Technical Report RT/18/2014

An Elastic Middleware Platform for Concurrent and Distributed Cloud and Map-Reduce Simulation-as-a-Service

Pradeeban Kathiravelu Luis Veiga
INESC-ID/IST, Rua Alves Redol N^{o} 9, 1000-029 Lisboa, Portugal

May 2014

An Elastic Middleware Platform for Concurrent and Distributed Cloud and Map-Reduce Simulation-as-a-Service

Pradeeban Kathiravelu
INESC-ID Lisboa
Instituto Superior Técnico,
Universidade de Lisboa
Lisbon, Portugal
pradeeban.kathiravelu@tecnico.ulisboa.pt

Luis Veiga
INESC-ID Lisboa
Instituto Superior Técnico,
Universidade de Lisboa
Lisbon, Portugal
luis.veiga@inesc-id.pt

ABSTRACT

Middleware platforms empower the developers and users with an additional layer, enhancing scalability and performance. Cloud computing offers variety of resources and involves a tremendous amount of entities. Due to the limited access, along with varying availability of cloud resources, prototypes are often tested against cloud simulation environments. While computing has been evolving with multicore programming, MapReduce paradigms, and middleware platforms, cloud and MapReduce simulations still fail to exploit these developments themselves. This paper describes the research for the design, development and evaluation of a complete fully parallel and distributed cloud simulation platform ($Cloud^2Sim$), which tries to fill the gap between current simulations and the actual technology that they are trying to simulate.

First, $Cloud^2Sim$ provides a concurrent, distributed and elastic cloud simulator, by extending CloudSim cloud simulator, using Hazelcast in-memory key-value store. Then, it also provides an assessment of the MapReduce implementations of Hazelcast and Infinispan, as well as elastically distributing their execution in a cluster, providing means of simulating MapReduce executions. A dynamic scaler solution scales out the cloud and MapReduce simulations to multiple nodes running Hazelcast and Infinispan, based on load. The distributed execution model and adaptive scaling solution could further be leveraged as a general purpose auto-scaler middleware for a multi-tenant deployment.

ïż£

1. INTRODUCTION

Cloud simulations are used in evaluating architectures, algorithms, topologies, and strategies that are under research and development, tackling many issues such as resource management, application scheduling, load balancing, workload execution, and optimizing energy consumption. While the exact environment of the cloud platform may not be accessible to the developers at the early stages of development, simulations give an overall idea on the related parameters, resource requirements, performance, and output. With the increasing complexity of the systems that are simulated, cloud simulations are getting larger, and the larger simulations tend to take longer and longer to complete being run in a single node. Also, cloud simulation environments require a considerable amount of memory and processing power to simulate a complex cloud scenario. Processors are increasingly becoming more powerful with multi-core architectures and the computing clusters in the research laboratories themselves could be used to run complicated large simulations in a distributed manner. However, current simulation tools provide very limited support to utilize these resources, as they are mostly written with a sequential execution model targeting to run on a single server.

Java In-memory data grids provide a distributed execution and storage model for problems in the grid-scale. They offer scalability and seamless integration with persistent storage. Hazelcast [16], Infinispan [25], and Terracotta BigMemory[1] are some of the currently most used platforms for distributed execution and storage [13]. Using these platforms, users could create data grids and distributed caches on utility computers, to execute much larger jobs that could not run on any single computer, or that would take a huge time to execute with slow response. Functionality and scalability of the cloud simulators could thus be extended using in-memory data grids solutions. Existing cloud simulators also lack the ability to simulate MapReduce tasks, while there are simulators just specific to MapReduce. However, a MapReduce simulator could be implemented along with the cloud simulator, to simulate complex scenarios involving MapReduce tasks and cloud applications, such as load balancing the MapReduce tasks into different datacenters and power-aware resource scheduling. Cloud and MapReduce simulations can be executed on top of in-memory data grids, that execute over the computer clusters.

Public resource sharing or cycle sharing models allow acquiring resources for computing and storage from the volunteers for the tasks that are heavy in such requirements. This volunteer computing paradigm enables distributed execution of embarrassingly parallel jobs on the private computers of geographically distributed volunteers. Specific CPU and memory intensive research areas have utilized the volunteer computing model, where millions of volunteers offer their computer resources, while they are idle. BOINC (Berkeley Open Infrastructure for Network Computing) is a software that enables scientists to operate on public resource sharing model [1]. It is a server, client, and statistics system, that is later used by SETI@home and other projects [5]. SETI (Search for Extraterrestrial Intelligence) [2], Folding@home [31, 5], and Gnome@home [20], tackle problems of different domains, with geographically distributed computing resources provided by the volunteers. Condor is a scheduling system that maximizes the utilization of the workstations. Under-utilized or idling workstations offer their computing resources to the workstations that are overloaded. This resource sharing increases the overall productivity of the research labs or the cluster of workstations [21].

Exploiting the existing simulation approaches that are heav-

[1] http://terracotta.org/products/bigmemory

ily centralized, and the distributed execution platforms, cloud simulations can be made distributed, such that they would be able to utilize the computer clusters in the research labs. Distributed simulations could enable larger simulations to execute in a shorter time with a better response, whilst making it possible to simulate scenarios that may not even be possible on a single instance. Utilizing distributed computers to share the cycles to the simulation, as required by the simulation, would enable simulating larger and more complex scenarios that could not be simulated effectively in a single node, or it could be a very time consuming execution. While cycle sharing and volunteer computing is used in scientific research and grid computing projects, the cycle sharing model is not utilized to provide computing resources for cloud simulations. Moreover, when the resource providers are inside a trusted private network such as a research lab, security concerns related to cycle sharing could be considered lightly. Hence, the cycle sharing model can be leveraged to operate in a private cluster to provide a scalable middleware platform.

This paper describes first *Cloud*^{2}*Sim*, an adaptively scaling middleware platform for concurrent and distributed cloud simulator, by leveraging CloudSim [8, 7] as the core module, whilst taking advantage of the distributed shared memory provided by Hazelcast and in-memory key-value data grid of Infinispan. The Hazelcast based distributed simulator is implemented along with prototype deployments and samples. Infinispan is integrated into CloudSim, such that it can be used to implement the middleware platform to scale the simulator. Second, we describe two distributed implementations of a MapReduce simulator, leveraging centralized Hazelcast and Infinispan MapReduce implementations as core modules. Cycle sharing of the instances in the cluster, inspired by volunteer computing, is used as the model to achieve a scalable, adaptive, and elastic middleware platform for all the simulations. Hazelcast and Infinispan are integrated into core CloudSim as a compatibility layer for a seamless integration and invocation of cloud simulations. The whole simulation platform is implemented as a scalable middleware platform for cloud and MapReduce simulations, but it can be extended for other applications as well. Therefore, being elastic and adaptive, and thus cloud-ready, *Cloud*^{2}*Sim* can be the basis of a concurrent and distributed Simulation-as-a-Service for Cloud and MapReduce simulations.

In the upcoming sections, we will further analyze the proposed adaptively scaling middleware platform for the simulations in a distributed and concurrent manner. Section 2 will address background information on cloud and MapReduce simulators, and distributed execution frameworks. Section 3 discusses the solution architecture of *Cloud*^{2}*Sim*, the proposed middleware platform, and how CloudSim is enhanced and extended as to become a distributed and concurrent cloud simulator. Section 4 deals with the implementation details of *Cloud*^{2}*Sim*. *Cloud*^{2}*Sim* was benchmarked against CloudSim and was evaluated on multiple nodes, with results discussed in Section 5. Finally, Section 6 closes the paper discussing the research current state and future enhancements.

ïż£

2. RELATED WORK

2.1 Cloud Simulators

CloudSim [8], EmuSim [6], and DCSim [32] are some of the mostly used cloud simulation environments. OverSim [4] and PeerSim [26] are simulation toolkits for overlay and peer-to-peer networks respectively. GridSim, a Grid Simulation tool, was later extended as CloudSim, a Cloud Simulation environment [7]. It is capable of simulating application scheduling algorithms, power-aware data centers, and cloud deployment topologies. It has been extended into different simulation tools such as CloudAnalyst [37], WorkflowSim [11], and NetworkCloudSim [15]. Simulation environments have a trade-off of accuracy/speed [33], with faster less-accurate simulators and slower accurate simulators. Researchers focus on enhancing the speed and accuracy of existing simulators. Extensions to CloudSim tend to address its limitations or add more features to it. NetworkCloudSim enables modeling parallel applications such as MPI and workflows in CloudSim [15]. WorkflowSim simulates scientific workflows, through a higher level workflow management layer [11].

Table 1: Comparison of Cloud Simulators

	CloudSim	**SimGrid**	**GreenCloud**
Programming Language(s)	Java	C	C++ and TCL (Tool Command Language)
User Interface	Console	Console	Graphical
Features			
Grid Simulations	✓	✓	✓
Cloud Simulations	✓	✓	✓
Application Scheduling	✓	✓	X
Modeling Data Centers	✓	✓	✓
Modeling Energy-Aware Computational Resources	✓	✓	✓
P2P Simulations	X	✓	X
MPI Simulations	X	✓	X
Packet-level Simulations	X	With ns-3	With ns-2
Modeling and Simulation of Federated Clouds	✓	✓	X

SimGrid [9] is a toolkit initially developed for simulation of application scheduling. As a generic versatile simulator for large scale distributed computing, SimGrid offers two APIs, for researchers and developers respectively [10]. SimGrid approximates the behavior of the TCP networks, using a flow-level approach [14]. GreenCloud is a packet level simulator that simulates energy-efficient cloud data centers [18]. The Network Simulator - ns-2, a real-time network emulation tool [24], is extended for simulating energy-aware clouds. VM power management and migration in Internet Data Centers (IDC) are modeled, using Xen[2] as the VMM [22].

Optimizing the energy consumption is one of the major targets in cloud systems. Simulating energy-aware solutions has become part of the cloud simulators such as CloudSim, to optimize the energy consumption of data centers. Simulators are also developed exclusively for power systems. Internet technology based Power System Simulator (InterPSS) is a distributed and parallel simulation tool for optimizing and enhancing the design, analysis, and operation of power systems [17]. Cloud simulators have overlapping features, while some of the features are specific to only a few simulators. While some simulators are quite generic, others tend to be more focused. A comparison of three cloud simulators, CloudSim, SimGrid, and GreenCloud is presented by Table 1.

MapReduce Simulators. As MapReduce applications are becoming widespread, the need to simulate them in order to study their performance, efficiency, scalability, and resource requirements became apparent. Some MapReduce simulators were built from scratch, while some were developed on top of existing simulation frameworks of cloud or network. We will look at some of the MapReduce simulators below. SimMR is a MapReduce simulator that can replay the tasks from the logs of the real workloads produced by Hadoop, executing the tasks within 5% of the time the MapReduce task originally takes to execute [34]. MRPerf is a simulator of the MapReduce implementation of Hadoop, built using ns-2 [35]. Job execution time, amount of data transferred, and time taken for each of the phases of the job are output from the sim-

[2] http://www.xenproject.org/

ulator [36]. HSim, another Hadoop MapReduce simulator, following the same design paradigm of MRPerf, claims to improve the accuracy of MapReduce simulations for the complex Hadoop applications [23]. MRSG is a MapReduce Simulator built on top of SimGrid, providing APIs to prototype MapReduce policies and evaluate the algorithms [19]. Since MapReduce tasks are often run on bigger clusters, energy becomes a most important concern to address. BEEMR (Berkeley Energy Efficient MapReduce) is a MapReduce workload manager that is energy efficient [12]. High memory and processing power requirements of MapReduce executions as well as the simulations indicate that a scalable platform can benefit the execution.

2.2 Distributed Execution

Multiple Java in-memory data grids exist, both open source and commercial. Hazelcast and Infinispan are two of the open source in-memory grids that are used in research.

Hazelcast is a distributed in-memory data grid that provides the distributed implementations for the java.util.concurrent package [16]. Computer nodes running Hazelcast can join or create a Hazelcast cluster using either multicast or tcp-ip based join mechanisms. Additionally, Amazon web service EC2 instances with a hazelcast instance running, can use the Hazelcast/AWS join mechanism to form a Hazelcast cluster as well. Multiple Hazelcast instances can also be created from a single node by using different ports, hence providing a distributed execution inside a single machine. As Hazelcast distributes the objects to remote JVMs, the distributed objects should be serializable, or custom serializers should be developed and registered for each of the classes that are distributed. Hazelcast custom serialization requires the classes to be serialized to have public setters and getters for the properties that should be serialized. Hazelcast is partition-aware, and exploiting its partition-awareness, related objects can be stored in the same instance, reducing data transmission and remote invocations. Hazelcast supports both synchronous and asynchronous backup for fault tolerance. Hazelcast has been already used in research to distribute the storage across multiple instances [27].

Infinispan is a distributed key/value data-grid [25]. As an in-memory data-grid, Infinispan has been used in many researches. Palmieri et al have developed a self-adaptive middleware platform to provide transactional data access services, based on the in-memory data management layer of Infinispan [28]. JBoss RHQ [3] provides an enterprise management solution for Infinispan as well as the other projects from JBoss, which can be used to monitor the state and health of the Infinispan distributed cache instances. Infinispan offers JCache[4] and MemCached[5] APIs. Goal-oriented self-adaptive scalable platforms are researched using Infinispan as an in-memory persistence and cache solution [29].

Typically, the simulators are sequential, and run on a single computer, where computer clusters and in-memory data grids can be leveraged to execute larger simulations that cannot be executed on a single computer.

ïż£

3. SOLUTION ARCHITECTURE

As designed to run top of a cluster, *Cloud*^{2}*Sim* attempts to execute larger and more complicated simulations that would not run on a single node or terminal, or consume huge amount of time. In-memory data grid libraries are used to form a cluster. The user can form an in-memory data grid on the computer cluster. Once the data grid is formed, simulations are executed on the cluster, utilizing the resources such as storage, processing power, and memory, provided by the individual nodes, as indicated by Figure 1. Hazelcast and Infinispan are used as the in-memory data grid libraries.

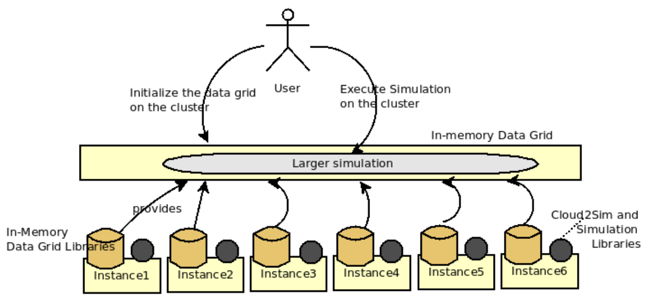

User
Initialize the data grid on the cluster
Execute Simulation on the cluster
In-memory Data Grid
Larger simulation
provides
In-Memory Data Grid Libraries
Cloud2Sim and Simulation Libraries
Instance1
Instance2
Instance3
Instance4
Instance5
Instance6

Figure 1: High Level Use-Case of *Cloud*^{2}*Sim*

Cloud^2*Sim* functions in two basic modes, as a concurrent and distributed simulator, for cloud simulations and for MapReduce simulations. It was decided to extend an existing cloud simulator to be concurrent and distributed, instead of writing a new cloud simulator from scratch. Developed as a Java open source project, CloudSim could be modified by extending the classes, with a few changes to the CloudSim core. Its source code is open and maintained. Hence, CloudSim was picked as the core module to build *Cloud*^{2}*Sim*. Cloud simulation further uses Hazelcast to distribute the storage of VM, Cloudlet, and datacenter objects and also to distribute the execution according to the scheduling to the instances in the cluster. Users have the freedom to choose Hazelcast based or Infinispan based distributed execution for the cloud and MapReduce simulator, as the simulator is implemented on top of both platforms following the same design. Classes of CloudSim are extended and a few are also modified to be able to extend CloudSim with further functionality. External dependencies such as Hazelcast and Infinispan are used unmodified. Cloud and MapReduce simulations are independent by design. Cloud and MapReduce simulations can be executed independently, though experiments could be run utilizing both cloud and MapReduce simulations.

3.1 Concurrent and Distributed Middleware Platform for Simulations

As multiple instances execute the single simulation, measures are taken to ensure that the output is consistent as simulating in a single instance, while having enhanced performance. Data is partitioned across the instances by leveraging Hazelcast and Infinispan. Each member of the cluster executes part of the logic on the objects that are stored in the local partitions of the respective nodes. Execution of simulations is leveraged, utilizing the multi-core environments, and exploiting multi-thread programming.

While CloudSim provides some means for a concurrent execution, its support is very limited. Applications should be written utilizing the multi-thread environments, while the simulator itself should run the tasks concurrently, whenever that is possible and efficient. Runnables and callables are used to submit tasks to be run in a separate thread, while the main thread is executing its task. The relevant checkpoints ensure that the threads have finished their execution and the values are returned from the callables, as required.

Instances form a cluster with the same cluster group. Multiple clusters could be used to execute parallel cloud or MapReduce simulations, as multiple tenants of the nodes. As each cluster is unaware of the other clusters, tenant-awareness is ensured that the parallel experiments can be independent and

[3] http://rhq.jboss.org/
[4] https://jcp.org/en/jsr/detail?id=107
[5] http://memcached.org/

secured from the other parallel simulations. Partitioning of data and execution is done by 3 different approaches, as listed below.

1. Simulator - Initiator based approach. Simulator is the complete $Cloud^2Sim$ with the simulation running. Hazelcast instance is started by $Cloud^2Sim$ Initiator, which keeps the computer node connected to the Hazelcast cluster, offering the resources of the node to the data grid. Simulator instance is run from the master instance, where an instance of Initiator is run from the other instances. Simulator acts as the master, distributing the logic to the Initiator instances. Part of the logic is executed in the master itself, and the execution is partitioned evenly among all the instances, using the ExecutorService. This approach is used for the tasks that are effectively scheduled by the single master to all the instances that are joined, such as the MapReduce simulator.

2. Simulator - SimulatorSub based approach. One instance contains the Simulator, which is the master, where others run SimulatorSub, which are the slave instances. Master coordinates the simulation execution. However, unlike the Simulator - Initiator based approach, execution is started by all the instances. Some of the non-parallelizable tasks could be delegated to the primary worker, which is an instance other than the master instance, that is decided upon the cluster formation. This mitigates overloading the master instance. This is implemented in the specific simulations where master and worker instances perform completely different tasks, when it is not effective to unify them to dynamically allocate the responsibilities.

3. Multiple Simulator instances approach. This is similar to the Simulator - SimulatorSub based approach, except that there is no specific Simulator master. The instance that joins first becomes the master at run time. Data is partitioned evenly across the instances, as in the previous cases. Logic is partitioned across the instances using the Partitioning algorithms defined in $Cloud^2Sim$. PartitionUtil manages the partitioning of the data and execution of the data structures across the instances. It provides the initial and final IDs of the data structure such as cloudlets and VMs, given the total number of structures and the initial offset.

Multi-tenancy in $Cloud^2Sim$. A multi-tenanted experiment executes over a deployment, composed of multiple clusters of (Hazelcast) instances, across multiple physical nodes. A tenant is a part of the experiment, represented by a cluster. An instance is attached to a single cluster, and is tenant-specific. Data specific to a tenant is stored in its instances of the cluster. The specific instance to store is defined by the $Cloud^2Sim$ design, leveraging and configuring the HazelcastInstanceAware and PartitionAware features to decide the optimal instance.

A coordinator node has instances in multiple clusters and hence enables sharing information across the tenants through the local objects of the JVM. Due to the 1:1 mapping between a cluster and a tenant, a tenant may not span across multiple clusters. This does not cause under-utilization, as multiple clusters can co-exist in and utilize the same nodes. Fault-tolerance is easily ensured by enabling synchronous backups, by just changing the configuration file. Thus, even if a node goes down, the tenants will not suffer.

Figure 2 depicts a sample deployment of 6 nodes configured into 7 clusters to run 6 experiments in parallel. Both cluster1 and cluster3 contain 2 nodes - the Master/Supervisor and one Initiator instance, running an experiment. Cluster2 contains 3 nodes, with the third node having 2 Initiator instances running. Cluster4 contains an instance of Initiator, ready to join a simulation job, when the Master instance joins. Cluster5 consists of node4, which hosts both Initiator and Master instances. Cluster6 contains node5 and node6, both running Initiator instances. Node1 hosts 2 Master instances, one in cluster1, and the other in cluster2. It also hosts an Initiator instance in cluster3. It has a Coordinator running on cluster0. Coordinator coordinates the simulation jobs running on cluster1 and cluster2 from a single point, and prints the final output resulting from both experiments or tenants. This is done externally from the parallel executions of the tenants, and enables a combined view of multi-tenanted executions.

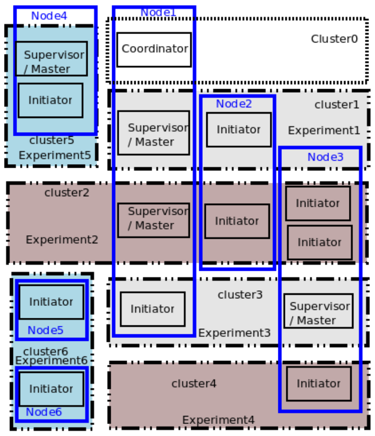


Figure 2: A Multi-tenanted Deployment of $Cloud^2Sim$

Cloud Simulations. $Cloud^2Sim$ is designed on top of CloudSim, where cloud2sim-1.0-SNAPSHOT could be built using maven independently without rebuilding CloudSim. Modifications to CloudSim are minimal. $Cloud^2Sim$ enables distributed execution of larger CloudSim simulations. The Compatibility layer of $Cloud^2Sim$ enables the execution of the same CloudSim simulations, either on top of the Hazelcast and Infinispan based implementations, as well as pure CloudSim distribution, by abstracting away the dependencies on Hazelcast and Infinispan. Partition-awareness features of Hazelcast are exploited in storing the distributed objects, such that the data that are associated with each other are stored in the same partition to minimize remote invocations. Distributed Executors pulled the logic to the data, reducing the communication costs.

MapReduce Simulations. Design of the MapReduce simulator is based on a real MapReduce implementation. A simple MapReduce application executes as the Simulator is started. The number of times map() and reduce() are invoked could easily be configured. MapReduce simulator is designed on two different implementations, based on Hazelcast and Infinispan, making it possible to benchmark the two implementations against each other. Multiple simulations are executed in parallel, without influencing others, where an instance of a coordinating class could collect the outputs from the independent parallel MapReduce jobs carried out by different clusters.

3.2 Scalability

$Cloud^2Sim$ achieves scalability through both static scaling and dynamic scaling. Static scaling is the scenario where $Cloud^2Sim$ uses the storage and resources that are initially made available, when instances are started and joined manually to the execution cluster. Multiple nodes can be started simultaneously at start-up time for large simulations that require high amount of resources. Initiator instances can also be started manually at a later time, to join the simulation that has already started. Simulations begin when the minimum number of instances specified have joined the simulation cluster. $Cloud^2Sim$ scales smoothly as more Hazelcast instances join the execution.

Scaling can also be done by $Cloud^2Sim$ itself dynamically without manual intervention, based on the load and simulation requirements. When the load of the simulation environment goes high, $Cloud^2Sim$ scales itself to handle the increased load. Dynamic scaling of $Cloud^2Sim$ provides an elastic, cost-effective, cloud-ready solution, instead of having multiple instances being allocated to the simulation even during the low load scenarios. Since scaling introduces the possibility of nodes joining and leaving the cluster, as opposed to the static execution or manual joins and exits of instances, scalable simulation mandates availability of synchronous backup replicas, to avoid losing the distributed objects containing the simulation data upon the termination of an instance.

A health monitor is designed to monitor the health of the instances, and trigger scaling accordingly. Health monitoring module runs from the master node and periodically checks the health of the instance by checking the process CPU load, system CPU load, and the load average. Based on the policies defined in the cloud2sim.properties file, the health monitor triggers the scaling action. During scale out, more instances are included into the simulation cluster, where scale in removes instances from the simulation cluster, as the opposite of scale out. Dynamic scaling is done in two modes - auto scaling and adaptive scaling, as discussed below.

Auto Scaling. By default, the auto scaler spawns new instances inside the same node/computer. This feature is available out-of-the-box for Hazelcast paid/enterprise versions. When the availability of clusters is limited to simulate a large scenario, $Cloud^2Sim$ can be run on an actual cloud infrastructure. Hazelcast can be configured to form a cluster on Amazon EC2 instances, with the Hazelcast instances running on the same AWS[6] security group. When using AWS join mechanism provided by Hazelcast to form the cluster, Hazelcast uses the access key and secret key to authorize itself into forming the cluster. If no AWS security group is mentioned, all the running EC2 instances will be tried, where mentioning a security group will limit the search to only the instances of the same security group. Ports involved in the Hazelcast clustering should be allowed in the EC2 instances. Scaling can be done using $Cloud^2Sim$ health monitoring or with scaling policies configured with AWS Auto Scaling and Amazon Cloud Watch.

Adaptive Scaling. Adaptive Scaling is a scenario where, in a clustered environment, more computer nodes will be involved in an application execution based on the load. More instances will be attached to the simulation cluster when the load is high, and instances will be detached or removed from simulation when the load is low. Scaling decisions are made in a separate cluster. Figure 3 shows the execution of two independent simulations in a cluster with adaptive scaling.

Health monitor in the main instance (cluster-main) monitors its load and shares with the AdaptiveScalerProbe thread in cluster-sub, using the local objects, as they are from the same JVM. AdaptiveScalerProbe shares this information with IntelligentAdaptiveScaler (IAS) instances, which are threads from all the other 5 nodes that are connected to Sub-Cluster.

When IAS from one instance notices the high load in the master, it spawns an Initiator instance in the cluster-main, and sets the flag to false to avoid further scaling outs/ins. Monitoring for scaling out happens when there is no Initiator instance in the node, and monitoring for scaling in happens when there is an Initiator instance, for each individual node. This ensures 0 or 1 of Initiator instance in each node, and avoids unnecessary hits to the Hazelcast distributed objects holding the health information. Since IAS is in a separate cluster (cluster-sub) from the simulation (cluster-main), the executions are independent.

Adaptive Scaling is used to create prototype deployments. When the simulations finish, the Hazelcast instances running in the main cluster will be terminated, and the distributed objects stored in the sub cluster will be released. These instances just require Hazelcast and the adaptive scaler thread to keep them connected, providing their CPU and storage for the simulation work voluntarily, in a BOINC-like cycle sharing model. The entire simulation code could be kept only on the master node. All the member nodes are from the same network, that they have joined by tcp-ip or multicast. Hence the cycle sharing used in $Cloud^2Sim$ is not public as in voluntary computing. Due to this nature, the security implications involved in voluntary computing are not applicable for $Cloud^2Sim$. The scaling decision flag should be get and set in a concurrent and distributed environment atomically, ensuring that exactly one instance takes action of it. Access to the object that is used as the flag should be locked upon the access, from any other instance in the distributed environment.

Multiple Hazelcast clusters can execute from a single computer cluster or a single machine. Exploiting this feature, multiple experiments could be run on $Cloud^2Sim$ in parallel, as different clusters are used for independent simulations. Different simulations are initialized from the same node. The adaptive scaling solution could be extended to have the node cluster providing its resources to different applications or simulations running on different Hazelcast clusters, as they are not expected to know the application logic.

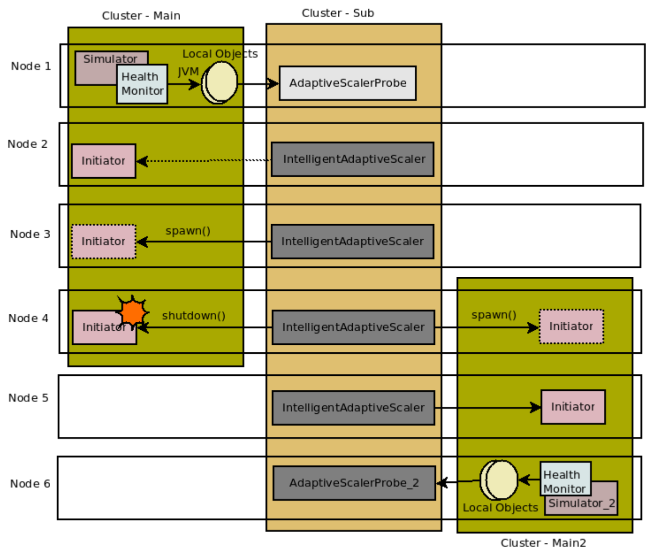


Figure 3: An Elastic Deployment of $Cloud^2Sim$

3.3 Software Architecture and Design

Distributed storage and execution for CloudSim simulations is achieved by exploiting Hazelcast. Infinispan integration with the compatibility layer ensures easy integration of Infinispan to replace Hazelcast as the in-memory data grid for CloudSim simulations. Figure 4 depicts a layered architecture

[6] https://aws.amazon.com/

Table 2: $Cloud^2Sim$ and CloudSim

$Cloud^2Sim$ Class	Extended CloudSim class	Core Responsibilities
HzCloudSim	CloudSim	* Core class of the Simulator * Initializes distributed data structures
HzDatacenterBroker	DatacenterBroker	* Implements distributed scheduling
$Cloud^2Sim$Engine	-	* Starts Simulation based on the configuration * Starts supportive threads for scaling and health monitoring
PartitionUtil	-	Calculates the partitions of the data structures
HzCloudlet	Cloudlet	* Extends Cloudlet
HzVm	Vm	* Extends Vm
HazelSim	-	* Singleton of Hazelcast integration
HzObjectCollection	-	* Provides unified access to distributed objects

overview of $Cloud^2Sim$, hiding the fine architectural details of the original CloudSim.

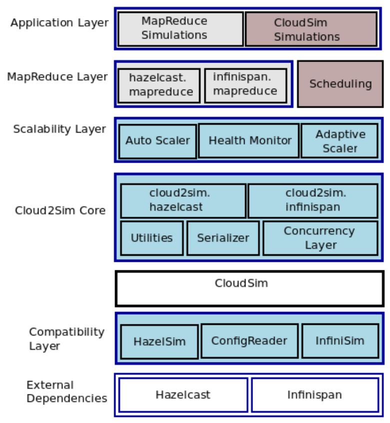


Figure 4: $Cloud^2Sim$ Architecture

Hazelcast monitoring and heart beats are run on a separate thread, hence not interfering with the main thread that runs the simulations. Simulation objects, cloudlets and VMs were ported from Java lists to Hazelcast distributed maps. This enabled storing these objects in a distributed shared memory provided by Hazelcast spanning across the cluster. Instances of Hazelcast IMap are used as the data structure. The core CloudSim class, CloudSim is extended as HzCloudSim to address the Hazelcast specific initializations. Similarly, Cloudlet and Vm are extended as HzCloudlet and HzVm respectively. This extended class hierarchy enabled modifying the internals of Vm and Cloudlet classes by sub-classing them to use Hazelcast distributed maps as the storage data structure, instead of lists. As extending CloudSim, $Cloud^2Sim$ provides an API compatible with CloudSim, for the cloud simulations. Classes of CloudSim are extended as shown by Table 2.

Scheduling. Scheduling package handles scheduling in complex scenarios that involves searching large maps consisting of VMs, cloudlets, and the user requirements. Distributed application scheduling is done by the extended datacenter brokers that are capable of submitting the tasks and resources in a distributed manner.

Compatibility Layer. A new package named "compatibility" composed of the core classes such as HazelSim was left inside CloudSim to integrate Hazelcast, Infinispan, and other new dependencies, and to enable multiple modes of operation (Such as Hazelcast or Infinispan based and regular CloudSim simulations). HazelSim is the single class that is responsible for initiating the Hazelcast clusters and ensuring that the minimum number of instances are present in the cluster before the simulation begins. Hazelcast could also be configured programmatically for $Cloud^2Sim$ using HazelSim. HzObjectCollection provides access to the distributed objects such as Hazelcast maps. InfiniSim provides similar functionality for the Infinispan based distribution. $Cloud^2Sim$.properties is used to input MapReduce and CloudSim specific parameters such as the number of resources and users to be present in the simulation, such that simulations could be run with varying loads, without recompiling.

Clustering and Grouping. Hazelcast and Infinispan based clustering is done via TCP and UDP multicasting. When groups are formed inside a single node, UDP multicasting is used for detecting the instances and creating the cluster. Properties of the Hazelcast cluster such as whether the caching should be enabled in the simulation environment, and when the unused objects should be evicted from the instances are configured by hazelcast.xml, and infinispan.xml is used to configure the Infinispan cache.

When running across the nodes in a physical cluster, TCP based discovery is used, where the instances are predefined, in hazelcast.xml for Hazelcast based implementation, and in jgroups-tcp-config.xml for Infinispan based implementation. JGroups [3] is configured as the core group communication technology beneath Infinispan clustering, and the respective TCP or UDP configuration files are pointed from infinispan.xml.

The packages cloudsim.hazelcast and cloudsim.infinispan respectively integrate Hazelcast and Infinispan into the simulator. Concurrency layer consists of callables and runnables for asynchronous invocations to concurrently execute. As complex objects should be serialized before sending them to other instances over the wire, custom serializers are needed for Vm, Cloudlet, Host, Datacenter, and the other distributed objects. Utilities module provides the utility methods used throughout $Cloud^2Sim$.

MapReduce Layer. MapReduce Layer provides MapReduce representation and implementations based on Hazelcast and Infinispan MapReduce modules. Two different implementations for Hazelcast and Infinispan exist following the same design. MapReduce Simulator can be configured with health monitoring and scaling. Hence, the execution time for varying the number of map() and reduce() invocations, as well as the health parameters, such as load average and CPU utilization could be measured.

Dynamic Scaling. Auto scaling and adaptive scaling are implemented by the packages scale.auto and scale.adaptive. To prevent loss of information when the cluster scales in, synchronous backups are enabled by marking synchronous backup count as 1 in hazelcast.xml. This makes $Cloud^2Sim$ able to tolerate crashes, and avoid wasted work in long simulations, due to the availability of backups in different hazelcast instances. Hazelcast stores the backups in different physical machines, whenever available, to minimize the possibility of losing all the backups during a hardware failure of a computer node. Distributed objects are removed by the user simulations as appropriate at the end of simulations. This enables the Initiator instances to join the other simulations without the need to restart them.

Application Layer. Application layer provides sample cloud and MapReduce simulations, and structures that assist developing further simulations on top of them. The execution flow of a distributed simulation of an application scheduling scenario with Round Robin algorithm is shown by Figure 5. It shows the simulation utilizing the core modules of $Cloud^2Sim$, and CloudSim to execute in a distributed manner.

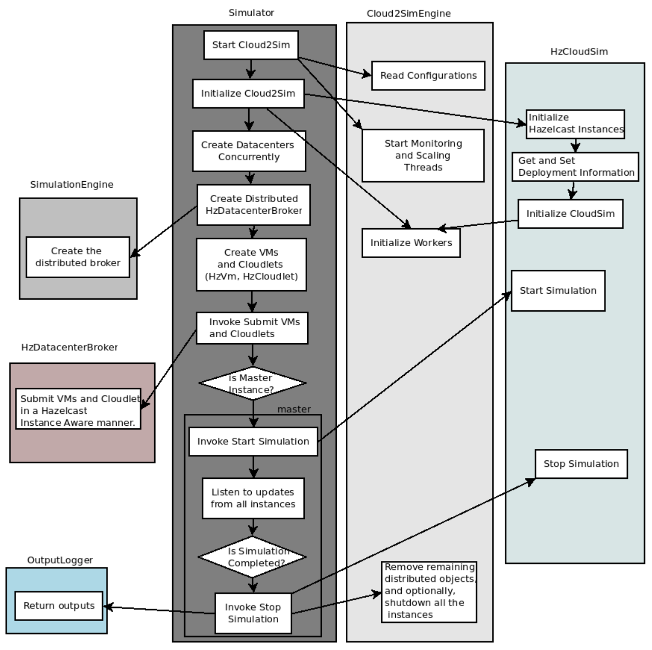

Simulator
Cloud2SimEngine
HzCloudSim
Start Cloud2Sim
Read Configurations
Initialize Cloud2Sim
Initialize Hazelcast Instances
Start Monitoring and Scaling Threads
Create Datacenters Concurrently
Get and Set Deployment Information
SimulationEngine
Create Distributed HzDatacenterBroker
Initialize CloudSim
Initialize Workers
Create the distributed broker
Create VMs and Cloudlets (HzVm, HzCloudlet)
Start Simulation
Invoke Submit VMs and Cloudlets
HzDatacenterBroker
Submit VMs and Cloudlets in a Hazelcast Instance Aware manner.
Is Master Instance?
master
Invoke Start Simulation
Stop Simulation
Listen to updates from all instances
Is Simulation Completed?
OutputLogger
Remove remaining distributed objects, and optionally, shutdown all the instances
Return outputs
Invoke Stop Simulation

Figure 5: Higher Level Execution flow of an application scheduler simulation

ïż£

4. IMPLEMENTATION

Based on the design, *Cloud*^{2}*Sim* was implemented as a concurrent and distributed cloud simulator using Hazelcast, by extending CloudSim.

4.1 Concurrent and Distributed Cloud Simulator

CloudSim simulations can run on *Cloud*^{2}*Sim* with minor changes to facilitate distribution. Distributing the simulation environment has been implemented using an incremental approach. CloudSim trunk version was forked and extended in the implementation. Hazelcast version 3.2 and Infinispan version 6.0.2 were used in the implementations and evaluations. JGroups is a reliable multicasting toolkit [3] that is used internally by Infinispan for clustering and grouping, and version 3.4.1 is used by the Infinispan 6.0.2. Built using Apache Maven, the project is hosted on SourceForge[7], with the Git distributed version control system.

Sample concurrent simulations were implemented, with concurrent data center creation, concurrent initialization of VMs and cloudlets, and submission of them to the brokers. Though the initialization of threads and executor frameworks introduced an overhead for small simulations, they provided a speed-up for the larger simulations. Very small simulations do not require distributed execution, as they perform reasonably well in a single node in a sequential manner. Simulations that fail to execute or perform poorly due to the processing power requirements on a single thread, perform much better on the concurrent environments utilized by *Cloud*^{2}*Sim*. Hence, the overheads imposed by the initializations is not a limitation to usability, as the performance gain is higher. Sample prototype developments show concurrent creation of data centers increases the performance, overcoming the overheads.

Hazelcast IExecutorService is utilized to make the execution distributed. Multiple instances are started and the first instance to join the cluster becomes the master and executes the core fractions of the logic which must not be distributed, decentralized, or run in parallel for a correct execution of the simulation. Callables and runnables were made to implement HazelcastInstanceAware interface, to ensure the members of the clusters executed part of the logic on the data partition that is stored in themselves. This minimizes remote invocation, by increasing data locality. As the CloudSim objects to be distributed are custom objects that cannot be serialized, custom serializers were written for them, extending Hazelcast StreamSerializer interface. The serializers are registered with the respective classes that they serialize using hazelcast.xml.

Partitioning of data and execution is calculated iteratively for each instance. The number of instances currently in the cluster is tracked by deploymentList, an instance of distributed map. An instance will have an offset value assigned to it, which is the number of instances that have joined previously. Hence the offset of the first instance will be zero and initial ID of the partition will be zero as well. Final ID of the instance that joins last, will be same as the last ID of the distributed data structure. The partition logic permits dynamic scaling, where the instances could join and leave during execution. This implementation enables easy integration of auto scaling implementations into the simulation. By default, the back up count is set to zero, though it is set to 1 when the dynamic scaling is enabled.

Cloud2SimEngine is started as the initial step of *Cloud*^{2}*Sim* cloud simulations, starting the timer and calls HzConfigReader to read the configurations. If health checks are enabled, health monitor thread starts to periodically monitor the instance status and report as configured. If adaptive scaling is enabled, AdaptiveScalerProbe is started in a separate thread, to communicate with the IntelligentAdaptiveScaler instances in the other nodes to adaptively scale the simulation. *Cloud*^{2}*Sim*Engine finally initializes HzCloudSim, where Hazelcast simulation cluster is initialized with the simulation job, and CloudSim simulation is started. Data centers are created concurrently. Brokers extending HzDatacenterBroker create instances of HzVm and HzCloudlet and start scheduling in a distributed manner, using all the instances in the simulation cluster. Core simulation is done by the master instance, invoking HzCloudSim.startSimulation(). When the simulation finishes, the final output is logged by the master instance. Based on the simulation implementation, the instances could either be terminated or the distributed objects are cleared and the instances are reset for the next simulation.

Scalable Middleware Platform. The health monitoring module exploits the system health information that can be retrieved using com.sun.management.OperatingSystemMXBean. It facilitates scaling based on a few parameters such as CPU load, and also provides an API to extend it further. Scaling policies are defined on the maximum and minimum thresholds of the defined properties, along with the maximum and minimum number of instances that should be present in the simulation cluster. Once a new instance is spawned, the adaptive scaler will wait for a user-defined period, which is usually longer than the time interval for health checks, for the next scaling action. This longer wait between scaling decisions prevents cascaded scaling where multiple instances are added or removed at once, or within a very short period of time interval, during the time taken for the scaling effect to be reflected on the simulation. The gap between the high and low thresholds is kept reasonably high, to prevent the jitter effect, where instances are added and removed frequently, as the high and low thresholds are frequently met. Health monitor configuration provides means to configure the scaling and monitoring to fit the application requirements and extend the module further to fine tune according to the application requirements.

[7]Checkout the source code at https://sourceforge.net/p/cloud2sim/code/ci/master/tree/, with user name, "cloud2sim" and password, "Cloud2Simtest".

Scaling decisions are made atomically in the distributed environment, to make the adaptive scaling work without simultaneously starting or shutting down multiple instances. This is implemented using the distributed flags, leveraging Hazelcast IAtomicLong data structure.

4.2 MapReduce Simulator

MapReduce Simulator has two different implementations, based on Hazelcast and Infinispan. A basic MapReduce application was implemented using Hazelcast and Infinispan and incorporated into $Cloud^2Sim$. Complex MapReduce scenarios were simulated using this small application. $Cloud^2Sim$ integrates MapReduce implementations with Hazelcast and Infinispan. Infinispan is integrated using the compatibility layer in CloudSim, to facilitate later distribution of CloudSim simulations with Infinispan. This also enables the same design and architecture for both Hazelcast and Infinispan based distributions. Infinisim in the compatibility layer configures the DefaultCacheManager of Infinispan, using the infinispan.xml pointed by $Cloud^2Sim$.properties. A transactional cache is created from the cache manager. An instance of cache in Infinispan is similar to the instances in Hazelcast. Hence, multiple instances of Cache form a cluster and execute the jobs. Cache instance initialized by the master node acts as the supervisor of the MapReduce jobs, and distributes the tasks across the Initiator instances.

InfJob and HzJob classes implement the Job interface. HzJob and InfJob get the job, and the real implementation is done by the classes in mapreduce.hazelcast.impl and mapreduce.infinispan.impl packages. The application behind the MapReduce simulations is a simple word count application that reads and counts big files that are stored in the folder pointed by the property loadFolder in cloud2sim.properties. This example implementation can be changed to different sample applications by the users. Huge text files such as the files collected from USENET Corpus were used [30]. Current implementation can stand as a working sample, following the CloudSim approach of providing examples.

Both Hazelcast and Infinispan based MapReduce implementations have an Initiator class that starts an instance of Hazelcast or Infinispan respectively and joins with the instance that runs the main simulation. While the HzMapReduceSimulator or InfMapReduceSimulator that runs from the master node coordinates and initiates the MapReduce jobs, the instances running Initiator join the cluster and do the equal share of the jobs. Master node hosts the supervisor of the MapReduce job. In the verbose mode, local progress of the individual map/reduce executions could be viewed from all the instances, where the final outcome is printed only to the master instance.

Customizing the MapReduce job invocations. In the simple MapReduce simulation, the number of invocations of map() is proportional to the number of files present in the load folder, or to a smaller number defined by the user. Similarly, reduce() is proportional to the number of lines in the file. By using duplicate files, invocations of map() are increased, keeping the reduce() invocations constant. Keeping the same number of files, and increasing the number of lines read, increases the reduce() invocations, keeping the map() invocations constant. By increasing both the size and number of files, both map() and reduce() invocations can be increased simultaneously.

This simple configuration helps to develop a sample MapReduce application with varying number of map() and reduce() invocations. More complex MapReduce applications can be visualized by this simulator, which is a simple MapReduce application implemented on Hazelcast and Infinispan. Scalability by increasing the number of physical nodes and its effects on the execution time and status change of the instances such as load average could be monitored. The execution flow of a MapReduce simulation using Infinispan implementation is shown by Figure 6. Hazelcast execution is similar, with the respective classes of Hazelcast implementation.

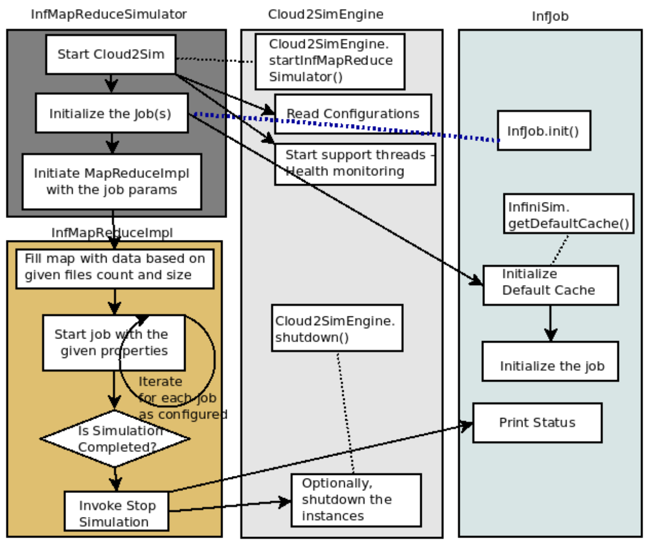

InfMapReduceSimulator
Cloud2SimEngine
InfJob
Start Cloud2Sim
Cloud2SimEngine. startInfMapReduce Simulator()
Initialize the Job(s)
Read Configurations
InfJob.init()
Start support threads - Health monitoring
Initiate MapReduceImpl with the job params
InfiniSim. getDefaultCache()
InfMapReduceImpl
Fill map with data based on given files count and size
Initialize Default Cache
Start job with the given properties
Cloud2SimEngine. shutdown()
Initialize the job
Iterate for each job as configured
Is Simulation Completed?
Print Status
Invoke Stop Simulation
Optionally, shutdown the instances

Figure 6: Execution flow of a MapReduce simulation with the Infinispan Implementation

5. EVALUATION

A cluster with 6 identical nodes (Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz and 12 GB memory) was used for evaluations. Multiple simulations were experimented on the system using 1 to 6 nodes. Each node executed one Hazelcast or Infinispan instance, except during the experiments involving auto scaling in a single node.

5.1 CloudSim Simulations

The master node always completed the last, as the final outcome is printed by the master node in the simulations considered. Time taken by the master node is noted down, as other nodes finished the execution before the master. nohup was used to start the process to avoid interrupts, and the output was directed to an output file called nohup.out.

Table 3 shows the time taken to simulate a round robin application scheduling simulation with 200 users, 15 datacenters, with and without workload for a varying number of VMs and cloudlets. CloudSim outperformed $Cloud^2Sim$ in the base execution without a workload, due to the inherent overheads involved in $Cloud^2Sim$. $Cloud^2Sim$ with multiple nodes showed a considerable 10-fold improvement in the execution time when the cloudlets contained a relevant workload to be simulated once scheduled. Time taken (in seconds) for an experiment in $Cloud^2Sim$ with 1, 2, 3, and 6 nodes as well as in CloudSim is depicted by Table 3 for 200 VMs and 400 cloudlets.

In a simulation where each cloudlet does a complex job, time taken for the simulation increases with the number of cloudlets. With the number of VMs fixed at 200, simulation time taken on 1 - 6 nodes was measured. Figure 7 depicts how the application scales with varying number of cloudlets.

5.2 Distributed Execution

The experiment was repeated with different combinations of VMs and Cloudlets, with and without a complex mathematical operation to be performed for each cloudlet as its

Table 3: Execution time (sec) for CloudSim Vs. *Cloud^{2}Sim*

Deployment	Simple Simulation	Simulation with a workload
CloudSim	3.678	1247.400
Cloud^{2}Sim (1 node)	20.914	1259.743
Cloud^{2}Sim (2 nodes)	16.726	120.009
Cloud^{2}Sim (3 nodes)	14.432	96.053
Cloud^{2}Sim (6 nodes)	20.307	104.440

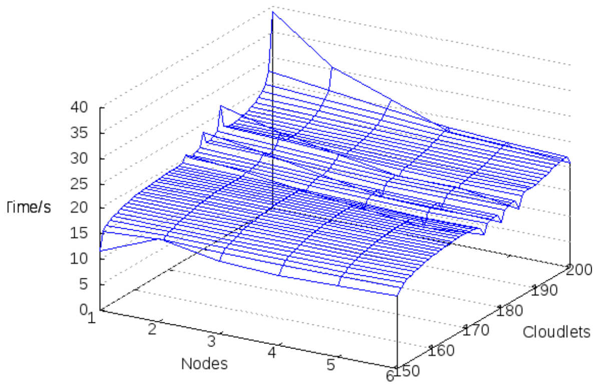


Figure 7: Simulation of Application Scheduling Scenarios

workload. Four different cases of scalability were noticed, as described below.

5.2.1 *Success Case (Positive Scalability)*

Figure 8 depicts the scenarios of (noOfVMs = 200, noOfCloudlets = 400, isLoaded = true) and (noOfVMs = 100, noOfCloudlets = 200, isLoaded = true), where the time taken for simulation is decreasing with the number of nodes. This is a desired scenario of scaling where the task is so much CPU intensive for each cloudlet to handle in a single node, such that introducing more nodes distribute the tasks, reducing the simulation time. As shown by Figure 8, time taken for the simulations is converging. Introducing further nodes beyond a certain maximum number of nodes may not be economically feasible, and at a point this may become the case 3, which is explained below.

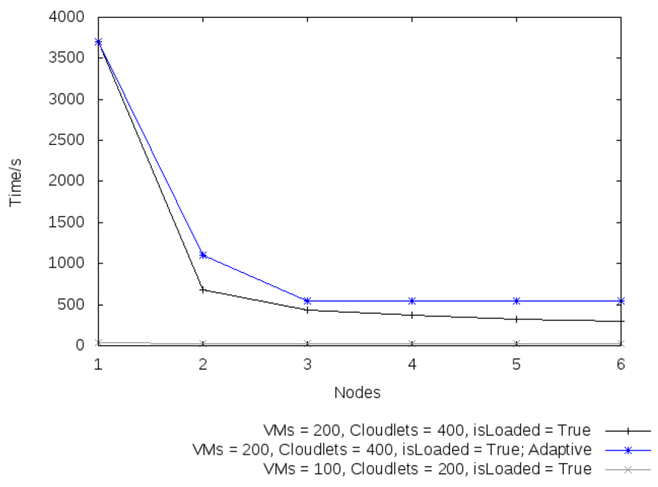


Figure 8: Distributed Execution - Positive Scalability

Table 4: Load averages with Adaptive Scaling on 6 nodes

No. of Instances	Master I0	I1	I2	Event
1	0.30	-	-	Spawning I1
2	0.30	0.24	-	Waiting Time
2	0.25	0.24	-	Spawning I2
3	0.23	0.23	0.13	Waiting Time
3	0.21	0.19	0.13	Monitoring
3	0.09	0.18	0.09	Monitoring
3	0.06	0.18	0.08	Monitoring

Dynamic Scaling. With the dynamic scaling enabled, this case introduced more instances into the execution, as the load goes high. In the evaluations, memory used by the application, as a percentage of the total memory used, was employed as the health monitoring measure. It could be replaced by load average, CPU or memory utilization. With the adaptive scaling, the environment of 200 VMs and 400 cloudlets with load scaled up to 3 instances, for a CPU utilization of 0.20, even when more than 3 instances were included in the sub-cluster. Reducing the maximum threshold made the main-cluster to scale out earlier, making the system closer to the static case, involving all the available instances to the simulation.

Adaptive scaling was not observed in the other cases, except when the maximum process CPU load is reduced below 0.15 from the configurations. This shows that a single instance was sufficient to run the sample simulations of the other 3 cases discussed below. The low threshold was kept low enough, such that there was no scale ins. When the minimum process CPU load was increased beyond 0.02, scale in was monitored in simulation scenarios. For the scenario of scale ins, synchronous backups should be enabled to prevent the data loss, which eliminates the possibility of a comparison, as the simulations with the fixed number of instances are run with no backups.

Load Average. With adaptive scaling configured, logs of load averages were noticed during and after the scaling events. Table 4 shows the load averages for the simulation environment with 6 nodes available. Up to 3 nodes were involved in the simulation by the IntelligentAdaptiveScaler. Waiting time acts as a buffer to prevent cascading scaling events. Health is monitored periodically, except during the buffer time introduced immediately following the scaling events. These intervals are configured to fit the requirements and the nature of the simulation.

5.2.2 *Other Cases of Scalability*

Figure 9 depicts 3 different cases of scalability, where the execution time changes in different patterns with the increasing number of nodes. The scenarios are analysed below.

Controlling/Base Case (Negative Scalability). Simulation time is increasing with the number of nodes, for the case of (noOfVMs = 200, noOfCloudlets = 400, isLoaded = false). This is because the cloudlets are not doing any task or no actual workload is attached to each cloudlet to perform. Hence, Hazelcast integration imposes a load, on an application for which a sequential and centralized execution is good enough. As in Case 1, the time is converging here as well. Introducing further nodes will not increase the time any more, after some number of nodes.

Common/Regular Case (Positive Scalability followed by Negative Scalability). Simulation time is decreasing with the number of nodes steadily till a number of nodes, and then

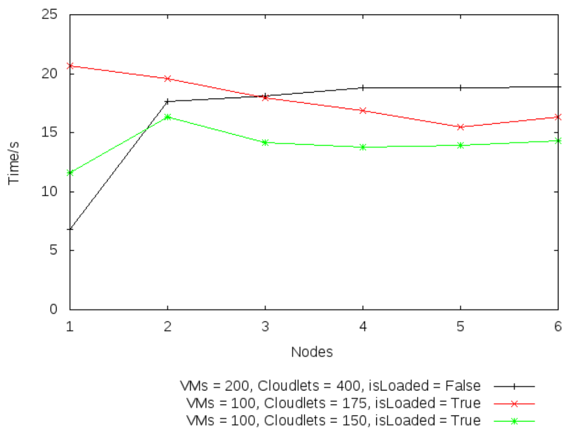


Figure 9: Distributed Execution - Different Patterns of Scaling

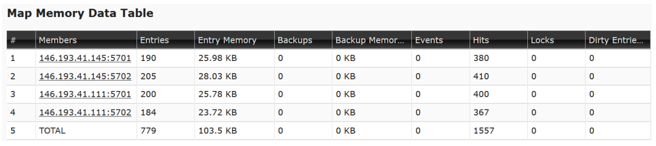
it starts to increase steadily, for the case of (noOfVMs = 100, noOfCloudlets = 175, isLoaded = true). This is one of the commonest cases, where a memory-hungry application that can hang (infinitely long time) a single node, runs faster (10x speedup) in 2 nodes and also in 3 nodes, where further nodes may decrease the performance. In this particular example, 5 nodes was the ideal scenario and introducing the 6th node created a negative scalability. Here the communication and serialization costs start to dominate the benefits of the scalability at latter stages.

Complex Case (Weird Patterns and borderline cases). Scenario (noOfVMs = 100, noOfCloudlets = 150, isLoaded = true) initially shows a negative scalability, followed by a positive scalability and then by a negative scalability again. Through repeating different experiments, a pattern was noticed in this rarely occurring scenario. Initially, introducing Hazelcast causes an overhead over the performance enhancements it provides, hence increasing the execution time. Then, the application starts to enjoy the advantages of scalability when the distribution dominates over the initial overheads of distribution, specially the serialization and initialization costs of Hazelcast. Later, communication costs tend to overtake the advantages of the distribution, causing negative scalability again. These are borderline cases, where an ideal number of nodes for the distribution cannot be easily predicted.

Among all the cases, there was a pattern, and it was possible to predict and adapt to the changing scalability pattern, based on the curves for the other number of cloudlets and VMs combinations, given that the application remained unchanged.

Hazelcast for $Cloud^2Sim$. The effectiveness of using Hazelcast to distribute the storage and execution for the simulation was evaluated by observing the overhead it imposes. Its distribution of execution and storage was measured by observing Hazelcast Management Center. Distributed objects and distributed execution were monitored by Hazelcast Management Center. Objects were evenly distributed among the available Hazelcast instances, consuming almost the same amount of entry memory from each instance. Partitions of different instances were equally hit or accessed. This shows an ideal partitioning of the distributed objects by Hazelcast. Figure 10 shows a screenshot of Hazelcast Management Center, while $Cloud^2Sim$ was running a sample simulation.

5.3 MapReduce Implementations



Map Memory Data Table

#	Members	Entries	Entry Memory	Backups	Backup Memor...	Events	Hits	Locks	Dirty Entrie...
1	146.193.41.145:5701	190	25.98 KB	0	0 KB	0	380	0	0
2	146.193.41.145:5702	205	28.03 KB	0	0 KB	0	410	0	0
3	146.193.41.111:5701	200	25.78 KB	0	0 KB	0	400	0	0
4	146.193.41.111:5702	184	23.72 KB	0	0 KB	0	367	0	0
5	TOTAL	779	103.5 KB	0	0 KB	0	1557	0	0

Figure 10: Distributed Objects as Observed by Hazelcast Management Center

Hazelcast-based and Infinispan-based MapReduce simulator implementations were benchmarked against multiple chunk files of 6 - 8 MB, each consisting of more than 125,000 lines. Figure 11 represents the time taken for both implementations on a single server with 3 map() invocations, along with the increasing number of reduce invocations with the size. Here the size is measured by the number of lines taken into consideration for the MapReduce task.

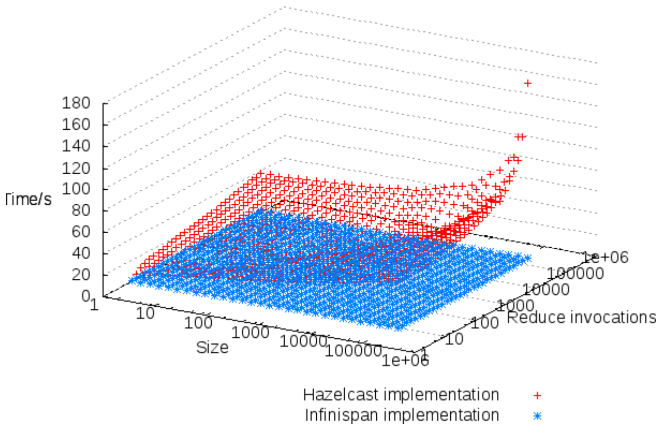


Figure 11: Reduce invocations and time taken for different sizes of MapReduce tasks

The results showed Infinispan outperforming Hazelcast by 10 to 100 folds. Infinispan based simulator was still fast, even when operating verbose. Infinispan MapReduce is matured. Hazelcast MapReduce is young, and still could be inefficient. Infinispan performs well in a single-node mode, as it functions better as a local cache. Hazelcast is optimized for larger set ups with very high number of real server nodes, and probably Hazelcast could outperform Infinispan, when large number of nodes (such as 50) are involved.

MapReduce simulations are distributed by nature, even in a single node. This is not the case for general applications like CloudSim simulations. Infinispan model is perfect for a single node (or even a few node) MapReduce tasks.

5.3.1 *Hazelcast MapReduce Implementation*

As Hazelcast based MapReduce simulator was slow when run in a single mode, it is tested on 1 - 6 nodes in verbose mode to check the improvements in execution time. One node starts the MapReduce simulator, where other nodes start the Initiator class, which just connects to the cluster. Hazelcast MapReduce implementation perfectly distributes the jobs. All the Initiator nodes were started to form a cluster, before starting the instance running the simulator. The time taken for different sizes of the task to run on different number of instances is shown in Figure 12. The number of map() invocations was kept constant at 3, as 3 files were used during the experiments. Infinispan with single node was noticed to be still faster than all 6 nodes running MapReduce in Hazelcast.

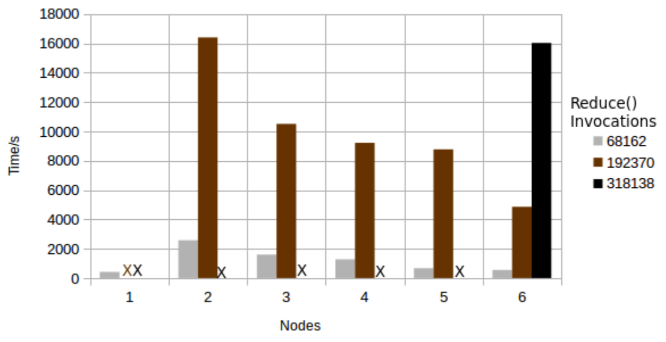


Figure 12: Distributing the Hazelcast MapReduce execution to multiple nodes

Table 5: Time (sec) taken for multiple Hazelcast instances to execute the same task

Instances	**1**	**2**	**3**	**4**
Time	416.687	2580.087	1600.655	1275.664
	6	**8**	**10**	**12**
	553.296	432.926	320.055	312.414

For the size of 10,000, Hazelcast running on a single instance was fast enough, and distributing the execution to multiple nodes started with a considerable negative scalability. This is because the communication costs were higher than the optimizations by the distributions. However, positive scalability, though not significant, was achieved when more than 8 instances were used, as shown by Table 5. Up to 2 Hazelcast instances were executed from each of the nodes during this. This shows that even for smaller applications, distribution may be advantageous overtaking the communication and other costs introduced by distributing the execution.

The sample application failed to run on single node for the size of 50,000 due to the heap space limitations. It ran smoothly on 2 instances, and showed a perfect positive scalability, when the nodes were joined to the cluster up to 6. The application failed to run on a single node for the size of 100,000, due to the out of memory issue in heap space. The issue persists even when the cluster size was increased up to 5 nodes. The application only ran successfully when 6 nodes were involved. The last two cases show the requirement of a distributed MapReduce simulations for larger tasks, as single or a fewer nodes in the cluster were proven to be insufficient for the higher memory requirements for the MapReduce tasks.

Bugs and Limitations. A few critical bugs were encountered during the evaluations of Hazelcast MapReduce implementation. If a new Hazelcast instance joins a cluster that is running a MapReduce job, it is noticed to crash the instance running the MapReduce task and hence failing the MapReduce task[8]. This was because of the newly joined instance not knowing the supervisor of the job, due to a missing null-check, according to the core Hazelcast/MapReduce developer. As a workaround, the master instance that starts the MapReduce jobs was started and joined the cluster, only after all the Initiator instances have started and formed the cluster. This prevented incorporation of the Hazelcast-based auto scaling and adaptive scaling that were already implemented during this work. Moreover, in a long running heavy task, instances were noticed to leave the cluster, to exhibit a split-brain syndrome[9]. This limited the usability of the MapReduce implementation to shorter MapReduce jobs. These issues were reported to the Hazelcast issue tracker.

5.3.2 *Infinispan MapReduce Implementation*

Infinispan implementation was tested for its scalability, with the same MapReduce job distributed to different number of nodes. Figure 13 shows the scaling of Infinispan MapReduce implementation to multiple nodes, with the time taken to execute different number of map() invocations. Number of reduce() invocations was kept constant at 159,069. Number of map() invocations is equal to the number of files present in the word count execution used. Hence, the numbers of files were increased for different scenarios. As the number of instances were increased, the jobs were distributed to the available instances.

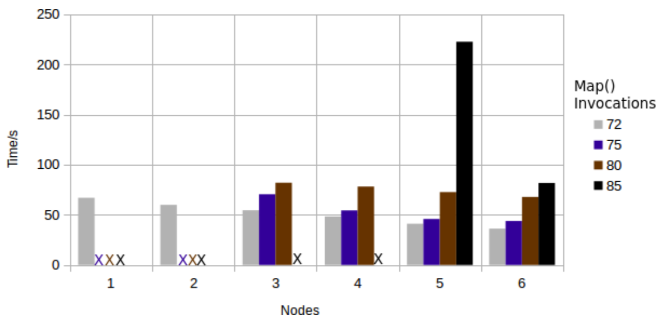


Figure 13: Distributing the Infinispan MapReduce execution to multiple nodes

When the number of map() invocations was increased, jobs started to fail in single instance, due to the out of memory (Java heap space) issue. Further, garbage collection (GC) overhead limit was exceeded in some scenarios. These issues prevented larger invocations to execute in smaller number of instances. When the number of instances was increased, the jobs that failed started to execute successfully. Moreover, a positive scalability was observed, when the number of nodes was increased. These evaluations prove that memory and processing requirements increase as the map() and reduce() invocations are increased. Further, distributing the execution enables larger executions, and makes the executions faster.

6. CONCLUSION AND FUTURE WORK

Typically, cloud and MapReduce simulators are sequential, and thus run on a single computer, where computer clusters and in-memory data grids could be leveraged to execute larger simulations that cannot be executed on a single computer. Even the simulations that could run on a single node can take advantage of more resources from the cluster, that it can run faster and more effectively. The cycle sharing model could be utilized to provide means of sharing the resources across the simulation instances, allowing multiple independent simulations to execute in parallel, in a multi-tenant way. A scalable middleware platform for concurrent and distributed cloud and MapReduce simulations can leverage an existing cloud simulator, whilst exploiting the in-memory data grid platforms for an elastic environment, deploying an adaptive scaling strategy inspired by the volunteer computing model.

Cloud^{2}Sim presents an architecture that enables the execution of larger simulations in a cluster, that could not be run on single nodes due to the requirement of huge heap space, and long execution times. *Cloud^{2}Sim* has the advantages of CloudSim while being efficient, faster, customizable, and scalable. By virtue of being elastic and adaptive, it is

[8] https://github.com/Hazelcast/Hazelcast/issues/2354
[9] https://github.com/Hazelcast/Hazelcast/issues/2359

cloud-ready and can be the basis of a concurrent and distributed Simulation-as-a-Service for Cloud and MapReduce simulations. MapReduce implementations stand as an extension and proof that the same distributed execution model could be extended beyond cloud simulations. The design of the adaptively scaling middleware platform can be extended to other cloud and MapReduce simulators, as the design of Hazelcast and CloudSim based $Cloud^2Sim$ distributed cloud simulator is not tightly coupled to CloudSim or Hazelcast. Currently, the Hazelcast based distributed cloud simulator is implemented completely, along with an Infinispan integration to facilitate distributed execution with Infinispan. Following the same design, a complete distributed cloud simulator with Infinispan can also be built. Moreover, the adaptive scaler design suits many applications, not just limited to simulations. Hence this can be extended to use on any application that has an elastic scaling requirement.

7. REFERENCES

[1] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[2] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] Bela Ban et al. Jgroups, a toolkit for reliable multicast communication. *URL: http://www. jgroups. org*, 2002.

[4] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[5] Adam L Beberg, Daniel L Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[6] Rodrigo N Calheiros, Marco AS Netto, César AF De Rose, and Rajkumar Buyya. Emusim: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, 43(5):595–612, 2013.

[7] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[8] Rodrigo N Calheiros, Rajiv Ranjan, César AF De Rose, and Rajkumar Buyya. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. *arXiv preprint arXiv:0903.2525*, 2009.

[9] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2001.

[10] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131. IEEE, 2008.

[11] Weiwei Chen and Ewa Deelman. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8. IEEE, 2012.

[12] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 43–56. ACM, 2012.

[13] Marco Ferrante. A java framework for high-level distributed scientific programming. 2003.

[14] Kayo Fujiwara and Henri Casanova. Speed and accuracy of network simulation in the simgrid framework. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, page 12. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[15] Saurabh Kumar Garg and Rajkumar Buyya. Networkcloudsim: Modelling parallel applications in cloud simulations. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 105–113. IEEE, 2011.

[16] Mat Johns. *Getting Started with Hazelcast.* Packt Publishing Ltd, 2013.

[17] Siddhartha Kumar Khaitan and Anshul Gupta. *High Performance Computing in Power and Energy Systems.* Springer, 2012.

[18] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[19] Wagner Kolberg, Pedro De B Marcos, Julio Anjos, Alexandre KS Miyazaki, Claudio R Geyer, and Luciana B Arantes. Mrsg–a mapreduce simulator over simgrid. *Parallel Computing*, 39(4):233–244, 2013.

[20] Stefan M Larson, Christopher D Snow, Michael Shirts, et al. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. 2002.

[21] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988.

[22] Liang Liu, Hao Wang, Xue Liu, Xing Jin, Wen Bo He, Qing Bo Wang, and Ying Chen. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38. ACM, 2009.

[23] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: a mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.

[24] Daniel Mahrenholz and Svilen Ivanov. Real-time network emulation with ns-2. In *Distributed Simulation and Real-Time Applications, 2004. DS-RT 2004. Eighth IEEE International Symposium on*, pages 29–36. IEEE, 2004.

[25] Francesco Marchioni. *Infinispan Data Grid Platform.* Packt Publishing Ltd, 2012.

[26] Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[27] Vishal Pachori, Gunjan Ansari, and Neha Chaudhary. Improved performance of advance encryption standard using parallel computing. *International Journal of Engineering Research and Applications (IJERA)*, 2(1):967–971, 2012.

[28] Roberto Palmieri, Pierangelo di Sanzo, Francesco Quaglia, Paolo Romano, Sebastiano Peluso, and Diego Didona. Integrated monitoring of infrastructures and applications in cloud environments. In *Euro-Par 2011: Parallel Processing Workshops*, pages 45–53. Springer, 2012.

[29] Liliana Rosa, Luís Rodrigues, and Antónia Lopes. Goal-oriented self-management of in-memory distributed data grid platforms. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 587–591. IEEE, 2011.

[30] C Shaoul and C Westbury. A reduced redundancy usenet corpus (2005-2011) edmonton, ab: University of alberta, 2013. Downloaded from: http://www.psych.ualberta.ca/ westburylab/-downloads/usenetcorpus.download.html.

[31] Michael Shirts, Vijay S Pande, et al. Screen savers of the world unite. *COMPUTING*, 10:43, 2006.

[32] Michael Tighe, Gaston Keller, Michael Bauer, and Hanan Lutfiyya. Dcsim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, pages 385–392. IEEE, 2012.

[33] Pedro Velho and Arnaud Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[34] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Play it again, simmr! In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 253–261. IEEE, 2011.

[35] Guanying Wang, Ali R Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*, pages 19–26. ACM, 2009.

[36] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[37] Bhathiya Wickremasinghe, Rodrigo N Calheiros, and Rajkumar Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 446–452. IEEE, 2010.