ABS-YARN: A Formal Framework for Modeling Hadoop YARN Clusters

Jia-Chun Lin(✉), Ingrid Chieh Yu, Einar Broch Johnsen,
and Ming-Chang Lee

Department of Informatics, University of Oslo, Oslo, Norway
{kellylin,ingridcy,einarj,mclee}@ifi.uio.no

Abstract. In cloud computing, software which does not flexibly adapt to deployment decisions either wastes operational resources or requires reengineering, both of which may significantly increase costs. However, this could be avoided by analyzing deployment decisions already during the design phase of the software development. Real-Time ABS is a formal language for executable modeling of deployed virtualized software. Using Real-Time ABS, this paper develops a generic framework called ABS-YARN for YARN, which is the next generation of the Hadoop cloud computing platform with a state-of-the-art resource negotiator. We show how ABS-YARN can be used for prototyping YARN and for modeling job execution, allowing users to rapidly make deployment decisions at the modeling level and reduce unnecessary costs. To validate the modeling framework, we show strong correlations between our model-based analyses and a real YARN cluster in different scenarios with benchmarks.

1 Introduction

Cloud computing changes the traditional business model of IT enterprises by offering on-demand delivery of IT resources and applications over the Internet with pay-as-you-go pricing [6]. The cloud infrastructure on which software is deployed can be configured to the needs of that software. However, software which does not flexibly adapt to deployment decisions either require wasteful resource over-provisioning or time-consuming reengineering, which may substantially increase costs in both cases. Shifting deployment decisions from the deployment phase to the design phase of a software development process can significantly reduce such costs by performing model-based validation of the chosen decisions during the software design [14]. However, virtualized computing poses new and interesting challenges for formal methods because we need to express deployment decisions in formal models of distributed software and analyze the non-functional consequences of these deployment decisions at the modeling level.

A popular example of cloud infrastructure used in industry is Hadoop [5], an open-source software framework available in cloud environments from vendors

Supported by the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems* (http://www.hyvar-project.eu) and FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu).

P. Stevens and A. Wąsowski (Eds.): FASE 2016, LNCS 9633, pp. 49–65, 2016.
DOI: 10.1007/978-3-662-49665-7_4

such as Amazon, HP, IBM, Microsoft, and Rackspace. YARN [27] is the next generation of Hadoop with a state-of-the-art resource negotiator. This paper presents ABS-YARN, a generic framework for modeling YARN infrastructure and job execution. Using ABS-YARN, modelers can easily prototype a YARN cluster and evaluate deployment decisions at the modeling level, including the size of clusters and the resource requirements for containers depending on the jobs to be executed and their arrival patterns. Using ABS-YARN, designers can focus on developing better software to exploit YARN in a cost-efficient way.

ABS-YARN is defined using Real-Time ABS, a formal language for the executable modeling of deployed virtualized software [10]. The basic approach to modeling resource management for cloud computing in Real-Time ABS is a separation of concerns between the resource costs of the execution and the resource provisioning at (virtual) locations [18]. Real-Time ABS has previously been used to model and analyze the management of virtual resources in industry [3] and compared to (informal) simulation tools [17]. Although Real-Time ABS provides a range of formal analysis techniques (e.g., [2,30]), our focus here is on obtaining results based on easy-to-use rapid prototyping, using the executable semantics of Real-Time ABS, defined in Maude [12], as a simulation tool for ABS-YARN.

To evaluate the modeling framework, we comprehensively compare the results of model-based analyses using ABS-YARN with the performance of a real YARN cluster by using several Hadoop benchmarks to create a hybrid workload and designing two scenarios in which the job inter-arrival time of the workload follows a uniform distribution and an exponential distribution, respectively. The results demonstrate that ABS-YARN models the real YARN cluster accurately in the uniform scenario. In the exponential scenario, ABS-YARN performs less well but it still provides a good approximation of the real YARN cluster.

The main contributions of this paper can be summarized as follows:

1. We introduce ABS-YARN, a generic framework for modeling software targeting YARN. Using Real-Time ABS, designers can develop software for YARN on top of the ABS-YARN framework and evaluate the performance of the software model before the software is realized and deployed on a real YARN cluster.
2. ABS-YARN supports dynamic and realistic job modeling and simulation. Users can define the number of jobs, the number of the tasks per job, task cost, job inter-arrival patterns, cluster scale, cluster capacity, and the resource requirement for containers to rapidly evaluate deployment decisions with the minimum costs.
3. We comprehensively evaluate and validate ABS-YARN under several performance metrics. The results demonstrate that ABS-YARN provides a satisfiable modeling to reflect the behaviors of real YARN clusters.

Paper Organization. Section 2 provides a background introduction to Real-Time ABS and YARN. Section 3 presents the details of the ABS-YARN framework. In Sect. 4, we validate ABS-YARN and compare it with a real YARN cluster. Section 5 surveys related work and Sect. 6 concludes the paper.

Syntactic categories.
T in `GroundType`
x in `Variable`
s in `Stmt`
a in `Annotation`
g in `Guard`
e in `Expression`

Definitions.

$$
\begin{aligned}
P &::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\,\} \\
IF &::= \mathbf{interface}\ I\,\{\,[\overline{Sg}]\,\} \\
CL &::= \mathbf{class}\, C\,[(\overline{T}\ \overline{x})]\,[\mathbf{implements}\,\overline{I}]\,\{\,[\overline{T}\ \overline{x};]\,\overline{M}\} \\
Sg &::= T\ m\ ([\overline{T}\ \overline{x}]) \\
M &::= Sg\ \{[\overline{T}\ \overline{x};]\ s\,\} \\
a &::= \mathtt{Deadline:}\ e \mid DC : e \mid Cost : e \mid a, a \\
g &::= b \mid x? \mid g \wedge g \\
s &::= s; s \mid \mathbf{skip} \mid \mathbf{if}\ b\ \{s\}\ \mathbf{else}\,\{s\} \mid \mathbf{while}\ b\,\{s\} \mid \mathbf{return}\ e \\
&\quad \mid \mathbf{duration}(e, e) \mid \mathbf{suspend} \mid \mathbf{await}\ g \mid \texttt{[}a\texttt{]}\ s \mid x = rhs \\
rhs &::= e \mid cm \mid \mathbf{new}\ C\ (\overline{e}) \mid \mathbf{new}\ DeploymentComponent\,(e, e) \\
cm &::= [e]!m(\overline{e}) \mid x.\mathbf{get}
\end{aligned}
$$

Fig. 1. Syntax for the imperative layer of Real-Time ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] denote optional elements.

2 Background

2.1 Modeling Deployed Systems Using Real-Time ABS

Real-Time ABS [10] is a formal, executable, object-oriented language for modeling distributed systems by means of concurrent object groups [16], akin to concurrent objects [11], Actors [1], and Erlang processes [7]. Concurrent objects groups execute in parallel and communicate by asynchronous method calls and futures. In a group, at most one process is active at any time, and a queue of suspended processes wait to execute on an object of the group. Processes, which stem from methods calls, are cooperatively scheduled, so active and reactive behaviors can be easily combined in the concurrent object groups. Real-Time ABS combines functional and imperative programming styles with a Java-like syntax and a formal semantics. Internal computations in an object are captured in a simple functional language based on user-defined algebraic data types and functions. A modeler may abstract from many details of the low-level imperative implementations of data structures, but maintain an overall object-oriented design. The semantics of Real-Time ABS is specified in rewriting logic [12], and a model written in Real-Time ABS can be automatically translated into Maude code and executed by the Maude tool.

The imperative layer of Real-Time ABS addresses concurrency, communication, and synchronization based on objects. The syntax is shown in Fig. 1. A program P consists of interfaces IF, classes CL with method definitions M, and a main block $\{[\overline{T}\ \overline{x};]\ s\,\}$. Our discussion focuses on interesting imperative language features, so we omit the explanations of standard syntax and the functional layer (see [16]).

In Real-Time ABS, communication and synchronization are decoupled. Communication is based on asynchronous method calls $f = o!m(\overline{e})$ where f is a future variable, o an object expression, m a method name, and $\overline{e}$ the parameter values for the method invocation. After calling $f = o!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Synchronization is controlled by operations on futures. The statement **await** $f?$ releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and the execution may resume. The return value is retrieved by the expression f.**get**, which blocks all execution in the object until the return value is available. The syntactic sugar $x =$ **await** $o!m(\overline{e})$ encodes the standard pattern $f = o!m(\overline{e});$ **await** $f?; x = f$.**get**.

In Real-Time ABS, the timed behavior of concurrent objects is captured by a *maximal progress* semantics. The execution time can be specified directly with *duration* statements, or be implicit in terms of observations on the executing model. Method calls have associated deadlines, specified by `deadline` annotations. The statement **duration**(e_1, e_2) will cause time to advance between a best case e_1 and a worst case e_2 execution time. Whereas duration-statements advance time at any location, Real-Time ABS also allows a separation of concerns between the *resource cost* of executing a task and the *resource capacity* of the location where the task executes. Cost annotations `[Cost: e]` are used to associate resource consumption with statements in Real-Time ABS models.

Real-Time ABS uses *deployment components* to capture the execution capacity of a location in the deployment architecture, on which a number of concurrent objects can be deployed [18]. Each deployment component has its own execution capacity, which will determine the performance of objects executing on the deployment component. Deployment components are dynamically created by $x =$ **new** $DeploymentComponent\,(descriptor, capacity)$, where x is typed by the DC interface, *descriptor* is a descriptor for the purpose of monitoring, and *capacity* specifies the initial CPU capacity of the deployment component. Objects are deployed on a deployment component using the `DC` annotation on the object creation statement.

2.2 YARN: Yet Another Resource Negotiator

YARN [27] is an open-source software framework supported by Apache for distributed processing and storage of high data volumes. It inherits the advantages of its well-known predecessor Hadoop [5], including resource allocation, code distribution, distributed data processing, data replication, and fault tolerance. YARN further improves Hadoop's limitations in terms of scalability, serviceability, multi-tenancy support, cluster utilization, and reliability.

YARN supports the execution of different types of jobs, including MapReduce, graph, and streaming. Each job is divided into tasks which are executed in parallel on a cluster of machines. The key components of YARN are as follows:

- *ResourceManager* (RM): RM allocates resources to various competing jobs and applications in a cluster, replacing Hadoop's JobTracker. Unlike JobTracker, the scheduling provided by RM is job level, rather than task level. Thus, RM does not monitor each task's progress or restart any failed task. Currently, the default job scheduling policy of RM is CapacityScheduler [23], which allows cluster administrators to create hierarchical queues for multiple tenants to share a large cluster while giving each tenant a capacity guarantee.

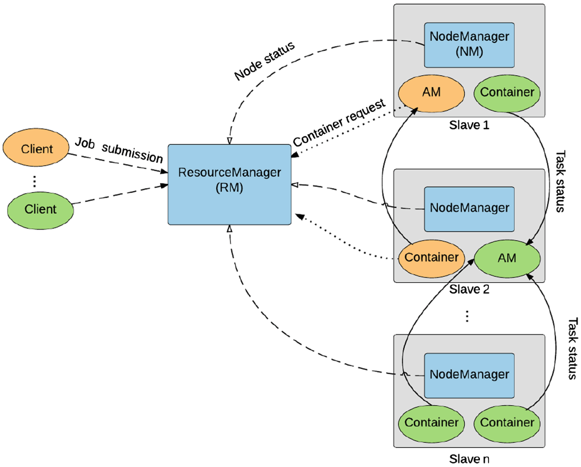


Fig. 2. The architecture of a YARN cluster.

The jobs in each queue are scheduled based on a First-in-First-out policy (FIFO), i.e., the first job to arrive is first allocated resources.

- *ApplicationMaster* (AM): This is an instance of a framework-specific library class for a particular job. It acts as the head of the job to manage the job's lifecycle, including requesting resources from RM, scheduling the execution of all tasks of the job, monitoring task execution, and re-executing failed tasks.
- *Containers*: Each container is a logical resource collection of a particular node (e.g., 1 CPU and 2GB of RAM). Clients can specify container resource requirements when they submit jobs to RM and run any kind of applications.

Figure 2 shows the architecture of a YARN cluster, which consists of RM and a set of slave nodes providing both computation resources and storage capacity to execute applications and store data, respectively. A slave node has an agent called NodeManager to periodically monitor its local resource usage and report its status to RM. The execution flow of a job on a YARN cluster is as follows:

1. Whenever receiving a job request from a client, RM follows a pre-defined job scheduling algorithm to find a container from an available slave and initiate the AM of the job on the container.
2. Once the AM is initiated, it starts requesting a set of containers from RM based on the client's container resource requirement and the number of tasks of the job. Basically, each task will be run on one container.
3. When RM receives a container request from the AM, it inserts the request into its queue and follows its job scheduling algorithm to allocate a desired container from an available slave node to the AM.
4. Upon receiving the container, the AM executes one task of the job on the container and monitors this task execution. If a task fails due to some errors such as an underlying container/slave node failure, the AM will re-request a container from RM to restart the task.
5. When all tasks of a job finish successfully, implying that the job is complete, the AM notifies the client about the completion.

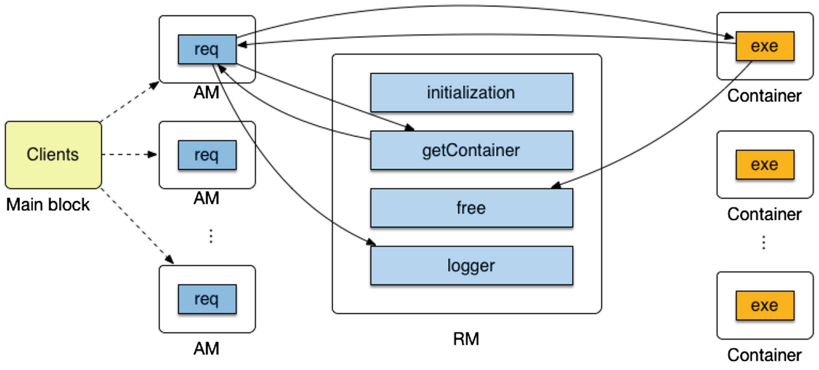


Fig. 3. The structure of the ABS-YARN framework.

3 Formal Model of the ABS-YARN Framework

Figure 3 shows the structure of ABS-YARN with classes RM, AM, and Container reflecting the main components of a YARN cluster. In our framework, RM is deployed as an independent deployment component with its own CPU capacity. To model the most general case, we assume that RM has a single queue for all job requests, implying that all jobs are served in a FIFO order. When a client submits a job, an AM object is created for this job, and its **req** method starts requesting containers from RM by invoking the **getContainer** method. If a slave has sufficient resources, a container will be created and returned to the AM. Then the AM submits one task of the job to the allocated container by invoking the **exe** method. When the task terminates, the result is returned to the associated AM, the **free** method is invoked to release the container, and the **logger** method is used to record execution statistics.

ABS-YARN allows modelers to freely determine the scale and resource capacity of a YARN cluster, including (1) the number of slave nodes in the cluster, (2) the CPU cores of each slave node, and (3) the memory capacity of each slave node. To support dynamic and realistic modeling of job execution, ABS-YARN also allows modelers to define the following parameters:

- Number of clients submitting jobs
- Number of jobs submitted by each client
- Number of tasks per job
- Cost annotation for each task
- CPU and memory requirements for each container
- Job inter-arrival pattern. Modelers can determine any kind of job inter-arrival distributions in ABS-YARN.

MapReduce jobs are the most common jobs in YARN, so we focus on modeling their execution in this paper. Each MapReduce job has a map phase followed by a reduce phase. In the map phase, all map tasks are executed in parallel. When all the map tasks have completed, the reduce tasks are executed (normally, each jobs has only one reduce task). The job is completed when all the map and reduce tasks have finished.

The execution time of a task in a real YARN cluster might be influenced by many factors, e.g., the size of the processed data and the computational complexity of the task. To reduce the complexity of modeling the task execution time, ABS-YARN adopts the cost annotation functionality of Real-Time ABS to associate cost to the execution of a task. Hence, the task execution time will be the cost divided by the CPU capacity of the container that executes the task.

In the following, we limit our code presentation to the main building blocks and functionalities to simplify the description.

3.1 Modeling ResourceManager (RM)

The ResourceManager implements the RM interface:

```
interface RM {
  Bool initialization(Int s, Int sc, Int sm);
  Pair<Int, Container> getContainer(Int c, Int m);
  Unit free(Int slaveID, Int c, Int m);
  Unit logger(...);}
```

Method `initialization` initializes the entire cluster environment, including RM and `s` slaves. Each slave is modeled as a record in a database `SlaveDB`, with a unique `SlaveID`, `sc` CPU cores, and amount `sm` of memory capacity. After the initialization, the cluster can start serving client requests. Method `getContainer` allows an AM to obtain containers from RM. The size of the required container core and container memory are given by `c` and `m`, respectively. Method `free` is used to release container resources whenever a container finishes executing a task, and method `logger` is used to record job execution statistics, including job ID and job execution time.

The `getContainer` method, invoked by an AM, tries to allocate a container with `c` CPU cores and `m` amount of memory capacity from an available slave to the AM. Each container request is allowed at most `thd` attempts. Hence, as long as `Find==False` and `attempt<=thd` (line 3), the `getContainer` method will keep trying to obtain the database token to ensure a safe database access. The built-in function `lookupDefault` checks each slave in `slaveDB` to find a slave with sufficient resources. If such a slave exists (line 11), the corresponding container will be created as a deployment component with `c` cores, and the slave's resources will be reduced and updated accordingly (lines 12–14). The successfully generated container is returned to the AM.

However, if no slaves have enough resources, the process will suspend (line 21), allowing RM to process other method activations. The suspended process will periodically check whether any slaves can satisfy the request. If the desired container cannot be allocated within `thd` attempts, the method terminates and RM is unable to provide the desired container to the AM.

```
Pair <Int, Container> getContainer (Int c, Int m) {
  Bool find=False; Int slaveID=1; Int attempt=1;
  while (find==False && attempt<=thd){
    await dbToken==True;
    dbToken==False;
    Int i=1;
    while (find==False && i<=size(keys(slaveDB))){
     Pair<Int,Int> slave= lookupDefault(slaveDB, i, Pair(1,1));
     Int free_core= fst(slave);
     Int free_mem= snd(slave);
     if (free_core>=c && free_mem >= m){
       slaveDB=put(slaveDB, i, Pair(free_core-c, free_mem-m));
       DC s=new DeploymentComponent("slave", map[Pair(CPU,c)]);
       [DC: s] Container container = new Container(this);
       find=True;
       slaveID=i;
     }
     i++;
    }
    ... // Release dbToken
    await duration(1,1);
    attempt++;
  }
  if (find==False){ container=null;}
  return Pair(slaveID, container);
}
```

3.2 Modeling ApplicationMaster (AM)

An AM implements the AM interface with a req method to acquire a container from RM and then execute a task on the container. For an AM, the total number of times that req is called corresponds to the number of map tasks of a job (e.g., if a job is divided into 10 map tasks, this method will be called 10 times).

```
interface AM {
  Unit req(Int mNum, Int c, Int m, Rat mCost, Rat rCost);}
```

The req method first invokes the getContainer method and sends a container-resource request (i.e., the parameters c and m) to acquire a container from RM. Since the call is asynchronous, the AM is able to request containers for other tasks of jobID while waiting for the response.

```
Unit req(Int mNum, Int c, Int m, Rat mCost, Rat rCost) {
  ...
  Pair<Int, Container> p= await rm!getContainer(c, m);
  Int slaveId=fst(p);
  Container container=snd(p);
  if (container!=null){
    Fut<Bool> f = container!exe(slaveID, c, m, mCost);
    await f?;
    Bool map_result = f.get;
    if (map_result==True){
      returned_map++;
      if (returned_map==mNum){
        Bool red_result;
        ...//Try to request a container and run the reduce task
        if (red_result==True){
          logging the job completion;
        }
18         else{ logging the reducde-task failure;}
19       }
20     }
21    else{ logging the map-task failure;}
22  }
23   else{ logging unsuccessful container request;}
24 }
```

When a container is successfully obtained, a map task with cost mCost can be executed on the container (line 7). The process suspends while waiting for the result of the task execution. Each time when map_result==True, the req method increases the variable returned_map by one. When all map tasks of the job have successfully completed (line 12), the AM proceeds with a container request to run the reduce task of the job with cost rCost. Only when all map and reduce tasks are completed (line 15), the job is considered completed.

3.3 Modeling Containers

A container implements the Container interface:

```
interface Container{
  Bool exe(Int slaveID, Int c, Int m, Rat tcost);}
```

Method exe is used to execute a task on a container. The formal parameters of exe consist of slaveID, CPU capacity c, memory capacity m, and the task cost tcost. Hence, the task execution time is tcost/c. When a task terminates, the free method of RM is invoked to release the container, implying that the corresponding CPU and memory resources will be returned back to the slave.

```
Bool exe(Int slaveID, Int c, Int m, Rat tcost){
  [Cost: tcost] ... //executing a task;
  rm!free(slaveID, c, m);
  return true;}
```

4 Performance Evaluation and Validation

To compare the simulation results of ABS-YARN against YARN, we established a real YARN cluster using Hadoop 2.2.0 [5] with one virtual machine acting as RM and 30 virtual machines as slaves. Each virtual machine runs Ubuntu 12.04 with 2 virtual cores of Intel Xeon E5-2620 2 GHz CPU and 2 GB of memory. To achieve a fair validation, we also created an ABS-YARN cluster with 30 slaves; each with 2 CPU cores and 2 GB of memory. To realistically compare job execution performance between ABS-YARN and YARN clusters, we used the following five benchmarks from YARN [23]: **WordCount**, which counts the occurrence of each word in data files; **WordMean**, which calculates the average length of the words in data files; **WordStandardDeviation (WordSD)**, which counts the standard deviation of the length of the words in data files; **GrepSort**, which sorts data files; and **GrepSearch**, which searches for a pattern in data files.

We created a hybrid workload consisting of 22 WordCount jobs, 22 WordMean jobs, 20 WordSD jobs, 16 GrepSort jobs, and 20 GrepSearch jobs. The submission orders of all jobs were randomly determined. Each job processes 1 GB of enwiki data [13] with 128 MB block size (the default block size of YARN [23]). Hence, each job was divided into 8 (=1 GB/128 MB) map tasks and one reduce task, implying that 9 containers are required to execute each job. We assume that the resource requirement for each container is 1 CPU core and 1 GB RAM for both the ABS-YARN and YARN clusters.

We considered two job inter-arrival patterns in our experiments: Uniform and exponential distribution [20]. In the former, the inter-arrival time between two consecutive jobs submitted by clients are equal. In the latter, job inter-arrival time follows a Poisson process [20], i.e., job submissions occur continuously and independently at a constant average rate. Reiss et al. [25] show that job arrival patterns in a Google trace approximates an exponential distribution. This distribution has also been widely used as job arrival pattern in the literature (e.g., [22,24]). Based on these distributions, two scenarios were designed:

- *Uniform scenario:* The job inter-arrival time of the workload is 150 sec in the real YARN cluster. In ABS-YARN, this is normalized into 2 time units.
- *Exponential scenario:* The job inter-arrival time of the workload follows an exponential distribution with the average inter-arrival time of 158 sec and a standard deviation of 153 sec in the real YARN cluster. This is normalized into the average inter-arrival time of 158/75 time units and a standard deviation of 153/75 time units in the ABS-YARN cluster.

The following metrics were used to evaluate how well ABS-YARN can simulate job scheduling, job execution behavior, and job throughput of YARN:

- Starting time of all jobs of the workload
- Finish time of all jobs of the workload
- The number of cumulative completed jobs
- Total number of completed jobs

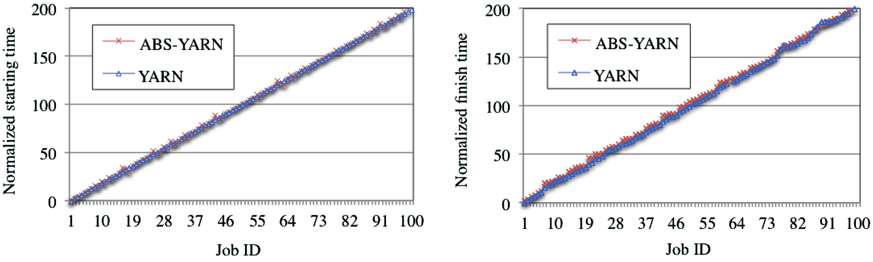
4.1 Validation Results in the Uniform Scenario

In order to achieve a fair comparison, we conducted the uniform scenario on the YARN cluster to obtain the average map-task execution time (AMT) and average reduce-task execution time (ART) for each job type. The results are listed in Table 1. After that, we respectively normalized each AMT and ART into a map-task cost and a reduce-task cost for ABS-YARN by dividing the AMT value by 75 and dividing the ART value by 75 (Note that 75 is half of the job inter-arrival time for the uniform scenario). With the corresponding map-task cost annotation (MCA) and reduce-task cost annotation (RCA), we simulated the uniform scenario on ABS-YARN.

Figure 4(a) shows the normalized starting time of all jobs in both clusters. We can see that the two curves are almost overlapping. The average time difference between ABS-YARN and YARN is 0.02 time units with a standard deviation of

Table 1. The average map-task execution time (AMT), average reduce-task execution time (ART), normalized map-task cost annotation (MCA), and normalized reduce-task cost annotation (RCA) in the uniform scenario.

Benchmark	AMT (sec)	ART (sec)	MCA	RCA
WordCount	162.64	251.01	2.17 (=162.64/75)	3.35 (251.01/75)
WordMean	107.10	139.94	1.43 (=107.10/75)	1.87 (=139.94/75)
WordSD	108.23	162.27	1.44 (=108.23/75)	2.16 (=162.27/75)
GrepSort	20.39	38.44	0.27 (=20.39/75)	0.51 (=38.44/75)
GrepSearch	31.22	55.97	0.42 (=31.22/75)	0.75 (=55.97/75)



(a) The normalized starting time

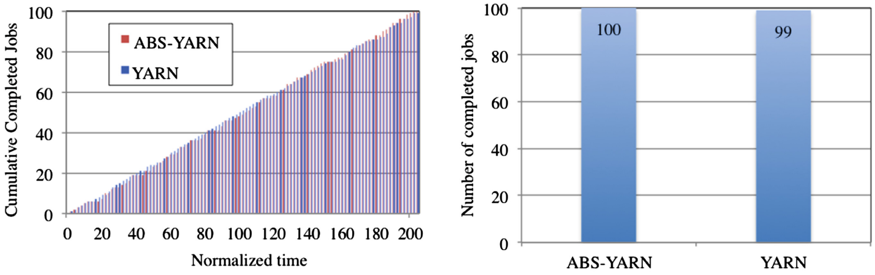
(b) The normalized finish time

Fig. 4. The normalized time points of all jobs in the uniform scenario.

1.73 time units, showing that ABS-YARN is able to precisely capture the job scheduling of YARN in the uniform scenario. Figure 4(b) depicts all job finish time in both clusters. The average difference between ABS-YARN and YARN is 2.67 time units with a standard deviation of 1.81 time units, indicating that the framework can accurately model how containers execute jobs in a real YARN cluster. Based on the results shown in Fig. 4, we can derive that the cumulative numbers of completed jobs between the two clusters are close (see Fig. 5(a)). The average error is approximately 2.52 %, implying that ABS-YARN can precisely reflect the operation of YARN in the uniform scenario. Figure 5(b) shows that 100 jobs of the workload successfully finished in the ABS-YARN cluster, but 99 jobs of the workload completed in the YARN cluster since the remaining one job could not obtain sufficient containers to execute its tasks. The job completion error of ABS-YARN is only 1.01 %. Based on the above-mentioned results, it is evident that the ABS-YARN framework offers a superior modeling of YARN in the uniform scenario.

4.2 Validation Results in the Exponential Scenario

In this section, we compare ABS-YARN and YARN under the exponential scenario. Similar to the uniform scenario, we performed a normalization by executing the exponential scenario on the YARN cluster to derive a map-task cost



(a) Cumulative completed jobs (b) Total number of completed jobs

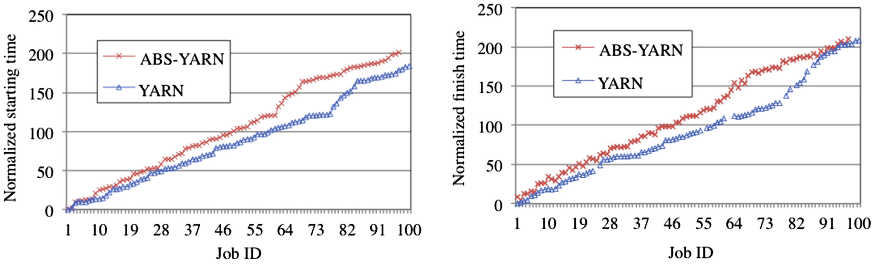
Fig. 5. The cumulative completed jobs and the total number of completed jobs in the uniform scenario.

annotation and a reduce-task cost annotation for each job type. The results are listed in Table 2. Note that regardless of which job type was tested, the corresponding average map-task and reduce-task execution time were apparently higher than those in the uniform scenario. The main reason is that the job inter-arrival time in the exponential scenario had a much higher standard deviation, implying that many jobs might compete for containers at the same time. However, due to the limited container resources, these jobs had to wait for available containers and hence prolonged their execution time.

Table 2. The AMT, ART, MCA, and RCA in the exponential scenario.

Benchmark	AMT (sec)	ART (sec)	MCA	RCA
WordCount	295.47	430.24	3.94 (=295.27/75)	5.74 (430.24/75)
WordMean	139.98	201.11	1.87 (=139.98/75)	2.68 (=201.11/75)
WordSD	238.46	312.38	3.18 (=238.46/75)	4.17 (=312.38/75)
GrepSort	37.38	62.06	0.50 (=37.38/75)	0.83 (=62.06/75)
GrepSearch	173.92	205.94	2.32 (=173.92/75)	2.75 (205.94/75)

The normalized job starting time illustrated in Fig. 6(a) show that the ABS-YARN cluster follows the same trend as the YARN cluster. However, as more jobs were submitted, their starting time in ABS-YARN were later than those in the YARN cluster. The average time difference is around 19.48 with standard deviation of 12.92. The key reasons are two. First, the normalized map-task (reduce-task) cost annotations used by ABS-YARN were based on average map-task (reduce-task) execution time of the entire workload, which were longer than the actual map-task (reduce-task) execution time spent by the real YARN cluster in the early phase of the workload execution. Second, the number of available containers gradually decreased when more jobs were submitted to the ABS-YARN cluster. For these two reasons, the starting time of the subsequent jobs were delayed.

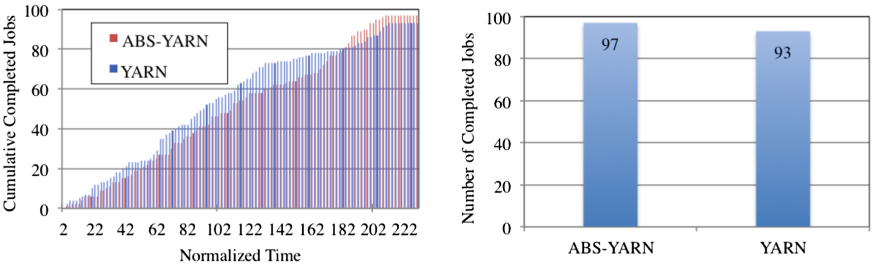


(a) The normalized starting time (b) The normalized finish time

Fig. 6. The time points of all jobs in the exponential scenario.

Figure 6(b) depicts the normalized job finish time of the two clusters under the exponential scenario. We can see that during the workload execution, many jobs in the ABS-YARN cluster finished later than the corresponding jobs in the YARN cluster. The reasons are the same, i.e., the map-task (reduce-task) cost annotation values were derived from the corresponding average map-task (reduce-task) execution time, which were usually higher than the actual execution time in the YARN cluster during the early stage of the workload. Nevertheless, the results show that even under a heavy and dynamic workload, the ABS-YARN framework can still adequately model YARN.

The cumulative number of completed jobs illustrated in Fig. 7(a) shows that during most of the workload execution, the ABS-YARN cluster finished fewer jobs than the YARN cluster for the above mentioned reasons. However, in the late stage, the ABS-YARN cluster had more completed jobs than the YARN cluster. This phenomenon can also be deduced from Fig. 6 since seven jobs could not complete by the YARN cluster. The average difference of the cumulative workload completion between ABS-YARN and YARN is 14.49 %. Due to failing to get containers, 97 jobs and 93 jobs (as shown in Fig. 7(b)) were finished by the ABS-YARN cluster and the YARN cluster, respectively. Although the job



(a) Cumulative completed jobs (b) Total number of completed jobs

Fig. 7. The cumulative completed jobs and the total number of completed jobs in the exponential scenario.

completion error of ABS-YARN is increased to 4.3 % from the uniform scenario to the exponential scenario, the above results still demonstrate that the ABS-YARN framework provides a satisfiable modeling for YARN.

5 Related Work

General-purpose modeling languages provide abstractions where the main focus has been on describing functional behavior and logical composition. However, this is inadequate for virtualized systems such as clouds when the software's deployment influences its behavior and when virtual processors are dynamically created. A large body of work on performance analysis using formal models can be found based on, e.g., process algebra [9], Petri Nets [26], and timed and probabilistic automata [4,8]. However, these works mainly focus on non-functional aspects of embedded systems without associating capacities with locations. A more closely related technique for modeling deployment can be found in an extension of VDM++ for embedded real-time systems [28], in which static architectures are explicitly modeled using buses and CPUs with fixed resources.

Compared to these languages, Real-time ABS [10,18] provides a formal basis for modeling not only timed behavior but also dynamically created resource-constrained deployment architectures, which enables users to model feature-rich object-oriented distributed systems with explicit resource management at an abstract yet precise level. Case studies validating the formalization proposed in Real-Time ABS include Montage [17] and the Fredhopper Replication Server [3]. Both case studies address resource management in clouds by combining simulation techniques and cost analysis. Different from these case studies, this paper uses Real-Time ABS to create a formal framework for YARN and comprehensively compare this framework with a real YARN cluster.

In recent years, many simulation tools have been introduced for Hadoop, including MRPerf, MRSim, and HSim. MRPerf [29] is a MapReduce simulator designed to understand the performance of MapReduce jobs on a specific Hadoop parameter setting, especially the impact of the underlying network topology, data locality, and various failures. MRSim [15] is a discrete event based MapReduce simulator for users to define the topology of a cluster, configure the specification of a MapReduce job, and simulate the execution of the job running on the cluster. HSim [21] models a large number of parameters of Hadoop, including nodes, cluster, and simulator parameters. HSim also allows users to describe their own job specification. All the above-mentioned simulators target Hadoop rather than YARN. Due to the fundamental difference between Hadoop and YARN, these simulators are unable to simulate YARN. Besides, these simulators concentrate on simulating the execution of a single MapReduce job and compare the corresponding simulation results with the actual results on real Hadoop systems. However, this is insufficient to confirm that they can faithfully simulate Hadoop when multiple jobs are running on Hadoop. Similar work can also be found in [19]. Different from all these simulators, the proposed ABS-YARN framework is designed to model a set of jobs running on YARN, rather than just one job. With ABS-YARN, users can comprehend the performance of YARN under a dynamic workload.

To our knowledge, the Yarn Scheduler Load Simulator (SLS) [31] is the only simulator currently designed for YARN, but it concentrates on simulating job scheduling in a YARN cluster. Besides, SLS does not provide any performance evaluation to validate its simulation accuracy. Compared with SLS, ABS-YARN provides a formal executable YARN environment. In this paper, we also present a comprehensive validation to demonstrate its applicability.

6 Conclusion and Future Work

This paper has presented the ABS-YARN framework based on the formal modeling language Real-Time ABS. ABS-YARN provides a generic model of YARN by capturing the key components of a YARN cluster in an abstract but precise way. With ABS-YARN, modelers can flexibly configure a YARN cluster, including cluster size and resource capacity, and determine job workload and job inter-arrival patterns to evaluate their deployment decisions.

To increase the applicability of formal methods in the design of virtualized systems, we believe that showing a strong correlation between model behaviors and real system results is of high importance. We validated ABS-YARN through a comprehensive comparison of the model-based analyses with the actual performance of a real YARN cluster. The results demonstrate that ABS-YARN is accurate enough to offer users a dependable framework for making deployment decisions about YARN at design time. In addition, the provided abstractions enable designers to naturally model and design virtual systems at this complexity, such as enhancing YARN with new algorithms.

In future work, we plan to further enhance ABS-YARN by incorporating multi-queue scheduler modeling, slave and container failure modeling, and distributed file-system modeling. Modeling different job types will also be considered. Whereas this paper has focussed on the accuracy of the ABS-YARN framework, our ongoing work on a more powerful simulation and visualization tool for Real-Time ABS will improve the applicability of ABS-YARN.

Acknowledgement. The authors thank NCLab at National Chiao Tung University, Taiwan for providing computation facilities for the YARN cluster used in our experiments.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
2. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014)

3. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: An industrial case study using Real-Time ABS. J. Serv. Oriented Comput. Appl. **8**(4), 323–339 (2014)
4. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004)
5. Apache Hadoop. http://hadoop.apache.org/
6. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
7. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, Raleigh (2007)
8. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)
9. Barbanera, F., Bugliesi, M., Dezani-Ciancaglini, M., Sassone, V.: Space-aware ambients and processes. Theor. Comput. Sci. **373**(1–2), 41–69 (2007)
10. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia, S.L.: Tarifa.: User-defined schedulers for real-time concurrent objects. Innov. Syst. Softw. Eng. **9**(1), 29–43 (2013)
11. Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer, New York (2005)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
13. enwiki. http://dumps.wikimedia.org/enwiki/
14. Hähnle, R., Johnsen, E.B.: Designing resource-aware cloud applications. IEEE Comput. **48**(6), 72–75 (2015)
15. Hammoud, S., Li, M., Liu, Y., Alham, N.K., Liu, Z.: MRSim: A discrete event based MapReduce simulator. In: Seventh International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2010, pp. 2993–2997. IEEE (2010)
16. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
17. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling resource-aware virtualized applications for the cloud in real-time ABS. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 71–86. Springer, Heidelberg (2012)
18. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebraic Methods Program. **84**(1), 67–91 (2015)
19. Kolberg, W., Marcos, P.D.B., Anjos, J.C., Miyazaki, A.K., Geyer, C.R., Arantes, L.B.: MRSG - a MapReduce simulator over SimGrid. Parallel Comput. **39**(4), 233–244 (2013)
20. Koralov, L.B., Sinai, Y.G.: Theory of Probability and Random Processes. Springer-Verlag, Berling (2007)
21. Liu, Y., Li, M., Alham, N.K., Hammoud, S.: HSim: a MapReduce simulator in enabling cloud computing. Future Gener. Comput. Syst. **29**(1), 300–308 (2013)

22. Luo, C., Zhan, J., Jia, Z., Wang, L., Lu, G., Zhang, L., Xu, C.-Z., Sun, N.: Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications. Front. Comput. Sci. **6**(4), 347–362 (2012)
23. Murthy, A., Vavilapalli, V., Eadline, D., Niemiec, J., Markham, J.: Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2. Addison-Wesley Professional, San Francisco (2014)
24. Palanisamy, B., Singh, A., Liu, L., Bryan, L.: Cura: A cost-optimized model for MapReduce in a cloud. In: IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 1275–1286. IEEE (2013)
25. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Towards understanding heterogeneous clouds at scale: Google traceanalysis.Technical Report ISTC-CC-TR-12-101, Intel Science and TechnologyCenter for Cloud Computing, Carnegie Mellon University, April 2012. http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/ISTC-CC-TR-12-101.pdf
26. Sgroi, M., Lavagno, L., Watanabe, Y., Sangiovanni-Vincentelli, A.: Synthesis of embedded software using free-choice petri nets. In: Proceedings of the Design Automation Conference, DAC 1999, pp. 805–810. ACM (1999)
27. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache Hadoop YARN: yet another resource negotiator. In: Lohman, G.M. (ed.) ACM Symposium on Cloud Computing (SOCC 2013), pp. 5:1–5:16 (2013)
28. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
29. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: A simulation approach to evaluating design decisions in MapReduce setups. In: IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2009, pp. 1–11. IEEE (2009)
30. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. J. Softw. Tools Technol. Transf. **14**(5), 567–588 (2012)
31. Yarn Scheduler Load Simulator (SLS). https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html