Modeling of Distributed File Systems for Practical Performance Analysis

Yongwei Wu, *Member*, *IEEE*, Feng Ye, Kang Chen, and Weimin Zheng, *Member*, *IEEE*

Abstract—Cloud computing has received significant attention recently. Delivering quality guaranteed services in clouds is highly desired. Distributed file systems (DFSs) are the key component of any cloud-scale data processing middleware. Evaluating the performance of DFSs is accordingly very important. To avoid cost for late life cycle performance fixes and architectural redesign, providing performance analysis before the deployment of DFSs is also particularly important. In this paper, we propose a systematic and practical performance analysis framework, driven by architecture and design models for defining the structure and behavior of typical master/slave DFSs. We put forward a configuration guideline for specifications of configuration alternatives of such DFSs, and a practical approach for both qualitatively and quantitatively performance analysis of DFSs with various configuration settings in a systematic way. What distinguish our approach from others is that 1) most of existing works rely on performance measurements under a variety of workloads/strategies, comparing with other DFSs or running application programs, but our approach is based on architecture and design level models and systematically derived performance models; 2) our approach is able to both qualitatively and quantitatively evaluate the performance of DFSs; and 3) our approach not only can evaluate the overall performance of a DFS but also its components and individual steps. We demonstrate the effectiveness of our approach by evaluating Hadoop distributed file system (HDFS). A series of real-world experiments on EC2 (Amazon Elastic Compute Cloud), Tansuo and Inspur Clusters, were conducted to qualitatively evaluate the effectiveness of our approach. We also performed a set of experiments of HDFS on EC2 to quantitatively analyze the performance and limitation of the metadata server of DFSs. Results show that our approach can achieve sufficient performance analysis. Similarly, the proposed approach could be also applied to evaluate other DFSs such as MooseFS, GFS, and zFS.

Index Terms—Distributed file system, architecture model, practical performance analysis, HDFS

1 Introduction

Data-intensive DFSs are any file system that allows multiple users to access to files distributed on multiple machines via a computer network, for the purpose of sharing files and storage resources [22]. DFSs are emerging as a key component of large-scale cloud computing platforms. Applications on such computing paradigms come with increasing challenges on how to transfer and where to store and compute data reliably and efficiently. Specifically, these challenges include data transfer bottlenecks, performance unpredictability, scalable storage and so on. To deal with these challenges, various DFSs such as Hadoop distributed file system (HDFS) [2], the Google file system (GFS) [11], MooseFS [3], and zFS [21], have been developed for large-scale distributed systems such as Facebook and Google.

Performance analysis is an important concern in the distributed system research area. Researchers have made a lot effort to evaluate, model, and analyze distributed systems for computing intensive or data intensive applications. There exist well-known evaluation benchmarks (e.g., LINPACK, mpiBLAST) for computing paradigms. However, similar kinds of widely accepted benchmarks are rarely seen in DFSs. For example, there is no specific benchmark proposed for evaluating DFSs (e.g., HDFS, GFS) for web services. Some other related works evaluate performance of DFSs by comparing them with similar kinds of DFSs (e.g., NFS) via running in-house benchmarks or application programs (e.g., [13]). Some researchers measured the performance of DFSs under a variety of workloads and strategies (e.g., [27], [15]).

In the field of evaluating the performance of DFSs, a typical approach taken is through experiments by running DFSs; therefore, it is mainly based on the analysis of the experiment results (e.g., [28]), or draw conclusion by comparing with existing DFSs (e.g., [13], [23]). Therefore, there rarely exist approaches that are capable of qualitatively and quantitatively analyzing the overall performance of a DFS or its individual step or component, prior to the deployment of the DFS, without running benchmarks or particularly designed or selected applications. In this paper, we propose such an approach that is driven by architecture and design models of DFSs.

System architects can use design-time performance to evaluate the resource utilization, throughput, and timing behavior of a system prior to the deployment due to the following reasons: 1) analyzing performance of the system is much less expensive than testing the performance of the system by running it, 2) it is simply infeasible to test all kinds of different configurations of the system by running it, and 3) performance analysis on models helps architects make configuration and deployment decisions to avoid costly redesign, reconfiguration or redeployment.

• *The authors are with the Department of Computer Science and technology, Tsinghua University, Qinghuayuan 1#, Haidian District, Beijing 100084, China and the Research Institute of Tsinghua University in Shenzhen, China. E-mail: wuyw@tsinghua.edu.cn.*

Manuscript received 9 Nov. 2012; revised 20 Dec. 2012; accepted 21 Dec. 2012; published online 11 Jan. 2013.
Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-11-1141.
Digital Object Identifier no. 10.1109/TPDS.2013.19.

Some model-driven performance analysis and prediction (MDPAP) approaches (e.g., [10], [19]) have been proposed in the literature and especially Balsamo et al. conducted a survey on MDPAP [9]. The survey results reveal that 1) most approaches make use of unified modeling language (UML) [18] or UML-like formalisms to describe behavioral models, 2) few approaches provide direct correspondence between the software specification abstraction and the performance model evaluation results, and 3) the performance model should be easy to apply in practice. The key challenge of MDPAP approaches is finding the right architecture and performance abstraction of the system under study.

Based on the above study, in this paper, we propose a practical performance analysis methodology particularly for DFSs. The approach is based on the UML specification of the DFS architecture and their key behaviors (see Section 2). Some elements of the architecture model are also characterized by some stereotypes from the MARTE profile [6], which is a UML profile for modeling and analysis of real-time and embedded systems. We define the following characteristics of our methodology:

1. DFS architecture and design models provide a common understanding of DFS, which is considered crucial as DFS practices lack of such a common knowledge base.
2. Based on the models, one can systematically and automatically derive configurable parameters. Without them, this activity will heavily rely on expert tacit knowledge, inevitably leading to the low-quality management of the process.
3. Configuring DFS systems is typically the first and most important step to set up experiments. Therefore, our methodology provides a way to design experiments whose results directly contribute to performance analysis.
4. Qualitative and quantitative performance analysis can be conducted, based on the models, system configurations and experiment results. Analysis results can be also interpreted based on the architecture and design models, therefore making the architecture and design refinement much easier.
5. Based on the models, both the overall system performance and individual component or execution step performance can be analyzed. This is because the message interactions and relationships between components are clearly specified in the architecture and design models.

As one popular master/slave structured DFS, HDFS is selected as the representative for evaluation. Three sets of real-world experiments were conducted to qualitatively assess the effectiveness of our performance analysis approach. We also conducted a set of experiments of HDFS on EC2 to quantitatively analyze the memory and CPU bottlenecks of the metadata server of HDFS and formulate the response time of the Read operation of the metadata server to client requests. Results show that our approach can achieve an acceptable level of qualitative and quantitative performance analysis.

In Section 2, we first present the architecture and design models we built for DFSs, based on which, we propose a configuration guideline (see Section 2.3) and the performance modeling and analysis approach (see Section 3). In Section 4, we report how we evaluated our approach and the evaluation results. The related work is provided in Section 5. We conclude the paper in Section 6.

2 Architecture, Design Modeling, and Performance-Relevant Configuration

The architecture and design models we present in this section are general for typical DFSs, but we use the terminologies from HDFS [2] (e.g., namenode, datanode) to discuss the common concepts of DFSs.

2.1 Structural Modeling

The structural model consists of three models: data model, node specification model, and application to node deployment model.

The data model specifies the structure of the software deployed to DFS nodes as a class diagram. It mainly contains classes `DataNode`, `NameNode`, `Client`, `File`, and `Block`. Classes `Client`, `DataNode`, and `NameNode` correspond to software deployed to client workstations, datanodes and the namenode (i.e., metadata server), respectively. `File` and `Block` are two important DFS concepts and are entities needed to be consistent with the behavioral model. Note that the software classes in the data model are the abstraction of the software deployed to DFS nodes for the purpose of capturing sufficient information to support performance analysis. Therefore, only configurable attributes (e.g., size of `Block`) are captured. Operations of each class are referenced by the behavioral model (see Section 2.2). This data model is a minimum in the sense that we only capture concepts that are relevant to performance analysis, nothing more.

The node specification model (see Fig. 1) models the internal structure of nodes and their properties related to performance analysis. For a DFS, all its contained nodes are physical processing devices capable of storing and executing program code. Therefore, their fundamental service is to compute [6]. In MARTE, stereotype <<HwComputingResource>> denotes an active execution resource. In DFSs, all the nodes are computing resources and hence all have it applied. Plus, they are all applied with <<CommunicationEndPoint>>, indicating that all the nodes provide a mechanism for connecting/delivering data to a communication media. Each node is composed of one or more `Processor`, `Disk`, and `Memory`. <<HwProcessor>> has attributes of `op_Frenquecies` (the clock frequency), nbCores (the number of cores), among others. <<HwDrive>> and <<HwRAM>> denote a mass storage memory and a processing memory, respectively. Note that only few attributes of the stereotypes are shown in Fig. 1 due to space limitation.

As shown in Fig. 1, the application to node deployment model of DFSs captures physical connections among nodes and the deployment of software components to nodes, as a UML deployment diagram. A client node is connected to the namenode and one or many datanodes through network,

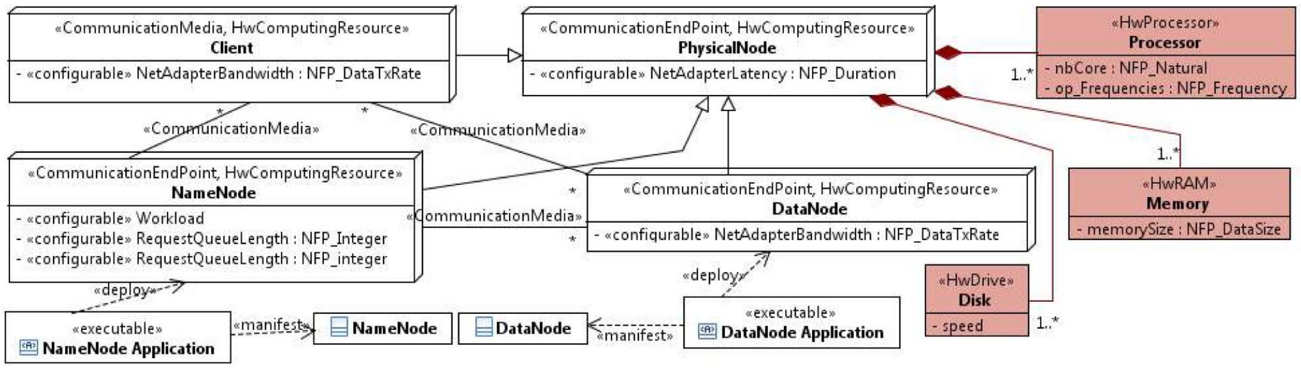

«CommunicationMedia, HwComputingResource» Client
- «configurable» NetAdapterBandwidth : NFP_DataTxRate
«CommunicationEndPoint, HwComputingResource» PhysicalNode
- «configurable» NetAdapterLatency : NFP_Duration
«HwProcessor» Processor
- nbCore : NFP_Natural
- op_Frequencies : NFP_Frequency
«CommunicationMedia»
«CommunicationEndPoint, HwComputingResource» NameNode
- «configurable» Workload
- «configurable» RequestQueueLength : NFP_Integer
- «configurable» RequestQueueLength : NFP_integer
«CommunicationEndPoint, HwComputingResource» DataNode
- «configurable» NetAdapterBandwidth : NFP_DataTxRate
«HwRAM» Memory
- memorySize : NFP_DataSize
«HwDrive» Disk
- speed
«deploy»
«executable» NameNode Application
«manifest»
NameNode
DataNode
«executable» DataNode Application
1..*

Fig. 1. Application to node deployment model and node specification model (red part).

which is specified as the associations between them stereotyped with <<CommunicationMedia>>. Configurable properties of the physical nodes are shown as attributes stereotyped with <<Configurable>>. This deployment diagram also indicates an instance of software component `NameNode application (DataNode application)` is deployed to physical node `NameNode (DataNode)`, which is linked to software class `Namenode (DataNode)` of the data model.

Note that in some DFSs, an instance of the client application software and an instance of the datanode software can be installed on the same machine.

2.2 Behavioral Modeling

Reading and writing files are two most important DFS operations. Therefore, the behavioral model consists of two UML sequence diagrams, each of which models the interactions between clients, datanodes and the namenode, which are realized in sequence diagrams as messages. Notice that DFSs are message-based systems. In DFSs, the delivery time of a message is determined by the file size being read or written from/to DFSs, and also the network conditions, and the current status of the receiver and sender of the message.

Besides messages, UML sequence diagrams also allow us to model loop, optional, alternative and parallel branches through advanced features: `CombinedFragments`. This is very important for modeling Read and Write of DFSs as we need to capture repeated steps, optional and alternative branches, and concurrent steps.

2.2.1 *Read*

As shown in Fig. 2, to read a file, a client first contacts the namenode and then it will translate the file name into a list of block information and return them to the client node. The block information contains a list of datanodes that store each block. Then the client connects itself to the "closest" datanode and requests a specific block ID. This can be done in parallel (the *par* combined fragment in Fig. 2), or sequentially (the *else* operand of the *alt* combined fragment in Fig. 2). Notice that the *alt* combined fragment specifies and distinguishes two types of realizations of the Read operation in DFSs. The *opt* combined fragment with condition "need more block info." shows that the first `getBlockLocations` operation might not get all the block information related to the file. In such cases, the `getBlockLocations` operations should be invoked multiple times. How many times required depends on the size of the file and how much information returned by one invocation of the operation and so on. The parameter N in the condition of the first loop combined fragment denotes the number of block information returned by a `getBlockLocations` invocation. Of course, the other approach is to gather all the required block information from the first `getBlockLocations` invocation.

2.2.2 *Write*

To write a file to DFS, a client application first creates a new file (see Fig. 3). This `creates` operation further sends a message to the namenode through a remote procedure call (RPC) to complete the creation of a new file in the namenode. At this time, there is no block related to the newly created file. Alternatively, a client can append new

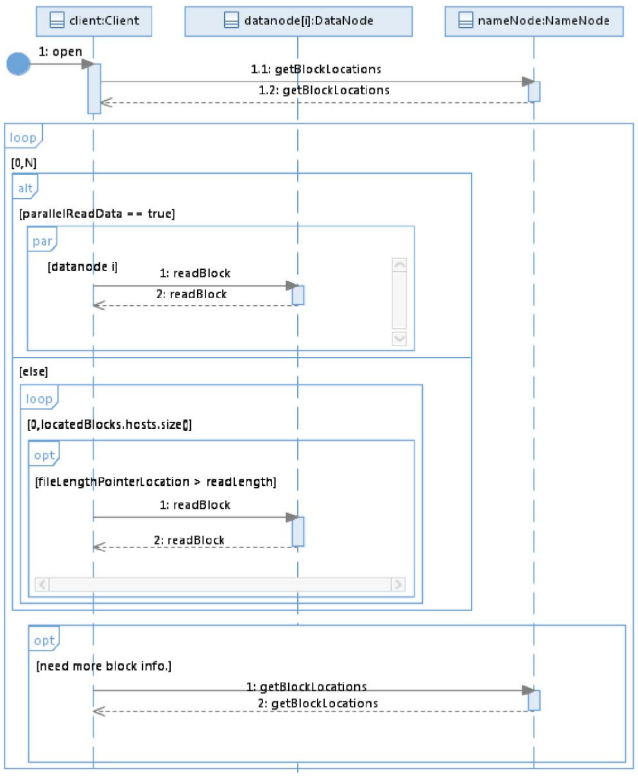


Fig. 2. Sequence diagram for the Read operation.

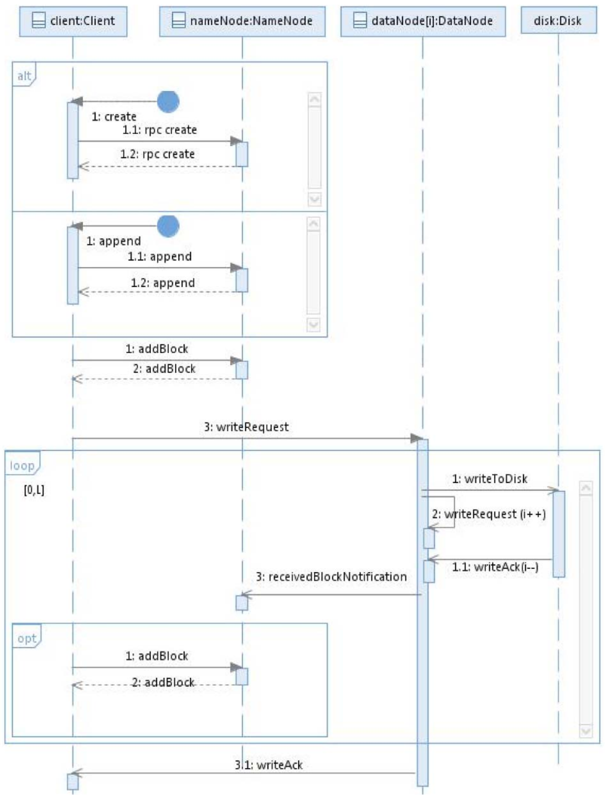


Fig. 3. Sequence diagram for the Write operation.

data to an existing file in the system. Then operation `append` is invoked instead of invoking operation `create`. This is captured in Fig. 3 as the *alt* combined fragment.

The client needs to send a message to the namenode to request for adding new blocks. This `addBlock` operation is also required as long as a block being written is full and there is a need for requesting a new block, as shown by the *opt* combined fragment in the *loop* combined fragment.

The client sends a write request to the first datanode (`writeRequest`), which, therefore, sends an asynchronized message to its local disk to write data (`writeToDisk`), while sending another asynchornized message to the next datanode to request writing (`writeRequest(i++)`). When the writing disk operation is completed, the datanode sends an acknowledgement message to its previous datanode (`writeAck(i-)`), and eventually an acknowledgment message is sent to the client (`writeAck`). Notice that the parameter L in the loop combined fragment denotes the total number of datanodes involved in a Write operation. Therefore, this parameter indirectly reflects the size of the file being written.

2.3 Performance-Relevant Configuration

For the structure configuration, one has to configure the DFS by making decisions such as number of nodes to deploy and what their internal topologies are. More specifically, we need to, based on the structural model described in Section 2.1, to configure the topology of the DFS nodes and their internal structures.

Regarding performance parameter configuration, performance-relevant parameters should be bound to concrete values when the system is configured. For example, parameters applied with stereotype <<configurable>> (e.g., `NetAdapterLatency` of `ClientNode`) (see Fig. 1). This type of configuration also includes configuring data model classes by assigning concrete values to their configurable attributes, for example, `size` of `Block`.

Based on the structural and behavioral models, we systematically derived a set of performance-relevant, configurable parameters, as shown in Table 1. They will be used to derive the analysis formulas (see Section 3).

TABLE 1
Configurable Parameters

Model	Acronyms	Configurable parameter
Application to Node Deployment Model	Dp1	Number of datanodes
	Dp2	Number of client nodes
	Dp3	PhysicalNode::NetAdapterLatency
	Dp4	NameNode::Workload
	Dp5	NameNode::RequestQueueLength
	Dp6	NameNode::ResponseQueueLength
	Dp7	DataNode::NetAdapterBandwidth
	Dp8	ClientNode::NetAdapterBandwidth
	Dp9	CommunimcationMedia::NetworkLatency
Node Specification Model	Np1	Number of processors/CPUs
	Np2	Processor::nbCores of each processor
	Np3	Processor::op_frequencies of each processor
	Np4	Memory::memorySize
Data model	Dap1	Block size
	Dap2	File size
Performance Model	Pp1	Number of invocations of getBlockLocations

3 Performance Modeling and Analysis

A performance model is an instance-based representation of a runtime system, focusing on how the system uses all kinds of resources and how the usage of them impacts the system performance, for example, response time. Therefore, we mainly reply on the Read and Write sequence diagrams (see Figs. 2 and 3), and the configuration parameters to derive performance analysis formulas for DFSs.

To analyze the performance of a DFS, we mainly consider time required to send messages between client nodes, the namenode and datanodes. Therefore, the total response time is determined by time required for each operation invocation (in messages) between two nodes.

3.1 Read

As one can see from Fig. 2, the overall Read response time (T_{t_read}) consists of three parts: time for operations `open`, `getBlockLocations`, and `readBlock`. We, therefore, can derive the following formula for the Read operation:

$$T_{t\_read} = T_{\text{open}} + N{*}T_{\text{getBlockLocations}} + M{*}T_{\text{readBlock}}, \tag{1}$$

where N and M denote the total number of `getBlockLocations`, and `readBlock` operations of a Read operation, respectively. As discussed in Section 2.2, the sequence diagram in Fig. 2 represents different types of DFS implementations and therefore values assigned to M and N are different for different implementations. Note that there is always one `open` operation. M and N also depends on the size of the block location information returned by each `getBlockLocations` operation. In other words, if more block location information returned by each getBlockLocations, then less number of invocations of getBlockLocations (Pp1) is required, therefore leading to shorter response time.

T_{open}, $T_{getBlockLocations}$, and $T_{readBlock}$ denote the time spent on each invocation of open, getBlockLocations, and readBlock, respectively. As shown in Fig. 2, each invocation of getBlockLocations requires a message sent from a client node to the namenode. Time of sending such a message depends on the network adapter bandwidth of the client node (Dp8), the current workload (Dp4), request queue length (Dp5) and response queue length (Dp6) of the namenode, and also the network latency (Dp9) between the client node and the namenode. One can easily observe from (1) that when the network condition is not good, the response time of the Read operation will be long, and if the size of the file to read is large, it of course takes longer time to read since more getBlockLocation and readBlock will be invoked. For readBlock, each invocation triggers a message sent from the client to a datanode. Therefore, the following factors have influences on the time of sending such a message: the network adapter bandwidth of the client side (Dp8) and the datanode side (Dp7), and the network latency (Dp9) between the client side and the datanode.

The block size (Dap1) is defined as a configurable attribute of class Block in the data model (see Section 2.1). When the system is configured, such attributes should be specified (see Section 2.3). When we derive performance prediction formula, this parameter should be considered as a factor as well. Taking into account all the above-mentioned factors, we can further detail (1) into the following set of formulas, which specify all these factors influencing the response time of reading a file:

$$T_{\text{open}} = f_1(\text{Dap1}, \text{Dap2}, \text{Dp8}), \tag{2}$$

$$T_{\text{getBlockLocations}} = f_2(\text{Dap1}, \text{Dap2}, \text{Dp4}, \text{Dp5}, \text{Dp6}, \text{Dp8}, \text{Dp9}), \tag{3}$$

$$T_{\text{readBlock}} = f_3(\text{Dap1}, \text{Dap2}, \text{Dp7}, \text{Dp8}, \text{Dp9}). \tag{4}$$

Functions f_1, f_2, and f_3 are very hard to obtain. We, however, can obtain estimated functions via conducting a series of experiments (see Section 4). These functions can then be used to analyze response times of the Read operation when different configurations are given.

In DFSs, it is usually a case that multiple clients read or write files concurrently. Therefore, there is a very important factor that greatly impacts the response time of the Read or Write operations: *count*, denoting the number of concurrent Read and Write operations. Considering *count*, (1) should be changed to

$$T_{t\_\text{read}} = \text{count}(T_{\text{open}} + N{*}T_{\text{getBlockLocations}} + M{*}T_{\text{readBlock})}). \tag{5}$$

3.2 Write

Similarly, we derive the performance model for Write. As shown in Fig. 3, the overall write response time (T_{t_write}) can be decomposed into five parts: time required for operations create, append, rpc create, writeRequest, and addBlock. Operation writeRequest consists of all the operations needed to write a file to datanodes. It is possible to further decompose writeRequst to its nested operations to achieve better performance analysis, which is considered as our future work. We can then derive the following formula for Write:

$$T_{t\_\text{write}} = T_{\text{create}} + T_{\text{rpc create}} + T_{\text{writeRequest}} + P{*}T_{\text{addBlock}}, \tag{6}$$

where P denotes the total number of addBlock invocations. P depends on the file size. Reading large files leads to large P. It also depends on the block size. If the block size is large, then less number of operation addBlock is invoked and therefore small P.

T_{create}, $T_{rpc\ create}$, $T_{writeRequest}$, and $T_{addBlock}$ denote the time required for each invocation of operations create, rpc create, writeRequest, and addBlock, respectively. As shown in Fig. 3, operation create invokes operation rpc create, which requires a message sent from a client node to the namenode to create a new file. Similar procedure is followed for append. Therefore, for create, and rpc create and append, time required for them depends on factors Dap1, Dap2, Dp4, Dp5, Dp6, Dp8, and Dp9. We can then obtain the following formulas:

$$T_{\text{create}} = f_4(\text{Dap1}, \text{Dap2}, \text{Dp4}, \text{Dp5}, \text{Dp6}, \text{Dp8}, \text{Dp9}). \tag{7}$$

$$T_{\text{rpc create}} = f_5(\text{Dap1}, \text{Dap2}, \text{Dp4}, \text{Dp5}, \text{Dp6}, \text{Dp8}, \text{Dp9}), \tag{8}$$

$$T_{\text{append}} = f_6(\text{Dap1}, \text{Dap2}, \text{Dp4}, \text{Dp5}, \text{Dp6}, \text{Dp8}, \text{Dp9}). \tag{9}$$

Operation writeRequest is more complicated than the others since it involves multiple steps, which send messages among the namenode, client node, and datanodes. Therefore, many factors have impact on the response time of the operation:

$$T_{\text{create}} = f_7(\text{Dap1}, \text{Dap2}, \text{Dp4}, \text{Dp5}, \text{Dp6}, \text{Dp7}, \text{Dp8}, \text{Dp9}). \tag{10}$$

As for Read, if we consider factor *count*, (6) should be changed to

$$T_{t\_\text{write}} = \text{count}(T_{\text{create}} + T_{\text{rpc create}} + T_{\text{writeRequest}} + P{*}T_{\text{addBlock}}). \tag{11}$$

4 Evaluation

4.1 Qualitative Performance Analysis

In this section, we present a series of real-world experiments that we conducted to evaluate the effectiveness of using our approach to qualitatively analyze the response time of Read and Write. All the experiments are based on HDFS [2], though the whole methodology described in the previous sections are general for all typical DFSs (see Section 4.3).

4.1.1 Experiment Settings

As shown in Table 2, the first experiment was deployed on one box of the Tsinghua Tansuo-100 High performance machine (abbreviated as *Tansuo*), which can have up to 104 TFlops computing power. We deployed one namenode

TABLE 2
Configurations of Three DFS Systems

Parameter	Tansuo	Inspur Cluster		Amazon EC2	
Dp1	17	8		60	
Dp2	17	8		60	
		Namenode	Datanodes	Namenode	Datanodes
Np1/node	2	1	1-2	1	1
CPU type	Intel Xeon	Intel Q6600	Intel x5650	Intel E5430	Intel E5430
Np3	2.93GHz	2.40GHz	2.67GHz	2.66GHz	2.66GHz
Np2/CPU	6	4	4-12	4	1
Dp9	Gigabit	Megabit	Gigabit	Gigabit	Gigabit
Np4	47G	2G	1G-24G	15G	1.66G

TABLE 4
Experiment Settings

Phase I									
Size	1K	2K	4K	8K	16K	32K	64K		
Count	1024								
Phase II									
Size	128K	256K	512K	1M	2M	4M	8M	16M	32M
Count	512	256	128	64	32	16	8	4	2
Phase III									
Size	64M	128M	256M	512M	1G	2G	4G		
Count	1								

and 17 datanodes to Tansuo. Each datanode also has a client application installed; therefore, these 17 datanodes also act as client nodes. All the namenode and datanodes have the same hardware configuration (see Fig. 1). The second system was deployed on the *Inspur* cluster, with one namenode and eight datanodes. The namenode in Inspur has lower hardware configuration (e.g., much less memory) to compare with Tansuo. The detailed hardware configuration of each datanode in Inspur is provided in Table 3, from which one can notice that datanodes D1, D2, and D3 have much smaller memory size than the others and the CPU of D4 has 12 cores, which is larger than the others. The third experiment was deployed on the Amazon Elastic Compute Cloud (EC2) (http://aws.amazon.com/ec2/). EC2 is a commercial web service and a virtual computing environment, as opposed to Tansuo and Inspur. We deployed 60 datanodes on it with the same configuration (see Table 2).

For each experiment, we conducted a series of tests, each of which was performed under a different combination of file *size* and *count*. Recall that count denotes the total number of concurrent Read or Write operations. The block size was set to 64M by HDFS.

As shown in Table 4, for each of the three experiments, we conducted 23 tests, which are divided into three phases: Phase I, Phase II, and Phase III. Each test was run four times and the consistency of the results was checked. We did not identify any significant inconsistencies and therefore in the remainder of the paper, we only report one group of the test results.

4.1.2 Experiment Results and Analysis

Read. During Phase I, `open` (2) has to be invoked as many times as the number of the files concurrently been read (i.e., count). Therefore, as shown in Fig. 4, during Phase I of the experiment, time for `open` and `getBlockLocations` are stable at log value 10. This is because Read invokes `open` and `getBlockLocations` once and therefore the time for these two operations is determined by count. As all the files in Phase I are small size, one `getBlockLocations` invocation often obtains all the required information for a Read operation. Also note that time for `open` and `getBlockLocations` is longer, compared to other two phases. This is again because Phase I has the largest count. Regarding `readBlock`, time increases slowly along with the increment of the total file size, simply because larger files need more time to read.

TABLE 3
Configurations of Inspur Cluster Datanodes

Parameters	Configuration		
	D1, D2, D3	D4	D5, D6, D7, D8
Np1/node	1	2	2
Np2/CPU	4	12	6
Np4	970M	23.5G	23.6G

During Phase II, the total size of all the files being read at each test remains same (i.e., 6), though file size increases and count decreases correspondingly. It is clear from Fig. 4 that time for `readBlock` remains stable since it mainly depends on the total size of the files. Note that this is only true when the size of each file is less than the block size. Time of invoking `getBlockLocations` and `open` decreases along with the decrement of count. This is because reading a file requires invoking open once and times of invoking `getBlockLocations` are small since the size of each file is still less than the block size (see Fig. 2). During Phase III, only one file was read in each test and therefore count equals 1 ($\log \text{size} = 0$). File size increases from 64M to 4G. Therefore, time for `readBlock` increases and more time to invoke `getBlockLocations` is required.

One can also observe from Fig. 4 that `readBlock` requires significantly more time than `getBlockLocations` and `open`, especially when the file size is large. This can be interpreted as `readBlock` is invoked more times than the other two ($N > M > 1$, (1)). One can also observe that it requires increasingly more time to read blocks when the file size increases. This is because larger files are divided into more blocks and therefore more invocations of

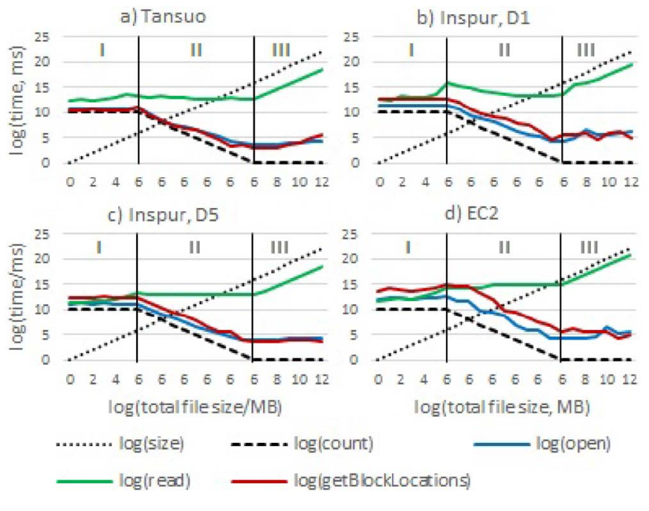


Fig. 4. Response time of the Read operations.

`readBlock` are occurred (i.e., N increases along with file size increment, formulas (1) and (4)). Very similar results were observed for all the datanodes in Tansuo and this is because they all have the same configuration.

Inspur has low-end hardware configuration than Tansuo (see Tables 2 and 3). The network connection between the namenode and datanodes is Megabit Ethernet with larger latency and the connections among datanodes are Gigabit. Hence, we expect that, for Read, `getBlockLocations` will take longer time in Inspur than Tansuo, since the operation is invoked through sending a message from the client to the namenode. The other two Read operations should not be much impacted by the configuration difference. We expect that as long as the hardware configuration of the namenode does not form a bottleneck to the whole system, the Read response time in Inspur should be very similar to the results we obtained from Tansuo, except that more time (but not significantly) is expected for `getBlockLocations`. Experiment results conducted on Inspur prove that our analysis is correct. The system performance is presented in Figs. 4b and 4c. Note that D1 (D5) has the lowest (highest) configuration among all the Inspur datanodes. From the figure, one can see that in all the phases, `getBlockLocations` took more time in Inspur than Tansuo.

The experiment we conducted on Amazon EC2 has larger scale than the first two. As shown in Table 4, 60 datanodes were deployed and they all have the same configuration. The namenode configuration of Amazon EC2 is higher than that in Inspur but lower than that in Tansuo. Therefore, we do not expect that the namenode would be the single-node bottleneck of the system performance because of memory limitation. However, we predict that overall EC2 should take longer time than Tansuo and Inspur because EC2 is a virtual machine-based environment. Therefore, the hard disk I/O speed of EC2 is slower and network communication also needs multiple routers, which should cause the overall lower performance of reading files in EC2 than Inspur and Tansuo. Our analysis again is proved correct; as shown in Fig. 4d: overall, all the Read operations took longer time during all phases than Tansuo and Inspur.

Write: In Fig. 5, during Phase I, time for operations `create`, `rpc create`, and `addBlock` does not increase along with the increment of the total file size, because the file size is small (1-64K) and therefore basically each of these three operations is invoked once only. Additionally, in Phase I, 1,024 files were written to the systems concurrently and hence these three operations were invoked 1,024 times. So comparing with Phase II and Phase III, the total time for each of these three operations in Phase I is the longest among all phases. In Phase I, time for writing files to blocks increases along with the increment of file size.

During Phase II, time required for `writeRequest` is roughly stable at log value 15, because the total size of the files remains at 64M. Along with the decrement of count, time for `create`, `rpc create`, and `addBlock` decreases. This is because the file size is still equal or smaller than 64M (the predefined block size); hence, count determines how many times are needed to invoke these three operations.

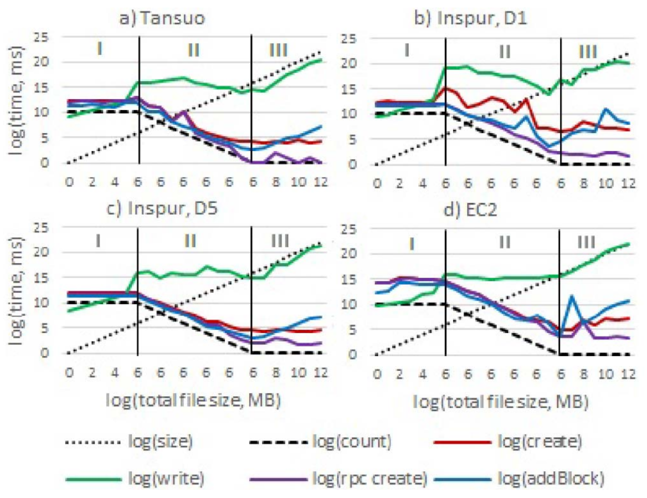


Fig. 5. Response time of the Write operations.

In Phase III, all files are over 64M; therefore, significantly more time is required for `writeRequest,` because when a file is larger than 64M, more than one block is needed to write the file. Therefore, the value of L in the loop combined fragment (see Fig. 3) will increase and consequently more times of invoking `writeToDisk` and `writeRequest` are needed. Due to the same reason, `addBlock` is invoked more times, hence time for `addBlock` increases long with the increment of the file size.

Due to the difference of the Inspur and Tansuo configurations, we expect that in Inspur, more time (but not significantly) should be taken on all the Write operations. This is because all the Write operations (i.e., `create`, `rpc create`, `writeRequest`, and `addBlock`) involving message interactions between clients/datanodes and the namenode (see Fig. 3). The performance of sending and receiving these messages is influenced by the network, which is Megabit Ethernet with larger latency, instead of Gigabit Ethernet with lower latency as in Tansuo and EC2. This analysis is proved correct, as shown in Figs. 5b and 5c. In addition, we also predict that the difference (mainly on Memory) among the configuration of the Inspur datanodes should not have significant impact on their performance as DFSs usually have no high requirements on datanode Memory. This analysis is again proved correct by comparing Figs. 5b and 5c. Note that the pulses in Fig. 5b are most probably caused by the instability of Inspur itself.

EC2 is a virtual machine-based environment which slows down the system performance. Therefore, we expect that the response time of the Write operations should be longer than that in Tansuo and Inspur. This analysis is again proved correct, as one can see from Fig. 5d.

4.2 Quantitative Performance Analysis

We did not observe any situation, where the namenode of a system reaches its bottleneck (see Section 4.1). But, being able to quantitatively analyze the limitation of the namenode beforehand is crucial. Hence, in this section, we discuss how we analyze the memory and computation limitations of the namenode, relying on the analysis of the response time of Read. Note that here we only analyze the performance of the namenode of a DFS and we particularly focus on the analysis of the `getBlockLocation` round trip message—the only type of message that the namenode has with clients.

4.2.1 Identify Namenode Memory and CPU Bottlenecks

As all user requests have to be processed by the namenode, there is a performance limitation for metadata operations, regardless how powerful its CPU is or how large its memory is. For the memory limitation, each file takes roughly 200 bytes of memory. Thus, the maximal number of files that the namenode can support is limited. For example, if the total memory of the namenode is 16G, the maximal number of files is up to 86 million, though this number may vary based on how long file names are and how many levels the files are down from root.

Regarding the CPU or computational limitation, we measure it as the maximal number of requests (Read or Write metadata operations) that the CPU can process per second. To see how this factor is important to the performance of the metadata server, consider there be a time period lasting for t seconds and during that period the rate of incoming requests is p. Assume the maximal number of requests per second that the CPU (we choose for the namenode) can process is q. If q is larger than p, each incoming request will be processed quite soon since there will be no queuing. However, if q is smaller than p, which is always the case when there comes a request peak, the average response time of requests in this period will increase by $\frac{t(p-q)}{2q}$, according to the queuing theory.

From the equation above, one can easily conclude that the amount of average response time will keep going up continuously as t or p increases. However, for any client using HDFS or other DFSs, there should always be an upper bound for the response time of a request, simply because one cannot wait for a request to be responded endlessly, hence making the whole client stuck in place. Therefore, if an upper bound of the average response time is given as B second and the latency of network (Dp9) for the round-trip message `getBlockLocation` is given as C second, we can infer, using equation $t(p-q) = 2q(B-C)$, how large scale (t and p) of the request peak (that the namenode can suffer but still keeps the average response time in bound) is. For example, if the lasting time of the request peak is T second, the maximal request rate of that period is

$$p_{\max} = \frac{2q(B-C)}{T} + q. \tag{12}$$

The above equation can really give a nice estimation of p when t is given or vice versa. However, q remains as an unknown factor for the given namenode. Though some experiments can be performed to obtain the value of this factor, it is not an easy task since many clients are required to push the namenode to its limit as well as a whole HDFS environment should be set up. In the next section, we discuss how to determine the value for q.

4.2.2 Determine the Maximal Number of Requests that the Namenode Can Process

In our experiments, we modified HDFS to do only metadata operations to test the performance of namenode. To keep its CPU as busy as possible, we use several clients to perform metadata operations in parallel. The number of clients varies from 10 to 99 and each client continuously sends read requests to namenode sequentially. We chose three different types of instances from Amazon EC2 as the namenodes and use the SPECjvm2008 benchmark to evaluate the performance of their CPUs. The hardware configurations and SPECjvm2008 results of the instances are given in Table 5.

TABLE 5
Quantitative Evaluation Experiment Settings

Instance type	CPU type	CPU speed	Memory	SPECjvm2008 Result
nn1(small)	Intel Xeon E5645	2.40GHz	1.7GB	7.7
nn2(medium)	Intel Xeon E5506*2	2.13GHz	1.7GB	27
nn3(large)	Intel Xeon E5645*4	2.40GHz	7.5GB	48

Based on (3) (see Section 3), it is easy to see that the response time of `getBlockLocation` is determined by seven factors: Dap1, Dap2, Dp4, Dp5, Dp6, Dp8, and Dp9. In our experiment, we intentionally let the file size (Dap1) smaller than the block size (Dap2), implying that reading one file only needs read one block. Dp5 is actually l and Dp6 is the response queue length, which is actually determined by clients, not the namenode when network condition does not cause the messages piled up in the namenode side—the case of our experiment. Therefore we do not consider Dp6 as an impact factor on the response time. Both Dp8 and Dp9 are set constant. Hence, we can easily draw the conclusion that the response time is only determined by Dp5 (l): the number of requests in the waiting queue when a Read request arrives. From Fig. 2, one can see that it takes time for the namenode to execute the `getBlockLocation` message (the bar on the namenode lifeline). This corresponds to the time of the namenode locating/searching file block information from a sorted block metadata record sequence, which is further determined by the number of file blocks (c) stored in the namenode with formula $\ln c$. So, based on our models and the analysis, we conclude that the response time of namenode to client requests is proportional to $l \ln c$.

Note that l is proportional to the number of clients. Moreover, to simplify our experiments, we make c also proportional to the number of clients. The rationale is that the number of files in HDFS is mostly expected to increase as the number of clients grows up. Therefore, what we can expect that the response time of metadata operations should be proportional to $x \ln x$, where x denotes the number of clients in the experiments.

Fig. 6 shows how the response time of Read varies with the number of clients when we chose nn1, nn2, and nn3 instances to be the namenode, respectively. A summary of the fit result is shown in Table 6, from which one can see that the $x \ln x$ curves fit quite well with our test points: all the R^2 are larger than 0.99. Moreover, when considering the reciprocal of coefficient of $x \ln x$, the ratio among three namenodes are 1:3.5089:6.6826, and the ratio of their SPECjvm2008 results is 1:3.5065:6.2338. These two ratios are fairly close, implying that the SPECjvm2008 results are a good estimation of how many requests the namenode can process per second. In addition, the conformance of our

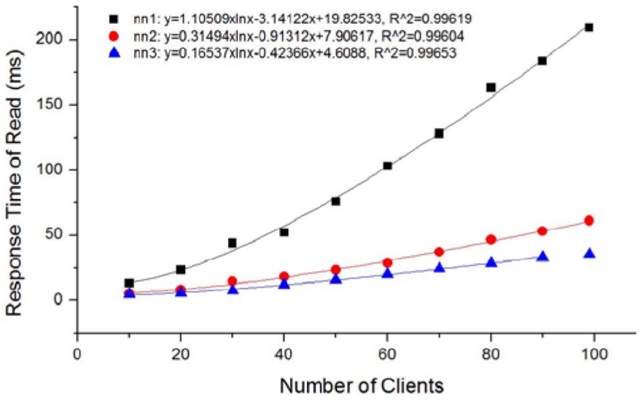


Fig. 6. Fit of response time of Read.

results to the SPECjvm2008 benchmark indicates that our analysis of $x \ln x$ is correct, which forms the foundation of the derivation of the fit curves.

As long as we can get a good estimation of how many requests that the namenode can process per second, we can further predict how large scale of the request peak that the namenode can suffer without compromising the response time upper bound, described as follows:

1. Run the SPECjvm2008 benchmark to get a score (s) for CPU. Using nn1 as the example, its $s = 7.7$.
2. Calculate the maximal request that the namenode can process per second according to equation $q = \frac{\frac{s}{7.7} * 0.90490}{\ln c} * 1{,}000$. Recall that c is the number of files. In this formula, we use the values of s and reciprocal of coefficient of $x \ln x$ of namenode nn1 to calculate the constant of the formula.

After we got the value for q, we can then use (12) to calculate the maximal incoming request rate p for a known time period t.

4.3 Discussion

All the systems deployed for the experiments are HDFS. However, the qualitative and quantitative performance analysis approach that we propose in this paper and applied in the experiments can be also applied for other typical DFSs such as MooseDFS, GFS, and zFS, with similar system structure and behavior as HDFSs.

In the qualitative performance analysis, we evaluate a DFS from the most critical aspect of its performance: response time of Read and Write. The same approach can also be applied for other operations such as *File List*, as long as the behavior of such operations is specified, for example, using UML sequence diagrams, as we did for Read and Write (see Figs. 2 and 3). In terms of the quantitative performance analysis, we only conducted the performance analysis for namenode. Similar kind of analyses can also be applied to evaluate the performance of datanodes. In the case that a DFS has multiple namenodes, we can reasonably consider the multiple namenodes as a whole (i.e., integrated CPU and Memory), when analyzing the performance of the system handling the metadata operations using our approach.

Replica strategies are commonly used in DFSs and introducing such strategies to the DFSs have impact on the performance of their namednode(s) and datanodes. However, our approach is still valid as long as the same strategy is applied in all the systems including DFS under analysis and deployed DFSs for collecting analysis data.

TABLE 6
Descriptive Statistics of Fit of Response Time of Read

Instance type	Coefficient of $x\ln x$	R^2	Reciprocal of coefficient of $x\ln x$
nn1	1.10509	0.99619	0.90490
nn2	0.31494	0.99604	3.17521
nn3	0.16537	0.99653	6.04705

5 Related Work

5.1 Model-Driven Methodologies for Performance Analysis

Balsamo et al. [9] conducted a survey on model-driven performance analysis and prediction in software development. The authors pointed out that the software performance prediction/analysis process is based on the availability of software artifacts describing suitable abstraction of the final software system. According to the survey, most of the methodologies that the authors studied make use of UML or UML-like formalisms to describe models, which is consistent with our work.

Tawhid and Petriu proposed an approach [26] to integrate performance analysis to the model driven development of software product lines. Annotated and parameterized UML sequence diagrams with the MARTE performance annotations (i.e., stereotypes) are required to be built for each use case of the system. The resulting sequence diagrams are then transformed into an LQN performance model for analysis [19]. A simulation-based approach is proposed in [10], which uses Palladio component models to specify component-based profile of a system. Pllana and Fahringer proposed a UML-based modeling approach (without analysis) for performance-oriented, parallel and distributed applications [20]. UML class, deployment and activity diagrams, extended with a large number of stereotypes, are mainly used to model architectures of parallel and distributed applications.

To compare with all these approaches, our approach aims to provide a practical means for the system analysts to analyze the system performance. The approach is simple in the sense that the effort required for modeling is much less than the modeling effort required by the approaches proposed in the related work.

5.2 Distributed System Performance Analysis

In benchmarks [12], [30] some HPC benchmarks (e.g., NPB, STREAM (for memory bandwidth), high performance LINPACK (HPL), IOR, and BT-IO (for parallel IO)) are used to evaluate distributed systems. They also test some real-world applications (e.g., POP, CCSM, mpiBLAST, CSFV) for investigating a variety of relevant issues such as performance (execution time, network latency and IO bandwidth), scalability and performance variability. Note that all of them rely on running benchmarks or application programs. Our approach, however, relies on architecture and design models of DFSs for their performance analysis, therefore saving cost and helping architects making decision of configuring and deploying DFSs prior to the deployment.

For distributed/parallel file systems (e.g., HDFS [2], GFS [11], MooseFS [3], PVFS [5]), there is no well-recognized benchmarks for evaluating or analyzing them. In [17], [24], and [26], in-house microbenchmarks (concurrent read/write from single/different files) are used and their evaluation results show that each of them is good at different aspects and they are quite different from each other. Research results in [16] provide a comparison of cloud storage systems (e.g., Amazon S3 [1], Openstack Swift [4], Microsoft SQL Azure [7]) from the aspects of high availability, scalability, data access performance, security and so on. As for evaluating the performance of computing paradigms, these approaches also rely on running application programs to test performance. Beker et al. [8] also analyzed the user-level file access patterns of a DFS by running application programs.

Another frequently applied performance analysis approach is to rely on performance measurements under a variety of workloads/strategies. For example, in [27], Weil et al. used this approach to measure a DFS, named Ceph, from various aspects such as I/O performance and write latency. Similarly, Ligon III and Ross [15] measured the raw transfer throughput in a variety of configurations to indicate the performance of PVFS.

Some researchers also evaluate/analyze the performance of a DFS by comparing with other DFSs. For example, Howard et al. [13] evaluated their DFS by comparing it with Sun Microsystem's NFS file system for the purpose of evaluating the whole file transfer strategy.

Regarding performance analysis and prediction of distributed computing and file systems, there are a lot of methods (e.g., machine learning [14], adaptive hybrid model with confidence window [29]) proposed. These approaches are mostly based on mathematical models.

6 Conclusion

Model-driven performance analysis has been recognized as an important tool to analyze system performance. In the paper, we presented such an approach, particularly tailored for distributed file systems (DFSs). Our approach mainly has several components: the architecture and design models and explicitly captured performance-relevant, configurable parameters, and the systematically derived performance model.

The related work in the field mainly evaluates the performance of DFSs and computing paradigms by, for example, relying on running benchmarks or application programs, performance measurements under a variety of workloads/strategies, and comparing with other DFSs. Our approach, however, qualitatively and quantitatively analyzes the DFS performance based on the models we constructed, such that early feedback on architectural design, configuration, and deployment of DFSs can be provided. Thereby one can avoid cost for architectural redesign and redeployment.

We conducted a series of real-world experiments deployed on EC2, Tansuo, and Inspur to demonstrate how our approach should be applied and to evaluate how effective it is. Results show that our approach is practical and can achieve sufficient performance analysis.

Acknowledgments

This work was supported by National Basic Research (973) Program of China (2011CB302505) and National High-Tech R&D (863) Program of China (2012AA012600).

References

[1] Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/S3/, 2013.
[2] Hadoop Distributed File System (HDFS), http://hadoop.apache.org/hdfs/, 2013.
[3] MooseFS, http://www.moosefs.org/, 2013.
[4] OpenStack SWIFT, http://openstack.org/software/openstack-storage/, 2013.
[5] Parallel Virtual File System, http://www.pvfs.org/, 2013.
[6] The UML MARTE Profile, http://www.omgmarte.org/, 2013.
[7] Windows Azure, http://www.windowsazure.com/, 2013.
[8] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. ACM SIGOPS Operating Systems Rev.*, vol. 25, pp. 198-212, 1991.
[9] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Trans. Software Eng.*, vol. 30, no. 5, pp. 295-310, May 2004.
[10] S. Becker, H. Koziolek, and R. Reussner, "The Palladio Component Model for Model-Driven Performance Prediction," *J. Systems and Software*, vol. 82, no. 1, pp. 3-22, 2009.
[11] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," vol. 37, pp. 29-43, 2003.
[12] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case Study for Running HPC Applications in Public Clouds," *Proc. 19th ACM Int'l Symp. High Performance Distributed Computing*, pp. 395-401, 2010.
[13] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Trans. Computer Systems*, vol. 6, no. 1, pp. 51-81, 1988.
[14] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling Virtualized Applications Using Machine Learning Techniques," *Proc. Eighth ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environment*, 2012.
[15] W. Ligon III and R. Ross, "Implementation and Performance of a Parallel File System for High Performance Distributed Applications," *Proc. IEEE Fifth Int'l Symp. High Performance Distributed Computing*, pp. 471-480, 1996.
[16] S. Mohammad, S. Breß, and E. Schallehn, "Cloud Data Management: A Short Overview and Comparison of Current Approaches," *Proc. 24th GI-Workshop Foundations of Databases (Grundlagen von Datenbanken)*, 2012.
[17] B. Nicolae, D. Moise, G. Antoniu, L. Boug, and M. Dorier, "BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, pp. 1-11, 2010.
[18] OMG, "UML 2.2 Superstructure Specification (formal/2009-02-04)."
[19] D. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications," *Proc. 12th Int'l Conf. Computer Performance Evaluation: Modelling Techniques and Tools*, pp. 183-204, 2002.
[20] S. Pllana and T. Fahringer, "On customizing the UML for Modeling Performance-Oriented Applications," *Proc. Winter Simulation Conf.*, pp. 83-102, 2002.
[21] O. Rodeh and A. Teperman, "zFS-a Scalable Distributed File System Using Object Disks," *Proc. IEEE/11th 20th NASA Goddard Conf. Mass Storage Systems and Technologies (MSST '03)*, pp. 207-218, 2003.

[22] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts,* vol. 89, Addison-Wesley, 1994.

[23] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M.F. Kaashoek, and R. Morris, "Flexible, Wide-Area Storage for Distributed Systems with WheelFS," *Proc. Sixth USENIX Symp. Networked Systems Design and Implementation,* pp. 43-58, 2009.

[24] W. Tantisiriroj, S.W. Son, S. Patil, S.J. Lang, G. Gibson, and R.B. Ross, "On the Duality of Data-Intensive File System Design: Reconciling HDFS and PVFS," *Proc. Int'l Conf. for High Performance Computing,* pp. 1-12, 2011.

[25] R. Tawhid and D. Petriu, "Integrating Performance Analysis in the Model Driven Development of Software Product Lines," *Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 490-504, 2008.

[26] W. Tantisiriroj, S. Patil, and G. Gibson, "Data-Intensive File Systems for Internet Services: A Rose by Any Other Name," Parallel Data Laboratory, Carnegie Mello Univ., 2008.

[27] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI),* pp. 307-320, 2006.

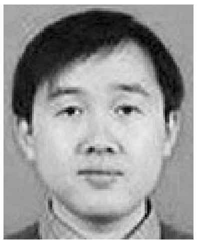
[28] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," *Proc. Sixth USENIX Conf. File and Storage Technologies,* p. 2, 2008.

[29] Y. Wu, K. Hwang, Y. Yuan, and W. Zheng, "Adaptive Workload Prediction of Grid Performance in Confidence Windows," *IEEE Trans. Parallel and Distributed Systems,* vol. 21, no. 7, pp. 925-938, July 2010.

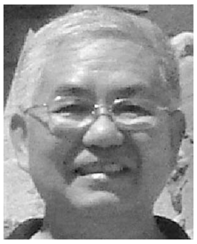
[30] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus In-House Cluster: Evaluating Amazon Cluster Compute Instances for Running MPI Applications," *Proc State of the Practice Reports Article. (SC '11),* p. 11, 2011.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing and Communication* of China Computer Federation. He has published more than 80 research publications and has received two Best Paper Awards. He is a member of the IEEE.

Feng Ye received the BS degree in computer science and technology from Tsinghua University in 2010. Currently, he is working toward the master's degree in computer science at Tsinghua University of China. His research interests include distributed file system, and cloud storage.



Kang Chen received the PhD degree in computer science and technology from Tsinghua University in 2004. He is currently an associate professor in computer science and technology at Tsinghua University of China. His research interests include distributed file system, parallel and distributed processing.



Weimin Zheng received the BS and MS degrees form the Department of Automatics, Tsinghua University, in 1970 and 1982, respectively. He is a professor of computer science and technology, Tsinghua University, China. He is the director of the Chinese Computer Society currently. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.