.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins. "Pig latin: a not-so-foreign language for data processing". The 34th ACM SIGMOD international conference on Management of data, 2008.

[6] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, N. Zhang. "Hive - A Petabyte Scale Data Warehousing Using Hadoop". The 26th IEEE International Conference on Data Engineering, 2010.

[7] Dryad. http://research.microsoft.com/en-us/projects/Dryad/

[8] "Hadoop and HBase at RIPE NCC". http://www.cloudera.com/blog/2010/11/hadoop-and-hbase-at-ripe-ncc/

[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, M. Stonebraker. "A comparison of approaches to large-scale data analysis". The 35th SIGMOD international conference on Management of data, 2009.

[10] S. L. Graham, P. B. Kessler, M. K. Mckusick. "Gprof: A call graph execution profiler". The 1982 ACM SIGPLAN Symposium on Compiler Construction, 1982.

[11] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/

[12] A. Rabkin, R. H. Katz. "Chukwa: A system for reliable large-scale log collection", Technical Report UCB/EECS-2010-25, UC Berkeley, 2010.

[13] Scribe. http://github.com/facebook/scribe

[14] Flume. https://github.com/cloudera/flume

[15] Java Instrumentation. http://download.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html

[16] Sort benchmark. http://sortbenchmark.org/

[17] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang. "The HiBench Benchmark suite: Characterization of the MapReduce-Based Data Analysis". IEEE 26th International Conference on Data Engineering Workshops, 2010.

[18] J. L. Hennessy, D. A. Patterson. "Computer Architecture: A Quantitative Approach". Morgan Kaufmann, 4th edition, 2006.

[19] Jeff Dean. "Designs, Lessons and Advice from Building Large Distributed Systems". The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.

[20] Jeffrey Dean. "Experiences with MapReduce, an abstraction for large-scale computation". The 15th International Conference on Parallel Architectures and Compilation Techniques, 2006.

[21] Ganglia. http://ganglia.sourceforge.net/

[22] A fair sharing job scheduler. https://issues.apache.org/jira/browse/HADOOP-3746

[23] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica. "Job Scheduling for Multi-User MapReduce Clusters". Technical Report UCB/EECS-2009-55, UC Berkeley, 2009.

[24] Hadoop Fair Scheduler Design Document. https://svn.apache.org/repos/asf/hadoop/mapreduce/trunk/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf

[25] Hadoop LZO patch. http://github.com/kevinweil/hadoop-lzo

[26] Hive performance benchmark. https://issues.apache.org/jira/browse/HIVE-396

[27] P. Barham, R. Isaacs, R. Mortier, D. Narayanan. "Magpie: online modelling and performance-aware systems". The 9th conference on Hot Topics in Operating Systems, 2003.

[28] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, I. Stoica. "X-trace: A pervasive network tracing framework". The 4th USENIX Symposium on Networked Systems Design & Implementation, 2007.

[29] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag. "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". Google Research, 2010.

[30] B. Chun, K. Chen, G. Lee, R. Katz, S. Shenker. "D3: Declarative Distributed Debugging". Technical Report UCB/EECS-2008-27, UC Berkeley, 208.

[31] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, R. Hundt. "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers". IEEE Micro (2010), pp. 65-79.

[32] "HPROF: a Heap/CPU Profiling Tool in J2SE 5.0". http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[33] Nagios. http://www.nagios.org/

[34] Cacti. http://www.cacti.net/

[35] V. Bhat, S. Gogate, M. Bhandarkar. "Hadoop Vaidya". Hadoop World 2009.

[36] J. Tan, X. Pan, S. Kavulya, R. Gandhi, P. Narasimhan. "Kahuna: Problem Diagnosis for MapReduce-Based Cloud Computing Environments". IEEE/IFIP Network Operations and Management Symposium (NOMS), 2010.

[37] G. Cretu, M. Budiu, M. Goldszmidt . "Hunting for problems with Artemis". First USENIX conference on Analysis of system logs, 2008.