Otus: Resource Attribution in Data-Intensive Clusters

Kai Ren
Carnegie Mellon University
kair@cs.cmu.edu

Julio López
Carnegie Mellon University
jclopez@cs.cmu.edu

Garth Gibson
Carnegie Mellon University
garth@cs.cmu.edu

ABSTRACT

Frameworks for large scale data-intensive applications, such as Hadoop and Dryad, have gained tremendous popularity. Understanding the resource requirements of these frameworks and the performance characteristics of distributed applications is inherently difficult. We present an approach, based on resource attribution, that aims at facilitating performance analyses of distributed data-intensive applications. This approach is embodied in Otus, a monitoring tool to attribute resource usage to jobs and services in Hadoop clusters. Otus collects and correlates performance metrics from distributed components and provides views that display time-series of these metrics filtered and aggregated using multiple criteria. Our evaluation shows that this approach can be deployed without incurring major overheads. Our experience with Otus in a production cluster suggests its effectiveness at helping users and cluster administrators with application performance analysis and troubleshooting.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging, Monitors*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Measurement, Management.

Keywords

Resource Attribution, Metrics Correlation, Data-Intensive Systems, Monitoring.

1. INTRODUCTION

Our modern world is data led. Practitioners from many fields, including science, business, government and academia, have turned to distributed data-intensive processing frameworks to deal with the data deluge. As a result, frameworks such as Hadoop, Dryad, Pig and Hive [1, 10, 14, 23] have gained tremendous popularity because they have enabled the analysis of massive amounts of data at unprecedented rates. However, understanding the resource requirements of these frameworks and the performance characteristics of the applications is inherently difficult due to the distributed nature and scale of the computing platform. In a typical data-intensive cluster installation, computations for user jobs are colocated with cluster services. For example, user tasks simultaneously run in the same hosts as the processes for distributed database and file system services. In addition, tasks from different user jobs may share the hosts at the same time. This mode of operation creates complex interactions and poses a considerable challenge for the understanding of resource utilization and application performance. In contrast, in typical HPC clusters, the storage and the computations run in separate hosts. Using a cluster job scheduler, the compute hosts are space-shared over coarse periods of time, i.e., a subset of the cluster's hosts are exclusively allocated to a job for a period of time. Application performance analysis in data-intensive systems is significantly harder because resources are not exclusively allocated to a job.

There are many implementations of cluster monitoring systems [5, 6, 11, 12, 15]. These tools collect OS generated metrics, such as CPU load, at host-level granularity and support extensions to monitor long-running processes such as a web server. As the collected metrics age, the data is thinned by lowering the resolution of older data through subsampling or averaging over time. The resulting resolution of the data is insufficient to analyze the performance of a job that executed sufficiently far in the past. In general, the readily available tools for analyzing the collected data are limited to the display of time series for individual metrics, and aggregated values for the entire cluster. These tools fail to provide the necessary information to answer the fundamental questions to understand application performance in data-intensive environments. These questions include among others: *How much of resource X is my application using through its execution? What is the (bottleneck) resource being exhausted by my application? What resources are in contention throughout my application's execution and what share of those resources is my application getting?*

We propose a simple *resource attribution* approach for monitoring and analyzing the behavior of applications and services in data-intensive clusters. Attributing the resource utilization to important components of interest is a key concept in performance analysis of computer systems. The techniques for resource attribution are embodied in a pro-

MapReduce'11, June 8, 2011, San Jose, California, USA.

totype monitoring system implementation named Otus. At a high-level, Otus uses metrics and events from different layers in the cluster software stack, from OS-level to high-level services such as HDFS, Hadoop MapReduce (MR), and Hbase [7, 2]. The data is correlated to infer the resource utilization for each service component and job process in the cluster. To effectively present these metrics to users, a flexible analysis and visualization system is provided to display the performance data using different selection and aggregation criteria. We have deployed our prototype on a 64-node production cluster that is used by researchers from fields such as computational biology, astrophysics, seismology, natural language processing and machine learning among others. Our initial experiences suggest that Otus can be a useful tool for helping users and cluster administrators with application performance analysis and troubleshooting.

This paper is structured as follows. In Section 2 we discuss goals and requirements for Otus. Section 3 describes the general architecture of monitoring systems and related work. Details about the implementation of Otus and the techniques used for attributing resource utilization are presented in Section 4. We present several case studies to illustrate how Otus can help users analyze the runtime behavior of their applications and troubleshoot performance problems (Section 5). We evaluate the overhead of running Otus in a production cluster and show that it imposes low per-node overheads (Section 6). Section 7 discusses how the system can be improved and future research directions.

2. ENABLING RESOURCE ATTRIBUTION

Our main goal is to help users, application programmers and system administrators gain insight about the performance of the applications and the distributed execution platform. The main approach used in Otus is to attribute resource consumption to user jobs and distributed cluster services by correlating and aggregating detailed information for individual processes running as part of those jobs or services. The following building blocks are required to provide clear, useful information about resource utilization that can lead to better understanding of application performance.

Collection of fine-grained resource utilization data: The collection of resource usage metrics needs to be performed at a granularity that is finer than host-wide metrics and include data from various levels of the software stack. Of particular importance is the data that helps distinguish the interaction between components and the logical relationships between services and user job processes. The metrics need to be collected for select processes on a per-process granularity, at time scales of 1–10 seconds, and need to be labeled such that they can be later identified and aggregated according to user job or system service.

Simple and flexible visual analysis: Collecting detailed resource utilization is not enough to explain the observed performance of an application. To be effective, as more data is collected, it is important to summarize and display information in a simple and easy to understand manner, using different filtering and aggregation criteria depending on the analysis. The Otus interface offers a set of views that display the data in time series graphs with various levels of aggregation and filtered by criteria such as Hadoop job id. Users can issue custom queries to specify aggregation criteria other than the pre-defined ones (see Section 4).

Scalable and efficient storage of historical data: The collected data needs to be displayed on-line as well as stored persistently for later analysis. For example, it is desirable to analyze the performance of a job that finished execution in the previous hour and compare it to a job that executed the previous week. Monitoring large scale cluster at fine granularity will produce large amounts of data. Storing and querying this data requires a scalable storage back-end. The system must scale performance-wise in order to support both on-line monitoring and flexible performance analysis. It must be efficient in terms of the size needed to represent the data and the computational resources needed to process queries.

Our initial Otus prototype focuses on the top two requirements. We present results for two storage back-end approaches. We have started exploring alternatives for storing the metrics in an efficient and scalable way.

3. BACKGROUND

Contemporary monitoring systems (e.g. [12], [15], [8], [11], [19]) for large scale clusters share many similarities in their architecture. The techniques used in Otus can be implemented by extending these systems and leveraging their overall architecture for transporting the metrics data in large clusters.

3.1 Monitoring System Architecture

The general architecture of today's monitoring systems consists of four components as shown in Fig. 1: Collector, aggregator, storage and visualizer. The *collector* is a background process running on each cluster host. It periodically reads performance metric values, e.g. number of disk I/O operations, and passes those to an aggregator. The *aggregator* processes metrics data received from multiple collectors, then it decides where and how to store the data. In some systems, the aggregator performs data thinning and aggregation of metrics. The *storage back-end* provides a central repository for storing and querying the metrics received by the aggregators. Its performance needs to scale up as the number of hosts, metrics and queries increases. The *visualizer* provides a graphical interface for users to query and analyze the metrics collected by the system. To illustrate

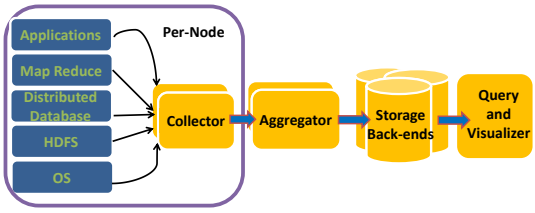

Figure 1: Architecture of monitoring systems

the basic data flow, consider the scenario of collecting the total number of bytes read from disk across the cluster hosts. In each node, the collector process obtains the I/O utilization data (i.e., number of bytes read and written) from the Linux `/proc` file system and sends the newest value to the aggregator. The aggregator sums the values received from all the collectors, and then writes the aggregate and individual values to the storage back-end. Through the visualizer, users can then get time-series graphs of the aggregate read bandwidth for their clusters.

3.2 Related Work

There is a large body of work in the area of monitoring of distributed system. Some approaches focus primarily on building scalable monitoring infrastructures, while others aim at pinpointing faults and diagnosing performance problems.

Ganglia [12], Collectd [5] and Munin [24], focus on collecting host-wide performance metrics, that is, the metrics are summed for all running processes on a node. RRDTool [13] has been used in these systems to bound storage space and generate time-series graphs for their user interfaces. They aid system administrators in performing tasks such as identifying loaded hosts or services. However, they do not provide enough support for more detailed application performance analysis. DataGarage [11] and OpenTSDB [19] aim at warehousing time-series data for large scale clusters by using a distributed storage system. Chukwa [15], Flume [8] and Scribe [17] are scalable monitoring systems that collect log-based data. These systems implement features for enabling scalable data gathering and transport. Otus leverages the infrastructure provided by these systems and extends it to enable resource attribution.

Path-tracing tools (e.g., Dapper [18], Xtrace [9], Stardust [22] and Spectroscope [16]), collect and analyze fined-grained event traces from distributed system. By comparing performance metrics in different traces, these tools can help to isolate problematic components in the system. Chopstix [3] collects low-level OS events continuously (e.g L2 cache misses) at the granularity of procedures and instructions, and combines human experience to do troubleshooting. Xu et al. [25] parses logs to create features and use machine learning techniques to detect abnormal events in the cluster. Specialized debugging tools for data-intensive system have been recently developed. Artemis [6] collects metrics for each task in Dryad system [10], and provides algorithms to detect outliers automatically. Tan et al. [21, 20] proposed a series of visualization tools to analyze Hadoop logs. These tools mainly focus on off-line debugging and analysis of logs, a function that is complementary to Otus.

4. IMPLEMENTATION

The main goal of Otus is to aid users in understanding the performance of their applications by providing useful resource usage information for their jobs. The resource attribution and metric correlation techniques embodied in Otus are implemented by extending the basic components of the monitoring system architecture described in Section 3.1. Below we explain the mechanisms employed in Otus to associate resource usage with jobs and cluster services.

4.1 Data Collector

The role of data collector is to retrieve necessary metric data that enables the analysis of resource utilization of applications in the cluster. The current prototype of Otus focuses on metric data about detailed resource utilization information of key processes such as task tracker, data node and MapReduce tasks. It collects the resource usage (like CPU, memory, etc) of each process from the Linux /proc file system. To associate resource usage on a process level to higher level entities such as MapReduce tasks and users, we instrument higher-level system to record additional metadata to these resource utilization data.

The data collector is implemented as a plug-in to a Collectd daemon. It is periodically invoked by the Collectd daemon, and scans /proc to retrieve metric data. In /proc, each process has a corresponding directory where the OS kernel exposes process performance counters such as CPU usage, memory usage, and I/O statistics. In addition, Otus extracts metadata information from a per-process file in /proc named *cmdline*, which contains the command line arguments passed to the process. From these arguments, Otus can identify the software layer to which the process belongs, e.g., MapReduce task, task tracker, HDFS, HBase, etc. For MapReduce tasks, Otus also extracts the corresponding task ID from the arguments, which is used to create the mapping between process ID and Hadoop jobs.

For frameworks like MapReduce, the collector performs local aggregations, such as summing the values for a process sub-tree that belongs to a single MapReduce task, and only sends the aggregated value to the aggregator. For example, in MapReduce system, local aggregation is enabled for jobs whose mapper/reducer tasks will spawn many child processes. Because users may be only interested in understanding the task process as a whole instead of each child process, local aggregation can reduce the amount of data that is sent over the network.

Another special case is the reuse of Java Virtual Machine (JVM) by the Hadoop framework. To avoid the overhead of JVM startup, Hadoop implements a feature that allows the JVMs from finished tasks to be used by new tasks of the same job. This causes a single JVM process ID to correspond to multiple MapReduce tasks. The correct task ID cannot be obtained by merely reading the process's cmdline file in /proc. A solution for this case is to instrument the per-node task tracker process to export the needed information so the collector can obtain the current mapping between task IDs and JVM processes.

All metrics data collected by the data collector are in the form of time-series values represented as tuples of the form <Local Time Stamp, Software Name, Metric Name, Value>. The aggregator can categorize incoming metrics based on these tags.

4.2 Aggregator

The metrics collected by Otus on a node are periodically transferred to aggregators where they are categorized, aggregated and written to a storage back-end. Aggregators have different ways of summarizing incoming data according to metric types. For metrics related to MapReduce jobs, since they are associated with task processes, the aggregator adds up the metric values over all tasks belonging to a job. This type of aggregation gives the resource usage of an entire MapReduce job. Another type of aggregation is to sample the raw metrics data at fixed-sized time-scales such as one hour, one day and etc. The data points over a fixed range of time are coalesced into a single data point. This aggregation reduces the number of data points transferred when querying data in a long time range at coarse granularity.

4.3 Storage Back-ends

In our implementation, we explored two different storage back-ends: One that uses RRDTool and another one that uses a MySQL database for storing the metrics. RRDTool is a lightweight database that stores data into a circular buffer, in a so called RRD file. Therefore, the system's stor-

age footprint remains constant over time. Another advantage of RRDTool is its simple interface and visualization tool kit. However, RRDTool has scalability problems when the number of metrics and nodes increases. A large number of metrics updates causes the RRDTool storage back-end to generate lots of small writes to disk and results in bad I/O performance. That is why we tried a MySQL database as an alternative storage back-end to increase the scalability of the monitoring system. Below, we describe the implementation of the storage back-ends using RRDTool and MySQL respectively.

RRDTool as storage back-end

RRDTool is designed to store multiple time-series variables in a single RRD file. To make the data schema simple and flexible, Otus only stores one metric per RRD file. The same strategy is used by Collectd and Ganglia.

We categorize the collected metrics into two types and use a RRD schema depending on the type as illustrated in Figure 2. Type 1 corresponds to system-wide long-term metrics that are collected all the time. They include host-level resource utilization metrics, e.g., CPU, memory usage, etc., and metrics of long-running processes, e.g., HDFS's data node daemons, Hadoop's task trackers, HBase's tablet servers, etc. Type 2 metrics correspond to the resource utilization of MapReduce task processes. MapReduce task processes only run for a limited amount of time, and each MapReduce job has different numbers of task processes. Thus, Type 2 are short-term metrics and their number cannot be pre-determined.

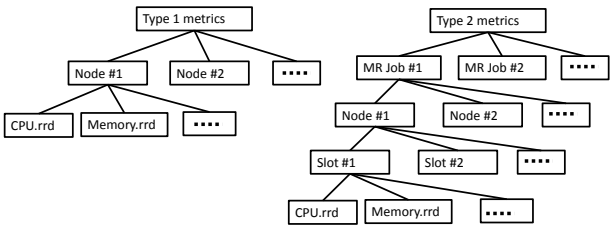

Figure 2: Data schema of RRDTool

All Type 1 metrics are stored in a central directory using the same approach as the base Collectd RRD back-end implementation. Under the central directory, a sub-directory per-host is used to store the metric data for a particular cluster node. Each metric is stored as a single RRD file in the sub-directory corresponding to the source host. Type 2 metrics are handled in a different manner as follows. The RRD files are grouped into directories by MapReduce job ID. Each MapReduce job has its own job directory, which contains sub-directories that are named by node IDs. In a Hadoop MapReduce system, there is a limit on the number of tasks that can concurrently execute in each node. To reduce the number of RRD files, we map MapReduce tasks into a fixed number of slots in every node. Thus each node subdirectory contains RRD files named by slot ID. Resource utilization information of tasks running in the same slot will be stored into the same RRD file.

When a new RRD file is created, RRDTool requires that the circular buffer size be specified. However, we cannot estimate the length of the appropriate RRD file for MapReduce tasks. Otus pre-allocates a delibrately large RRD file to store metrics for each task slot. If the actual length of the job is shorter than the pre-allocated value, additional disk space is reclaimed after the MapReduce job finishes, by coping the metric data to a new RRD file with the appropriage length according to the job's duration. Then, the original pre-allocated RRD files are deleted.

MySQL as storage back-end

Similar to the RRD back-end, the MySQL back-end uses two different schemas to store the metrics according to their type. For Type 1 metrics, each node has a single table to store metrics related to that host. Since the number of metrics can be known in advance, we use a "wide table" schema, as follows: $< TimeStamp, Metric_1, \ldots, Metric_n >$. For Type 2 metrics, each MapReduce job has a table to store metrics. The schema for these tables is the following: $< TimeStamp, TaskID, MetricName, MetricValue >$. This schema is different from the schema used in the RRD back-end, and allows metric data for a large number of tasks to be combined into a single "narrow" table. To reduce the required storage, we assign an integer ID to each type of metric instead of a string with the explicit metric name.

4.4 Visualizer

The goal of graphically displaying the metrics data is to aid with gaining insight about how jobs and services use the available cluster resources. The Otus visualizer is web-based and implemented in PHP. In the RRD version, RRDTool returns query results as graphs that are displayed by the web browser. In the MySQL version, a query result is a list of data, which is rendered by the browser using JavaScript. The Otus user interface provides two pre-defined views that show performance metrics at different levels: node view and group view. Users are also able to issue customized queries to the storage back-end to get metrics data.

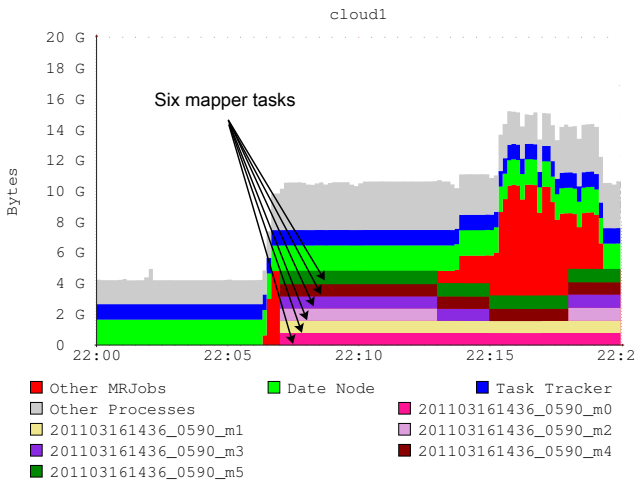

Figure 3: Memory usage decomposed in one node

A node view can show resource utilization of selected processes for a particular node in the cluster. Figure 3 shows an example of a node view for a particular MapReduce job. This view shows the virtual memory usage of six mapper task slots over a period of time. Each slot corresponds to multiple mapper tasks and is labeled with different colors.

We can infer from the figure that most of the mapper tasks consumed about 700MB of virtual memory. It also shows the virtual memory of other long-running processes like the task tracker and HDFS data node. As all metrics are gathered from every node, a node view provides the most fine-grained view of data.

In contrast, group views display aggregate metrics for a group of nodes, so users can understand the system behavior quickly without looking at every node in a large cluster. A group view aggregates a metric by summing the values for the selected group of nodes. Figure 4 shows a sample group view. We can infer from this figure that MapReduce job 201101170028_0332 dominated the memory usage for over a 5-minute period, and the peak memory usage reached 35 percent of the total memory available in the cluster.

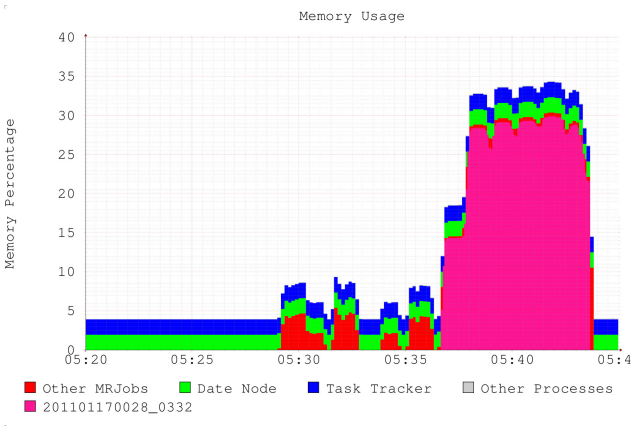

Figure 4: Memory usage of the whole cluster

5. CASE STUDIES

We deployed Otus in a production cluster at CMU. The cluster is a Hadoop cluster that runs MapReduce jobs used by researchers from fields such as computational biology, machine learning and astrophysics among others. We have been using Otus to diagnose system and application performance problems that arise in the cluster. Below are three cases that illustrate how visual correlation of resource metrics help users understand application performance and infer problem causation.

5.1 Resource Exhaustion in Cluster Hosts

Detailed resource usage data is useful for diagnosing the causes of abnormal conditions such as resource exhaustion. For example, during the course of cluster operations, a number of nodes crashed after several MapReduce jobs executed. The nodes were soon rebooted and put back into service. A few hours later the problem recurred on different nodes. In conjunction with the system administrators, we used Otus to examine the historical resource utilization of the crashed nodes and found that their CPU utilization was moderate prior to the crashes. However, they showed low levels of free memory. Otus allowed us to dive deeper and attribute the memory utilization to specific processes as shown in Fig. 5. A large portion of the physical memory was consumed by unknown processes that did not belong to the running MapReduce jobs or services.

Then, we queried Otus to retrieve a list of live nodes that had similar memory usage patterns. The system administrator logged into a few of these nodes and found out that the memory was being consumed by orphan Python processes. These processes were spawned by Hadoop streaming jobs[1], but were not properly cleaned up by the framework when the respective jobs were killed. Otus classified the stray processes as "other" since they were orphan and could not be traced back to specific MapReduce jobs. The orphan processes accumulated over time. Although they did not use the CPU, they consumed valuable memory until eventually very little memory was available to run other jobs.

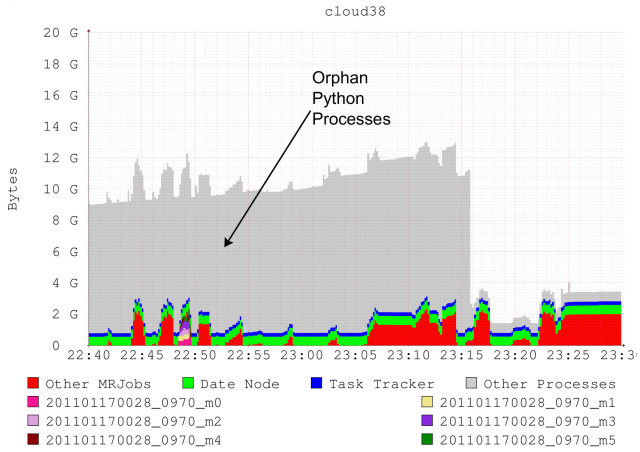

Figure 5: Abnormal resource utilization

5.2 Hadoop Memory Management

Memory management, as an important part of resource control in Hadoop MapReduce system, is designed to prevent multiple MapReduce tasks in the same node from competing for memory resources. In Hadoop MapReduce system, users and cluster administrators can set three parameters to limit the memory usage of MapReduce jobs:

- *Process maximum memory*: This parameter limits memory used by each task process, and every child process it spawns.
- *Task maximum memory*: It limits the total amount of virtual memory and physical memory used by the entire process tree for a task. The process tree includes all processes launched by the task or its children, (e.g in streaming jobs).
- *Node maximum memory*: This limits the total virtual and physical memory of all map-reduce tasks and their children processes in each node of the cluster.

The first two parameters mentioned above are user definable. The users can use them to specify the memory requirement of their jobs. The last one is specified by the cluster administrator.

When Hadoop schedules MapReduce jobs, it monitors the memory usage of each mapper/reducer task. If any of the

[1]Hadoop streaming [1] is a mode of operation that allows users to execute a script as part of a MapReduce job.

aforementioned memory thresholds is exceeded, the corresponding task is killed. This mechanism makes it necessary for users to appropriately specify memory limit parameters for their jobs. Improperly setting these parameters results in jobs failing due to *out of memory* (OOM) errors. Often, users cannot accurately estimate the memory requirements for their MapReduce programs due to the large number of factors that affect memory consumption, e.g., input size, number of records, dynamic allocation of small structures, memory allocated by the framework, etc. In lieu of useful memory usage information, users commonly overestimate the memory limit creating too high of a demand for memory resources in the cluster hosts. Often, as a side effect of this situation. task trackers launch fewer mapper/reducer tasks per node, increasing job running time and wait time for jobs in the queue. Users benefit in multiple ways from the detailed resource usage data available through Otus.

- Monitor the memory utilization of MapReduce jobs continously to help users set the correct memory parameters in future runs.
- Understand "Out of Memory" failures. The memory metrics in Otus can differentiate between the following causes: 1) *killed by MapReduce framework:* because the total memory of one process or the total process tree exceeds the limit; 2) *killed by the Operating System:* because processes in the node used up the available physical memory and swap space. To infer failure causes, one can get information from Otus about the memory usage of killed processes. If the memory usage of the killed process exceeds one of the above limits, the process may have been killed by the MapReduce framework. Otherwise, if the total memory consumed by all processes exceeds the size of physical or virtual memory of the node, the OS may have killed one of the processes.

Memory Stress Benchmark

The memory stress benchmark consists of a MapReduce program that stress tests the memory management in Hadoop. We use this benchmark to test the memory limit parameters for jobs and verify the behavior of memory management specified in the Hadoop documentation. We also use this benchmark to recreate job failures related to memory issues. The benchmark program executes as a map-only job. Each map task dynamically allocates two-dimensional arrays and continuosly traverses the arrays to prevent the OS from paging out the allocated memory. Users can specify the task duration as well as the size of the array according to the memory parameters in their cluster.

Discovering Pitfalls in JVM Reuse

While running the memory stress benchmark in our cluster testbed, we discovered a memory starvation situation caused by Hadoop's JVM reuse feature, resulting in high failure rates for memory intensive jobs. In our memory stress experiment, a particular memory-intensive job required 4 GB of virtual memory per task. The maximum memory per node parameter was set to be 12 GB. In this case the framework allows simultaneous execution of up to three of those tasks. Figure 6 shows the memory used by Hadoop tasks in one of the cluster hosts. Initially, only three tasks were executing as expected. After the initial tasks finished, a new

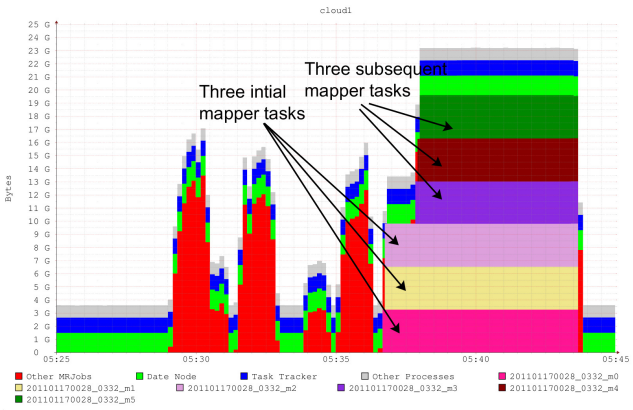

Figure 6: Potential pitfall in memory management

set of JVMs was spawned for new tasks. The original JVMs were not immediately reused nor terminated, and remained on standby. All these JVMs consumed the available physical memory and triggered process failures as they could not allocate memory. The display of detailed memory usage information was instrumental to track down this problem.

5.3 Resource Usage in Cloud Databases

One important feature of Otus is to account the resource usage of different components of the distributed system, and correlate the variance of resource utilization with system behaviors. During the course of benchmarking experiments for distributed cloud database systems, we encountered a performance degradation in one of the benchmarked systems. The benchmark program continuously inserted and updated rows in a table. After performing these operations for a period of time, the performance of the system degraded resulting in lower query throughput as the benchmark progressed.

The benchmarked system was a new Java implementation of BigTable [4] built on top of HDFS [1]. A tablet server executes on each cluster host and manages disjoint partitions of the tables stored in the database. Each tablet server maintains an in-process memory cache for recently accessed records, and writes dirty records to data files when there is memory pressure. Periodically, the system also performs *major compactions* to merge small data files into larger ones.

We used Otus to analyze the database's resource utilization in two continuous rounds of the benchmarking experiments. These experiments were carried out in a small subset of our cluster consisting of six tablet server hosts and a separate host for the master processes. Figures 7 and 8 show two types of resource utilization under the group view. Figure 7 shows the aggregated disk I/O throughput for the HDFS data server processes where the tablet servers were executing. The timeline (X axis) corresponds to two consecutive experiments. We noticed that the read throughput gradually dropped, which corresponds to the decrease in query throughput. There was no write activity during the experiments, the writes only occurred at the end of the experiments after manually triggering major compactions. Figure 8 shows the aggregate memory utilized by different components of the system, e.g., HDFS data server, tablet server and logger. The table servers consumed the largest portion of the memory available on the testbed hosts. We formulated a hypothesis that the degradation was caused by the

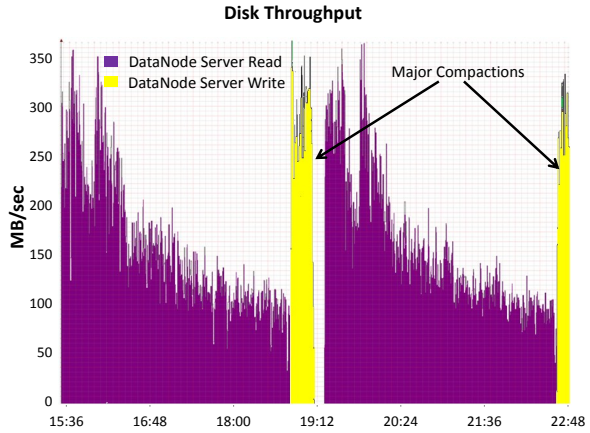

Figure 7: Aggregated disk I/O throughput

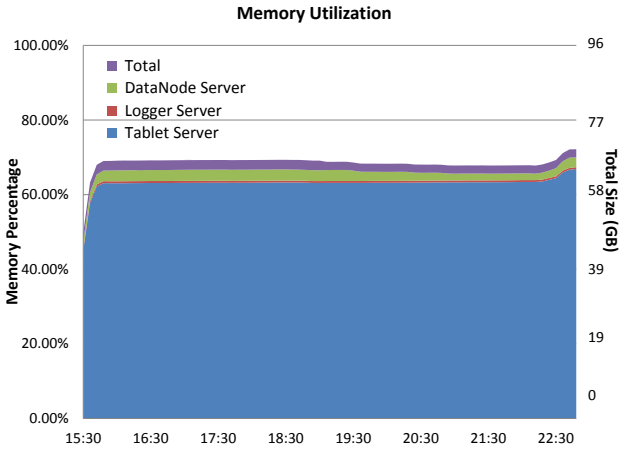

Figure 8: Aggregated memory utilization

growth and fragmentation of a data structure in memory. This hypothesis was later confirmed by the developers.

5.4 Case Studies Summary

Our initial experience suggests that Otus is useful for application performance analysis and troubleshooting. By using resource utilization graphs generated with Otus, we are able to narrow down the number of potential sources of performance problems and rule out potential causes such as system misconfiguration. Then, we can combine data from other other tools to find the actual cause.

6. EVALUATION

We ran a set of experiments in our production cluster. Each node of the cluster runs a collector that scans /proc and extracts metrics every five seconds. The collectors send the data to the aggregator every ten seconds. Since the size of our cluster is sufficiently small, the aggregator, storage back-end and web front-end all run in a single monitoring node. We also use Otus to monitor its own resource consumption during the experiment.

The production cluster, where we run experiments, consists of 64 nodes, each containing two quad-core 2.83 GHz Xeon processors, 16 GB of memory, and four 7200 RPM 1 TB Seagate Barracuda ES SATA disks. Nodes are interconnected by 10 Gigabit Ethernet. The monitoring node has a 2 GHz Xeon processor, 2 GB of memory, and two 7200 RPM 1 TB Seagate Barracuda SATA disks with software RAID 1. All nodes run the Linux 2.6.32 kernel.

In our evaluation of data collectors, the baseline is the Collectd [5] daemon without the Otus plug-in that collects fine-grained resource utilization metrics from a Hadoop system. We use two workloads: idle and busy. In the idle workload no MapReduce jobs run in the cluster. In the busy workload, multiple jobs use up all the available task slots in the compute hosts. Since the complexity of monitoring a Hadoop system is proportional to the number of MapReduce jobs and service processes, the busy workload represents the heaviest load for our monitoring system. In the busy workload, there are over 80 performance metrics per host.

	CPU	PhyMem	VMem	Network
No Plug-in (Idle)	0.28%	776 KB	117 MB	83 B/s
No Plug-in (Busy)	0.33%	902 KB	121 MB	85 B/s
With Plug-in (Idle)	0.85%	29.9 MB	155 MB	274 B/s
With Plug-in (Busy)	1.29%	34.5 MB	165 MB	571 B/s

Table 1: Average resource utilization for collectors

Table 1 shows the resource requirements for the collectors running on each compute node. The first column (CPU) shows the CPU utilization percentage. The *PhyMem* column contains the collector's average resident memory size, and *VMem* denotes the virtual memory size. The *Network* column has the average network bandwidth consumed by the collector, not accounting for packet headers, when sending the data to the aggregator. The data shows that the CPU utilization is small, and overall the resource requirements are modest for current server-class hosts. Running the Otus plug-in incurs a minor memory overhead because collectd needs to execute the Python interpreter to run the plug-in. This requirement can be alleviated by implementing the Otus functionality as a C plug-in.

		CPU (%)	PhyM (MB)	VirtM (MB)	Disk Writes (KB/s)	Disk Writes (ops)
Idle	RRD	44	7	261	66	47
	MySQL	52	313	1160	54	31
Busy	RRD	50	14	293	144	1097
	MySQL	75	369	1162	180	63

Table 2: Aggregator resource utilization

For the aggregator resource utilization, we compared two implementations: one with a RRD storage back-end, and the other one with a MySQL back-end. Table 2 shows the aggregator's resource utilization on the monitoring node for each implementation. The data for the Idle and Busy scenarios is shown for both implementations. The last two columns show the disk write activity in bytes written/s and number of writes/s. Notice that the RRD implementation generates a large number of writes during busy periods. The MySQL back-end successfully reduces the number of write operations through write-ahead logging. However, the MySQL version consumes much more CPU and memory resources than the RRD one. The high memory usage is caused by the MySQL

back-end's large memory buffer. The high CPU usage, we speculate, is an implementation artifact of our aggregator's use of Python rather than the RRDTool's use of C. We believe the performance of the MySQL version can be greatly improved.

	Size MB (Factor)	
	Type 1	Type 2
RRD	3 MB	8 MB
MySQL	3.5 MB (1.2 X)	31 MB (3.9 X)

Table 3: Daily storage requirement per host

Another evaluation of the two storage back-ends is to compare their in-disk storage cost. As mentioned in Section 4.3, metrics of type one are related to long running processes, and metrics of type two are related to MapReduce task processes. The total storage space needed for storing metrics is proportional to the number of hosts in the cluster. Table 3 contains the storage requirements per monitored host per day. In this case, twenty four metrics of type one and sixty metrics of type two are stored in the storage back-ends. The MySQL back-end has a much higher storage requirement for metrics of type two, up to 3.9 times that of the RRD back-end. This is due to the "narrow" table used in the MySQL back-end to store type 2 metrics, which has an index on the time stamp and explicitly includes a metric id in each record. In contrast, in the RRD back-end there is no index on the time stamp field, and the metric id is not explicitly stored in each record, but instead encoded in the RRD metadata.

Overall, the evaluation shows that it is feasible to implement, with relatively little overhead, the mechanisms that are needed to perform resource attribution. However, in order to deploy these technique in larger clusters, the storage back-end needs to scale up.

7. FUTURE WORK

We intend to improve the scalability of our system by adopting OpenTSDB [19], an open source scalable database for time-series data built atop HBase [2]. With data stored in HBase, we can also leverage Hive as a query engine on top of HBase. This could further enhance functionality by flexibly allowing users to analyze their performance data and search for root causes of performance problems.

We are also collaborating with a team that specializes on automatic strategies to detect abnormal behaviors and performance problems in large clusters [20, 21]. We plan to develop new strategies that take advantage of the richer resource metrics collected by Otus.

8. CONCLUSION

Understanding performance and resource requirements of data-intensive applications is increasingly hard due to the complex interactions between distributed system components. Otus is a monitoring system that helps address this challenge by attributing resource utilization to jobs and services executing in a cluster. It is feasible to use Otus in production clusters without introducing significant overheads. Our experience deploying Otus in a production cluster shows its utility for troubleshooting and monitoring data-intensive clusters.

Acknowledgements

The work in this paper is based on research supported in part by the National Science Foundation, under award CCF-1019104, the Qatar National Fund, under award NPRP 09-1114-1-172 in the QCloud project, and the Gordon and Betty Moore Foundation, in the eScience project. We also thank the member companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

9. REFERENCES

[1] Apache (ASF). Hadoop MapReduce and HDFS. http://hadoop.apache.org/core.
[2] Apache (ASF). HBase. http://hbase.apache.org.
[3] Sapan Bhatia et al. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI*, 2008.
[4] Fay Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[5] Collectd: The system statistics collection daemon. http://collectd.org.
[6] Gabriela F. Cretu-Ciocarlie et al. Hunting for problems with Artemis. In *USENIX WASL*, 2008.
[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[8] Flume. https://github.com/cloudera/flume.
[9] Rodrigo Fonseca et al. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.
[10] M. Isard and M. Budiu. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.
[11] Charles Loboz et al. DataGarage: Warehousing massive performance data on commodity servers. *VLDB*, 2010.
[12] Matthew L. Massie et al. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 2003.
[13] Tobi Oetiker. RRDTool. http://www.mrtg.org/rrdtool.
[14] Christopher Olston et al. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.
[15] Ariel Rabkin and Randy H. Katz. Chukwa: A large-scale monitoring system. In *Cloud Computing and Its Applications (CCA)*, 2008.
[16] Raja R. Sambasivan et al. Diagnosing performance changes by comparing system behaviours. In *NSDI*, 2011.
[17] Scribe. https://github.com/facebook/scribe.
[18] B. Sigelman et al. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.
[19] StumbleUpon. OpenTSDB: Open time series database. http://opentsdb.net.
[20] Tan et al. SALSA: Analysing logs as state machines. In *USENIX Workshop on Analysis of System Logs*, 2008.
[21] Tan et al. Mochi: Visual log-analysis based tools for debugging Hadoop. In *HotCloud*, 2009.
[22] Eno Thereska et al. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS*, 2006.
[23] Ashish Thusoo et al. Hive: A petabyte scale data warehouse using hadoop. In *ICDE*, 2010.
[24] v. G. Pohl and M. Renner. *Munin: Graphisches Netzwerk-und System-Monitoring.* opensource press, 2008.
[25] Wei Xu et al. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.