Model Driven Performance Simulation of Cloud Provisioned Hadoop MapReduce Applications

Hanieh Alipour, Yan Liu
Concordia University
Montreal.Canada
h_alipou@encs.concordia.ca;
yan.liu@concordia.ca

Abdelwahab Hamou-Lhadj
Concordia University
Montreal.Canada
wahab.hamou-lhadj@concordia.ca

Ian Gorton
College of Computer and Information Science
Northeastern University
igorton@neu.edu

ABSTRACT

Hadoop is a widely adopted open source implementation of MapReduce. A Hadoop cluster can be fully provisioned by a Cloud service provider to provide elasticity in computational resource allocation. Understanding the performance characteristics of a Hadoop job can help achieve an optimal balance between resource usage (cost) and job latency on a cloud-based cluster. This paper presents a method that estimates the performance of a MapReduce application in a Cloud provisioned Hadoop cluster. We develop a model-driven approach that models a cloud provided independent Hadoop MapReduce model and customizes it for a specific Cloud deployment. These models are further transformed into a simulation model that produces estimations of end-to-end job latency. We explore this method in the design space of MapReduce applications to estimate the performance for different sizes of input data. Our approach provides a model-to-simulation-to-prediction method for observing the performance behaviour of MapReduce applications given a configuration of a MapReduce platform.

Keywords

MapReduce; Hadoop; Performance; Model Driven; Cloud Computing

1. INTRODUCTION

MapReduce is a programming model and associated framework introduced by Google to process large data volumes in parallel in distributed clusters [4]. This programming model is divided into two main functions: the Map and Reduce function. The map function involves input split, map and combine phases, while the reduce function comprises shuffle, sort and reduce phases. Hadoop is an open-source implementation of the MapReduce model and is the most widely used implementation for processing large amounts of data in various domains such as the 2012 US Presidential Election, Healthcare, Telecoms, Energy, Retail and Software-as-a-Service (SaaS) Cloud.

Hadoop partitions large data sets stored in the distributed Hadoop File System (HDFS) across a number of mapper tasks that form the Map function. Each mapper performs identical processing on its partition of the data, and produces an intermediate collection of key-value pairs. This intermediate data is then shuffled to reducer tasks that encapsulate the Reduce function and aggregate the intermediate data to generate the final output. Application programmers are only required to write the Map and Reduce functions, with the Hadoop framework implementing the necessary mechanisms to reliably read, process and shuffle the data at scale. For deployment, a Hadoop cluster can be fully provisioned by a Cloud service provider to benefit from integrated service platforms. For example, Amazon Web Services provide Elastic MapReduce (EMR) and Simple Storage Service (S3) to distribute and process application data across Amazon Elastic Compute Cloud (EC2) instances.

In our previous work [10],we developed a Queuing Network Model to estimate the performance of a MapReduce application. The model was manually built and there was no direct mapping from the architecture design and deployment environment to the modelling constructs. In this work, we employ a model-driven approach that models a cloud provided independent Hadoop MapReduce model and customizes it for a specific Cloud deployment. These models are further transformed into a simulation model that produces estimations of end-to-end job latency. Our new work provides a model-to-simulation-to-prediction method for observing the performance behavior of MapReduce applications.

This paper aims to develop a method that follows model-driven principles and provides mapping between the design models to performance models. This method has abstractions of the most critical aspects of a MapReduce design with platform independent models (PIMs) and platform specific models (PSMs). The mapping between PIMs and PSMs, as well as transformation to performance models enables platform specific performance analysis of a MapReduce application design. More precisely, this paper addresses the following research questions:

R1. How to capture the entities of a MapReduce programming model in a PIM?

R2. How to model a configuration of the Hadoop execution framework and thus the resulting behaviour of MapReduce jobs?

R3. How to transform the design models to a simulation

MiSE'16, May 16-17, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4164-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896982.2896989

model to predict job latency?

R4. When the size of the input data increases, can we predict the performance of MapReduce jobs?

In our approach, we first develop a cloud provider independent model that represents entities involved in the Hadoop MapReduce phases. We further customize this model for a specific cloud deployment. These models are further transformed into a simulation model that produces estimations of end-to-end job latency. We demonstrate this method to estimate the performance of a movie recommendation application using Netflix data. The application is programmed using Hadoop streaming APIs on Python and deployed on Amazon EMR. In this evaluation, we vary the size of input data and compare the modeling results with experiments. The results establish a base for further research on a comprehensive framework for predicting MapReduce applications for a wide range of Hadoop configurations.

2. DEVELOPING CLOUD PROVIDER INDEPENDENT MODEL (CPIM)

MapReduce is a programming model designed to process large volumes of data in parallel by dividing the work into a set of independent tasks. In a nutshell, a MapReduce program processes data with *map* and *reduce* tasks. The map task takes the input data and produces intermediate data tuples. Each tuple consists of a key and a value. The reduce task aggregates the values of the data tuples with the same key according to the application logic. The basic functionality provided by simply writing *Mapper* and *Reducer* functions are further extended by refinements including input and output types, partition and combiner functions, sorting, merging and shuffling [4].

Apache Hadoop is an execution framework that provides the aforementioned extensions and refinements to launch a MapReduce program and manage its inputs and outputs. Hadooop launches MapReduce programs as jobs. A MapReduce job in Hadoop consists of one *JobTracker* and multiple *TaskTrackers*. When Hadoop jobs are submitted to the *JobTracker*, they are split into map tasks and reduce tasks, representing the *Mapper* and *Reducer* function respectively. These tasks are then pushed to available *TaskTracker* nodes. Each *TaskTracker* has a number of map slots and reduce slots. Every active *map* or *reduce* task occupies one slot.

2.1 Interface Model of MapReduce

In the interface model depicted in Figure 1, each interface defines relevant operations in each phase of *map* task and *reduce* task. The workflow of the *map* task includes the following phases.

Input phase : The input data to be processed by a MapReduce job are split so that each split is processed by a map task; then repeatedly an input split record is read and converted into an object of the type <*key,value*>;

Map phase : The *Mapper* is an implementation of the map function developed by the programmer. This function receives as input an object of <*key, value*> for each record of the input split;

Output phase : the outputs of the mapper function are written to the *Sort Buffer* that is in local memory with a predefined size. When the size of output data reaches a certain threshold, they are divided into partitions and saved in temporary files on the local disk.

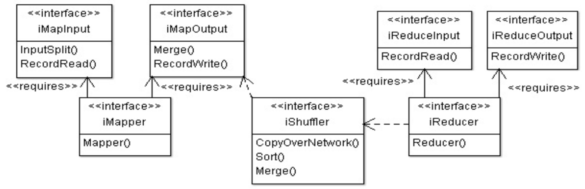


Figure 1: Interfaces of MapReduce programming model

To model the *map* task, we define three interfaces, namely *iMapInput, iMapper* and *iMapOutput*. The relationship defines that *iMapper* requires *iMapInput* and *iMapOutput*.

Next, in the shuffle phase, for each *Reducer*, the relevant partition of the output of all the *Mappers* is fetched via HTTP. They are sorted and distributed to *Reducers* by their keys. During the sorting, the inputs to *Reducers* are merged by keys since different *Mappers* may have output the same key. The shuffling and sorting occur simultaneously, that is while outputs are being fetched they are merged. The interface *iShuffler* is defined to model the shuffle phase. Interface *iShuffler* depends on interface *iMapOutput*.

The *Reduce* task depends on the workflow of the shuffle phase to fetch a record for each key, containing all their respective values. Similar to the map task, the workflow of the reduce task involves the input phase that takes records generated from the shuffle phase and converts the records into objects in the format <*key, list* <*value*> > for processing by the reduce function. Finally in the output phase, the result of the reduce function is output to the default file system. Likewise, interface *iReducer* depends on interface *iShuffler* and requires interface *iReduceInput* and interface *iReduceOutput* respectively.

2.2 Component Model of Hadoop Framework

The Hadoop execution framework provides the runtime components for a MapReduce program. Components of the Hadoop framework are modelled in Figure 2. We use a modelling tool called Palladio Component Model (PCM) [8] that supports UML notation. The binding between a component and an interface is through *provided role* or *required role* .

Hadoop supports several types of input formatting classes that take input files from Hadoop Distributed File System (HDFS) and split text or binary files into records to generate the key/value data tuples. The input formatting component is modeled as the component *FileInputFormatter*. It provides the interface *iMapInput* that is required by the *UserMapper* component. Hadoop also supports output formatting classes for producing the output from the *Mapper* function to the local file system. This is modeled as the *TextOutPutFormatter* component. Likewise, the *FileOutPutFormatter* component represents the Hadoop output formatting classes for producing output from the Reducer function to HDFS.

In addition, Hadoop provides the option of pluggable shuffling to replace the built in shuffling and sorting logic with alternative implementation. For example, using a different protocol other than HTTP such as RDMA for shuffling data from the *Map* nodes to the *Reducer* nodes. Thus the component *UserShuffler* models a user defined shuffling behavior.

The bottom layer of Figure 2 represent a user defined MapReduce program that requires the MapReduce interfaces. A MapReduce program often only needs to define the *UserMapper* and *UserReducer* components as the Hadoop *Mapper* class and *Reducer* class respectively.

2.3 Modeling MapReduce Configuration

To describe the runtime interactions as a result of a Hadoop configuration, we annotate the component model of Hadoop framework with calls in the PCM model of Service Effect Specifications (SEFF in short). Thus each component is further referred to a SEFF diagram. SEFF provides basic control flow constructs including sequences, alternatives, loops, and parallel executions (forks). Alternatives or branches split the control flow with an XOR semantics, while forks split the control flow with an AND semantic, i.e., all following actions are executed concurrently.

In Hadoop, reduce tasks need to process the output data of map tasks. Thus the data processing of reduce tasks cannot start until all the map tasks have finished. However, the shuffling of intermediate result data from map tasks to reduce tasks can start much earlier than the actual reduce processing. Hence reduce tasks occupy allocated reduce slots and begin data shuffling when some percentage of map tasks are complete. This is known as a slow start of reduce tasks. The parameter *mapred.reduce.slowstart.completed.maps* specifies the percentage of map tasks that should be completed before the process of shuffling data to reduce tasks can start.

Assume *mapred.reduce.slowstart.completed.maps* is set to the percentage of r and the total number of map tasks is *N*. Therefore $r * N$ of map tasks should be completed before the process of shuffling data can start. The forked behaviors of the component *UserMapper* have two compositions in two groups. One has *r*N* map tasks (in the left group in Figure 3) and the other has *(1-r)*N* map tasks (in the right group in Figure 3). The former group of *r*N* map tasks have internal actions of

1. loadfile: Load input file partition from the file system;
2. mapping: User-defined map processing on the input file;
3. localwrite: If the intermediate result from the map processing is bigger than a specified spill threshold, it will be written to the local disk of the map slot.

The former group of map tasks also have additional external calls to shuffling, while the remaining group of map tasks do not. This models the behaviour that partial map tasks complete and the shuffling phase begins.

Finally, the defined Hadoop components are composed into an *AssemblyContexts* in the system model, which are connected by *AssemblyConnectors*. An *AssemblyConnector* links a Required-Role of one component to a Provided-Role of another component.

The aforementioned models are Cloud provider independent since a Hadoop execution framework can be deployed in most nodes provisioned by a Cloud service provider.

3. MODELING CLOUD SPECIFIC HADOOP PLATFORM

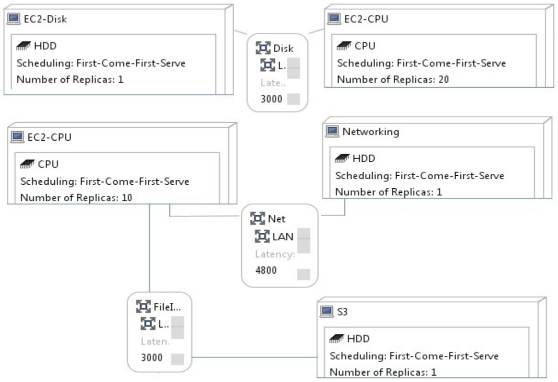


Figure 4: Resource environment of AWS Hadoop cluster

To launch user defined MapReduce components as depicted in Figure 3, these components need to be deployed on a Hadoop cluster. Amazon Web Services (AWS) provides the built-in Infrastructure-as-a-Service, called Elastic MapReduce (EMR) for setting up the Hadoop cluster and running MapReduce applications in a hosted cloud environment.

In the EMR configuration, the number of core instances can be specified for running Hadoop jobs. A master instance manages the Hadoop cluster. The master assigns tasks and distributes data to core instances. Core instances run map and reduce tasks and store data. The capacity of each core instance can be chosen from a range of EC2 instance types. In this paper, we assume the Hadoop cluster is homogenous. All core instances are of the same type. With EMR, both the input and output data are stored on Amazon S3 rather than an HDFS. This is the main difference between Amazon EMR and a standalone Hadoop cluster.

We model the physical resources in the resource environment diagram. Figure 4 illustrates the resource environment based on the AWS deployment architecture. The resource container EC2-CPU represents an EC2 instance and has CPUs as inner resources. The resource container EC2-Disk symbolizes the local disk of the EC2 instance. In addition, the resource containers of EC2 instances are connected to AWS S3 through the AWS networking. The attributes of each resource container further specify the processing rate, the capacity, the number of replicas and the scheduling policy of the inner resource.

In the resource environment diagram, we define a *LinkingResource* component that connects two resource containers. The connection links resource containers whose components are of a pair of *Provided-Role* and *Required-Role*. The service time or throughput of a component with the providing role is specified as an attribute of the *LinkingResource*. A component may have linkages to more than one component in a hierarchy. Thus by navigating through the linkages via *LinkingResource* components, the service time of dependent components can be determined.

The next step is allocating components modeled in CPIM to specific resource containers and their inner resource. The PCM Allocation diagram in Figure 5 specifies that a resource container should execute an *AssemblyContext* that contains components defined in the CPIM. The components of *FileInputFormatter* and *FileOutputFormatter* are allocated to the

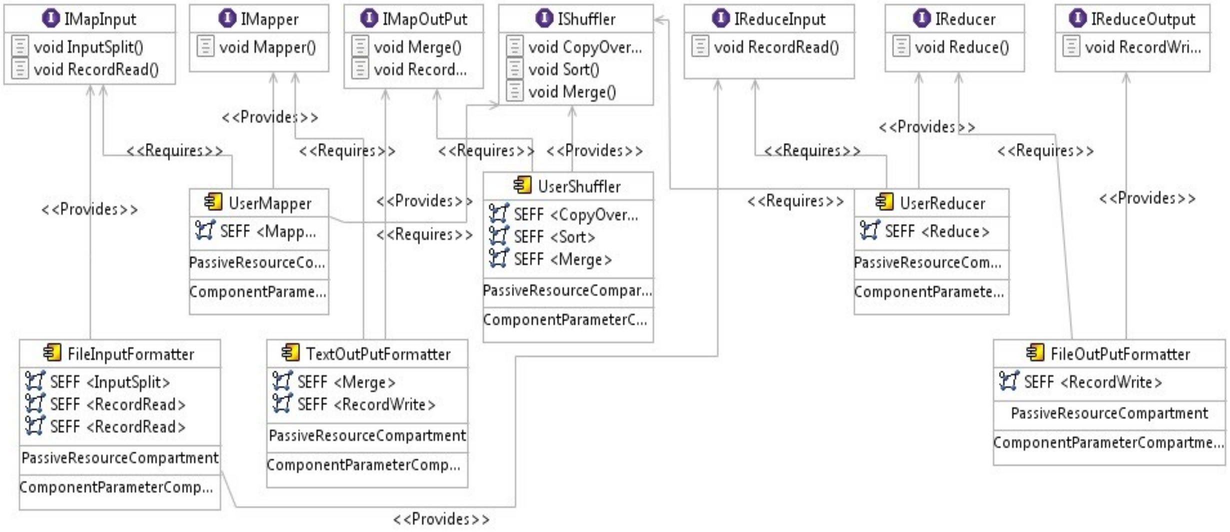


Figure 2: Cloud Platform Independent Model (CPIM) of MapReduce

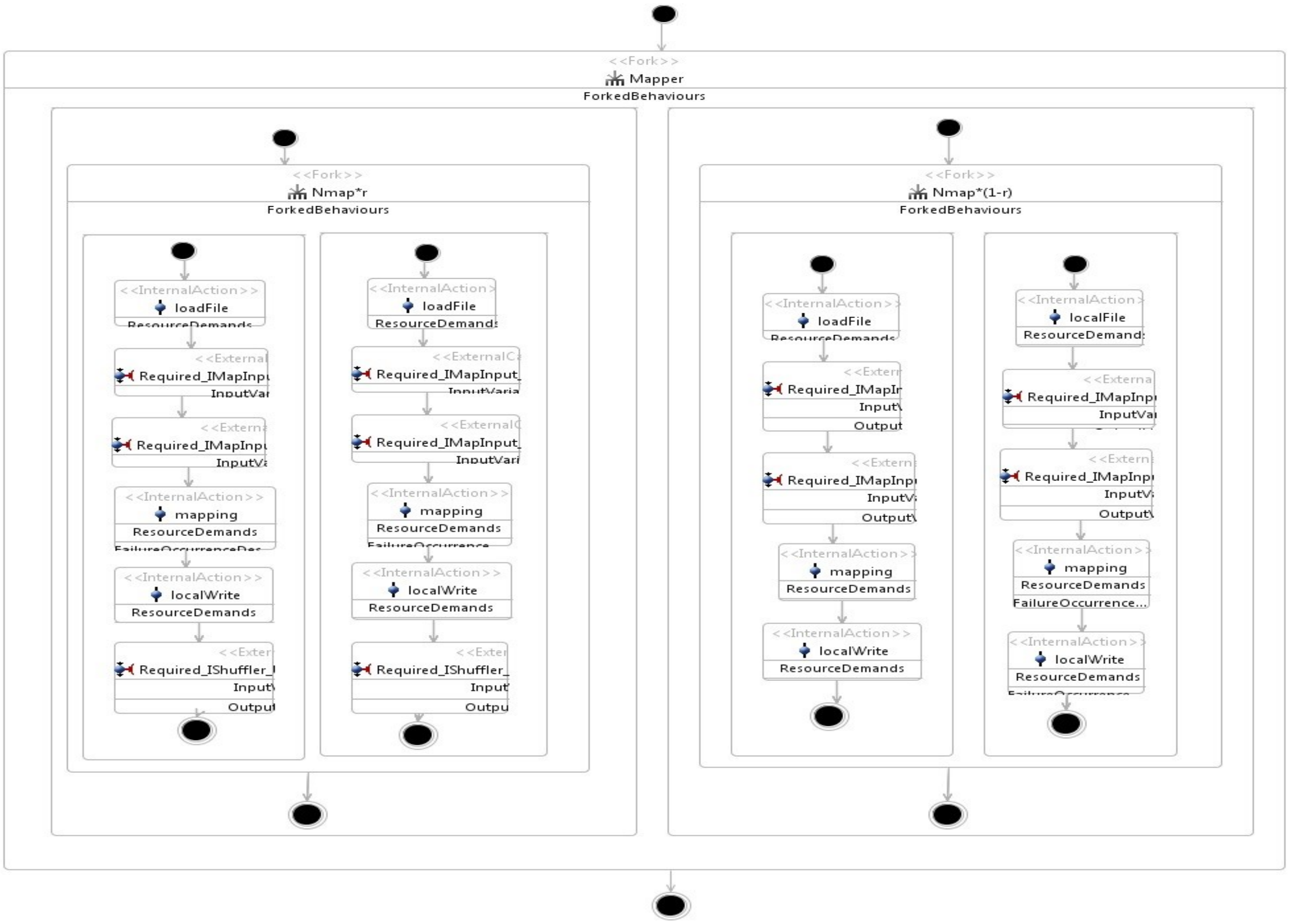
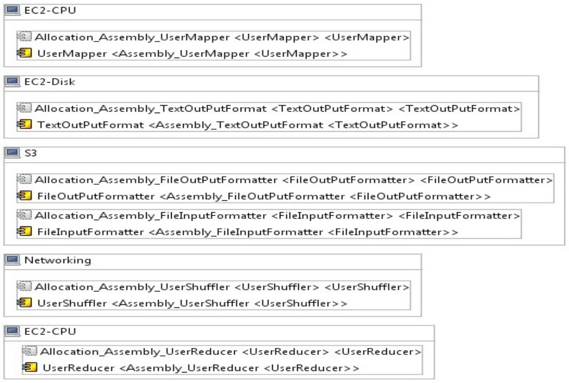


Figure 3: Model of slow-start behavior in SEFF



EC2-CPU

Allocation_Assembly_UserMapper <UserMapper> <UserMapper>
UserMapper <Assembly_UserMapper <UserMapper>>

EC2-Disk

Allocation_Assembly_TextOutPutFormat <TextOutPutFormat> <TextOutPutFormat>
TextOutPutFormat <Assembly_TextOutPutFormat <TextOutPutFormat>>

S3

Allocation_Assembly_FileOutPutFormatter <FileOutPutFormatter> <FileOutPutFormatter>
FileOutPutFormatter <Assembly_FileOutPutFormatter <FileOutPutFormatter>>
Allocation_Assembly_FileInputFormatter <FileInputFormatter> <FileInputFormatter>
FileInputFormatter <Assembly_FileInputFormatter <FileInputFormatter>>

Networking

Allocation_Assembly_UserShuffler <UserShuffler> <UserShuffler>
UserShuffler <Assembly_UserShuffler <UserShuffler>>

EC2-CPU

Allocation_Assembly_UserReducer <UserReducer> <UserReducer>
UserReducer <Assembly_UserReducer <UserReducer>>

Figure 5: Allocation of components to resource containers

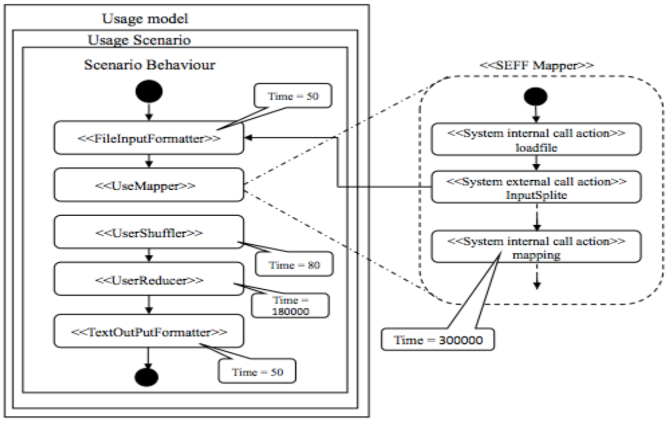


Figure 6: Service time specification in usage model

S3 resource container.

3.1 Service Demand Specification

The service time is specified in the usage diagram. A component may have linkages to more than one component in a hierarchy. Thus the request from one component to another involves the overall service call of components with dependencies. The example usage model in Figure 6 illustrates the workflow that is started by calling the *FileInputFormatter* component. Next, the call requires the *UserMapper* component, which has its own SEFF diagram.

The *load file* action is an internal action of the *UserMapper* component. It requires an external call in the *FileInputFormatter* component. The *FileInputFormatter* system contains three other SEFFs and the combination of service call for those three together make the service time for the *FileInputFormatter* component. The service time for *UserMapper* interface is the sum of the time required for *load file* action, *mapping* action, and *localwrite* action. Since the *load file* and *local write* need external calls, their service times are calculated by summing up the service time of external components.

4. MODEL TRANSFORMATION

The ultimate goal of developing CPIMs and CPSMs for a Hadoop MapReduce application is to estimate the end-to-end performance. Thus we further transform these models to a simulation model, called Layered Queueing Network (LQN) for performance estimation [2]. LQN models extend queueing network models to support nested queueing networks, which are suitable for performance modeling of complex distributed systems.

We chose the LQNs because in general the LQN model elements are semantically close to elements of our abstract model. This helps to automate the process of extracting a fundamental LQN performance model. Briefly, the notation of LQN models includes *Task*, *Processor*, *Entry*, *Call*, and *Activity*. The software components that provide services are *Entries*. A *Task* can have multiple *Entries*. The *Entries* can *Call* (or send requests to) *Entries* of other *Tasks*. When the operations of a *Task* become complex, the *Task* can be modeled as a precedence graph of *Activities*. An *Activity* can also call one or more *Entries* of other *Tasks*. The physical entities that perform the operations are *Processors*.

The one-on-one mapping rules between a PCM and a LQN model are as follows:

1. Components are mapped to Entries in a LQN model;
2. Each SEFF reference of a Component is mapped to Tasks in a LQN model;
3. Inter actions of a SEFF are mapped to Activities in a LQN model;
4. *AssemblyConnector* that links a Required-Role of one component to a Provided-Role of another component is modeled as a Call in a LQN model;
5. A *Resource Container* in a resource environment diagram is mapped to a Processor in a LQN model;
6. The allocation of an *AssemblyContext* to a *Resource Container* is mapped to the link from a Task to a Processor in a LQN model;

Given above mapping rules, Figure 7 shows the corresponding LQN model of a MapReduce application specified in the CPIM and CPSM.

In the transformation, the service times of individual components are retrieved mainly from the usage model. By navigating through the workflow in usage scenario, the service time of dependent components is available.

The LQN model presented in Figure 7 can be analyzed by two means, namely the LQN model solver or the LQN simulator [2] . We choose to use an LQN simulator. The main reason is both Palladio and the LQN solver have the limitation of presenting concurrent instances of tasks. They both require repetitive manual composing of the number of concurrent tasks. Since a MapReduce job can lead to arbitrary number of map tasks and reduce tasks depending on the partition of the input data, it is not practical to model such a behavior and then have a precise input to the LQN solver. Using a LQN simulator allows us to develop a program that generates simulation model segments of the concurrent tasks from the description of CPIM and CPSM.

We develop a program that automates the model transformation. In this program, PCM diagrams are first exported and stored in XML files. The transformation program parses the XML files and extracts elements involved in the mapping rules aforementioned. The extracted data from multiple diagrams are stored in a single JSON file. Finally segments of

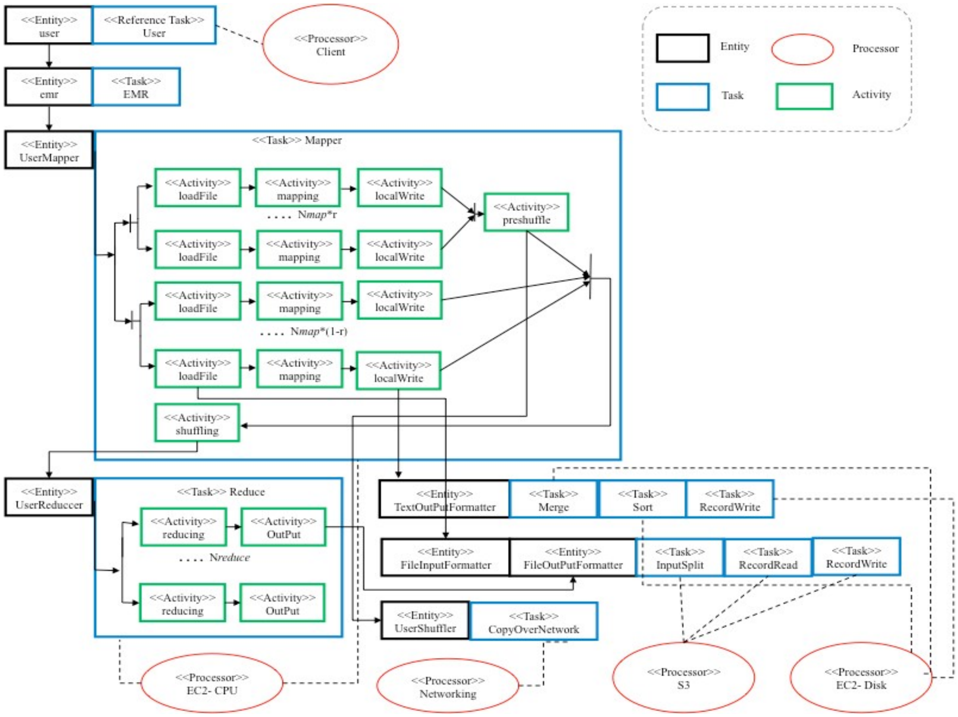


Figure 7: LQN model of a MapReduce job on Amazon EMR

the simulation model are generated from the JSON file to specify Processors, Tasks, and service times of each Task as well as concurrent calls from activities to activities and from tasks to processors. One of the advantages is we can extract the number of tasks from JSON files and generate the segments files automatically. For example, we model that the Amazon EMR launches 51 map tasks and 17 reduce tasks in Figure 3, then 51 concurrent calls from *Activity loadFile*, to *Activity mapping*, to *Activity localWrite* are generated in segments automatically from JSON files.

5. EVALUATION

We consider different sizes of data of a movie recommendation application to evaluate the ability of our method in estimating the MapReduce job performance [10]. The recommendation application provides a list of interested movies that a user may like. This application consists of three rounds of MapReduce processing as follows: Round 1 maps and sorts the movie pairs. Then in Round 2, the number of mutual fans of each movie is calculated. A mapper finds all the mutual fans of each movie and outputs a line for every mutual fan. Then the reducer combines the result based on the movie pair that has one mutual fan and counts the number of mutual fans. Finally, in Round 3, the movie pairs in the result of Round 2 extracts and top-N movies that have the most mutual fans of each movie are founded.

Our deployment of this application on AWS EMR includes one master instance and ten core instances. The master instance manages the Hadoop cluster. It assigns tasks and distributes data to core instances. Core instances run map and reduce tasks and access the movie input data from Amazon S3. All instances are in the type of m1.small, which has 1.7 GB of memory, 1 virtual core of EC2 Compute Unit, and 160 GB of local instance storage.

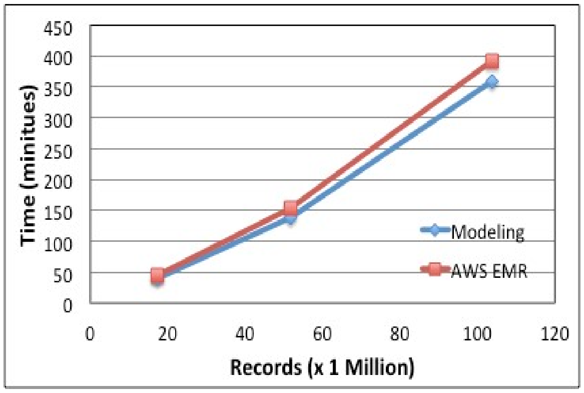


Figure 8: Modeling vs Experiment Performance Results

The input sample data are from Netflix Prize [1] . In total, the data set contains 17,770 files. Each file contains three fields, namely UserID, Rating and Date. The UserID ranges from 1 to 2649,429 with gaps. There are in total 480,189 users. Ratings are on a five star scale from 1 to 5. We merged all the 17,770 files into one file, with each line in the format of "UserID MovieID Rating Date", ordered by the ascending date, to create the input data for our application.

Assume 200 ratings made by users every second, the data set is of 200 ratings per second * 24 (hours) * 3600 (seconds) * N records on the Nth day. We run the movie recommendation jobs over the period of days. First we set N = 1, and then calibrate the models with parameters derived from the logs. We use Amazon CloudWatch and the Hadoop log files to collect monitoring data and calibrate the service times of components and actions using one-day data. We run the model-to-model transformation and produce the segments illustrated in List 1,2,3 as input to the LQN simulator.

We run the LQN Simulator [2] to process the simulation segments produced from our models. We simulate the performance for N = 3 and N = 6. Finally we run the movie recommendation application for N= 3 and N=6. We compare the processing time of the MapReduce job from the simulator with the actual processing time measured from the Amazon EMR. The results are plotted in Figure 8.

As the data size increases, we assume a linear growth of the service time of map tasks and reduce tasks. The error of simulation result is approximately 8% for six-day data, which leads to the absolute error of 33 minutes. There is a contributing factor to this error from the cloud environment. One factor is due to the processing time of EMR varies even with the same input and job configuration, potentially due to the facts that the virtual resources on AWS are not completed isolated. Also EMR modifies the original Hadoop to interact with S3. We also observe that the CPU utilization of the Hadoop cluster is at 100% most of the time of processing 3-day data. This indicates that the computing resource is already saturated and becomes bottleneck when processing 6-day data. The contention at the hardware level could also contribute to this error.

6. RELATED WORK

Model driven approaches are applied at both the infrastructure level and application level to help with the performance of cloud-based applications, including Hadoop MapReduce applications. Becker et al. [5, 6] employ model-driven approach to introduce SimuLizar as self-adaptive systems. Simulizar extends Palladio in modeling and design-time performance analysis of self-adaptive systems. Hadoop MaReduce applications are not their essential concern, hence it is not explicit of the steps to model Hadoop MapReduce applications.

Song at el. [7] propose a model to predict the performance of Hadoop jobs. This is composed of two parts: a job analyzer and a prediction model. The model can predict task execution times and job performance metrics. Wang at el. [9] use a simulation approach to capturing design parameters that affect the performance of MapReduce setups. The result of simulator is then used for optimizing existing MapReduce setups. We introduce model-driven approach to derive a prediction model from design models to estimate end-to-end job latency.

Franceschelli et al. [3] define meta-models of cloud resource provisioning and integrate both Cloud Provider Independent Model and Cloud Provider Specific Model with the Palladio modeling environments. This model-driven approach is applied to estimate the cost and performance of deploying cloud-based applications.

7. CONCLUSION

In this paper, we present a model driven method to simulate the performance of an MapReduce application running on a Cloud provisioned Hadoop cluster. This method focuses on capturing the runtime behavior of a MapReduce job given a setting of the Hadoop configuration. We first derive the essential interfaces of the MaReduce workflow. These interfaces are further realized by components built on the Hadoop execution framework. Thus we develop Cloud Provider Independent Models to represent the parallelism of map tasks and reduce tasks. The resource environment of a Hadoop cluster is customized in Cloud Provider Specific Models. We develop a workflow to transform both CPIMs and CPSMs to a LQN model for simulation. The simulation results provide performance indications given the size of input data. This research remains active to pursue a comprehensive modeling framework that can address other settings of Hadoop configurations.

8. REFERENCES

[1] http://www.netflixprize.com/.

[2] http://www.sce.carleton.ca/rads/lqns.

[3] Davide Franceschelli, Danilo Ardagna, M. C., and Nitto., E. D. Space4cloud: A tool for system performance and costevaluation of cloud systems. *in Proceedings of the 2013 international workshop on Multi-cloud applications and federated cloudsA.* (2013), 27–34.

[4] Dean, J., and Ghemawat., S. Mapreduce: simplified data processing on large clusters. *ACM* (2008), 107–113.

[5] Matthias Becker, J. M., and Becker., S. Simulizar: Design-time modeling and performance analysis of self-adaptive systems. *In Proceedings of the Software Engineering Conference (SE 2013)* (2013).

[6] Matthias Becker, M. L., and Becker., S. Performance analysis of self-adaptive systems for requirements validation at design-time. *In Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures (QoSA '13)* (2013), 43–52.

[7] Song, G., Meng, Z., Huet, F., Magoules, F., Yu, L., and Lin., X. A hadoop mapreduce performance prediction method,. *IEEE International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing.* (2013).

[8] Steffen Becker, H. K., and Reussnerr., R. The palladio component model for model-driven performance prediction. *Journal of Systems and Software, 82(1)* (2009).

[9] Wang, G., R.Butt, A., Pandey, P., and Gupta, K. A simulation approach to evaluating design decisions in mapreduce setups. *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium.* (2009), 21–23.

[10] Xing Wu, Y. L., and Gorton., I. Exploring performance models of hadoop applications on cloud architecture. *In Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15),ACM, New York, NY, USA* (2015), 93–101.