OpenStack Neat: A Framework for Dynamic and Energy-Efficient Consolidation of Virtual Machines in OpenStack Clouds

Anton Beloglazov and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computing and Information Systems
The University of Melbourne, Australia
anton.beloglazov@gmail.com

December 23, 2013

Abstract

Dynamic consolidation of Virtual Machines (VMs) is an efficient approach for improving the utilization of physical resources and reducing energy consumption in Cloud data centers. Despite the large volume of research published on this topic, there are very few open source software systems implementing dynamic VM consolidation. In this paper, we propose an architecture and open source implementation of OpenStack Neat, a framework for dynamic VM consolidation in OpenStack Clouds. OpenStack Neat can be configured to use custom VM consolidation algorithms, and transparently integrates with existing OpenStack deployments without the necessity in modifying their configuration. In addition, to foster and encourage further research efforts in the area of dynamic VM consolidation, we propose a benchmark suite for evaluating and comparing dynamic VM consolidation algorithms. The proposed benchmark suite comprises OpenStack Neat as the base software framework, a set of real-world workload traces, performance metrics, and evaluation methodology. As an application of the proposed benchmark suite, we conduct experimental evaluation of OpenStack Neat and several dynamic VM consolidation algorithms on a 5-node testbed, which shows significant benefits of dynamic VM consolidation resulting in up to 33% energy savings.

1 Introduction

Cloud computing has revolutionized the Information and Communication Technology (ICT) industry by enabling on-demand provisioning of elastic computing resources on a pay-as-you-go basis. However, Cloud data centers consume huge amounts of electrical energy resulting in high operating costs and carbon dioxide (CO_2) emissions to the environment. It is estimated that energy consumption by data centers worldwide has risen by 56% from 2005 to 2010, and in 2010 was accounted to be between 1.1% and 1.5% of the global electricity use [24]. Furthermore, carbon dioxide emissions of the ICT industry were estimated to be 2% of the global emissions, which is equivalent to the emissions of the aviation industry [19].

To address the problem of high energy use, it is necessary to eliminate inefficiencies and waste in the way electricity is delivered to computing resources, and in the way these resources are utilized to serve application workloads. This can be done by improving the physical infrastructure of data centers, as well as resource allocation and management algorithms. Recent advancement in the data center design resulted in a significant increase of the infrastructure efficiency. As reported by the Open Compute project, Facebook's Oregon data center achieved a Power Usage Effectiveness (PUE) of 1.08 [2], which means that approximately 91% of the data center's energy consumption are consumed by the computing resources. Therefore, now it is important to focus on optimizing the way the resources are allocated and utilized to serve application workloads.

One method to improve the utilization of resources and reduce energy consumption is dynamic consolidation of virtual machines (VMs) [5, 10, 12–15, 17, 18, 20–23, 25–27, 29, 30, 32–36, 38] enabled by *live migration*, the capability of transferring a VM between physical servers (referred to as hosts, or nodes)

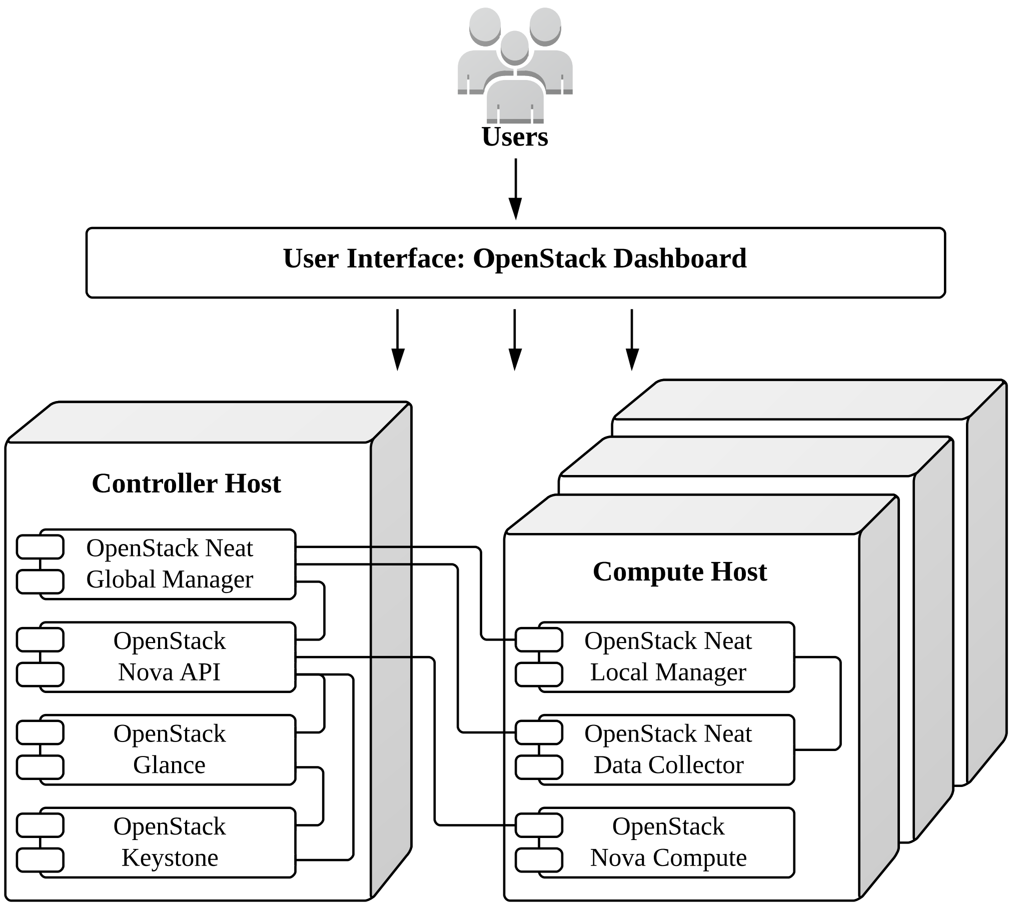

Users
User Interface: OpenStack Dashboard
Controller Host
OpenStack Neat Global Manager
OpenStack Nova API
OpenStack Glance
OpenStack Keystone
Compute Host
OpenStack Neat Local Manager
OpenStack Neat Data Collector
OpenStack Nova Compute

Figure 1: The combined deployment of OpenStack and OpenStack Neat

with a close to zero downtime. Dynamic VM consolidation consists of two basic processes: migrating VMs from underutilized hosts to minimize the number of active hosts; and offloading VMs from hosts when those become overloaded to avoid performance degradation experienced by the VMs, which could lead to a violation of the Quality of Service (QoS) requirements. Idle hosts are automatically switched to a low-power mode to eliminate the static power and reduce the overall energy consumption. When required, hosts are reactivated to accommodate new VMs or VMs being migrated from other hosts. Even though a large volume of research has been published on the topic of dynamic VM consolidation, there are very few open source software implementations.

In this work, we introduce an architecture and implementation of *OpenStack Neat*[1]: an open source software framework for distributed dynamic VM consolidation in Cloud data centers based on the OpenStack platform[2]. Figure 1 depicts a typical deployment of the core OpenStack services, OpenStack Neat services, and their interaction, which will be discussed in detail in the following sections. The deployment may include multiple instances of compute and controller hosts. The OpenStack Neat framework is designed and implemented as a transparent add-on to OpenStack, which means that the OpenStack installation need not be modified or specifically configured to benefit from OpenStack Neat. The framework acts independently of the base OpenStack platform and applies VM consolidation processes by invoking public Application Programming Interfaces (APIs) of OpenStack. The purpose of the OpenStack Neat framework is twofold: (1) providing a fully operational open source software for dynamic VM consolidation that can be applied to existing OpenStack Clouds; and (2) providing an extensible software framework for conducting research on dynamic VM consolidation.

OpenStack Neat is designed and implemented following the distributed approach to dynamic VM consolidation introduced and evaluated in our previous works [7–9]. The target environment is an Infrastructure as a Service (IaaS), e.g., Amazon EC2, where the provider is unaware of applications and workloads served by the VMs, and can only observe them from outside. We refer to this property of

[1]The OpenStack Neat framework. http://openstack-neat.org/
[2]The OpenStack Cloud platform. http://openstack.org/

IaaS environments as being application-agnostic. The proposed approach to distributed dynamic VM consolidation consists in splitting the problem into 4 sub-problems [8]:

1. Deciding if a host is considered to be underloaded, so that all VMs should be migrated from it, and the host should be switched to a low-power mode.
2. Deciding if a host is considered to be overloaded, so that some VMs should be migrated from it to other active or reactivated hosts to avoid violating the QoS requirements.
3. Selecting VMs to migrate from an overloaded host.
4. Placing VMs selected for migration on other active or reactivated hosts.

This approach has two major advantages compared with traditional fully centralized VM consolidation algorithms: (1) splitting the problem simplifies its analytical treatment by allowing the consideration of the sub-problems independently; and (2) the approach can be implemented in a partially distributed manner by executing the underload / overload detection and VM selection algorithms on compute hosts, and the VM placement algorithm on the controller host, which can optionally be replicated. Distributed VM consolidation algorithms enable the natural scaling of the system to thousands of compute nodes, which is essential for large-scale Cloud providers. For instance, Rackspace, a well-known IaaS provider, currently manages tens of thousands of servers. Moreover, the number of servers continuously grows: Rackspace has increased the total server count in the second quarter of 2012 to 84,978 up from 82,438 servers at the end of the first quarter [3].

In addition, to facilitate research efforts and future advancements in the area of dynamic VM consolidation, we propose a benchmark suite for evaluating and comparing dynamic VM consolidation algorithms comprising OpenStack Neat as the base software framework, real-world workload traces from PlanetLab, performance metrics, and evaluation methodology (Section 6).

The **key contributions** of this work are the following:

- An architecture of an extensible software framework for dynamic VM consolidation designed to transparently integrate with OpenStack installations and allowing configuration-based substitution of multiple implementations of algorithms for each of the 4 defined sub-problems of dynamic VM consolidation.
- An open source software implementation of the framework in Python released under the Apache 2.0 license and publicly available online.
- An implementation of several algorithms for dynamic VM consolidation proposed and evaluated by simulations in our previous works [7–9].
- An initial version of a benchmark suite comprising the software framework, workload traces, performance metrics, and methodology for evaluating and comparing dynamic VM consolidation solutions following the distributed model.
- Experimental evaluation of the framework on a 5-node OpenStack deployment using real-world application workload traces collected from more than a thousand PlanetLab VMs hosted on servers located in more than 500 places around the world [31]. According to the estimates of potential energy savings, the evaluated algorithms reduce energy consumption by up to 33% with a limited performance impact.

The remainder of the paper is organized as follows. In the next section we discuss the related work, followed by the overall design and details of each component of the OpenStack Neat framework in Section 3. In Section 4, we describe the implemented VM consolidation algorithms. In Section 6, we propose a benchmark suite for evaluating distributed dynamic VM consolidation algorithms. We experimentally evaluate the framework and analyze the results in Section 7. We conclude the paper with a discussion of scalability and future research directions in Section 8 and conclusions in Section 9.

2 Related Work

Research work related to this paper can be divided into two categories: (1) practically implemented and publicly available open source software systems; and (2) theoretical work on various approaches to dynamic VM consolidation. Despite the large volume of research published on the topic of dynamic VM consolidation, there are very few software implementations publicly available online. To the best of our knowledge, the earliest open source implementation of a VM consolidation manager is the Entropy project[3]. Entropy is an open source VM consolidation manager for homogeneous clusters developed by Hermenier et al. [22] and released under the LGPL license.

Entropy is built on top of Xen and focused on two objectives: (1) maintaining a configuration of the cluster, where all VMs are allocated sufficient resources; and (2) minimizing the number of active hosts. To optimize the VM placement, Entropy periodically applies a two-phase approach. First, a constraint programming problem is solved to find an optimal VM placement, which minimizes the number of active hosts. Then, another optimization problem is solved to find a target cluster configuration with the minimal number of active hosts that also minimizes the total cost of reconfiguration, which is proportional to the cost of VM migrations. Instead of optimizing the VM placement periodically as Entropy, OpenStack Neat detects host underload and overload conditions and dynamically resolves them, which allows the system to have more fine-grained control over the host states. Although, Entropy may find a more optimal VM placement by computing a globally optimal solution, all aspects of VM placement optimization must be computed by a central controller, thus limiting the scalability of the system.

Feller et al. [17] proposed and implemented a framework for distributed management of VMs for private Clouds called Snooze[4], which is open source and released under the GPL v2 license. In addition to the functionality provided by the existing Cloud management platforms, such as OpenStack, Eucalyptus, and OpenNebula, Snooze implements dynamic VM consolidation as one of its base features. Another difference is that Snooze implements hierarchical distributed resource management. The management hierarchy is composed of three layers: local controllers on each physical node; group managers managing a set of local controllers; and a group leader dynamically selected from the set of group managers and performing global management tasks. The distributed structure enables fault-tolerance and self-healing by avoiding single points of failure and automatically selecting a new group leader if the current one fails. Snooze also integrates monitoring of the resource usage by VMs and hosts, which can be leveraged by VM consolidation policies. These policies are intended to be implemented at the level of group managers, and therefore can only be applied to subsets of hosts. This approach partially solves the problem of scalability of VM consolidation by the cost of losing the ability of optimizing the VM placement across all the nodes of the data center. OpenStack Neat enables scalability by distributed underload / overload detection and VM selection, and potentially replicating the VM placement controllers. In contrast to Snooze, it is able to apply global VM placement algorithms for the selected for migration VMs by taking into account the full set of hosts. Another difference is that OpenStack Neat transparently integrates with OpenStack, a mature open source Cloud platform widely adopted and supported by the industry, thus ensuring long-term development of the platform.

The second category of related work includes various theoretical approaches to the problem of dynamic VM consolidation. These can be further divided into application-specific and application-agnostic approaches. In application-specific approaches, the authors make assumptions about the execution environment or applications. For example, there are a number of works, where the authors focus on enterprise environments by developing static and semi-static consolidation techniques [32, 33]; assuming the knowledge of application priorities [14]; or applying offline profiling of applications [25]. There has been extensive research on VM consolidation tailored to specific application types, such as focusing on web applications [5, 21, 23, 26, 27, 34], HPC applications [12, 36], or managing both web and HPC applications [15, 20]. In contrast, our approach is application-agnostic meaning that there are no application-specific assumptions. This makes the system easily applicable to IaaS environments, where various types of user applications share computing resources.

One of the first works, in which dynamic VM consolidation was applied to minimize energy consumption in an application-agnostic way, has been done by Nathuji and Schwan [30]. The authors have proposed an architecture of a system, where the resource management is divided into local and global policies. At the local level the system leverages power management strategies of the guest Operating System (OS). The global manager applies its policy to decide whether the VM placement needs to be adapted. Zhu et al. [38] studied a similar problem of automated resource allocation and capacity planning. They proposed three individual controllers each operating at a different time scale, which place compatible workloads onto groups of servers, react to changing conditions by reallocating VMs, and allocate resources to VMs within the servers to satisfy the Service Level Agreements (SLAs).

[3]The Entropy VM manager. http://entropy.gforge.inria.fr/
[4]The Snooze Cloud manager. http://snooze.inria.fr/

Bobroff et al. [13] proposed an algorithm for dynamic VM consolidation under SLA constraints based on time-series analysis and forecasting of the resource demand. Nandi et al. [29] formulated the problem of VM placement as a stochastic optimization problem with a constraint on the probability of a host overload taking into account three dimensions of resource constraints. The authors introduced two versions of the optimization problem and corresponding greedy heuristics depending on the distribution model of the VM resource usage: for known distributions, and for unknown distributions with the known mean and variance. A drawback of all the described application-agnostic approaches is that they are centralized: a single algorithm running on the master node leverages the global view of the system to optimize the VM allocation. Such a centralized approach limits the scalability of the system, when the number of physical machines grows over thousands and tens of thousands, like in Rackspace [3].

In this work we follow a distributed approach proposed in our previous works [7–9], where every compute host locally solves the problems of underload / overload detection and VM selection. Then, it sends a request to a global manager to place only the selected for migration VMs on other hosts. A similar approach was followed by Wood et al. [35] in their system called Sandpiper aimed at load balancing in virtualized data centers using VM live migration. The main objective of the system is to avoid host overloads referred to as hot spots by detecting them and migrating overloaded VMs to less loaded hosts. The authors applied an application-agnostic approach, referred to as a black-box approach, in which VMs are observed from outside, without any knowledge of applications resident in the VMs. A hot spot is detected when the aggregate usage of a host's resources exceeds the specified threshold for k out of n last measurements, as well as for the next predicted value. Another proposed approach is gray-box, when a certain application-specific data are allowed to be collected. The VM placement is computed heuristically by placing the most loaded VM to the least loaded host. The difference from our approach is that VMs are not consolidated; therefore, the number of active hosts is not reduced to save energy.

Feller et al. [18] proposed a set of algorithms for dynamic VM consolidation and evaluated them using the Snooze Cloud manager discussed above. The approach and algorithms proposed in the paper can be seen as an extension of our previous work [7]. The authors proposed threshold-based heuristics for underload and overload detection using averaged resource usage data collected from the Central Processing Unit (CPU), Random-Access Memory (RAM), and network interface of the hosts. In contrast to our previous works [7, 9], the VM placement algorithm is applied to only subsets of hosts, therefore, disallowing the optimization of the VM placement across all the hosts of the data center.

3 System Design

The aim of the OpenStack Neat project is to provide an extensible framework for dynamic consolidation of VMs based on the OpenStack platform. Extensibility in this context means the ability to implement new VM consolidation algorithms and apply them in OpenStack Neat without the necessity to modify the source code of the framework itself. Different implementations of the algorithms can be plugged into the framework by modifying the appropriate options in the configuration file.

OpenStack Neat provides an infrastructure required for monitoring VMs and hypervisors, collecting resource usage data, transmitting messages and commands between the system components, and invoking VM live migrations. The infrastructure is agnostic of VM consolidation algorithms in use and allows implementing custom decision-making algorithms for each of the 4 sub-problems of dynamic VM consolidation: host underload / overload detection, VM selection, and VM placement. The implementation of the framework includes the algorithms proposed in our previous works [7–9]. In the following sections, we discuss the requirements and assumptions, integration of the proposed framework with OpenStack, and each of the framework's components.

3.1 Requirements and Assumptions

The components of the framework are implemented in the form of OS services running on the compute and controller hosts of the data center in addition to the core OpenStack services. The current implementation of OpenStack Neat assumes a single instance of the controller responsible for finding a new placement of the VMs selected for migration. However, due to distributed underload / overload detection and VM selection algorithms, the overall scalability is significantly improved compared with existing completely centralized VM consolidation solutions.

OpenStack Neat relies on live migration to dynamically relocate VMs across physical machines. To enable live migration, it is required to set up a *shared storage* and correspondingly configure OpenStack

Nova (i.e. the OpenStack Compute service) to use this storage for storing VM instance data. For instance, a shared storage can be provided using the Network File System (NFS), or the GlusterFS distributed file system [11].

OpenStack Neat uses a database (which can be distributed) for storing information about VMs and hosts, as well as resource usage data. It is possible to use the same database server used by the core OpenStack services. In this case, it is only required to create a new database and user for OpenStack Neat. The required database tables are automatically created by OpenStack Neat on the first launch of its services.

Another requirement is that all the compute hosts must have a user, which is enabled to switch the host into a low-power mode, such as *Suspend to RAM*. This user account is used by the global manager to connect to the compute hosts via the Secure Shell (SSH) protocol and switch them into the sleep mode when necessary. More information on deactivating and reactivating physical nodes is given in Section 3.4.2.

3.2 Integration with OpenStack

OpenStack Neat services are installed independently of the core OpenStack services. Moreover, the activity of the OpenStack Neat services is transparent to the core OpenStack services. This means that OpenStack does not need to be configured in a special way to be able to take advantage of dynamic VM consolidation implemented by OpenStack Neat. It also means, that OpenStack Neat can be added to an existing OpenStack installation without the need to modify its configuration.

The transparency is achieved by the independent resource monitoring implemented by OpenStack Neat, and the interaction with the core OpenStack services using their public APIs. The OpenStack APIs are used for obtaining information about the current state of the system and performing VM migrations. In particular, the APIs are used to get the current mapping of VMs to hosts, hardware characteristics of hosts, parameters of VM flavors (i.e., instance types), VM states, and invoke VM live migrations. Although OpenStack Neat performs actions affecting the current state of the system by relocating VMs across hosts, it is transparently handled by the core OpenStack services since VM migrations are invoked via the public OpenStack APIs, which is equivalent to invoking VM migrations manually by the system administrator.

One of the implications of this integration approach is that the VM provisioning and destruction processes are handled by the core OpenStack services, while OpenStack Neat discovers new VM instances through OpenStack APIs. In the following sections, we refer to hosts running the Nova Compute service, i.e., hosting VM instances, as compute hosts; and a host running the other OpenStack management services but not hosting VM instances as the controller host.

3.3 System Components

OpenStack Neat is composed of a number of components and data stores, some of which are deployed on the compute hosts, and some on the controller host, which can potentially have multiple replicas. As shown in Figure 2, the system is composed of three main components:

- *Global manager* – a component that is deployed on the controller host and makes global management decisions, such as mapping VM instances to hosts, and initiating VM live migrations.
- *Local manager* – a component that is deployed on every compute host and makes local decisions, such as deciding that the host is underloaded or overloaded, and selecting VMs to migrate to other hosts.
- *Data collector* – a component that is deployed on every compute host and is responsible for collecting data on the resource usage by VM instances and hypervisors, and then storing the data locally and submitting it to the central database, which can also be distributed.

The deployment model may vary for each particular system depending on its requirements. For instance, the central database can be deployed on a separate physical node, or be distributed across multiple physical nodes. The location and deployment of the database server is transparent to OpenStack Neat, which only requires a configuration parameter to be set to the network address of the database front-end server. For simplicity, in our experimental testbed, the database server is deployed on the same physical node hosting the global manager, as shown in Figure 2.

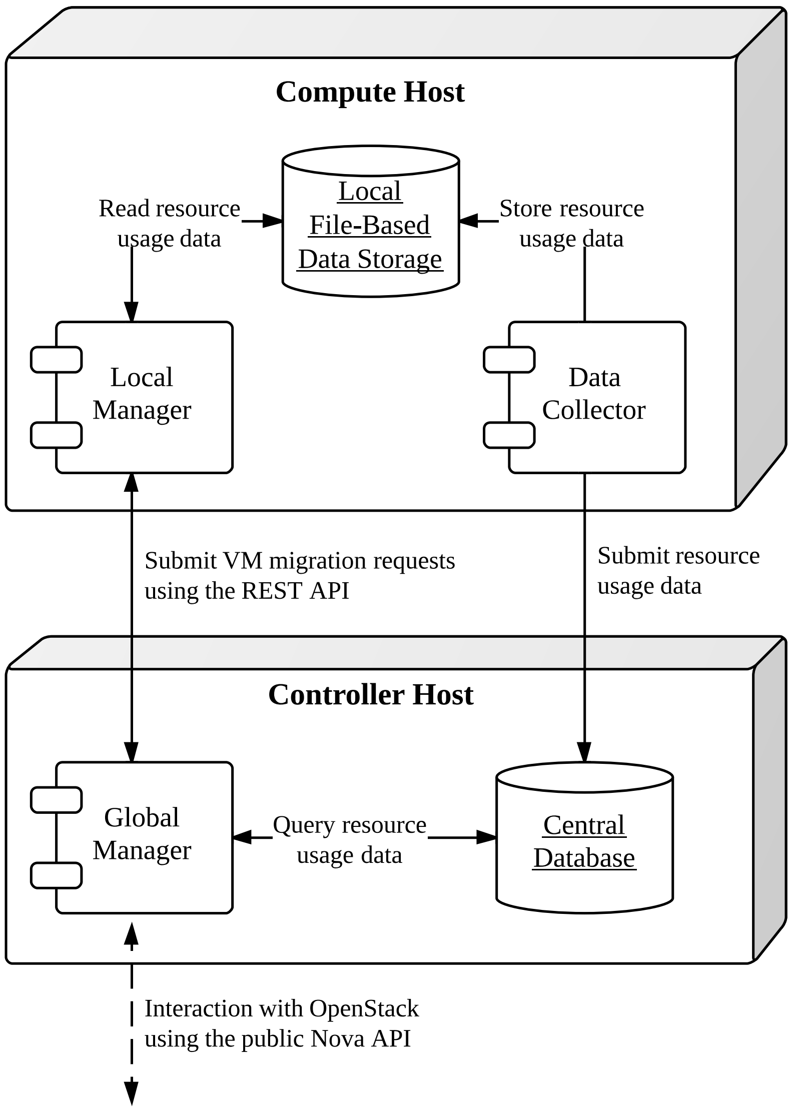

Compute Host
Local File-Based Data Storage
Read resource usage data
Store resource usage data
Local Manager
Data Collector
Submit VM migration requests using the REST API
Submit resource usage data
Controller Host
Global Manager
Query resource usage data
Central Database
Interaction with OpenStack using the public Nova API

Figure 2: The deployment diagram

3.4 Global Manager

The global manager is deployed on the controller host and is responsible for making VM placement decisions and initiating VM migrations. It exposes a Representational State Transfer (REST) web service, which accepts requests from local managers. The global manager processes two types of requests: (1) relocating VMs from an underloaded host; and (2) offloading a number of VMs from an overloaded host.

Figure 3 shows a sequence diagram of handling a host underload request by the global manager. First, a local manager detects an underload of the host using the specified in the configuration underload detection algorithm. Then, it sends an underload request to the global manager including the name of the underloaded host. The global manager calls the OpenStack Nova API to obtain the list of VM currently allocated to the underloaded host. This is required to obtain a view of the actual state of the system, as some VMs currently allocated to the host may still be in migration. Once the list of VMs is received, the global manager invokes the VM placement algorithm with the received list of VMs along with their resource usage and states of hosts fetched from the database as arguments. Then, according to the VM placement generated by the algorithm, the global manager submits the appropriate VM live migration requests to the OpenStack Nova API, and monitors the VM migration process to determine when the migrations are completed. Upon the completion of the VM migrations, the global manager switches the now idle source host into the *Suspend to RAM* state. Since the *Suspend to RAM* state provide low-latency transitions, it is possible to quickly re-enable the host if required as discussed in Section 3.4.2.

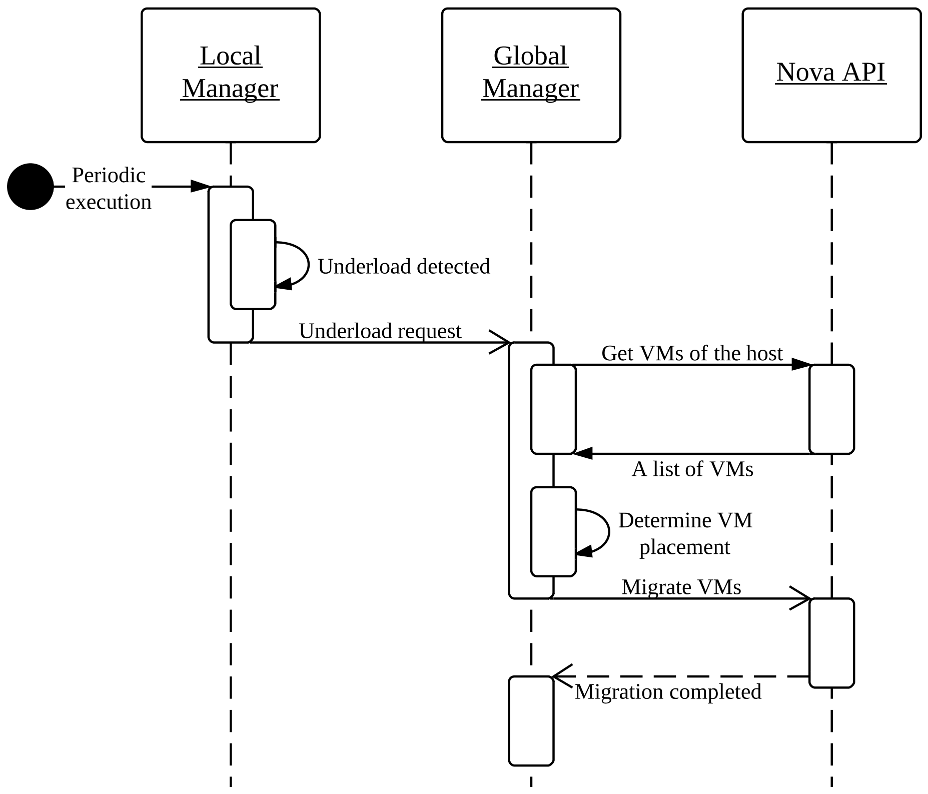


Figure 3: The global manager: a sequence diagram of handling an underload request

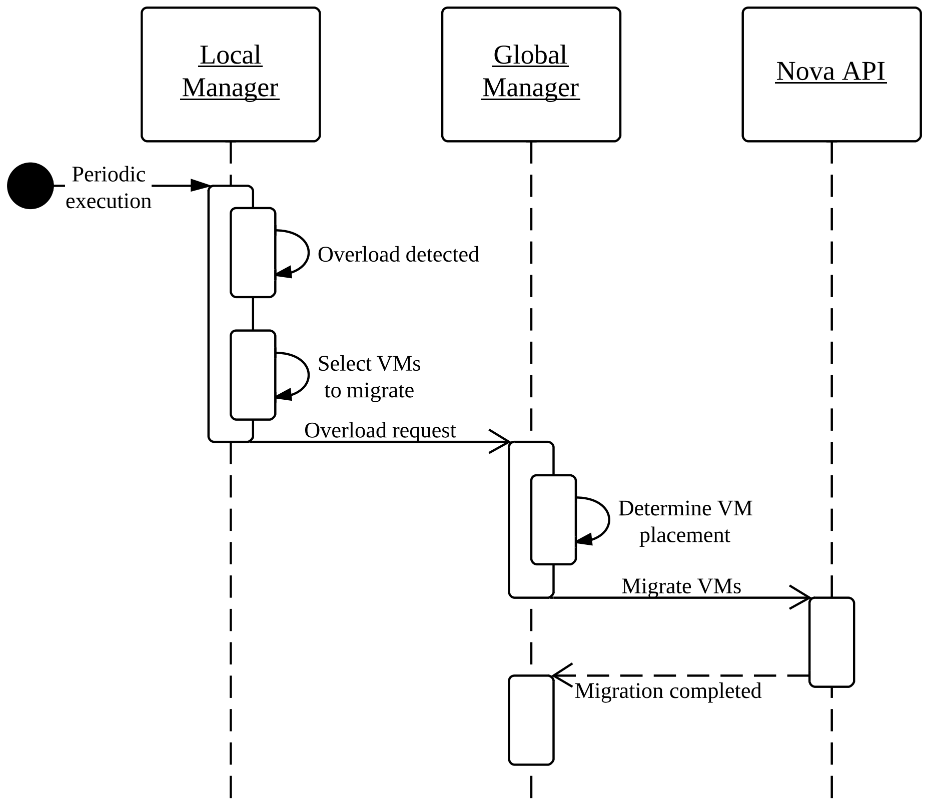


Figure 4: The global manager: a sequence diagram of handling an overload request

As shown in Figure 4, handling overload requests is similar to underload requests. The difference is that instead of sending just the host name, the local manager also sends a list of UUIDs of the VMs selected by the configured VM selection algorithm to be offloaded from the overloaded host. Once the request is received, the global manager invokes the specified in the configuration VM placement algorithm and passes as arguments the list of VMs received from the local manager to be placed on other hosts along with other system information. If some of the VMs are placed on hosts that are currently in the low-power mode, the global manager reactivates them using the Wake-on-LAN technology, as described in Section 3.4.2. Then, similarly to handling underload requests, the global manager submits VM live migration requests to the OpenStack Nova API.

3.4.1 REST API.

The global manager exposes a REST web service (REST API) for processing VM migration requests sent by local managers. The service Uniform Resource Locator (URL) is defined according to configuration options specified in */etc/neat/neat.conf*, which is discussed in detail in Section 3.8. The two relevant options are:

- *global_manager_host* – the name of the host running the global manager;
- *global_manager_port* – the port that should be used by the web service to receive requests.

Using these configuration options, the service URL is composed according to the following template: *http://global_manager_host:global_manager_port/*. The global manager processes two types of requests from local managers: host underloads, and host overloads discussed in the previous section. Both types of requests are served at a single resource *'/'* accessed using the PUT method of the Hypertext Transfer Protocol (HTTP). The type of a received request is determined by the global manager by analyzing the parameters included in the request. The following parameters are common to both types of requests:

- *username* – the admin user name specified in the configuration file, which is used to authenticate the client making the request as being allowed to access the web service. This parameter is sent SHA-1-encrypted to avoid sending the user name in the open form over the network.
- *password* – the admin password specified in the configuration file, which is used to authenticate the client making the request as being allowed to access the web service. Similarly to *username*, this parameter is also sent encrypted with the SHA-1 algorithm.
- *time* – the time when the request has been sent. This parameter is used by the global manager to identify and enforce time-outs, which may happen if a request has been sent a long time ago rendering it non-representative of the current state of the system.
- *host* – the host name of the overloaded or underloaded host, where the local manager sending the request is deployed on.
- *reason* – an integer specifying the type of the request, where 0 represents a host underload request, and 1 represents a host overload request.

If the request type specified by the *reason* parameter is 1 (i.e., denoting an overload request), there is an extra mandatory parameter *vm_uuids*. This is a string parameter, which must contain a coma-separated list of Universally Unique Identifiers (UUIDs) of VMs selected for migration from the overloaded host.

If a request contains all the required parameters and the provided credentials are correct, the service responds with the HTTP status code *200 OK*. The service uses standard HTTP error codes to respond in cases of errors. The following error codes are used:

- 400 – bad input parameter: incorrect or missing parameters;
- 401 – unauthorized: user credentials are missing;
- 403 – forbidden: user credentials do not much the ones specified in the configuration file;
- 405 – method not allowed: the request has been made with a method other than the only supported PUT method;
- 422 – precondition failed: the request has been sent more than 5 seconds ago, which means that the states of the hosts or VMs may have changed – a retry is required.

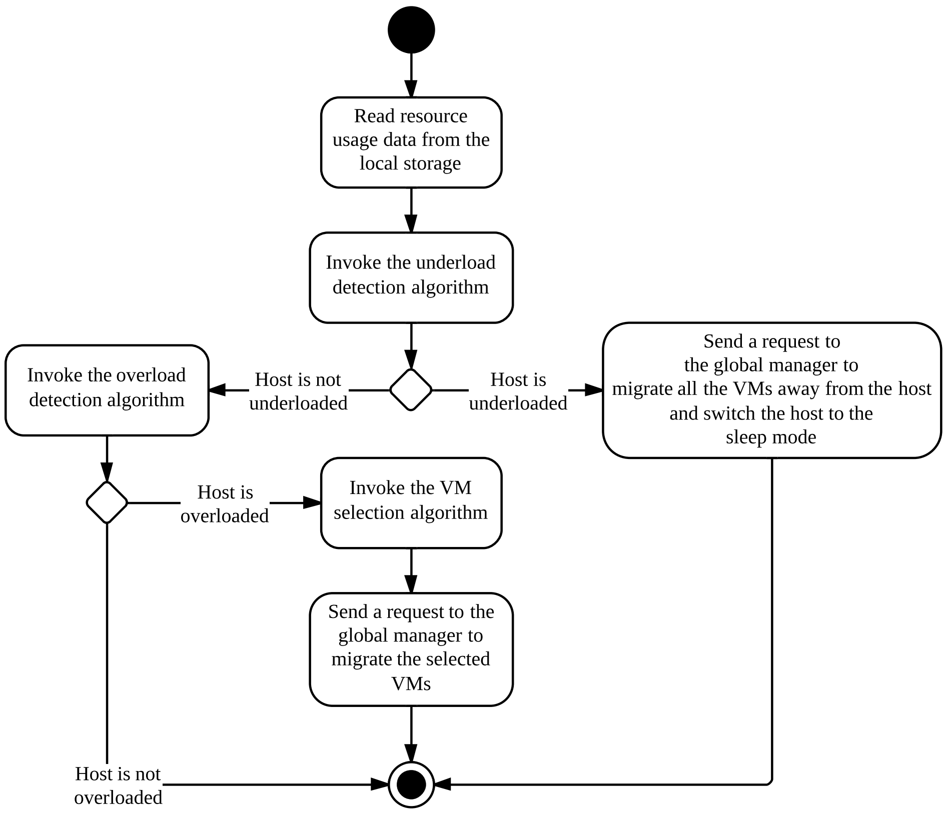

Read resource usage data from the local storage
Invoke the underload detection algorithm
Host is not underloaded
Host is underloaded
Invoke the overload detection algorithm
Send a request to the global manager to migrate all the VMs away from the host and switch the host to the sleep mode
Host is overloaded
Invoke the VM selection algorithm
Send a request to the global manager to migrate the selected VMs
Host is not overloaded

Figure 5: The local manager: an activity diagram

3.4.2 Switching Power States of Hosts.

One of the main features required to be supported by the hardware and OS in order to take advantage of dynamic VM consolidation to save energy is the Advanced Configuration and Power Interface (ACPI). The ACPI standard defines platform-independent interfaces for power management by the OS. The standard is supported by Linux, the target OS for the OpenStack platform. ACPI defines several sets of power states, the most relevant of which is the sleep state S3, referred to as *Suspend to RAM*. Meisner et al. [28] showed that power consumption of a typical blade server can be reduced from 450 W in the active state to just 10.4 W in the S3 state. The transition latency is currently mostly constrained by the Power Supply Unit (PSU) of the server, which leads to the total latency of approximately 300 ms. This latency is acceptable for the purposes of dynamic VM consolidation, as VM live migrations usually take tens of seconds.

The Linux OS provides an API to programmatically switch the physical machine into the sleep mode. In particular, CentOS supports a *pm-utils* package, which includes command line programs for changing the power state of the machine. First of all, to check whether the Suspend to RAM state is supported, the following command can be used: `pm-is-supported --suspend`. If the command returns 0, the Suspend to RAM state is supported, otherwise it is not supported. If the state is supported, the following command can be used to enable it: `pm-suspend`.

It is possible to reactivate a physical machine over the network using the Wake-on-LAN technology. This technology has been introduced in 1997 by the Advanced Manageability Alliance (AMA) formed by Intel and IBM, and is currently supported by most modern servers. To reactivate a server using Wake-on-LAN, it is necessary to send over the network a special packet, called the *magic packet*. This can be done using the *ether-wake* Linux program as follows: `ether-wake -i interface mac_address`, where `interface` is replaced with the name of the network interface to send the packet from, and `mac_address` is replaced with the actual Media Access Control (MAC) address of the host to be reactivated.

3.5 Local Manager

The local manager component is deployed on every compute host as an OS service. The service periodically executes a function that determines whether it is necessary to reallocate VMs from the host.

A high-level view of the workflow performed by the local manager is shown in Figure 5. At the beginning of each iteration it reads from the local storage the historical data on the resource usage by the VMs and hypervisor stored by the data collector. Then, the local manager invokes the specified in the configuration underload detection algorithm to determine whether the host is underloaded. If the host is underloaded, the local manager sends an underload request to the global manager's REST API to migrate all the VMs from the host and switch the host to a low-power mode.

If the host is not underloaded, the local manager proceeds to invoking the specified in the configuration overload detection algorithm. If the host is overloaded, the local manager invokes the configured VM selection algorithm to select VMs to offload from the host. Once the VMs to migrate from the host are selected, the local manager sends an overload request to the global manager's REST API to migrate the selected VMs. Similarly to the global manager, the local manager can be configured to use custom underload detection, overload detection, and VM selection algorithms using the configuration file.

3.6 Data Collector

The data collector is deployed on every compute host as an OS service and periodically collects the CPU utilization data for each VM running on the host, as well as data on the CPU utilization by the hypervisor. The collected data are stored in the local file-based data store, and also submitted to the central database. The data are stored as the average MHz consumed by a VM during the last measurement interval of length τ. In particular, the CPU usage $C_i^v(t_0, t_1)$ of a VM i, which is a function of the bounds of a measurement interval $[t_0, t_1]$, is calculated as shown in (1).

$$C_i^v(t_0, t_1) = \frac{n_i^v F(\tau_i^v(t_1) - \tau_i^v(t_0))}{t_1 - t_0}, \tag{1}$$

where n_i^v is the number of virtual CPU cores allocated to the VM i; F is the frequency of a single CPU core in MHz; and $\tau_i^v(t)$ is the CPU time consumed by the VM i up to the time t. The CPU usage of the hypervisor $C_j^h(t_0, t_1)$ is calculated as a difference between the overall CPU usage and the CPU usage by the set of VMs allocated to the host, as shown in (2).

$$C_j^h(t_0, t_1) = \frac{n_j^h F(\tau_j^h(t_1) - \tau_j^h(t_0))}{t_1 - t_0} - \sum_{i \in \mathcal{V}_j} C_i^v(t_0, t_1), \tag{2}$$

where n_j^h is the number of physical cores of the host j; $\tau_j^h(t)$ is the CPU time consumed by the host overall up to the time t; and $\mathcal{V}_j$ is the set of VM allocated to the host j. The CPU usage data are stored as integers. This data format is used due to its portability: the stored values can be approximately converted to the CPU utilization percentages for any host or VM type. In general, to more accurately handle heterogeneous environments, this performance metric can be replaced with a more abstract measure, such as EC2 Compute Units (ECUs) in Amazon EC2.

The actual data are obtained using libvirt's API[5] in the form of the CPU time consumed by VMs and hosts overall to date. Using the CPU time collected at the previous time step, the CPU time for the last time interval is calculated. According to the CPU frequency of the host and the length of the time interval, the CPU time is converted into the required average MHz consumed by the VM over the last time interval. Then, using the VMs' CPU utilization data, the CPU utilization by the hypervisor is calculated. The collected data are stored both locally and submitted to the database.

At the beginning of every iteration, the data collector obtains the set of VMs currently running on the host using the Nova API and compares them to the VMs running on the host at the previous time step. If new VMs have been found, the data collector fetches the historical data about them from the central database and stores the data in the local file-based data store. If some VMs have been removed, the data collector removes the data about these VMs from the local data store.

While OpenStack Neat oversubscribes the CPU of hosts by taking advantage of information on the real-time CPU utilization, it does not overcommit RAM. In other words, RAM is still a constraint in placing VMs on hosts; however, the constraint is the maximum amount of RAM that can be used by a VM statically defined by its instance type, rather than the real-time RAM consumption. The reason for that is that RAM is a more critical resource compared with the CPU, as an application may fail due to insufficient RAM, whereas insufficient CPU may just slow down its execution.

[5]The libvirt virtualization API. http://libvirt.org/

3.7 Data Stores

As shown in Figure 2, the system contains two types of data stores:

- *Central database* – a database server, which can be deployed either on the controller host, or on one or more dedicated hosts.
- *Local file-based data storage* – a data store deployed on every compute host and used for temporary caching the resource usage data to use by the local managers in order to avoid excessive database queries.

The details about the data stores are given in the following subsections.

3.7.1 Central Database.

The central database is used for storing historical data on the resource usage by VMs and hypervisors, as well as hardware characteristics of hosts. To ensure scalability, the database can be provided by any existing distributed database system, e.g., the MySQL Cluster [1]. The database is populated by the data collectors deployed on compute hosts. There are two main use cases when the data are retrieved from the central database instead of the local storage of compute hosts. First, it is used by local managers to fetch the resource usage data after VM migrations. Once a VM migration is completed, the data collector deployed on the destination host fetches the required historical data from the database and stores them locally to use by the local manager.

The second use case of the central database is when the global manager computes a new placement of VMs on hosts. VM placement algorithms require information on the resource consumption of all the hosts in order to make global allocation decisions. Therefore, every time there is a need to place VMs on hosts, the global manager queries the database to obtain the up-to-date data on the resource usage by hypervisors and VMs.

Field	**Type**
id	Integer
hostname	String(255)
cpu_mhz	Integer
cpu_cores	Integer
ram	Integer

(a) The *hosts* table

Field	**Type**
id	Integer
host_id	Integer
timestamp	DateTime
cpu_mhz	Integer

(b) The *host_resource_usage* table

Field	**Type**
id	Integer
uuid	String(36)

(c) The *vms* table

Field	**Type**
id	Integer
vm_id	Integer
timestamp	DateTime
cpu_mhz	Integer

(d) The *vm_resource_usage* table

Table 1: The database schema

As shown in Table 1, the database schema contains 4 main tables: *hosts*, *host_resource_usage*, *vms*, and *vm_resource_usage*. The *hosts* table stores information about hosts, such as the host names, CPU frequency of a physical core in MHz, number of CPU cores, and amount of RAM in MB. The *vms* table stores the UUIDs of VMs assigned by OpenStack. The *host_resource_usage* and *vm_resource_usage* tables store data on the resource consumption over time by hosts and VMs respectively.

3.7.2 Local File-Based Data Store.

A local manager at each iteration requires data on the resource usage by the VMs and hypervisor of the corresponding host in order to pass them to the underload / overload detection and VM placement algorithms. To reduce the number of queries to the database over the network, apart from submitting the data into the database, the data collector temporarily stores the data locally. This way, the local manager can just read the data from the local file storage and avoid having to retrieve data from the central database.

The data collector stores the resource usage data locally in *local_data_directory/vms/* as plain text files, where *local_data_directory* is defined in the configuration file discussed in Section 3.8. The data for each VM are stored in a separate file named after the UUID of the VM. The data on the resource usage by the hypervisor are stored in the *local_data_directory/host* file. The format of the files is a new line separated list of integers representing the average CPU consumption in MHz during the subsequent measurement intervals.

3.8 Configuration

The configuration of OpenStack Neat is stored in the */etc/neat/neat.conf* file in the standard INI format using the '#' character for denoting comments. It is assumed that this file exists on all the compute and controller hosts and contains the same configuration. The available configuration options, default values, and descriptions of the options are given in Table 2.

One of the ideas implemented in OpenStack Neat is providing the user with the ability to change the implementation and parameters of any of the 4 VM consolidation algorithms simply by modifying the configuration file. This provides the means of adding to the system and enabling custom VM consolidation algorithms without modifying the source code of the framework. The algorithms are configured using the options with the *algorithm_* prefix. More information on adding and enabling VM consolidation algorithms is given in Section 3.9.

Table 2: OpenStack Neat's configuration options

Configuration option = default value	**Description**
log_directory=/var/log/neat	The directory to store log files.
log_level=3	The level of emitted log messages (0-3).
sql_connection= mysql://neat:neatpassword@controller/neat	The host name and credentials for connecting to the database server specified in the format supported by SQLAlchemy.
os_admin_tenant_name=admin	The admin tenant name authenticating in Nova.
os_admin_user=admin	The admin user name for authenticating in Nova.
os_admin_password=adminpassword	The admin password for authenticating in Nova.
os_auth_url=http://controller:5000/v2.0/	The OpenStack authentication URL.
vm_instance_directory=/var/lib/nova/instances	The directory, where OpenStack Nova stores the data of VM instances.
compute_hosts=compute1, compute2, ...	A coma-separated list of compute host names.
global_manager_host=controller	The global manager's host name.
global_manager_port=60080	The port of the REST web service exposed by the global manager.
db_cleaner_interval=7200	The time interval between subsequent invocations of the database cleaner in seconds.
local_data_directory=/var/lib/neat	The directory used by the data collector to store data on the resource usage by the VMs and hypervisor.
local_manager_interval=300	The time interval between subsequent invocations of the local manager in seconds.
data_collector_interval=300	The time interval between subsequent invocations of the data collector in seconds.
data_collector_data_length=100	The number of the latest data values stored locally by the data collector and passed to the underload / overload detection, and VM placement algorithms.
host_cpu_overload_threshold=0.8	The threshold on the overall (all cores) utilization of the physical CPU of a host, above which the host is considered to be overloaded. This is used for logging host overloads.
host_cpu_usable_by_vms=1.0	The threshold on the overall (all cores) utilization of the physical CPU of a host available for allocation to VMs.

Table 2: OpenStack Neat's configuration options (continued)

Configuration option = default value	**Description**
compute_user=neat	The user name for connecting to the compute hosts for switching them into the sleep mode.
compute_password=neatpassword	The password of the user account used for connecting to the compute hosts for switching them into the sleep mode.
sleep_command=pm-suspend	A shell command used to switch a host into the sleep mode, the *compute_user* must have permissions to execute this command.
ether_wake_interface=eth0	The network interface to send a magic packet from the controller host using the ether-wake program.
network_migration_bandwidth=10	The network bandwidth in MB/s available for VM live migrations.
algorithm_underload_detection_factory= neat.locals.underload.trivial. last_n_average_threshold_factory	The fully qualified name of a Python factory function that returns a function implementing an underload detection algorithm.
algorithm_underload_detection_parameters= {"threshold": 0.5, "n": 2}	JSON encoded parameters to be parsed and passed to the specified underload detection algorithm factory.
algorithm_overload_detection_factory= neat.locals.overload.mhod.core.mhod_factory	The fully qualified name of a Python factory function that returns a function implementing an overload detection algorithm.
algorithm_overload_detection_parameters= {"state_config": [0.8], "otf": 0.2, "history_size": 500, "window_sizes": [30, 40, 50, 60, 70, 80, 90, 100], "bruteforce_step": 0.2, "learning_steps": 10}	JSON encoded parameters to be parsed and passed to the specified overload detection algorithm factory.
algorithm_vm_selection_factory= neat.locals.vm_selection.algorithms. minimum_migration_time_max_cpu_factory	The fully qualified name of a Python factory function that returns a function implementing a VM selection algorithm.
algorithm_vm_selection_parameters= {"n": 2}	JSON encoded parameters to be parsed and passed to the specified VM selection algorithm factory.
algorithm_vm_placement_factory= neat.globals.vm_placement. bin_packing.best_fit_decreasing_factory	The fully qualified name of a Python factory function that returns a function implementing a VM placement algorithm.
algorithm_vm_placement_parameters= {"cpu_threshold": 0.8, "ram_threshold": 0.95, "last_n_vm_cpu": 2}	JSON encoded parameters to be parsed and passed to the specified VM placement algorithm factory.

3.9 Extensibility of the Framework

One of the main points of the framework's extensibility is the ability to add new VM consolidation algorithm to the system and enable them by updating the configuration file without the necessity in modifying the source code of the framework itself. There are 4 algorithms that can be changed through a modification of the configuration file: underload / overload detection, VM selection, and VM placement algorithms. The values of the corresponding configuration options should be fully qualified names of functions available as a part of one of the installed Python libraries. The fact that the functions are specified by their fully qualified names also means that they can be installed as a part of a Python library independent from OpenStack Neat. The 4 corresponding configuration options are the following:

1. *algorithm_underload_detection_factory*
2. *algorithm_overload_detection_factory*
3. *algorithm_vm_selection_factory*

4. *algorithm_vm_placement_factory*

Since an algorithm may need to be initialized prior to its usage, we applied the factory function pattern. The functions specified as values of any of the *algorithm_*_factory* configuration options are not functions that actually implement VM consolidation algorithms, rather they are functions that return initialized instances of functions implementing the corresponding VM consolidation algorithms. Algorithm 1 shows a simple example of a factory function for a *threshold* overload detection algorithm implemented in Python.

Algorithm 1: A factory function of a threshold overload detection algorithm

```
def threshold_factory (time_step, migration_time, params):
    return lambda utilization, state = None: (
        threshold (params['threshold'], utilization),
        {})
```

All functions implementing VM consolidation algorithms and their factories should adhere to the corresponding predefined interfaces. For example, all factory functions of overload detection algorithms must accept a time step, migration time, and algorithm parameters as arguments, as shown in Algorithm 1. The function must return another function that implements the required consolidation algorithm, which in turn must follow the interface of an overload detection algorithm.

Every function implementing an overload detection algorithm must: (1) accept as arguments a list of CPU utilization percentages and dictionary representing the state of the algorithm; and (2) return a tuple containing the decision of the algorithm as a boolean and updated state dictionary. In the simple example shown in Algorithm 1, the *threshold* algorithm is stateless; therefore, it returns an empty dictionary as the state. Definitions of the interfaces of functions implementing VM consolidation algorithms and their factories are given in Table 3. The types and descriptions of the arguments are given in Table 4.

Using the *algorithm_*_parameters* configuration options, it is possible to pass arbitrary dictionaries of parameters to VM consolidation algorithm factory functions. The parameters must be specified as an object in the JSON format on a single line. The specified JSON strings are automatically parsed by the system and passed to factory functions as Python dictionaries. Apart from being parameterized, a consolidation algorithm may also preserve state across invocations. This can be useful for implementing stateful algorithms, or as a performance optimization measure, e.g., to avoid repeating costly computations. Preserving state is done by accepting a state dictionary as an argument, and returning the updated dictionary as the second element of the return tuple.

Table 3: Interfaces of VM consolidation algorithms and their factory functions

Algorithm	**Factory arguments**	**Algorithm arguments**	**Algorithm return**
Underload detection	1. time_step: int, ≥ 0 2. migration_time: float, ≥ 0 3. params: dict(str: *)	1. cpu_utilization: list(float) 2. state: dict(str: *)	1. decision: bool 2. state: dict(str: *)
Overload detection	1. time_step: int, ≥ 0 2. migration_time: float, ≥ 0 3. params: dict(str: *)	1. cpu_utilization: list(float) 2. state: dict(str: *)	1. decision: bool 2. state: dict(str: *)
VM selection	1. time_step: int, ≥ 0 2. migration_time: float, ≥ 0 3. params: dict(str: *)	1. vms_cpu: dict(str : list(int)) 2. vms_ram: dict(str : list(int)) 3. state: dict(str: *)	1. vms: list(str) 2. state: dict(str: *)
VM placement	1. time_step: int, ≥ 0 2. migration_time: float, ≥ 0 3. params: dict(str: *)	1. hosts_cpu_usage: dict(str : int) 2. hosts_cpu_total: dict(str : int) 3. hosts_ram_usage: dict(str : int) 4. hosts_ram_total: dict(str : int) 5. inactive_hosts_cpu: dict(str : int) 6. inactive_hosts_ram: dict(str : int) 7. vms_cpu: dict(str : list(int)) 8. vms_ram: dict(str : list(int)) 9. state: dict(str: *)	1. alloc.: dict(str: str) 2. state: dict(str: *)

Table 4: Arguments of VM consolidation algorithms and their factory functions

Argument name	Type	Description
time_step	int, ≥ 0	The length of the time step in seconds.
migration_time	float, ≥ 0	The VM migration time in time seconds.
params	dict(str: *)	A dictionary containing the algorithm's parameters parsed from the JSON representation specified in the configuration file.
cpu_utilization	list(float)	A list of the latest CPU utilization percentages in the $[0, 1]$ range calculated from the combined CPU usage by all the VMs allocated to the host and the hypervisor.
state	dict(str: *)	A dictionary containing the state of the algorithm passed over from the previous iteration.
vms_cpu	dict(str : list(int))	A dictionary of VM UUIDs mapped on the lists of the latest CPU usage values by the VMs in MHz.
vms_ram	dict(str : int)	A dictionary of VM UUIDs mapped on the maximum allowed amounts of RAM for the VMs in MB.
host_cpu_usage	dict(str : int)	A dictionary of host names mapped on the current combined CPU usage in MHz.
host_cpu_total	dict(str : int)	A dictionary of host names mapped on the total CPU capacities in MHz calculated as a multiplication of the frequency of a single physical core by the number of cores.
host_ram_usage	dict(str : int)	A dictionary of host names mapped on the current amounts of RAM in MB allocated to the VMs of the hosts.
host_ram_total	dict(str : int)	A dictionary of host names mapped on the total amounts of RAM in MB available to VMs on the hosts.
inactive_hosts_cpu	dict(str : int)	A dictionary of the names of the currently inactive hosts mapped on the total CPU capacities in MHz.
inactive_hosts_ram	dict(str : int)	A dictionary of the names of the currently inactive hosts mapped on the total amounts of RAM in MB available to VMs on the hosts.

Currently, the data collector only collects data on the CPU utilization. It is possible to extend the system to collect other types of data that may be passed to the VM consolidation algorithms. To add another type of data, it is necessary to extend the *host_resource_usage* and *vm_resource_usage* database tables by adding new fields for storing the new types of data. Then, the *execute* function of the data collector should be extended to include the code required to obtain the new data and submit them to the database. Finally, the local and global managers need to be extended to fetch the new type of data from the database to be passed to the appropriate VM consolidation algorithms.

3.10 Deployment

OpenStack Neat needs to be deployed on all the compute and controller hosts. The deployment consists in installing dependencies, cloning the project's Git repository, installing the project, and starting up the services. The process is cumbersome since multiple steps should be performed on each host. The OpenStack Neat distribution includes a number of Shell scripts that simplify the deployment process. The following steps are required to perform a complete deployment of OpenStack Neat:

1. Clone the project's repository on the controller host by executing:

   ```
   git clone git://github.com/beloglazov/openstack-neat.git
   ```

2. Install the required dependencies by executing the following command from the cloned repository if the OS of the controller is CentOS: `./setup/deps-centos.sh`

3. In the cloned repository, modify *neat.conf* to meet the requirements. In particular, it is necessary to enter the names of the available compute hosts. It is also necessary to create a database on the database server accessible with the details specified in the configuration file.

4. Install OpenStack Neat on the controller host by executing the following command from the project's directory: `sudo python setup.py install`. This command will also copy the modified configuration file to */etc/neat/neat.conf*.

5. Using the scripts provided in the package, it is possible to install OpenStack Neat on all the compute hosts specified in the configuration file remotely from the controller. First, the following command can be used to clone the repository on all the compute hosts: `./compute-clone-neat.py`.

6. Once the repository is cloned, OpenStack Neat and its dependencies can be installed on all the compute hosts by executing the two following commands on the controller: `./compute-install-deps.py; ./compute-install-neat.py`

7. Next, it is necessary to copy the modified configuration file to the compute hosts, which can be done by the following command: `./compute-copy-conf.py`

8. All OpenStack Neat services can be started on the controller and compute hosts with the following single command `./all-start.sh`

Once all the steps listed above are completed, OpenStack Neat's services should be deployed and started up. If any service fails, the log files can be found in */var/log/neat/* on the corresponding host.

4 VM Consolidation Algorithms

As mentioned earlier, OpenStack Neat is based on the approach to the problem of dynamic VM consolidation, proposed in our previous work [8], which consists in dividing the problem into 4 sub-problems: (1) host underload detection; (2) host overload detection; (3) VM selection; and (4) VM placement. The implementation of the framework includes several algorithms proposed in our previous works [7–9] with slight modifications. The overview of the implemented algorithms is given for reference only. It is important to note that the presented algorithms are not the main focus of the current paper. The focus of the paper is the design of the framework for dynamic VM consolidation, which is capable of handling multiple implementations of consolidation algorithms, and can be switched between the implementations through configuration.

4.1 Host Underload Detection

In our experiments, we use a simple heuristic for the problem of underload detection shown in Algorithm 2. The algorithm calculates the mean of the n latest CPU utilization measurements and compares it to the specified threshold. If the mean CPU utilization is lower than the threshold, the algorithm detects a host underload situation. The algorithm accepts 3 arguments: the CPU utilization threshold, the number of last CPU utilization values to average, and a list of CPU utilization measurements.

Algorithm 2: The averaging threshold-based underload detection algorithm

```
1 Input: threshold, n, utilization   Output: whether the host is underloaded
2 if not empty utilization then
3     utilization ← last n values of utilization
4     mean_utilization ← sum(utilization) / len(utilization)
5     return mean_utilization ≤ threshold
6 return False
```

4.2 Host Overload Detection

OpenStack Neat includes several overload detection algorithms, which can be enabled by modifying the configuration file. One of the simple included algorithms is the averaging Threshold-based (THR) overload detection algorithm. The algorithm is similar to Algorithm 2, while the only difference is that it detects overload situations if the mean of the n last CPU utilization measurements is *higher* than the specified threshold.

Another overload detection algorithm included in the default implementation of OpenStack Neat is based on estimating the future CPU utilization using *local regression* (i.e., the Loess method), referred to as the Local Regression Robust (LRR) algorithm [9] shown in Algorithm 3. The algorithm calculates the Loess parameter estimates, and uses them to predict the future CPU utilization at the next time step taking into account the VM migration time. In addition, the LR algorithm accepts a safety parameter, which is used to scale the predicted CPU utilization to increase or decrease the sensitivity of the algorithm to potential overloads.

Algorithm 3: The Local Regression Robust (LRR) overload detection algorithm

```
Input: threshold, param, n, migration_time, utilization
Output: whether the host is overloaded
if len(utilization) < n then
    return False
estimates ← loess_parameter_estimates(last n values of utilization)
prediction ← estimates[0] + estimates[1] * (n + migration_time)
return param * prediction ≥ threshold
```

A more complex overload detection algorithm included in OpenStack Neat is the Markov Overload Detection (MHOD) algorithm, which enables the system administrator to explicitly specify a constraint on the OTF value as a parameter of the algorithm, while maximizing the time between VM migrations, thus, improving the quality of VM consolidation [8].

4.3 VM Selection

Once a host overload has been detected, it is necessary to determine what VMs are the best to be migrated from the host. This problem is solved by VM selection algorithms. An example of such an algorithm is simply randomly selecting a VM from the set of VMs allocated to the host. Another algorithm shown in Algorithm 4 is called Minimum Migration Time Maximum CPU utilization (MMTMC). This algorithm first selects VMs with the minimum amount of RAM to minimize the live migration time. Then, out of the selected subset of VMs, the algorithm selects the VM with the maximum CPU utilization averaged over the last n measurements to maximally reduce the overall CPU utilization of the host.

Algorithm 4: The Minimum Migration Time Maximum CPU utilization (MMTMC) algorithm

```
Input: n, vms_cpu_map, vms_ram_map   Output: a VM to migrate
min_ram ← min(values of vms_ram_map)
max_cpu ← 0
selected_vm ← None
foreach vm, cpu in vms_cpu_map do
    if vms_ram_map[vm] > min_ram then
        continue
    vals ← last n values of cpu
    mean ← sum(vals) / len(vals)
    if max_cpu < mean then
        max_cpu ← mean
        selected_vm ← vm
return selected_vm
```

4.4 VM Placement

The VM placement problem can be seen as a bin packing problem with variable bin sizes, where bins represent hosts; bin sizes are the available CPU capacities of hosts; and items are VMs to be allocated with an extra constraint on the amount of RAM. As the bin packing problem is NP-hard, it is appropriate to apply a heuristic to solve it. OpenStack Neat implements a modification of the Best Fit Decreasing

(BFD) algorithm, which has been shown to use no more than $11/9 \cdot OPT + 1$ bins, where OPT is the number of bins of the optimal solution [37].

The implemented modification of the BFD algorithm shown in Algorithm 5 includes several extensions: the ability to handle extra constraints, namely, consideration of currently inactive hosts, and a constraint on the amount of RAM required by the VMs. An inactive host is only activated when a VM cannot be placed on one of the already active hosts. The constraint on the amount of RAM is taken into account in the first fit manner, i.e., if a host is selected for a VM as a best fit according to its CPU requirements, the host is confirmed if it just satisfies the RAM requirements. In addition, similarly to the averaging underload and overload detection algorithms, the algorithm uses the mean values of the last n CPU utilization measurements as the CPU constraints. The worst-case complexity of the algorithm is $(n + m/2)m$, where n is the number of physical nodes, and m is the number of VMs to be placed. The worst case occurs when every VM to be placed requires a new inactive host to be activated.

Algorithm 5: The Best Fit Decreasing (BFD) VM placement algorithm

```
Input: n, hosts_cpu, hosts_ram, inactive_hosts_cpu, inactive_hosts_ram, vms_cpu, vms_ram
Output: a map of VM UUIDs to host names
vm_tuples ← empty list
foreach vm, cpu in vms_cpu do
    vals ← last n values of cpu
    append a tuple of the mean of vals, vms_ram[vm], and vm to vm_tuples
vms ← sort_decreasing(vm_tuples)
host_tuples ← empty list
foreach host, cpu in hosts_cpu do
    append a tuple of cpu, hosts_ram[host], host to host_tuples
hosts ← sort_increasing(host_tuples)
inactive_host_tuples ← empty list
foreach host, cpu in inactive_hosts_cpu do
    append a tuple of cpu, inactive_hosts_ram[host], host to inactive_host_tuples
inactive_hosts ← sort_increasing(inactive_host_tuples)
mapping ← empty map
foreach vm_cpu, vm_ram, vm_uuid in vms do
    mapped ← False
    while not mapped do
        allocated ← False
        foreach _, _, host in hosts do
            if hosts_cpu[host] ≥ vm_cpu and hosts_ram[host] ≥ vm_ram then
                mapping[vm_uuid] ← host
                hosts_cpu[host] ← hosts_cpu[host]- vm_cpu
                hosts_ram[host] ← hosts_ram[host]- vm_ram
                mapped ← True
                allocated ← True
                break
        if not allocated then
            if not empty inactive_hosts then
                activated_host ← pop the first from inactive_hosts
                append activated_host to hosts
                hosts ← sort_increasing(hosts)
                hosts_cpu[activated_host[2]] ← activated_host[0]
                hosts_ram[activated_host[2]] ← activated_host[1]
            else
                break
if len(vms) == len(mapping) then
    return mapping
return empty map
```

5 Implementation

OpenStack Neat is implemented in Python. The choice of the programming language has been mostly determined by the fact that OpenStack itself is implemented in Python; therefore, using the same programming language could potentially simplify the integration of the two projects. Since Python is a dynamic language, it has a number of advantages, such as concise code, no type constraints, and *monkey patching*, which refers to the ability to replace methods, attributes, and functions at run-time. Due to its flexibility and expressiveness, Python typically helps to improve productivity and reduce the development time compared with statically typed languages, such as Java and C++. The downsides of dynamic typing are the lower run-time performance and lack of compile time guarantees provided by statically typed languages.

To compensate for the reduced safety due to the lack of compile time checks, several programming techniques are applied in the implementation of OpenStack Neat to minimize bugs and simplify maintenance. First of all, the functional programming style is followed by leveraging the functional features of Python, such as higher-order functions and closures, and minimizing the use of the object-oriented programming features, such as class hierarchies and encapsulation. One important technique that is applied in the implementation of OpenStack Neat is the minimization of mutable state. Mutable state is one of the causes of side effects, which prevent functions from being referentially transparent. This means that if a function relies on some global mutable state, multiple calls to that function with the same arguments do not guarantee the same result returned by the function for each call.

The implementation of OpenStack Neat tries to minimize side effects by avoiding mutable state where possible, and isolating calls to external APIs in separate functions covered by unit tests. In addition, the implementation splits the code into small easy to understand functions with explicit arguments that the function acts upon without mutating their values. To impose constraints on function arguments, the Design by Contract (DbC) approach is applied using the PyContracts library. The approach prescribes the definition of formal, precise, and verifiable interface specifications for software components. PyContracts lets the programmer to specify contracts on function arguments via a special format of Python docstrings. The contracts are checked at run-time, and if any of the constraints is not satisfied, an exception is raised. This approach helps to localize errors and fail fast, instead of hiding potential errors. Another advantage of DbC is comprehensive and up-to-date code documentation, which can be generated from the source code by automated tools.

To provide stronger guarantees of the correctness of the program, it is important to apply unit testing. According to this method, each individual unit of source code, which in this context is a function, should be tested by an automated procedure. The goal of unit testing is to isolate parts of the program and show that they perform correctly. One of the most efficient unit testing techniques is implemented by the Haskell QuickCheck library. This library allows the definition of tests in the form of properties that must be satisfied, which do not require the manual specification of the test case input data. QuickCheck takes advantage of Haskell's rich type system to infer the required input data and generates multiple test cases automatically.

The implementation of OpenStack neat uses Pyqcy, a QuickCheck-like unit testing framework for Python. This library allows the specification of *generators*, which can be seen as templates for input data. Similarly to QuickCheck, Pyqcy uses the defined templates to automatically generate input data for hundreds of test cases for each unit test. Another Python library used for testing of OpenStack Neat is Mocktest. This library leverages the flexibility of Python's monkey patching to dynamically replace, or *mock*, existing methods, attributes, and functions at run-time. Mocking is essential for unit testing the code that relies on calls to external APIs. In addition to the ability to set artificial return values of methods and functions, Mocktest allows setting expectations on the number of the required function calls. If the expectations are not met, the test fails. Currently, OpenStack Neat includes more than 150 unit tests.

OpenStack Neat applies Continuous Integration (CI) using the Travis CI service[6]. The aim of the CI practice is to detect integration problems early by periodically building and deploying the software system. Travis CI is attached to OpenStack Neat's source code repository through Git hooks. Every time modifications are pushed to the repository, Travis CI fetches the source code and runs a clean installation in a sandbox followed by the unit tests. If any step of the integration process fails, Travis CI reports the problem.

Despite all the precautions, run-time errors may occur in a deployed system. OpenStack Neat implements multi-level logging functionality to simplify the post-mortem analysis and debugging process. The

[6]OpenStack Neat on Travis CI. http://travis-ci.org/beloglazov/openstack-neat

Table 5: The OpenStack Neat codebase summary

Package	Files	Lines of code	Lines of comments
Core	21	2,144	1,946
Tests	20	3,419	260

Table 6: Open source libraries used by OpenStack Neat

Library	License	Description
Distribute	Python 2.0	A library for managing Python projects and distributions. `http://bitbucket.org/tarek/distribute`
Pyqcy	FreeBSD	A QuickCheck-like unit testing framework for Python. `http://github.com/Xion/pyqcy`
Mocktest	LGPL	A Python library for mocking objects and functions. `http://github.com/gfxmonk/mocktest`
PyContracts	LGPL	A Python library for Design by Contract (DbC). `http://github.com/AndreaCensi/contracts`
SQLAlchemy	MIT	A Python SQL toolkit, also used by the core OpenStack services. `http://www.sqlalchemy.org/`
Bottle	MIT	A micro web-framework for Python. `http://bottlepy.org/`
Requests	ISC	A Python HTTP client library. `http://python-requests.org/`
libvirt	LGPL	A virtualization toolkit with Python bindings. `http://libvirt.org/`
Python-novaclient	Apache 2.0	A Python Nova API client implementation. `http://github.com/openstack/python-novaclient`
NumPy	BSD	A library for scientific computing. `http://numpy.scipy.org/`
SciPy	BSD	A library of extra tools for scientific computing. `http://scipy.org/`

verbosity of logging can be adjusted by modifying the configuration file. Table 5 provides information on the size of the current codebase of OpenStack Neat. Table 6 summarizes the set of open source libraries used in the implementation of OpenStack Neat.

6 A Benchmark Suite for Evaluating Distributed Dynamic VM Consolidation Algorithms

Currently, research in the area of dynamic VM consolidation is limited by the lack of a standardized suite of benchmark software, workload traces, performance metrics, and evaluation methodology. Most of the time, researchers develop their own solutions for evaluating the proposed algorithms, which are not publicly available later on. This complicates further research efforts in the area due to the limited opportunities for comparing new results with prior solutions. Moreover, the necessity in implementing custom evaluation software leads to duplication of efforts. In this work, we propose an initial version of a benchmark suite for evaluating dynamic VM consolidation algorithms following the distributed approach of splitting the problem into 4 sub-problems discussed earlier. The proposed benchmark suite consists of 4 major components:

1. OpenStack Neat, a framework for distributed dynamic VM consolidation in OpenStack Clouds providing a base system implementation and allowing configuration-based switching of different implementations of VM consolidation algorithms.

2. A set of workload traces containing data on the CPU utilization collected every 5 minutes from more than a thousand PlanetLab VMs deployed on servers located in more than 500 places around the world [31].

3. A set of performance metrics capturing the following aspects: quality of VM consolidation; quality of service delivered by the system; overhead of VM consolidation in terms of the number of VM

migration; and execution time of the consolidation algorithms.

4. Evaluation methodology prescribing the approach of preparing experiments, deploying the system, generating workload using the PlanetLab traces, as well as processing and analyzing the results.

We believe that the availability of such a benchmark suite will foster and facilitate research efforts and future advancements in the area of dynamic VM consolidation. In addition, researchers are encouraged to publicize and share the implemented consolidation algorithms to simplify performance comparisons with future solutions. One approach to sharing algorithm implementations is the addition of an extra package to the main branch of OpenStack Neat that will contain contributed algorithms. This would provide a central location, where anyone can find the up-to-date set of consolidation algorithms to use in their research. Therefore, processing and managing the inclusion of such submissions into the main public repository of OpenStack Neat will be done as a part of the project. The following sections provide more information on the workload traces, performance metrics, and evaluation methodology of the proposed benchmark suite. The performance evaluation discussed in Section 7 is an example of application of the benchmark suite.

6.1 Workload Traces

To make experiments reproducible, it is important to rely on a set of input traces to reliably generate the workload, which would allow the experiments to be repeated as many times as necessary. It is also important to use workload traces collected from a real system rather than artificially generated, as this would help to reproduce a realistic scenario. This chapter uses workload trace data provided as a part of the CoMon project, a monitoring infrastructure of PlanetLab [31]. The traces include data on the CPU utilization collected every 5 minutes from more than a thousand VMs deployed on servers located in more 500 places around the world. 10 days of workload traces collected during March and April 2011 have been randomly chosen, which resulted in the total of 11,746 24-hour long traces. The full set of workload traces is publicly available online[7].

The workload from PlanetLab VMs is representative of an IaaS Cloud environment, such as Amazon EC2, in the sense that the VMs are created and managed by multiple independent users, and the infrastructure provider is not aware of what particular applications are executing in the VMs. Furthermore, this implies that the overall system workload is composed of multiple heterogeneous applications, which also corresponds to a typical IaaS environment.

To stress the system in the experiments, the original workload traces have been filtered to leave only the ones that exhibit high variability. In particular, only the traces that satisfy the following two conditions have been selected: (1) at least 10% of time the CPU utilization is lower than 20%; and (2) at least 10% of time the CPU utilization is higher than 80%. This significantly reduced the number of workload traces resulting in only 33 out of 11,746 24-hour traces left. The set of selected traces and filtering script are available online [6].

The resulting number of traces was sufficient for the experiments, whose scale was limited by the size of the testbed described in Section 7.1. If a larger number of traces is required to satisfy larger scale experiments, one approach is to relax the conditions of filtering the original set of traces. Another approach is to randomly sample with replacement from the limited set of traces. If another set of suitable workload traces becomes publicly available, it can be included in the benchmark suite as an alternative.

6.2 Performance Metrics

For effective performance evaluation and comparison of algorithms it is essential to define performance metrics that capture the relevant characteristics of the algorithms. One of the objectives of dynamic VM consolidation is the minimization of energy consumption by the physical nodes, which can be a metric for performance evaluation and comparison. However, energy consumption is highly dependent on the particular model and configuration of the underlying hardware, efficiency of power supplies, implementation of the sleep mode, etc. A metric that abstracts from the mentioned factors, but is directly proportional and can be used to estimate energy consumption, is the time of a host being idle, aggregated over the full set of hosts. Using this metric, the quality of VM consolidation can be represented by the increase in the aggregated idle time of hosts. However, this metric depends on the length of the overall evaluation period and the number of hosts. To eliminate this dependency, we

[7]The PlanetLab traces. http://github.com/beloglazov/planetlab-workload-traces

propose a normalized metric referred to as the Aggregated Idle Time Fraction (AITF) defined as shown in (3).

$$AITF = \frac{\sum_{h \in \mathcal{H}} t_i(h)}{\sum_{h \in \mathcal{H}} t_a(h)}, \tag{3}$$

where $\mathcal{H}$ is a set of hosts; $t_i(h)$ is the idle time of the host h; and $t_a(h)$ is the total activity time of the host h. To quantify the overall QoS delivered by the system, we apply the Aggregated Overload Time Fraction (AOTF), which is based on the OTF metric [8] and defined as shown in (4).

$$AOTF(u_t) = \frac{\sum_{h \in \mathcal{H}} t_o(h, u_t)}{\sum_{h \in \mathcal{H}} t_b(h)}, \tag{4}$$

where $t_o(h, u_t)$ is the overload time of the host h calculated according to the overload threshold u_t; and $t_b(h)$ is the total busy (non-idle) time of the host h. We propose evaluating the overhead of dynamic VM consolidation in the system in terms of the number of VM migrations initiated as a part of dynamic consolidation. Apart from that, the execution time of various components of the system including the execution time of the VM consolidation algorithms is evaluated.

6.3 Performance Evaluation Methodology

One of the key points of the proposed performance evaluation methodology is the minimization of manual steps required to run an experiment through automation. Automation begins from scripted installation of the OS, OpenStack services and their dependencies on the testbed's nodes, as described in the OpenStack installation guide [11]. The next step is writing scripts for preparing the system for an experiment, which includes starting up the required services, booting VM instances, and preparing them for starting the workload generation.

While most of the mentioned steps are trivial, workload generation is complicated by the requirement of synchronizing the time of starting the workload generation on all the VMs. Another important aspect of workload generation is the way workload traces are assigned to VMs. Typically, the desired behavior is assigning a unique workload trace out of the full set of traces to each VM. Finally, it is necessary to create and maintain a specific level of CPU utilization for the whole interval between changes of the CPU utilization level defined by the workload trace for each VM.

This problem is addressed using a combination of a CPU load generation program[8], and a workload distribution web service and clients deployed on VMs [6]. When a VM boots from a pre-configured image, it automatically starts a script that polls the central workload distribution web service to be assigned a workload trace. Initially, the workload distribution web service drops requests from clients deployed on VMs to wait for the moment when all the required VM instances are booted up and ready for generating workload. When all clients are ready, the web service receives a command to start the workload trace distribution. The web service starts replying to clients by sending each of them a unique workload trace. Upon receiving a workload trace, every client initiates the CPU load generator and passes the received workload trace as an argument. The CPU load generator reads the provided workload trace file, and starts generating CPU utilization levels corresponding to the values specified in the workload trace file for each time frame.

During an experiment, OpenStack Neat continuously logs various events into both the database and log files on each host. After the experiment, the logged data are used by special result processing scripts to extract the required information and compute performance metrics discussed in Section 6.2, as well as the execution time of various system components. This process should be repeated for each combination of VM consolidation algorithms under consideration. Once the required set of experiments is completed, other scripts are executed to perform automated statistical tests and plotting graphs for comparing the algorithms.

The next section presents an example of application of the proposed benchmark suite, and in particular applies: (1) OpenStack Neat as the dynamic VM consolidation framework; (2) the filtered PlanetLab workload traces discussed in Section 6.1; (3) the performance metrics defined in Section 6.2; and (4) the proposed evaluation methodology. The full set of scripts used in the experiments is available online [6].

[8]The CPU load generator. http://github.com/beloglazov/cpu-load-generator

7 Performance Evaluation

In this section, we apply the benchmark suite proposed in Section 6 to evaluate OpenStack Neat and several dynamic VM consolidation algorithm discussed in Section 4.

7.1 Experimental Testbed

The testbed used for performance evaluation of the system consisted of the following hardware:

- 1 x Dell Optiplex 745
  - Intel(R) Core(TM) 2 CPU (2 cores, 2 threads) 6600 @ 2.40GHz
  - 2GB DDR2-667
  - Seagate Barracuda 80GB, 7200 RPM SATA II (ST3808110AS)
  - Broadcom 5751 NetXtreme Gigabit Controller
- 4 x IBM System x3200 M3
  - Intel(R) Xeon(R) CPU (4 cores, 8 threads), X3460 @ 2.80GHz
  - 4GB DDR3-1333
  - Western Digital 250 GB, 7200 RPM SATA II (WD2502ABYS-23B7A)
  - Dual Gigabit Ethernet (2 x Intel 82574L Ethernet Controller)
- 1 x Netgear ProSafe 16-Port 10/100 Desktop Switch FS116

The Dell Optiplex 745 machine was chosen to serve as the controller host running all the major OpenStack services and the global manager of OpenStack Neat. The 4 IBM System x3200 M3 servers were used as compute hosts, i.e. running OpenStack Nova, and local managers and data collectors of OpenStack Neat. All the machines formed a local network connected via the Netgear FS116 network switch.

Unfortunately, there was a hardware problem preventing the system from taking advantage of dynamic VM consolidation to save energy. The problem was that the compute nodes of our testbed did not support the Suspend to RAM power state, which is the most suitable for the purpose of dynamic VM consolidation. This state potentially provides very low switching latency, on the order of 300 ms, while reducing the energy consumption to a negligible level [28]. Another approach would be to shut down inactive nodes; however, the latency is too high quickly re-enable them. Therefore, rather than measuring the actual energy consumption by the servers, the AITF metric introduced in Section 6.2 was used to compare algorithms and estimate potential energy savings.

7.2 Experimental Setup and Algorithm Parameters

From the point of view of experimenting with close to real world conditions, it is interesting to allocate as many VMs on a compute host as possible. This would create a more dynamic workload and stress the system. At the same time, it is important to use full-fledged VM images representing realistic user requirements. Therefore, the Ubuntu 12.04 Cloud Image [4] was used in the experiments, which is one of the Ubuntu VM images available in Amazon EC2.

Since the compute hosts of the testbed contained limited amount of RAM, to maximize the number of VMs served by a single host, it was necessary to use a VM instance type with the minimum amount of RAM sufficient for Ubuntu 12.04. The minimum required amount of RAM was empirically determined to be 128 MB. This resulted in the maximum of 28 VMs being possible to instantiate on a single compute host. Therefore, to maximize potential benefits of dynamic VM consolidation on the testbed containing 4 compute nodes, the total number of VM instances was set to 28, so that in an ideal case all of them can be placed on a single compute host, while the other 3 hosts are kept idle. Out of the 33 filtered PlanetLab workload traces discussed in Section 6.1, 28 traces were randomly selected, i.e., one unique 24-hour trace for each VM instance. The full set of selected traces is available online [6].

During the experiments, all the configuration parameters of OpenStack Neat were set to their default values except for the configuration of the overload detection algorithm. The overload detection algorithm was changed for each experiment by going through the following list of algorithms and their parameters:

Table 7: The experimental results (mean values with 95% CIs)

Algorithm	AITF	AOTF	VM migrations
THR-0.8	36.9% (35.6, 38.2)	15.4% (12.5, 18.3)	167.7 (152.7, 182.6)
THR-0.9	43.0% (42.6, 43.5)	27.0% (25.7, 28.1)	75.3 (70.2, 80.5)
THR-1.0	49.2% (49.2, 49.4)	42.2% (33.0, 51.3)	11.3 (9.9, 12.8)
LRR-1.1	37.9% (37.9, 38.0)	17.8% (12.8, 22.7)	195.7 (158.3, 233.0)
LRR-1.0	40.3% (38.1, 42.4)	23.8% (21.4, 26.1)	93.7 (64.6, 122.8)
LRR-0.9	47.3% (45.2, 49.4)	34.4% (28.8, 40.0)	28.3 (23.2, 33.5)
MHOD-0.2	37.7% (36.8, 38.5)	16.0% (13.5, 18.5)	158.3 (153.2, 163.5)
MHOD-0.3	38.1% (37.7, 38.5)	17.9% (16.8, 18.9)	138.0 (81.6, 194.4)
MHOD-0.4	40.7% (37.0, 44.4)	21.4% (16.7, 26.0)	116.3 (26.6, 206.0)
MAX-ITF	49.2% (49.1, 49.3)	40.4% (35.8, 44.9)	14.0 (7.4, 20.6)

1. MAX-ITF algorithm – a base line algorithm, which never detects host overloads leading to the maximum ITF for the host, where the algorithm is used.
2. The THR algorithm with the n parameter set to 2, and the CPU utilization threshold set to 0.8, 0.9, and 1.0.
3. The LRR algorithm with the safety parameter set to 0.9, 1.0, and 1.1.
4. The MHOD algorithm with the OTF parameter set to 0.2, 0.3, and 0.4.

Each experiment was run 3 times to handle the variability caused by random factors, such as the initial VM placement, workload trace assignment, and component communication latency. All the system initialization and result processing scripts, along with the experiment result packages are available online [6].

7.3 Experimental Results and Analysis

The results of experiments are graphically depicted in Figure 6. The mean values of the obtained AITF and AOTF metrics, and the number of VM migrations along with their 95% Confidence Intervals (CIs) are displayed in Table 7. The results of MAX-ITF show that for the current experiment setup it is possible to obtain high values of AITF of up to 50%, while incurring a high AOTF of more than 40%. All the THR, LRR, and MHOD allow tuning of the AITF values by adjusting the algorithm parameters. For the THR algorithm, the mean AITF increases from 36.9% to 49.2% with the corresponding decrease in the QoS level from 15.4% to 42.2% by varying the CPU utilization threshold from 0.8 to 1.0. The mean number of VM migrations decreases from 167.7 for the 80% threshold to 11.3 for the 100% threshold. The THR algorithm with the CPU utilization threshold set to 100% reaches the mean AITF shown by the MAX-ITF algorithm, which is expected as setting the threshold to 100% effectively disables host overload detection. Similarly, adjusting the safety parameter of the LRR algorithm from 1.1 to 0.9 leads to an increase of the mean AITF from 37.9% to 47.3% with a growth of the mean AOTF from 17.8% to 34.4% and decrease of the mean number of VM migrations from 195.7 to 28.3. THR-1.0 reaches the mean AITF of 49.2% with the mean AOTF of 42.2%, while LRR-0.9 reaches a close mean AITF of 47.3% with the mean AOTF of only 34.4%, which is a significant decrease compared with the AOTF of THR-1.0.

Varying the OTF parameter of the MHOD algorithm from 0.2 to 0.4 leads to an increase of the mean AITF from 37.7% to 40.7% with an increase of the mean AOTF from 16.0% to 21.4%. First of all, it is important to note that the algorithm meets the specified QoS constraint by keeping the value of the AOTF metric below the specified OTF parameters. However, the resulting mean AOTF is significantly lower than the specified OTF parameters: 17.9% for the 30% OTF, and 21.4% for the 40% OTF. This can be explained by a combination of two factors: (1) the MHOD algorithm is parameterized by the per-host OTF, rather than AOTF, which means that it meets the OTF constraint for each host independently; (2) due to the small scale of the experimental testbed, a single underloaded host used for offloading VMs from overloaded hosts is able to significantly skew the AITF metric. The AITF metric is expected to be closer to the specified OTF parameter for large-scale OpenStack Neat deployments. A comparison of the results produced by LRR-1.1 and LRR-1.0 with MHOD-0.2 and MHOD-0.4 reveals that the MHOD algorithm leads to lower values of the AOTF metric (higher level of QoS) for approximately equal values of AITF.

Table 8: Energy consumption estimates

Algorithm	**Energy, kWh**	**Base energy, kWh**	**Energy savings**
THR-0.8	25.99	34.65	24.99%
THR-0.9	24.01	33.80	28.96%
THR-1.0	22.09	32.93	32.91%
LRR-1.1	25.66	34.50	25.63%
LRR-1.0	24.96	34.18	26.97%
LRR-0.9	22.60	33.20	31.93%
MHOD-0.2	25.70	34.53	25.59%
MHOD-0.3	25.59	34.48	25.76%
MHOD-0.4	24.72	34.12	27.54%
MAX-ITF	22.07	32.94	33.01%

Table 9: The execution time of components in seconds (mean values with 95% CIs)

Algorithm	**GM underload**	**GM overload**	**LM overload**	**DC**
THR	33.5 (26.4, 40.5)	60.3 (54.0, 66.7)	0.003 (0.000, 0.006)	0.88 (0.84, 0.92)
LRR	34.4 (27.6, 41.1)	50.3 (47.8, 52.8)	0.006 (0.003, 0.008)	0.76 (0.73, 0.80)
MHOD	41.6 (27.1, 56.1)	53.7 (50.9, 56.6)	0.440 (0.429, 0.452)	0.92 (0.88, 0.96)
MAX-ITF	41.7 (9.6, 73.7)	–	0.001 (0.000, 0.001)	1.03 (0.96, 1.10)

Using the obtained AITF and AOTF metrics for each algorithm and data on power consumption by servers, it is possible to compute estimates of potential energy savings relatively to a non-power-aware system assuming that hosts are switched to the sleep mode during every idle period. To obtain a lower bound on the estimated energy savings, it is assumed that when dynamic VM consolidation is applied, the CPU utilization of each host is 80% when it is active and non-overloaded, and 100% when it is overloaded. According to the data provided by Meisner et al. [28], power consumption of a typical blade server is 450 W in the fully utilized state, 270 W in the idle state, and 10.4 W in the sleep mode. Using the linear server power model proposed by Fan et al. [16] and the power consumption data provided by Meisner et al. [28], it is possible to calculate power consumption of a server at any utilization level.

To calculate the base energy consumption by a non-power-aware system, it is assumed that in such a system all the compute hosts are always active with the load being distributed across them. Since, the power model applied in this study is linear, it is does not matter how exactly the load is distributed across the servers. The estimated energy consumption levels for each overload detection algorithm, along with the corresponding base energy consumption by a non-power-aware system, and percentages of the estimated energy savings are presented in Table 8.

According to the estimates, MAX-ITF leads to the highest energy savings over the base energy consumption of approximately 33% by the cost of substantial performance degradation (AOTF = 40.4%). The THR, LRR, and MHOD algorithms lead to energy savings from approximately 25% to 32% depending on the specified parameters. Similarly to the above comparison of algorithms using the AITF metric, LRR-0.9 produces energy savings close to those of THR-1.0 (31.93% compared with 32.91%), while significantly reducing the mean AOTF from 42.2% to 34.4%. The MHOD algorithm produces approximately equal or higher energy savings than the LRR algorithm with lower mean AITF values, i.e., higher levels of QoS, while also providing the advantage of specifying a QoS constraint as a parameter of the algorithm. The obtained experimental results confirm the hypothesis that dynamic VM consolidation is able to significantly reduce energy consumption in an IaaS Cloud with a limited performance impact.

Table 9 lists mean values of the execution time along with 95% CIs measured for each overload detection algorithm during the experiments for some of the system components: processing underload and overload requests by the Global Manager (GM), overload detection algorithms executed by the Local Manager (LM), and iterations of the Data Collector (DC). Request processing by the global manager takes on average between 30 and 60 seconds, which is mostly determined by the time required to migrate VMs. The mean execution time of the MHOD algorithm is higher than those of THR and LRR, while still being under half a second resulting in a negligible overhead considering that it is executed at most once in 5 minute. The mean execution time of an iteration of the data collector is similarly under a second, which is also negligible considering that it is executed only once in 5 minutes.

8 Scalability Remarks and Future Directions

Scalability and eliminating single points of failure are important benefits of designing a dynamic VM consolidation system in a distributed way. According to the approach adopted in the design of OpenStack Neat, the underload / overload detection and VM selection algorithms are able to inherently scale with the increased number of compute hosts. This is due to the fact that they are executed independently on each compute host and do not rely on information about the global state of the system. In regard to the database setup, there exist distributed database solutions, e.g., the MySQL Cluster [1].

On the other hand, in the current implementation of OpenStack Neat, there assumed to be only one instance of the global manager deployed on a single controller host. This limits the scalability of VM placement decisions and creates a single point of failure. However, even with this limitation the overall scalability of the system is significantly improved compared with existing completely centralized VM consolidation solutions. Compared with centralized solutions, the only functionality implemented in OpenStack Neat by the central controller is the placement of VMs selected for migration, which constitute only a fraction of the total number of VMs in the system. To address the problem of a single point of failure, it is possible to run a second instance of the global manager, which initially does not receive requests from the local managers and gets automatically activated when the primary instance of the global manager fails. However, the problem of scalability is more complex since it is necessary to have multiple independent global managers concurrently serving requests from local managers.

Potentially it is possible to implement replication of the global manager in line with OpenStack's approach to scalability by replication of its services. From the point of view of communication between the local and global managers, replication can be simply implemented by a load balancer that distributes requests from the local managers across the set of replicated global managers. A more complex problem is synchronizing the activities of the replicated global managers. It is necessary to avoid situations when two global managers place VMs on a single compute host simultaneously, since that would imply that they use an out-of-date view of the system state. One potential solution to this problem could be a continuous exchange of information between global managers during the process of execution of the VM placement algorithm, i.e., if a host is selected by a global manager for a VM, it should notify the other global managers to exclude that host from their sets of available destination hosts.

There are several future directions for improvements of the current work. First of all, it is important to propose a synchronization model and implement replication of global managers to achieve a completely distributed and fault-tolerant dynamic VM consolidation system. Next, the data collector component should be extended to collect other types of data in addition to the CPU utilization that can be used by VM consolidation algorithms. Another direction is performance optimization and testing of OpenStack Neat on large scale OpenStack deployments.

It is important to further refine the proposed initial version of the benchmark suite, e.g., to provide a more concrete evaluation methodology along with tools for conducting automated experiments. It would also be beneficial to add alternative sets of workload traces, which would help to evaluate VM consolidation algorithms with a variety of workloads. As mentioned before, we accept contributions of implementations of VM consolidation algorithms that could be included in OpenStack Neat as a part of the proposed benchmark suite. Such a central repository of VM consolidation algorithms would greatly benefit future research in the area by simplifying and facilitating performance comparisons of alternative solutions, and avoiding duplication of research efforts.

9 Conclusions

In this paper, we have proposed a design and implementation of an open source framework for dynamic VM consolidation in OpenStack Clouds, called OpenStack Neat. The framework follows a distributed model of dynamic VM consolidation, where the problem is divided into 4 sub-problems: host underload detection, host overload detection, VM selection, and VM placement. Through its configuration, OpenStack Neat can be customized to use various implementations of algorithms for each for the 4 sub-problems of dynamic VM consolidation. OpenStack Neat is transparent to the base OpenStack installation by interacting with it using the public APIs, and not requiring any modifications of OpenStack's configuration. We have also proposed a benchmark suite comprising OpenStack Neat as the base software framework, a set of PlanetLab workload traces, performance metrics, and methodology for evaluating and comparing dynamic VM consolidation algorithms following the distributed model.

The experimental results and estimates of energy consumption have shown that OpenStack Neat is able to reduce energy consumption by the compute nodes of a 4-node testbed by 25% to 33%, while resulting in a limited application performance impact from approximately 15% to 40% AOTF. The MHOD algorithm has led to approximately equal or higher energy savings with lower mean AOTF values compared with the other evaluated algorithms, while also allowing the system administrator to explicitly specify a QoS constraint in terms of the OTF metric.

The performance overhead of the framework is nearly negligible taking on average only a fraction of a second to execute iterations of the components. The request processing of the global manager takes on average between 30 and 60 seconds and is mostly determined by the time required to migrate VMs. The results have shown that dynamic VM consolidation brings significant energy savings with a limited impact on the application performance. The proposed framework can be applied in both further research on dynamic VM consolidation, and real OpenStack Cloud deployments to improve the utilization of resources and reduce energy consumption.

References

[1] MySQL Cluster. (online; accessed on 23/11/2012).

[2] The Open Compute project – energy efficiency. (online; accessed on 21/11/2012).

[3] Rackspace hosting reports second quarter 2012 results. (online; accessed on 06/11/2012).

[4] Ubuntu 12.04 (Precise Pangolin) Cloud images. (online; accessed on 22/11/2012).

[5] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. Energy-aware autonomic resource allocation in multitier virtualized environments. *IEEE Transactions on Services Computing (TSC)*, 5(1):2–19, 2012.

[6] Anton Beloglazov. Scripts for setting up and analyzing results of experiments using OpenStack Neat. (accessed on 10/11/2013).

[7] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems (FGCS)*, 28(5):755–768, 2011.

[8] Anton Beloglazov and Rajkumar Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in Cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2012. (in press, accepted on August 2, 2012).

[9] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers. *Concurrency and Computation: Practice and Experience (CCPE)*, 24(13):1397–1420, 2012.

[10] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. A taxonomy and survey of energy-efficient data centers and Cloud computing systems. *Advances in Computers, M. Zelkowitz (ed.)*, 82:47–111, 2011.

[11] Anton Beloglazov, Sareh Fotuhi Piraghaj, Mohammed Alrokayan, and Rajkumar Buyya. Deploying OpenStack on CentOS using the KVM hypervisor and GlusterFS distributed file system. Technical report, CLOUDS-TR-2012-3, CLOUDS Laboratory, The University of Melbourne, Australia, 2012.

[12] J. L Berral, Goiri, R. Nou, F. Juli, J. Guitart, R. Gavald, and J. Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proc. of the 1st Intl. Conf. on Energy-Efficient Computing and Networking*, pages 215–224, 2010.

[13] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proc. of the 10th IFIP/IEEE Intl. Symp. on Integrated Network Management (IM)*, pages 119–128, 2007.

[14] M. Cardosa, M. Korupolu, and A. Singh. Shares and utilities based power consolidation in virtualized server environments. In *Proc. of the 11th IFIP/IEEE Integrated Network Management (IM)*, 2009.

[15] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. Autonomic placement of mixed batch and transactional workloads. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 23(2):219–231, 2012.

[16] X. Fan, W. D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 13–23, 2007.

[17] E. Feller, L. Rilling, and C. Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proc. of the 12th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 482–489, 2012.

[18] E. Feller, C. Rohr, D. Margery, and C. Morin. Energy management in IaaS Clouds: A holistic approach. In *Proc. of the 5th IEEE Intl. Conf. on Cloud Computing (IEEE CLOUD)*, pages 204–212, 2012.

[19] Gartner, Inc. *Gartner estimates ICT industry accounts for 2 percent of global CO2 emissions.* Gartner Press Release (April 2007).

[20] Í. Goiri, J.L. Berral, J. Oriol Fitó, F. Julià, R. Nou, J. Guitart, R. Gavaldà, and J. Torres. Energy-efficient and multifaceted resource management for profit-driven virtualized data centers. *Future Generation Computer Systems (FGCS)*, 28(5):718–731, 2012.

[21] Brian Guenter, Navendu Jain, and Charles Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *Proc. of the 30st Annual IEEE Intl. Conf. on Computer Communications (INFOCOM)*, pages 1332–1340, 2011.

[22] F. Hermenier, X. Lorca, J.M. Menaud, G. Muller, and J. Lawall. Entropy: A consolidation manager for clusters. In *Proc. of the 2009 ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, pages 41–50, 2009.

[23] Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Richard D. Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in Cloud infrastructures. In *Proc. of the 30th Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 62–73, 2010.

[24] Jonathan Koomey. *Growth in data center electricity use 2005 to 2010.* Oakland, CA: Analytics Press, 2011.

[25] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vManage: Loosely coupled platform and virtualization management in data centers. In *Proc. of the 6th Intl. Conf. on Autonomic Computing (ICAC)*, pages 127–136, 2009.

[26] D. Kusic, N. Kandasamy, and G. Jiang. Combined power and performance management of virtualized computing environments serving session-based workloads. *IEEE Transactions on Network and Service Management (TNSM)*, 8(3):245–258, 2011.

[27] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009.

[28] D. Meisner, B.T. Gold, and T.F. Wenisch. PowerNap: eliminating server idle power. *ACM SIGPLAN Notices*, 44(3):205–216, 2009.

[29] B.B. Nandi, A. Banerjee, S.C. Ghosh, and N. Banerjee. Stochastic VM multiplexing for datacenter consolidation. In *Proc. of the 9th IEEE Intl. Conf. on Services Computing (SCC)*, pages 114–121, 2012.

[30] R. Nathuji and K. Schwan. VirtualPower: Coordinated power management in virtualized enterprise systems. *ACM SIGOPS Operating Systems Review*, 41(6):265–278, 2007.

[31] K. S Park and V. S Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):65–74, 2006.

[32] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Transactions on Services Computing (TSC)*, 3(4):266–278, 2010.

[33] A. Verma, G. Dasgupta, T. K Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proc. of the 2009 USENIX Annual Technical Conf.*, pages 28–28, 2009.

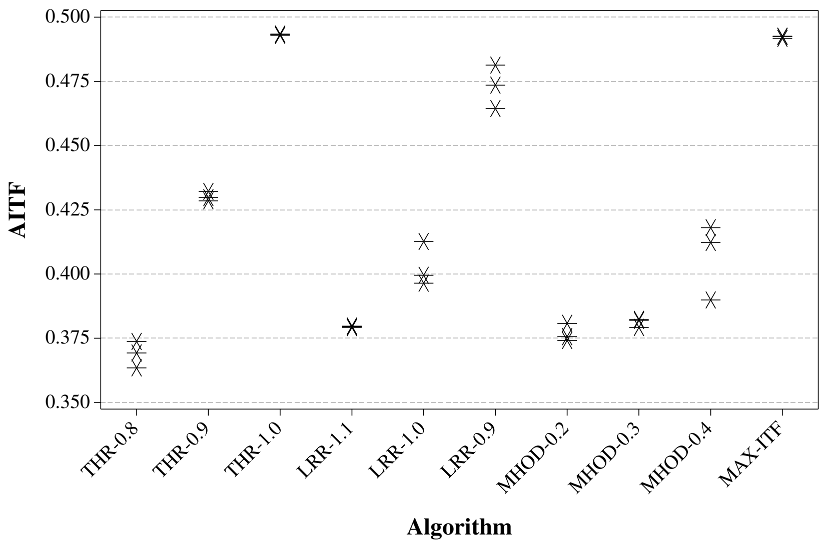
[34] Xiaorui Wang and Yefu Wang. Coordinating power control and performance management for virtualized server clusters. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(2):245–259, 2011.

[35] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. of the 4th USENIX Symp. on Networked Systems Design & Implementation*, pages 229–242, 2007.

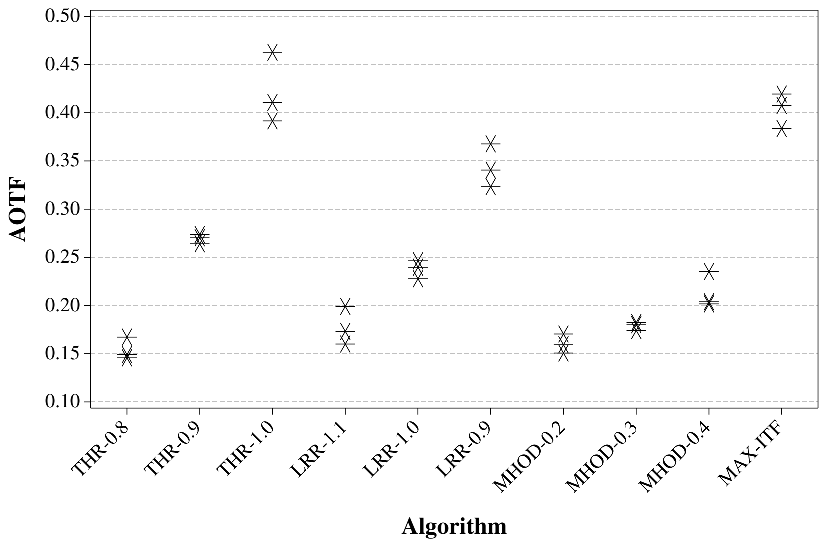
[36] J. Yang, K. Zeng, H. Hu, and H. Xi. Dynamic cluster reconfiguration for energy conservation in computation intensive service. *IEEE Transactions on Computers*, 61(10):1401–1416, 2012.

[37] M. Yue. A simple proof of the inequality FFD (L)< 11/9 OPT (L)+ 1,for all l for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica (English Series)*, 7(4):321–331, 1991.

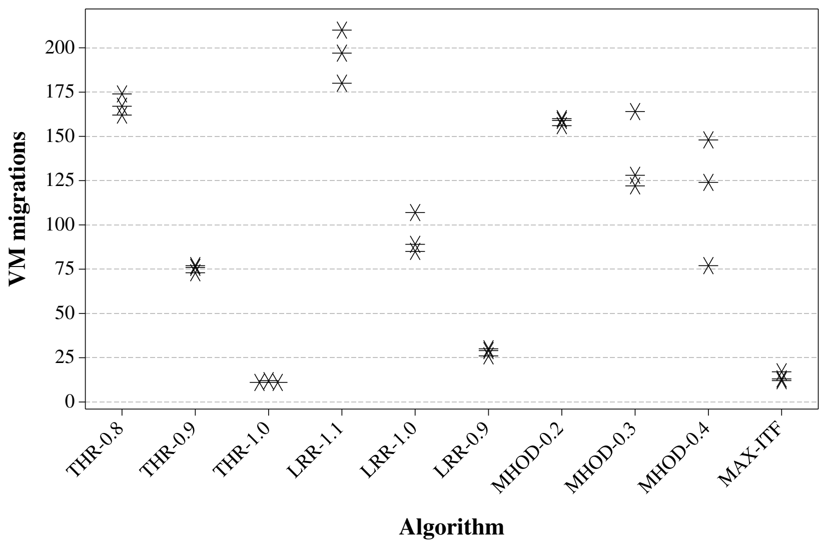
[38] X. Zhu, D. Young, B. J Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, et al. 1000 Islands: Integrated capacity and workload management for the next generation data center. In *Proc. of the 5th Intl. Conf. on Autonomic Computing (ICAC)*, pages 172–181, 2008.



(a) The AITF metric



(b) The AOTF metric



(c) The number of VM migrations

Figure 6: The experimental results