Hint-based Execution of Workloads in Clouds with *Nefeli*

Konstantinos Tsakalozos, Mema Roussopoulos, and Alex Delis

Abstract—*Infrastructure-as-a-Service* clouds offer entire virtual infrastructures for distributed processing while concealing all physical underlying machinery. Current cloud interface abstractions restrict users from providing information regarding usage patterns of their requested virtual machines (*VM*s). In this paper, we propose *Nefeli*, a *virtual infrastructure gateway* that lifts this restriction. Through *Nefeli*, cloud consumers provide deployment *hints* on the possible mapping of *VM*s to physical nodes. Such *hints* include the collocation and anti-collocation of *VM*s, the existence of potential performance bottlenecks, the presence of underlying hardware features (e.g., high availability), the proximity of certain *VM*s to data repositories, or any other information that would contribute in a more effective placement of *VM*s to physical hosting nodes. Consumers designate only properties of their virtual infrastructure and remain at all times agnostic to the cloud internal physical characteristics. The set of consumer-provided *hints* is augmented with high-level placement policies specified by the cloud administration. Placement policies and *hints* form a constraint satisfaction problem that when solved, yields the final *VM*-to-host placement. As workloads executed by the cloud may change over time, *VM*-to-host mappings must follow suit. To this end, *Nefeli* captures such events, changes *VM* deployment, helps avoid bottlenecks, and ultimately, improves the quality of the rendered services. Using our prototype, we examine overheads involved and show significant improvements in terms of time needed to execute scientific and real application workloads. We also demonstrate how power-aware policies may reduce the energy consumption of the physical installation. Finally, we compare *Nefeli*'s placement choices with those attained by the open-source cloud middleware, *OpenNebula*.

Index Terms—Distributed Systems, Cloud Computing, IaaS Cloud, Virtual Machine Scheduling

1 Introduction

Computing clouds allow for the transparent access to diverse physical resources available in the form of services. In this work, we focus on *IaaS*-clouds [1] that exploit virtual machines (*VM*s) to deploy computing systems on-demand [2]–[5]. We examine the effective placement of *VM*s on the physical infrastructure so that multiple and diverse workloads are efficiently handled. The key benefit in using an *IaaS*-cloud is that it shields users and/or applications from all administrative tasks and resource sharing policies of the underlying machinery. Moreover, the decoupling of physical resources from system software offers enhanced server-utilization through collocation of *VM*s and effective options for node recovery in light of failure(s). However, sharing physical resources may yield peak performance rates that are below expectation due to *VM* contention on particular physical nodes.

Virtualization as used in current *IaaS*-clouds makes deployment of *VM*s a straightforward task. However, the large number of options of *where* within the cloud to (re)deploy *VM*s renders the problem of infrastructure tuning a real challenge. To this date, there have been a number of efforts that attempt to fine-tune virtual infrastructure placement for executing specific types of jobs [6]–[9]. In these efforts, users "evaluate" the mapping quality of computational resources to *VM*s [10]–[12] by using either fixed service-level agreements (SLAs) or high-level utility functions. In general, producing an "evaluation function" is a nontrivial task for it requires knowledge of both the application at hand and the policies regulating resource sharing within the physical infrastructure [10].

In this paper, we present the design, implementation, and evaluation of a *cloud gateway*, *Nefeli*. *Nefeli* performs intelligent placement of *VM*s onto physical nodes by exploiting user-provided deployment *hints*. Hints realize placement preferences based on knowledge only the cloud consumer has regarding the intended usage of the requested *VM*s. By modeling workloads as patterns of data flows, computations, control/synchronization points and necessary network connections, users can identify favorable *VM* layouts. These layouts translate to deployment hints. Such hints articulate *1)* resource consumption patterns among *VM*s, *2) VM*s that may become a performance bottleneck and *3)* portions of the requested virtual infrastructure that can be assisted by the existence of special hardware support. For instance, the fact that two *VM*s in a virtual infrastructure will hold mirrors of a database is only known to the cloud consumer. This information should be communicated to the cloud as a deployment hint so that the respective *VM*s will not be deployed on the same host. We refer to *VM* layout patterns as *task-flows* to distinguish them from the traditional workflow concept [13]. Specifically, task-flows illustrate "ideal" deployments of *VM*s described by the cloud consumers using deployment hints. *Nefeli* exploits these hints so as to (re)deploy *VM*s in the cloud

• *The authors are with the University of Athens, Athens, GR15784, Greece. E-mail: {k.tsakalozos, mema, ad}@di.uoa.gr*

and achieve efficient task-flow execution. However, hints must not reveal any cloud internal properties to the consumers. Although hints may offer a desired *VM*-deployment for consumer workloads, *Nefeli* may ultimately elect to ignore part or all of them based on the available physical resources. In addition to hints, *Nefeli* also takes into account high-level *VM* placement policies, set by the cloud administration, whose objectives may entail energy efficiency and load balancing.

The main contribution of our approach is that we present a complete solution in extracting and exploiting the knowledge cloud consumers posses regarding the operational aspects of their virtual infrastructures. Our approach is compatible with the cloud abstractions that dictate users are kept agnostic of the physical infrastructure properties at all times. Furthermore, our approach is able to adapt to dynamic environments where both task-flows and user preferences change over time. *Nefeli* produces suitable *VM* to physical node mappings in response to signals coming from the infrastructures (both physical and virtual) or any other external notification mechanism. The produced mappings are applied through appropriate *VM* placement calls to an underlying cloud middleware.

We have created a detailed prototype and experimented with both simulated and real cloud environments. We compare *Nefeli VM*-placement against *a)* random placement, *b)* a placement that evenly distributes *VM*s among physical nodes, *c)* a policy that minimizes the number of physical nodes used and thus reduces the power footprint of the cloud, and *d)* the match making policy used by the open–source cloud middleware *OpenNebula v.1.2.0*. Our approach consistently displays significant performance improvements when compared to the aforementioned policies. In video transcoding, *Nefeli* achieves 17% reduced processing times compared to the *VM* placement decided by *OpenNebula*. In scientific task-flows and for a variety of simulated clouds, *Nefeli* demonstrates significantly higher throughput rates compared to other *VM* placement policies. Noteworthy savings in terms of power consumption are reported as well. We also present the performance overheads involved in the operation of *Nefeli* as the cloud infrastructure scales out. The rest of this paper is organized as follows: Section 2 states the problem we address. Sections 3–6 present in detail all the architectural elements of *Nefeli*. Section 7 discusses our experimental findings. Section 8 outlines earlier related work and finally, Section 9 offers concluding remarks.

2 Managing IaaS–Cloud Resources

IaaS-clouds provide for their users a separation of concerns at the level of hardware as their respective services are confined to the provision of *VM*s; the latter collectively form virtual infrastructures. Users may consume *IaaS*-cloud services, yet, they are unable to impose changes on the fundamental aspects and functional characteristics of the elements of the underlying physical substrate. Users may only offer minimal information to influence the performance of the infrastructure by indicating how *VM*s are to be actually deployed on the physical resources. Cloud providers undertake all administrative actions on physical computing nodes including setting the policy with which consumer requests are handled.

Both service consumers and producers possess fragments of information and maintain knowledge in their own sphere of operation that if combined could jointly improve the effectiveness of the cloud. Knowledge of the underlying hardware features, the make-up of the virtual infrastructure as well as the characterization of the workload in execution could all contribute to a more effective resource sharing. As the cloud-contract "prevents" the physical substrate from revealing most of its organizational features, user preferences and desired operational conditions can be expressed by the *IaaS* consumer to the provider. Perhaps the most critical parameter about which users have to alert the cloud is the nature of the *task-flows* submitted. A task-flow includes the set of *VM*s requested by the cloud consumer combined with information regarding a desirable *VM* deployment layout. This layout emerges from analyzing *VM* usage patterns which are known only to the consumer. In general, the consumer is aware of how various elements of her workload should be ideally pegged to *VM*s.

In this paper, we take the view that consumers may communicate the task-flow information in the form of *hints*. The latter could be used while trying to appropriately deploy *VM*s. For instance, consider a user who requests a *VM* that will play the role of a single network-bridge between her virtual infrastructure and the Internet. This bridge inherently becomes a single point of failure and a potential performance bottleneck. Therefore, the *VM* in question would be best placed on an offloaded physical node equipped with redundant hardware. In similar spirit, *VM*s that are to perform parallel jobs –very much in the *MapReduce* fashion– should be spread across different nodes[1]. Hence, it is important for the cloud to be aware of the user's intended use of particular *VM*s.

We also emphasize that *IaaS* consumers have no explicit control over *VM* migrations. Migrations reshuffle the way *VM*s share the same computing nodes so they may radically hurt or significantly enhance the virtual infrastructure's performance. The actual placement of *VM*s on physical hosting nodes should be able to address the needs of changing workloads. For instance in a video-encoding application, it might be beneficial to use a highly distributed setup for *VM*s across various physical nodes to harness as many CPU-cycles as possible. Occasionally however, the aforementioned layout might gen-

1. Cloud providers such as Amazon [3] allow users to ask for *VM*s deployed on different sites. Such ad-hoc engineering solutions, however, cover only a portion of the needs of a user and even worse, they disclose information about the cloud's internal structure.

erate significant network traffic calling for opportunistic collocation of *VM*s. Thus, the cloud must take actions to dynamically redeploy *VM*s to better serve continuously changing workloads. Overall, the challenge *IaaS*-clouds face is how to permit more sophisticated interaction with users while keeping them agnostic of cloud internals. In their quest to offer entirely transparent operations, contemporary clouds inadvertently prevent their users from exploiting salient virtualization features such as *VM* migration. By accepting hints, *Nefeli* plays a major role in helping attain user-favorable *VM* deployments. The user remains unaware of the cloud internals as any piece of his information arriving at *Nefeli* (the *cloud gateway*) strictly refers to the type of the workload(s) the virtual infrastructure is to serve.

3 Overview of Nefeli

Nefeli adds a layer between the user and the infrastructure providing *IaaS*-cloud services, shown in Figure 1. *Nefeli* interfaces with the lower level cloud services that handle the *VM* lifecycle and perform fundamental administrative tasks. This interface, denoted as a Cloud *API*, allows us to query for specific aspects of the hardware resources as well as manage the *VM* deployment and migration. During operation, *Nefeli* has to obtain

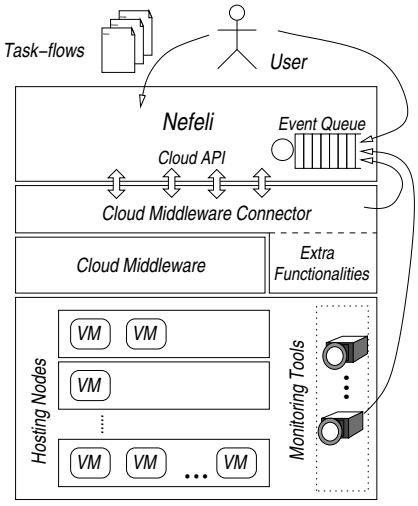
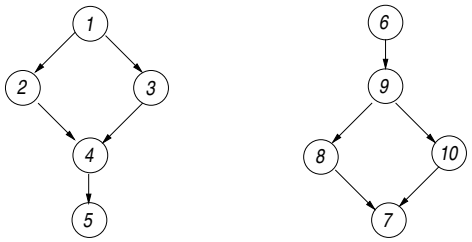


Fig. 1: *Nefeli*'s structured layout and interaction model

the following information:

- *Physical node properties:* these properties include free memory, total memory, CPU utilization, the name/ID of each hosting node, the amount of free disk space and redundant hardware enhancing the node's availability.
- *Physical infrastructure properties: Nefeli* takes into account the network topology of the physical substrate, the cloud's gateways towards the Internet and any data repositories available through the network.
- *The current status of each VM:* in our approach each *VM* may find itself in either *STAGING* or *RUNNING* state. A *VM* is considered to be *STAGING* when management operations such as disk image copying during a *VM* migration do not permit the *VM* to run.
- *VM properties:* these are similar to the properties acquired for physical hosting nodes. *VM* properties include the memory usage and the disk space reserved for each virtual machine. *Nefeli* also acquires the *IP*-address of each *VM* through the cloud *API* and forwards it to the user.



(a) *Task-flow A* (b) *Task-flow B*

Fig. 2: Two different sample task-flows

VM deployment is handled through the cloud *API* depicted in Figure 1 and includes the following operations:

- Spawn a new *VM*.
- Shut down a *VM*.
- Migrate a *VM*. The names/IDs of the hosting nodes are needed for this operation.

Nefeli may interact with the physical infrastructure through a cloud middleware [2], [4], [5]. However, the cloud middleware may not provide all the functionality required by *Nefeli*. For instance, *OpenNebula v.1.2.0* does not expose all host-related information it gathers. In such cases, we have to realize any missing functionality and incorporate it in the "Cloud Middleware Connector" component (denoted as "Extra Functionalities" in Figure 1).

Nefeli has the role of an *IaaS*-cloud *gateway*. Users contacting *Nefeli* request virtual infrastructures created by *instantiating sets of VM*s. Two sample graphs of task-flows executed in such an infrastructure are displayed in Figure 2. In the task-flow's graphical representation, each node corresponds to a single *VM* while edges indicate control and data flows. The *VM* specifications are accompanied by user-provided deployment *hints*. Hints are expressed as sets of conditions or constraints pointing out a deployment favoring specific task-flows within the virtual infrastructure. As the user must be kept agnostic of the internal deployment decision algorithms of the cloud, all available constraint types are provided by *Nefeli*. Constraints, even though important, may occasionally be contradicting or even impossible to satisfy all at the same time. Therefore, each constraint is coupled with a weight value indicating its importance relative to the other hints provided. For *task-flow A* of Figure 2a, possible deployment hints include a) *VM*s 2 and 3 should preferably be deployed on different hosting nodes and b) *VM* 4 should be favored by deploying it

on a host without any other *VM*s. The latter indicates a possible CPU performance bottleneck of the task-flow at hand. In addition to cloud consumer constraints, *Nefeli* also accepts hints that articulate high-level policies imposed by the cloud administration. Table 1 depicts a number of such consumer and administrative types of constraints that we have frequently used in our work.

TABLE 1: Commonly used constraints.

Cloud Consumer Constraints	
`FavorVM`	Try to reserve a single hosting node for a specific *VM*.
`MinTraf`	Deploy on the same host a set of *VM*s so as to minimize traffic over the physical network.
`ParVMs`	Try to deploy a set of *VM*s on separate physical nodes so as not to compete over the same resources
`PinVM`	Try not to migrate a specific *VM*.
`HighAvail`	Try to deploy a specific *VM* on a host with high availability features.
`UsesDataRepo`	Try to reduce the network distance between a *VM* and a data repository.
Cloud–Administration Constraints	
`PowerSave`	Reduce the number of hosting nodes used for *VM* deployment
`EmptyNode`	Offload a specific hosting node
`EvenLoad`	Distribute *VM*s evenly among hosting nodes
`StopPingPong`	Cease the same *VM*s from migrating back and forth among hosting nodes.
`ReduceDist`	Reduce the network hops among a set of *VM*s.

We use a single *XML* document to describe all consumer-provided information. Listing 1 presents all aspects related to *task-flow A* of Figure 2a. In the first section, the consumer provides the specifications of the requested *VM*s. Each *VM* is assigned a system-wide identifier. The user also sets RAM requirements and points to the *VM* type that needs to be instantiated by providing the proper disk image pointer. The second *XML* section outlines the constraints to be taken into account for the deployment of the virtual infrastructure. As mentioned earlier, there are two constraints, one for *VM* deployment in separate nodes (`ParVMs`) and one for favoring the deployment of *VM* with ID 4 (`FavorVM`). In the second *XML* section, *VM* identifiers are used whenever constraints have to refer to specific *VM*s. Since the performance impact of specific constraints may be greater than that of others, the third *XML* section contains pertinent user-assigned weights. In this example, the constraint with ID 1 is more important than that with ID 2 and thus, it receives a weight of 0.4 while constraint 2 gets a 0.3. Note that the correctness neither of the constraints nor the respective weights is questioned. We trust the user has some knowledge of the performance bottlenecks in her task-flows.

Administrative constraints are introduced by the cloud provider in the same way as consumer constraints. However, the administrative constraints do not refer to any specific set of *VM*s, rather they refer to aspects of cloud internals only the cloud administration is permitted to know. In what follows, we discuss how *Nefeli* handles a single task-flow and then, we look at how our approach offers simultaneous execution of multiple task-flows running on the same physical nodes.

```
<Task–flow>
 <VirtualMachines>
  <VM id="1"><RAM>512</RAM><Disk>VM1.img</Disk></VM>
  <VM id="2"><RAM>512</RAM><Disk>VM2.img</Disk></VM>
  <VM id="3"><RAM>512</RAM><Disk>VM3.img</Disk></VM>
  <VM id="4"><RAM>512</RAM><Disk>VM4.img</Disk></VM>
  <VM id="5"><RAM>512</RAM><Disk>VM5.img</Disk></VM>
 </VirtualMachines>

 <Constraints>
  <ParVMs id="1">
   <VMid>2</VMid><VMid>3</VMid>
  </ParVMs>
  <FavorVM id="2"><VMid>4</VMid></FavorVM>
 </Constraints>

 <Profiles>
  <Profile id="1">
   <Weights>
    <Constr id="1"><Weight>0.4</Weight></Constr>
    <Constr id="2"><Weight>0.3</Weight></Constr>
   </Weights>
  </Profile>
 </Profiles>
</Task–flow>
```

Listing 1: *Nefeli* input describing the sample *task-flow A* of Figure 2a.

4 Single Task-flow Execution

Figure 3 shows the key steps followed starting from the user input until we reach a *VM*-to-host mapping, termed *deployment profile*. With V being all the *VM*s to be deployed and H the set of physical nodes, a profile M is a function from V to H ($M : V \mapsto H$). *Nefeli* chooses, out of all possible profiles M_{all}, one that best suits the constraints expressed for the task-flow at hand. Profile production exploits information gathered from user hints, as well as information emanating from the cloud administration and the physical infrastructure. Combining the user-provided constraints with the *VM* specifications, as described in *XML*-documents such as the one of Listing 1, results in deployment patterns. The deployment patterns are used to match *VM* requirements to physical node resources. This matching phase creates the actual final deployment profile by taking into

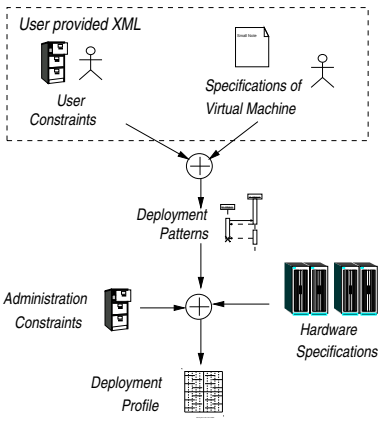


Fig. 3: *Nefeli*'s operational model

account cloud administration constraints and hardware node specifications.

4.1 Constraints

Constraints express user (cloud consumer) and administration preferences. Each constraint is realized as a utility function F that evaluates a single deployment profile. F takes as input a deployment profile and returns the degree of constraint satisfaction in the range of $[0, 1]$

$$F : M_{all} \mapsto [0, 1],$$

where M_{all} is the set of all possible deployment profiles.

In *Nefeli*, each such function has at its disposal all information regarding the characteristics of both physical and virtual nodes. An example of the utility function `HighAvail` is presented by Algorithm 1. When a cloud consumer uses `HighAvail` for a *VM*, she indicates that the specified *VM* is of great importance for the virtual infrastructure and it should always be available (i.e., ideally no down time). From the provider's perspective, this translates to hosting the *VM* on a physical node that is unlikely to fail. Algorithm 1 quantifies the success of a profile in mapping a specific *VM* to a hosting node with high availability properties. If the selected host is a high-availability (HA) server, then the `HighAvail` constraint is fully satisfied and 1.0 is returned by line 5. Otherwise, we check the redundancy of the host's hardware. We increase the degree of *satisfaction* if we find RAID setup (line 7), additional power supply (line 10) or multiple network interfaces (line 13). In our implementation of this `HighAvail` function we elect to increase the degree of *satisfaction* by 0.2 for each redundant device available.

The implementation details of the `HighAvail` constraint are not revealed to the consumer as they are specific to each cloud infrastructure. Whenever one or more assumptions regarding the high availability of the hardware are outdated, possibly due to major changes in the infrastructure (e.g., all hosting nodes become equipped with RAID), we have to provide new implementations for the affected utility functions.

Administrative constraints are also realized as utility functions. These constraints serve a dual purpose as they can introduce high-level policies and assist in administration tasks. For instance, `EmptyNode` relieves a hosting node of *VM*s. `ReduceDist` enforces the high-level policy of clustering *VM*s of the same user on hosts that may not be far apart in terms of network hops; this is done to limit major traffic to be routed over long-haul physical networks.

We have realized both user and administration constraints of Table 1 as utility functions in similar fashion to Algorithm 1; for brevity, we omit their detailed discussion here. Such functions are expected to work in a *plug-and-play* fashion.

Algorithm 1 `HighAvail` Utility Function

Input: `VM_ID`: ID of the *VM* to deploy in a high availability node
$M()$: Deployment profile function
Output: Satisfaction degree of constraint

```
1: host_ID := M(VM_ID)
2: satisfaction := 0;
3: if (HostIsHAServer(host_ID)) then
4:    satisfaction := 1.0;
5:    return satisfaction;
6: end if
7: if (HostHasRAID(host_ID)) then
8:    satisfaction += 0.2;
9: end if
10: if (HostHasRedundantPowerSupply(host_ID)) then
11:    satisfaction += 0.2;
12: end if
13: if (HostHasMultipleNetwork(host_ID)) then
14:    satisfaction += 0.2;
15: end if
16: return satisfaction;
```

4.2 Deployment Profile Production

Each possible deployment profile m is assigned a score computed by the formula:

$$Score(m) = \sum_{Const_i \in Cs} w_i Const_i(m),$$

where Cs is the set of all constraints and w the respective weights derived from the user-provided *XML*. In the example of Listing 1 where the two constraints `ParVMs` and `FavorVM` with weights 0.4 and 0.3 are used, the *Score* of a deployment profile m becomes:

$$Score(m) = 0.4 * \texttt{ParVMs}(m) + 0.3 * \texttt{FavorVM}(m)$$

The optimal profile (m_{opt}) is the one with the highest score:

$$Score(m_{opt}) \geq Score(m_q),\ \forall m_q \in M_{all}$$

where M_{all} is the set of all possible deployment profiles.

Finding optimal deployment profiles is *NP*-hard [8] so we employ simulated annealing [14] to attain plausible approximations. In Algorithm 2, we start from a random *VM* deployment, produced by `GetRandomProfile`, and visit gradually higher-scoring neighboring deployment profiles. The neighbors of each deployment profile are generated by a call to `GetNeightborOf`. The neighborhood N_m of a deployment profile m is the set:

$$N_m = \{N \in M_{all} | Prob(N(v) \neq m(v)) = d, \forall v \in V\},$$

Here, V is the set of all *VM*s, M_{all} is the set of all profiles, d is the probability for a *VM* v to be deployed on a hosting node other than the one set by profile m. Increasing d results in wider neighborhoods and prevents us from getting trapped at local optima. Yet, too wide neighborhoods result in almost randomly generated neighbors and thus, deployment profiles of low quality.

Algorithm 2 chooses to update current_profile with one of its neighbors based on a probability factor: $e^{D/T} > \text{Random}()$, where D is the score improvement

we get using the neighboring profile and T the temperature. Using this formula, we handle local minimum pits by allowing "jumps" to lower scoring profiles. However, when the temperature drops near zero (10^{-5}) only higher scoring neighbors are visited. Apart from the starting temperature and the number of non-improving iterations performed before returning the best profile (`same_iterations`), another option for enhancing the profile quality is the number of times *Nefeli* runs simulated annealing. Starting from a different initial *VM* deployment, allows our approach not to get trapped at locally optimum solutions.

Algorithm 2 Simulated Annealing - Profile Production

Input: `same_iterations`: After how many iterations showing no improvement will we stop our search
`T`: Temperature
`Score()`: Deployment profile score function
Output: A near-optimal deployment profile

```
1: same = 0
2: best_profile = current_profile = GetRandomProfile()
3: while same <same_iterations do
4:    new_profile = GetNeightborOf(current_profile)
5:    D = Score(new_profile) - Score(current_profile)
6:    if ( T > 10^-5 AND e^(D/T) > Random()) OR
          (T < 10^-5 AND D > 0) then
7:       current_profile = new_profile
8:    end if
9:    if Score(new_profile) > Score(best_profile) then
10:      best_profile = new_profile
11:      same = 0
12:   end if
13:   same++
14:   T = 0.99 * T
15: end while
16: return best_profile
```

Our approach decouples the profile evaluation and generation from the process of finding a near-optimal *VM*-to-host mapping. This allows us to place constraints into two categories:

- **Soft Constraints:** the degree of satisfaction of constraints that belong in this class contributes to the overall quality of the produced profile.
- **Hard Constraints:** conditions placed in this group have to be satisfied to their full extent. Otherwise, task-flows featuring such constraints are simply not admitted for execution and receive no further consideration.

Escalating the severity of a soft constraint to hard requires setting its weight to 1.0 in the respective task-flow *XML*-description. Soft constraints are used for the computation of each profile score. Hard constraints are taken into consideration during the generation of new profiles from functions `GetNeightborOf` and `GetRandomProfile` of Algorithm 2. These two functions also take into account the obvious constraints raising from the limited availability of hardware resources such as the available main-memory on each hosting node.

4.3 Computational Requirements for *Nefeli*

Typically, the provision of a *VM* is a process that takes several minutes. For a *VM* instantiation, one or more disk images need to be copied from the image repository, where all *VM*s are stored, to the physical node that will provide the resources needed at runtime. *Nefeli* incurs an additional overhead to the *VM* provision since a constraint satisfaction problem (CSP) needs to be solved for producing a deployment profile. Depending on the time requirements set by each *IaaS*-provider, the acceptable time overhead in general may range from a few seconds to at most a few minutes.

In this work, we formulate the *VM*-to-host mapping production as a CSP so as to take advantage of the evolution of CSP solvers and avoid the use of preset heuristics. The selected simulated annealing solver displays two fundamental properties that render *Nefeli* suitable for a wide range of small to medium sized clouds. Our approach has two salient features. First, it allows the cloud administration to specify the maximum amount of time to be expended on deployment planning. This is achieved by properly adjusting the temperature `T` and `same_iterations` in Algorithm 2. In this way, performance is tuned to match *VM* provision requirements. Second, our approach is parallelizable in an intuitive way. Multiple, separate executions of the simulated annealing algorithm may commence simultaneously, each one with a different start-up seed. Different seeds make sure that even if one execution gets trapped at a local optimum, a good solution will ultimately be found by some other execution.

In cases where the size of the cloud infrastructure is too large or the number of the involved constraints is very high, simulated annealing may not yield high quality profiles within strict time limits. Here, the provider has two options: either reduce the search space of simulated annealing or solve the CSP with other more effective solvers and possibly heuristics. In [15], we outline an approach that follows the first option above; it reduces the search space and harvests cloud resources to realize an elastic distributed solver and yield scalable profile production. *Nefeli*'s modular design can readily facilitate the second of the above choices by replacing simulated annealing with other alternatives envisaged.

5 Introducing Multiple Task-flows

As clouds serve many users, each one in need of his own private infrastructure, multiple task-flows may have to be active for simultaneous execution. In its simplest form, multiple task-flow execution occurs when *Nefeli* serves a single task-flow while a new one is submitted. In this case, a single deployment profile must be produced taking into consideration constraints for both task-flows. Listing 2 shows the *XML*-description for *task-flow B* of Figure 2b. For this task-flow, there are two constraints: *a) VM*s 6 and 9 would better be co-located since they will be producing too much network traffic and *b) VM*s

8 and 10 are to be deployed on different hosting nodes. *VM* IDs are system-wide identifiers and thus the task-flows of Figure 2 make use of different *VM*s. *Nefeli* never reveals the set of *VM* identifiers to users. In collaborative environments, where users share *VM*s, to produce task-flows, users must be assisted by higher level components operating outside *Nefeli*.

Producing a single deployment profile for both task-flows of Figure 2, is done by combining the respective descriptions of Listings 1 and 2.

```
<Task-flow>
 <VirtualMachines>
  <VM id="6"><RAM>512</RAM><Disk>VM6.img</Disk></VM>
  <VM id="7"><RAM>512</RAM><Disk>VM7.img</Disk></VM>
  <VM id="8"><RAM>512</RAM><Disk>VM8.img</Disk></VM>
  <VM id="9"><RAM>512</RAM><Disk>VM9.img</Disk></VM>
  <VM id="10"><RAM>512</RAM><Disk>VM10.img</Disk></VM>
 </VirtualMachines>

 <Constraints>
  <MinTraf id="1">
   <VMid>6</VMid><VMid>9</VMid>
  </MinTraf>
  <ParVMs id="2">
   <VMid>8</VMid><VMid>10</VMid>
  </ParVMs>
 </Constraints>

 <Profiles>
  <Profile id="1">
   <Weights>
    <Constr id="1"><Weight>0.4</Weight></Constr>
    <Constr id="2"><Weight>0.3</Weight></Constr>
   </Weights>
  </Profile>
 </Profiles>
</Task-flow>
```

Listing 2: *Nefeli* input derived from sample *task-flow B* of Figure 2b

In this case, the set of constraints to be considered is the union of all constraints. Constraint weights handling policies may need to take into account the financial gain from satisfying specific users. Such policies are out of the scope of *Nefeli* as we expect them to be enforced at a higher level. The score function for a deployment profile m becomes:

$$\begin{aligned} Score(m) &= 0.4 * \texttt{ParVMs}_A(m) + 0.3 * \texttt{FavorVM}(m) + \\ &+ 0.4 * \texttt{MinTraf}(m) + 0.3 * \texttt{ParVMs}_B(m) \end{aligned}$$

where $\texttt{ParVMs}_A$ and $\texttt{ParVMs}_B$ are the `ParVMs` constraints of *task-flow*s *A* and *B* respectively.

A task-flow departure also calls for the production of a new deployment profile. This time the constraints used will have to be the ones referring to the task-flows remaining for execution. The *VM*s used explicitly by the terminated task-flow alone will also have to be removed.

A transition between deployment profiles (as in the case of adding or removing task-flows) involves *VM* migrations that in the absence of live migration result in some downtime of the virtual infrastructures. In this case, *VM*s have to be suspended and copied to other hosting nodes where they can resume their normal operation. To tackle such inefficiency, the profile creation procedure may trade profile quality for migrating less *VM*s. To this end, we define the distance of two profiles to be the number of *VM*s deployed on different hosting nodes in the profiles compared.

Definition: The distance $Dist$ between two deployment profiles $M_1, M_2 \in M_{all}$, that map *VM*s to hosting nodes, is:

$$Dist(M_1, M_2) = |\{v \in V : M_1(v) \neq M_2(v)\}|$$

Given an initial deployment profile m_s, to reduce *VM* migration overheads, *Nefeli* first produces the k highest scoring profiles and then, it picks the one whose transition from m_s requires migrating fewer *VM*s. With k regulating the tradeoff between migration overhead and the time spent in producing the final deployment profile, we are able to express the virtual infrastructures' sensitivity to downtime. From the set (M_b) of the k highest scoring profiles, the final deployment profile (m_q) used is:

$$m_q : Dist(m_q, m_s) \leq Dist(m_i, m_s), \forall m_i, m_q \in M_b$$

Our decision to reduce the number of migrating *VM*s may not not always yield the swiftest transitions. Choosing a deployment profile based on the transition time would require us to consider several cloud properties such as the *VM* disk size, the network topology, and also schedule migrations according to projections on the available network bandwidth. We expect that our approach of reducing the number of migrating *VM*s will affect fewer users. *VM*s that would yield long transition times can be excluded from migrations using the `PinVM` constraint.

6 Nefeli in Dynamic Environments

The overall goal of *Nefeli* is to make choices regarding the deployment profile based on the user's needs and the system's performance. As both needs and actual preferences change over time, *Nefeli* must act accordingly and produce updated deployment profiles. To this end, our approach features a notification mechanism that relays events towards *Nefeli*.

6.1 Event Types and their Handling in *Nefeli*

Events are used to signal when the virtual infrastructures should be reorganized -through *VM* migration operations- so as to reflect the changes in *Nefeli*'s environment. We group events into two classes according to their origin:

• *Events activated by direct human intervention:* they include the submission or removal of any number of task-flows served. This class also includes events that help administrators effectively control the operation of both cloud and *Nefeli*. Consider for example node maintenance tasks that require specific parts of the hardware infrastructure to be shut down. The cloud administration must migrate the hosted *VM*s to nodes that will not be affected. *Nefeli* must provide the means to support this kind of activity and it does so by responding to

events set by the administrator combined with hard constraints included in task-flow descriptions. In similar spirit, activation of constraints such as `PowerSave`, may be performed on demand.

• *Events triggered by any monitoring activity in the context of physical/virtual infrastructure or any authorized third party component:* *VM* redeployment may take place after a threshold in a resource utilization is exceeded. Through the cloud middleware connector, *Nefeli* offers hooks for monitoring CPU utilization on both *VM*s and hosting nodes. Other internal activities such as network traffic are monitored through third party *monitoring tools* (Figure 1) such as *Nagios* [16]. Receiving this type of events may indicate that the deployment profile currently used is ineffective. For instance, long time periods with specific hosting nodes displaying high CPU loads while others staying idle, mean that the *VM*s hosted on those nodes have become a performance bottleneck. Such bottlenecks can be handled by a redeployment of *VM*s. This class of events includes events coming from both the physical and the virtual infrastructure. The virtual infrastructure may signal the end of a task-flow or even the initiation of a new one. This event class allows for the development of cloud-efficient applications while keeping applications agnostic to the infrastructure on which they are executed.

All events can be combined in Boolean expressions using *AND*, *OR* and *NOT* operations. In this regard, both consumers and administrators may express complex conditions calling for *VM* re-organization. Such Boolean expressions are placed in the *XML* descriptions of task-flows.

6.2 Application Driven Operation

Users frequently want to execute different task-flows at different time periods on top of their virtual infrastructure. These time periods are delineated by specific events occurring in the system. These events should be appropriately registered so that *Nefeli* carries out the respective optimization of the virtual infrastructure.

Listing 3 depicts the way such events are introduced to *Nefeli* through an extended *XML* input file. Similarly to the single task-flow description, the `VirtualMachines` section provides all *VM*s of the virtual infrastructure. The `Constraints` section describes all constraints, regardless of the task-flow to which they refer. In the `Profiles` section, the consumer provides one set of weights for each separate task-flow; there exist two profiles corresponding to two distinct task-flows. In more detail, the *XML* input file involves two *VM*s with IDs 1 and 2. Each of the two *VM*s is referenced by a separate `FavorVM` constraint. The user's intention is to have two deployment profiles each one favoring a different *VM*. To this end, there are two deployment profiles in the `Profiles` section; each profile assigns a 0.9 weight to the constraint to be active and 0.0 to the one that should be inactive. As clarified in Section 4.2, constraints with 0.0 weight have no impact on the evaluation of deployment profile score.

The next two sections of the input *XML* refer to the transition between the deployment profiles. The `Events` section points out events whose occurrence will cause a change in the deployment followed. The two events in question are: *a)* a time-based event that periodically sends a signal, and *b)* a network-based event that starts a server listening for a predefined message to arrive. Both events belong to the second class of Section 6.1. The first event, with ID 1, will be triggered every 1,000 seconds as defined within the `Time` tag. The second event (`Net` tag) will have *Nefeli* listen for messages coming in on port 2324. If the message received is the string `Change`, the event will be triggered.

```
<Altenative_Task-flows>
 <VirtualMachines>
  <VM id="1"><RAM>512</RAM><Disk>VM1.img</Disk></VM>
  <VM id="2"><RAM>512</RAM><Disk>VM2.img</Disk></VM>
 </VirtualMachines>

 <Constraints>
  <FavorVM id="1"><VMid>1</VMid></FavorVM>
  <FavorVM id="2"><VMid>2</VMid></FavorVM>
 </Constraints>

 <Profiles>
  <Profile id="1">
   <Weights>
    <Constr id="1"><Weight>0.9</Weight></Constr>
    <Constr id="2"><Weight>0.0</Weight></Constr>
   </Weights>
  </Profile>
  <Profile id="2">
   <Weights>
    <Constr id="1"><Weight>0.0</Weight></Constr>
    <Constr id="2"><Weight>0.9</Weight></Constr>
   </Weights>
  </Profile>
 </Profiles>

 <Events>
  <Time id="1"><Period>1000</Period></Time>
  <Net id="2">
   <Port>2324</Port><Msg>Change</Msg>
  </Net>
 </Events>

 <Transitions>
  <Transition>
   <From>1</From><To>2</To><Event>1</Event>
  </Transition>
  <Transition>
   <From>2</From><To>1</To><Event>2</Event>
  </Transition>
 </Transitions>
</Alternative_Task-flows>
```

Listing 3: *Nefeli* extended *XML* input sample.

Transitions caused by the above two events are sketched in the last segment of the *XML* description. Event with ID 1 causes a transition from deployment profile 1 to deployment profile 2, while activation of the event 2 has the opposite effect. The outcome of this "Alternative Task-flow" description is that *VM* 1 will be favored for 1,000 seconds and then *VM* 2 will be "promoted" until the message `Change` is received. This alternate usage of profiles continues as long as the virtual infrastructure remains on-line.

6.3 Coordinating Nefeli's Operation

Figure 4 depicts the key *Nefeli* components and their role in a cloud environment. *Deployer* is active the entire time the cloud is available and keeps track of all task-flows. As events may cause the *VM*-to-host mapping to change, the *Deployer* takes appropriate action to produce and apply updated deployment profiles. To do so, this component a) contacts the *Planner* to acquire a high scoring deployment profile and b) makes use of a cloud connector to interact with the underlying middleware.

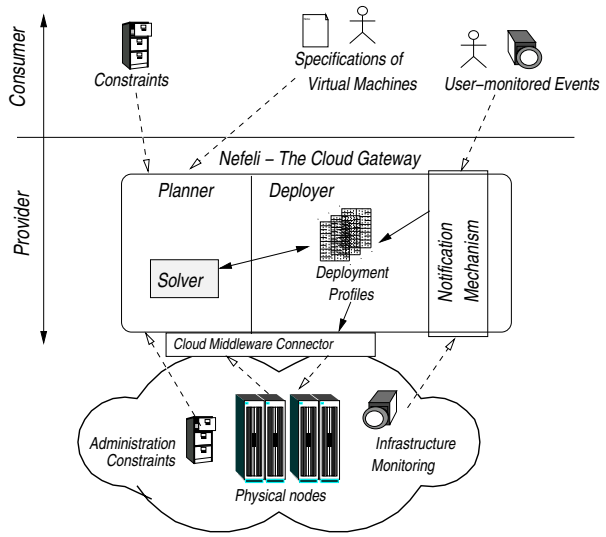


Fig. 4: The environment *Nefeli* operates in.

At bootstrap, *Nefeli* starts listening for two key events: requests for new virtual infrastructures as well as calls for purging virtual infrastructures that have run their work to completion. The *Deployer* maintains a list of all deployed virtual infrastructures. Each such infrastructure is paired with a corresponding task-flow for which the *VM* deployment is optimized; this task-flow is termed as *active*. Also, for each infrastructure there may exist *inactive* task-flows that get activated only if appropriate events occur. The transition between deactivating a task-flow and activating another –*inactive* thus far– necessitates a different *VM*-to-host deployment.

Changing the *VM*-to-host mapping calls for respective *VM* migration requests to be issued to the underlying cloud middleware. The precise migration requests are unknown until the target deployment profile is produced by the *Planner*. The *Deployer* contacts the *Planner*, passes the constraints of all active task-flows and receives the *VM*-to-host mapping. The *Deployer* subsequently orchestrates the transition to the produced deployment profile through proper migration requests. This cycle of event monitoring and profile transition continues until all virtual infrastructures (and thus task-flows) are purged.

7 Evaluation

We have implemented *Nefeli* as a Java library and a web service to allow both integration with user applications and easy embedding in cloud management systems. The key objectives of our experimental evaluation are to:

- examine the efficiency of our system as compared with existing placement alternatives as far as CPU utilization and throughput are concerned.
- investigate the behavior of *Nefeli* as the number and features of virtual resources available for processing change over time.
- evaluate the overheads involved in using *Nefeli*.

Our experimentation entails diverse scientific task-flows executed on simulated infrastructures as well as applications executed in a private *Iaas*-cloud. The difference between simulation and real application evaluation is in the infrastructure used. We have implemented two cloud middleware connectors (Figure1). The first simulates a physical infrastructure and the second interacts with *OpenNebula* [2] through *XML-RPC*. At this time, *OpenNebula* along with *Eucalyptus* [4] and *OpenStack* [5] are all key open-source *IaaS*-cloud middleware projects with similar if not identical objectives. In what follows, we first examine performance and scalability issues using the simulated infrastructure and subsequently present the benefits of using *Nefeli* in a real application.

7.1 Nefeli in a Simulated Cloud Environment

The physical nodes of the simulated infrastructure are assumed to be connected over a 10 $Mbps$ switch in a star network topology. Each node provides two types of resources: RAM and CPU-cycles. *VM*s reserve RAM upon their deployment and consume CPU-cycles to transform input data to output data. The number of available cycles per second to be shared among hosted *VM*s allows us to designate the CPU performance rate. Increasing the available CPU-cycles per second appointed to each host results in producing more output data per unit of time (more bytes per second). We set physical nodes to have 8 GB of RAM and virtual nodes to have 512 MB.

The behavior of each *VM* is designated by two ratios:

- The *input-to-output* size ratio (input *KBytes*/output *KBytes*). This ratio quantifies how much of the input data must be consumed to produce a single unit of the output data.
- The *cycles-to-output* ratio indicates how many cycles have to be expended to produce a single unit of output (i.e., *Byte*). By increasing the cycles per second available to each CPU and keeping the same *cycles-to-output* ratio, we allow more output *Bytes* to be produced (in the duration of a second). Such an increase essentially corresponds to an upgrade of the respective CPU.

Output bytes are forwarded to other virtual machines via the network and thus consume network bandwidth. Using the above modeling, a task-flow creation requires the following: first, setting up characteristics of each *VM* with both *input-to-output* and *cycles-to-output* rates and second, defining the network connections describing the

TABLE 2: *VM* characteristics for the *SIPHT*-inspired task-flow.

VM IDs	Input-to-Output	Cycles-to-Output
1	11.34	78.1
2	0.11	21.27
3	4.64	43.93
4	59.96	772.89
5	4.31	59.87
6	5.3	1.69
7	2.44	24.48
8	137	576
9	57.54	298.86
10	0.97	3.66
11	3.12	1.73
12	1	0.09
$\{13, 14, 15, 16, 17, 18\}$	430	132

data paths of the specific task-flow(s) at hand.

***SIPHT* and *CyberShake* task-flows:** we present two task-flows inspired by the *SIPHT* search engine [17] and the *CyberShake* [18] workflow.

• The sRNA identification protocol using high-throughput technology (*SIPHT*) program searches over a large database of RNA encoded genes. In doing so, it combines a variety of individual algorithms in a well formed workflow. This workflow has been split into tasks so that it can be conveniently executed in cluster environments. Figure 5 shows the processing nodes, corresponding to respective *VM*s, and network topology used by this task-flow. The *input-to-output* and *cycles-to-output* ratios for all *VM*s, as extracted from [19], are shown in Table 2. Taking into account the data flow ratios of Table 2, the layout of the nodes and knowledge of bottlenecks acquired from preliminary test runs we are able to formulate a number of hints to be passed to *Nefeli*. Here, *VM*s $\{1, 2, 3, 4\}$ would better be deployed on different hosting nodes as they operate *in parallel*. Nodes 5 and 11 may develop into bottlenecks and are thus better placed on dedicated hosting nodes. Table 3 presents our choice of weights for each of the constraints used to generate the deployment profiles.

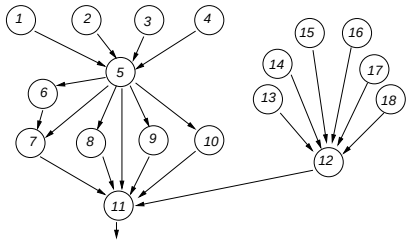


Fig. 5: *SIPHT*-inspired task-flow

• *CyberShake* workflow characterizes earthquake hazards using the Probabilistic Seismic Hazard Analysis (PSHA) technique. Figure 6 shows the layout of *VM*s involved in *CyberShake*-inspired task-flow while the *input-to-output* and *cycles-to-output* ratios are presented in Table 4. The user hints for this task-flow are presented in Table 5. As

TABLE 3: Two sets of User Weighted Constraints for *SIPHT*

Constraints	*Nefeli*	*Nefeli-power*
`ParVMs` on *VM*s $\{1, 2, 3, 4\}$	0.50	0.50
`FavorVM` on *VM*s 5	0.80	0.80
`FavorVM` on *VM*s 11	0.80	0.80
`PowerSave`	0.0	0.40

TABLE 4: *VM* properties for the *CyberShake*-inspired task-flow.

VM IDs	Input-to-Output	Cycles-to-Output
$\{1, 11\}$	87.75	0.80
$\{2, 3, 4, 5, 12, 13, 14, 15\}$	39, 573	21.27
$\{6, 7, 8, 9, 10, 16, 17, 18, 19, 20\}$	20	1, 360
21	1	1

*VM*s 1 and 11 act as top level data-producers of the task-flow, they have to be placed on separate hosting nodes and they should be favored over the rest of the *VM*s. *VM*s $10, 20$ and 21 should also be favored since they are on the receiving end of multiple data flows as shown in Figure 6. Finally, we ask for *VM*s operating in parallel ($\{2, 3, 4, 5, 12, 13, 14, 15\}$ and $\{6, 7, 8, 9, 16, 17, 18, 19\}$) to be hosted on different nodes.

In Tables 3 and 5 there are two sets of constraint weights: the first is mostly concerned with throughput attained by the infrastructure and is termed *Nefeli*, while the second includes the `PowerSave` constraint, indicating that we want to reduce the number of active hosting nodes. The latter has to do with the consumption of power, often of high concern in computing installations. This second set of constraints is termed *Nefeli-power*.

The simulated environment allows us to easily change key cloud properties affecting the performance of the

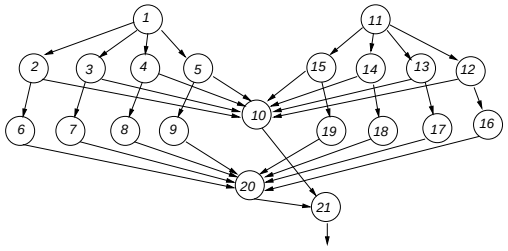


Fig. 6: *CyberShake*-inspired task-flow

TABLE 5: Two sets of User Weighted Constraints for *CyberShake*

Constraints	*Nefeli*	*Nefeli-power*
`ParVMs` on *VM*s $\{1, 11\}$	0.25	0.25
`ParVMs` on *VM*s $\{2, 3, 4, 5, 12, 13, 14, 15\}$	0.25	0.25
`ParVMs` on *VM*s $\{6, 7, 8, 9, 16, 17, 18, 19\}$	0.25	0.25
`FavorVM` on *VM*s $\{1, 11\}$	0.10	0.10
`FavorVM` on *VM*s $\{10, 20, 21\}$	0.30	0.30
`PowerSave`	0.0	0.40

underlying physical infrastructure. We can: *a)* selectively "increase" the CPU performance and *b)* offer additional hosting nodes. We are interested in the throughput of the entire flow as measured by the outcome of the trailing node. In the *SIPHT* task-flow this is the *VM* with ID 11 and in the *CyberShake*, the *VM* with ID 21. Our two configurations –*Nefeli* and *Nefeli-power*– are compared against implementations of the following scheduling policies:

- *Power Saving:* when instantiating *VM*s, we exclusively use the clause that the number of active hosting nodes must be the smallest possible.
- *Random:* schedule *VM*s randomly. This policy bears minimal overhead.
- *Balance VMs:* attempt to distribute *VM*s equally across all hosting nodes.

CPU performance: we select 6 hosting nodes in this experiment and gradually increase the CPU performance rate up to 20 times. We do so by increasing the CPU cycles each hosting node has available every second.

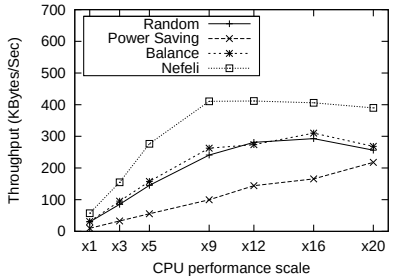
Figures 7a and 7d depict the performance gains obtained for the *SIPHT* and *CyberShake* task-flows using the *Nefeli* configuration of Tables 3 and 5. In both task-flows, *Random* and *Balance VMs* schedulers demonstrate approximately the same performance. Since the *Balance VMs* policy iterates over the hosting nodes for placing newly instantiated *VM*s, one would expect that it should be more beneficial than its *Random* counterpart. However, it is not since it does not discriminate between *VM*s that are to be deployed on the hosts. *VM*s are chosen randomly, albeit evenly distributed across hosts. The *Power Saving* policy uses fewer physical nodes to host the same number of *VM*s compared to *Balance VMs* and *Random* policies. Thus, its *VM* placement is effective in light of high performing CPUs. The conservative *Power Saving* scheduler is an underachiever in overall performance even though the utilization of its active hosting nodes is typically high. *Nefeli* consistently manages to outperform all other schedulers showing that the potential bottlenecks that have been user-"hinted" through pertinent constraints have been addressed successfully.

Provide additional physical nodes: increasing the number of physical nodes results in *a)* having more CPU-cycles available, *b)* increased overall network bandwidth and *c)* increased power consumption. We start with 3 hosting nodes and we gradually reach an infrastructure consisting of 10 physical machines. In Figures 7b and 7e, we present the mean throughput delivered by the *SIPHT* and *CyberShake* task-flows respectively. In both cases the three schedulers and the two *Nefeli* configurations are used. In infrastructures with very few nodes, the options for optimal deployment are limited. This is the reason why all scheduling policies perform equally well when 3 hosting nodes are available. As more nodes are added, the *Nefeli* configuration outperforms the other schedulers. All schedulers except the *Power Saving* display notable performance improvements since they take advantage of additional nodes. The *Power Saving* scheduler always uses a fixed number of hosting nodes, in our case two. Thus, it displays no improvement.

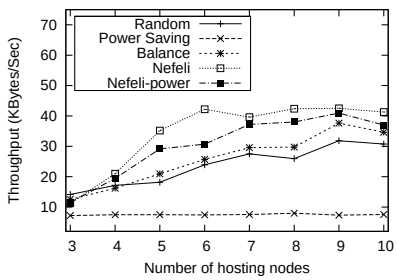
Power saving schedulers [20]: are popular amongst cloud operators because they reduce the maintenance cost of the physical plant. Hosting nodes serving no *VM*s may enter a "deep-sleep" state in which they consume far less energy compared to their normal operation. *Nefeli-power* may assist in reducing the number of active hosting nodes through its power saving constraint in the production of deployment profiles. Given that power is consumed only by the active hosting nodes and only during the period the virtual infrastructure is available, Figures 7c and 7f show the task-flow throughput achieved normalized by the number of active nodes. The normalized throughput is computed by dividing the total throughput reported in Figures 7b and 7e by the number of physical nodes that host *VM*s. This average throughput rate per active host captures the power efficiency of the physical substrate. The lower the average throughput per active host is, the longer the amount of time the nodes have to remain on-line to produce the same amount of output data.

For both task-flows, the *Power Saving* scheduler uses exactly two hosting nodes to deploy all *VM*s and therefore, it remains largely unaffected by the addition of extra nodes. *Random*, *Balance VMs* and *Nefeli* use as many hosting nodes as possible. For the *SIPHT* task-flow the trend displayed in Figure 7c is a decrease in the average throughput per node as nodes are added. Figure 7b shows that *Nefeli* cannot enhance overall throughput when more than 6 nodes are available. This is because 6 nodes can fully satisfy all constraints of Table 3. Therefore, *Nefeli-power* offers an improvement over *Nefeli* in terms of average throughput achieved per active node (Figure 7c) as the additional nodes are not used for hosting *VM*s. In the case of *CyberShake*, the extra resources offered by the additional hosting nodes are effectively harvested. Here, *Nefeli-power* offers total throughput similar to *Random* and *Balance VMs* (Figure 7e) but uses fewer hosting nodes as we show in Figure 7f. Overall, *Nefeli-power* presents a compromise between the high throughput rates achieved by *Nefeli* and the number of active nodes. Figures 7b and 7e in combination with Figure 7c and 7f depict the tradeoff between overall performance achieved in terms of throughput and the use of hosting nodes.

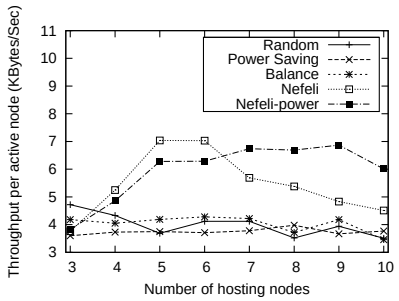
Multiple *SIPHT* task-flows: to evaluate the performance overhead incurred by profile production, we simulate a physical infrastructure of 500 nodes and we request the deployment of gradually increasing numbers of virtual infrastructures. Each such infrastructure serves a separate *SIPHT* task-flow. We allow the amount of *SIPHT* task-flows to range from 1 to 200 (involving 18 to 3,600 *VM*s). The three lines shown in Figure 8 depict the time needed to decide on the *VM* placement for three different values of the `same_iterations` used as input in Algorithm 2. Using a single thread of a *Core(TM)2 Duo*



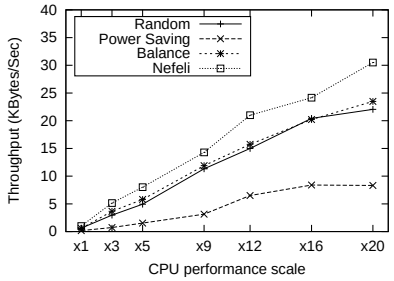
(a) *SIPHT* throughput under increasing CPU performance



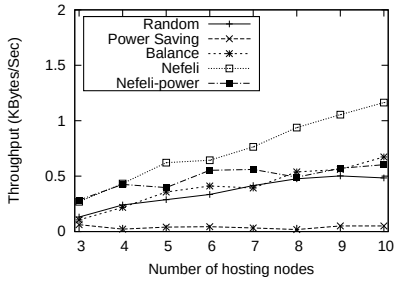
(b) *SIPHT* throughput under increasing number of hosting nodes



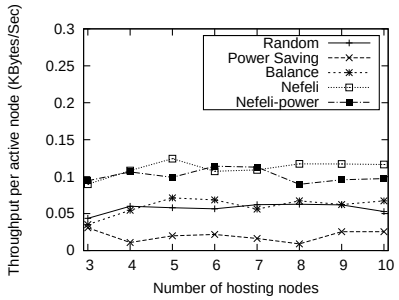
(c) *SIPHT* throughput per active node



(d) *CyberShake* throughput under increasing CPU performance



(e) *CyberShake* throughput under increasing number of hosting nodes



(f) *CyberShake* throughput per active node

Fig. 7: Evaluation of *Nefeli* under the *SIPHT* and *CyberShake* task-flows

CPU T7100 at 1.80GHz, our *Nefeli* prototype implementation produces any deployment profile in less than 45 seconds. The lower the `same_iterations` is, the less time is needed to reach to a placement decision. Yet, this has an impact on the deployment profile's score. In Figure 9 we show the effect of the `same_iterations` parameter on the profile score. In this experiment, we fix the number of *SIPHT* instances to 200 and we vary the `same_iterations` from 5 to 200. More iterations result in higher scoring profiles.

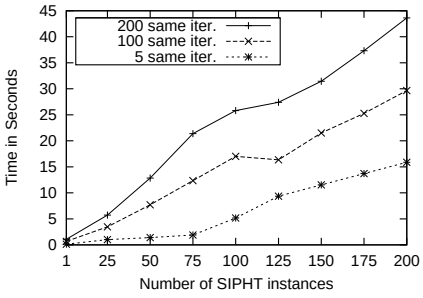


Fig. 8: Plan production time under increasing numbers of task-flows.

The outcome of experimenting with simulated cloud environments shows the potential of our approach. When cloud consumers indicate likely bottlenecks using hints or constraints, *Nefeli* can drastically enhance the overall performance of the equipment used.

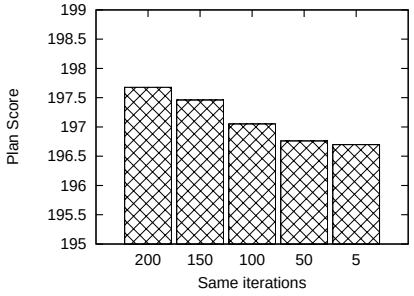


Fig. 9: Plan score of 200 *SIPHT* instances for different numbers of `same_iterations`.

7.2 Nefeli in a Real Private Cloud Environment

We now outline our evaluation using a real private-cloud environment running *Nefeli* and show the gains obtained when compared with the scheduler of a widely-deployed open-source cloud middleware [2]. We have created a cloud-enabled application that performs video and audio transformations while offering deployment hints to *Nefeli*. Such applications are very well suited to cloud execution as many *VM*s can simultaneously operate on separate fragments of the input media. In addition, the elongated processing time ameliorates the *VM* scheduling and deployment delays.

The transformation application serves one user at a time in a *first-come first-serve* fashion. Users must provide a media file comprised of a video and an audio stream. Our application accepts the input streams and

transforms them to a user-selected format. The available output formats are: DVD, SVCD or VCD. The user also provides the encoding property, either PAL or NTSC, regarding the display standard. The combination of the format and display standard specifies the compression algorithm (MPEG-1 & 2) and the video resolution of the output. A transformation request is served by following a four-step procedure:

1) the input file is split into equally sized parts. The number of parts is equal to the number of *VM*s capable of processing them (3 in our virtual infrastructure).
2) each part is dispatched to *VM*s performing the appropriate video transformation,
3) once video transformation completes, all parts are forwarded to the *VM*s that perform the audio transformation, and finally,
4) all segments are merged into one transformed video.

All valid combinations of the three output formats (DVD, SVCD, VCD) and the two display settings (PAL, NTSC) for both audio and video transformation yield in summary 12 distinct capabilities (6 for video and 6 for audio) shown in Table 6. Our virtual infrastructure is made of 6 *VM*s with IDs 1 to 6. Shown in Table 6, each of the video/audio transformations of steps 2 and 3 above is carried out by 3 *VM*s. During a video or audio transformation, 3 *VM*s can work in parallel on three parts of the input file. Therefore, step 1 splits the input file into 3 equally sized parts. We note that *VM*s do not feature identical capabilities. For example, *VM* 1 can produce only DVD/PAL, SVCD/PAL, and SVCD/NTSC video as well as DVD/PAL, VCD/PAL and VCD/NTSC audio streams. The rest of the six capabilities are not installed on *VM* 1.

In an optimal *VM* deployment, *VM*s simultaneously performing video/audio transformations should be distributed among different hosting nodes so that they do not compete for CPU cycles. We allocate transformation programs (or capabilities) to *VM*s in a manner such that there is no single optimal *VM*-to-host mapping for all transformation operations. Table 6 indicates what is the optimal deployment profile for each transformation operation. For example, in the optimal mapping of DVD/-PAL, *VM*s *1, 2, 3* are distributed among different hosting nodes as they perform the video transformations. Along these lines, *VM*s *1, 4, 5* also have to be placed on different hosting nodes while producing the audio stream.

TABLE 6: Mapping of transformational tasks to *VM*s.

Transformation	VMs for Video	VMs for Audio
DVD/PAL	1, 2, 3	1, 4, 5
DVD/NTSC	2, 3, 4	2, 5, 6
VCD/PAL	3, 4, 5	3, 6, 1
VCD/NTSC	4, 5, 6	4, 1, 2
SVCD/PAL	5, 6, 1	5, 2, 3
SVCD/NTSC	6, 1, 2	6, 3, 4

During the experiment, the mapping of the capabilities-to-*VM*s remains fixed while *Nefeli* dynamically performs the *VM*-to-host mapping. We use an *XML* extended input file (as the one in Section 6.2) to indicate six different task-flows corresponding to the 6 rows of Table 6. Every time, we encounter, for example, the creation of a VCD/NTSC stream the respective deployment hints are used by *Nefeli* to produce a suitable deployment profile. Once this piece of work completes, another request through a respective event might appear, for example DVD/NTSC. Again, an appropriate profile is produced and applied to attain an optimal *VM*-to-host mapping for the specific transformation function.

The communication between the application and *Nefeli* is based on the event mechanism. We assign one event for each of the output formats (DVD, VCD, SVCD) and two more for the display standard (PAL or NTSC). These events are triggered by sending a signal to ports on which *Nefeli* listens. Format and display events are combined using Boolean operators before asking the *Deployer* to take appropriate action. For instance, requesting a DVD/PAL transformation will trigger both events "DVD" and "PAL".

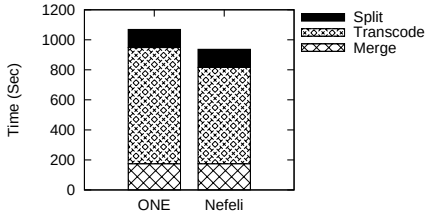


Fig. 10: Comparing *Nefeli* to *OpenNebula* in the three phases of the encoding.

The physical substrate, in this experiment, is made of 3 nodes connected via a 1 $GBps$ Ethernet switch. Each node is equipped with 8 GB of RAM and an Intel(R) Core(TM)2 CPU 6600 at 2.40 GHz CPU. Live migration is not available and *VM* images are fetched from a file server. We use *Xen 3.2.1* [21] as *VM* hypervisor and *OpenNebula v.1.2.0* [2] is the cloud middleware. *Nefeli* interacts with *OpenNebula* through its *API* that is exposed with the assistance of the *XML-RPC* protocol. *VM*s use 512 MB of RAM and face no restriction on the CPU resource usage.

Figure 10 shows the time required to process a DivX (MPEG-4 video/MPEG layer-3 audio) file in our cloud infrastructure using either *Nefeli* or simply employing the default *OpenNebula VM* scheduler. The *VM* placement of *OpenNebula* is based on a match-making policy that takes into account the free memory and CPU each hosting node reports and the respective *VM* requirements. Each *VM* occupies a percentage of the hosting CPU and uses a portion of its memory. The *OpenNebula* match-making approach proves ineffective for our application as *VM* CPU requirements are known only at

run-time and not at deployment time. *Nefeli* achieves a 17% improvement in the time required to have video and audio transformation complete. File splitting and merging display no gains from an optimal deployment since both operations are performed on a single node. In our infrastructure featuring hosts with Core(TM)2 CPU 6600 at 2.40GHz processors, we found the overhead for the production of the deployment profile to be negligible when compared with virtual disk copying and *VM* booting operations.

8 Related Work

In the pure virtualized environments of [6], [7], [22] the satisfaction of SLAs is examined in light of changing workloads. Local utility functions provide feedback to a global and system-wide optimization two-level mechanism that decides on resource provisioning. In contrast, *Nefeli* exclusively uses a single optimization scoring function in which all deployment preferences are included.

The system described in [9] employs smart component regeneration through *VM* instantiation to achieve high availability and low response time of multi-tier applications. Placing and instantiating *VM*s is done based on load predictions using queue modeling. Anti-collocation and resource constraints are used in [23] to guarantee high availability. Through the introduction of shadow *VM*s the placement algorithm reserves locations on separate nodes that can be used to evacuate *VM*s in case of a failing host. In *Nefeli* high availability is only one of the placement aspects we target through constraints. With constraints that function in a plug and play fashion we serve several versatile placement goals at the same time. Advice, similar to the hints of our approach, regarding the placement of *VM*s are used in [24]. Components, termed *Domain Advisors*, offer their advice to a constraint satisfaction solver that yields the final placement of *VM*s to hosts. Compared to [24], our approach is better aligned with the cloud abstractions as we classify hints as those used by the consumer and those available to the administration. Furthermore, our event-based mechanism allows for the implementation of cloud enabled applications that dynamically adjust their deployment on the cloud. The *Plasma* [25] consolidation manager makes a clean distinction between the roles of the user requesting *VM*s and the administrator responsible for the physical infrastructure. This distinction, available also in *Nefeli*, is vital for the transparent operation of the cloud. Yet, compared to our approach, *Plasma* has limited options in the constraints available. A total of four constraints is used, two of them are available to the cloud consumers and two to the administrators. The cloud software by *VMware* [26] offers support for a limited set of constraints in the placement of *VM*s. Its *Distributed Resource Manager (DRM)* can exploit collocation and anti-collocation deployment hints. However, the cloud consumer is not given a wide range of constraints to use in describing an ideal deployment. The concept of resource pools can serve certain properties required by the customers (e.g., high availability). Contrary to the platform offered by *VMware*, with *Nefeli* we take advantage of the flexibility that pluggable constraints offer in matching user needs with the facilities offered by the cloud. *VM* placement under SLA guarantees and power efficiency is examined in *pMapper* [20]. SLAs and the cost of live migration are taken into account while *VM*s are continuously reorganized to balance load. In our approach, power efficiency is considered as an administrative preference. Furthermore, our *VM* placement decisions are based on deployment hints and not on load predictions. Finally, the system described in [27] also rearranges *VM*s based on SLAs, high-level policies, and performance forecasts. *Nefeli* does not employ load predictions, rather user hints are exploited in handling peak load and high-level policies can be readily enforced by administrative deployment hints.

The problem of constraint satisfaction in *VM* placement is shown to be hard and a hierarchical placement approach is suggested to handle scalability in [8]. In our work, we formulate the *VM* placement issue as a constraint satisfaction problem so as to benefit from the evolution of respective contemporary solvers. In [15], we offer an approach that elaborates on the scalability of such solvers.

The behavior of many distributed applications can be modeled as recurrent data and control flows (or collectively workflows) that often follow distinct and specific patterns [28]. In *Nefeli*, we offer the means to express the existence of such patterns as task-flows; *Nefeli* exploits these patterns to attain improved *VM* deployment.

The allocation of resources in dynamic distributed environments [29] where load and resource availability change over time requires adaptive policies. In [30], [31], such resource sharing policies are proposed for the execution of jobs on the Grid. Utility functions [31] are proposed to help quantify the efficient execution of jobs in light of different resource sharing disciplines. *Grid*-jobs frequently form large *DAG*s often split before they are dispatched for execution.

Rearrangement of *VM*s aims at harvesting cloud resources in the most effective way. In similar spirit, data stream processing systems [32]–[34] aim to produce the most efficient placement of operators in the network for processing of data flowing from data sources to interested data consumers.

In many respects, *Nefeli* realizes a number of features envisaged by autonomic computing [35]. Autonomic systems attempt to self-adjust according to the needs of the applications they process. Specific application requirements are expressed in a high-level language which are then interpreted by the tuning component of the systems. In enterprise infrastructures, these requirements are described with the help of service-level agreements (SLAs). The degree to which an SLA is satisfied is quantified through user-furnished utility functions [11],

[12]. Although, the stated objective of SLAs is to make applications agnostic of the system they run on, this regularly fails because defining an appropriate utility function is a nontrivial task. This definition requires both application expertise and detailed knowledge of the autonomic model used. Moreover, complex SLA requirements frequently require significant human intervention [10]. In contrast, *Nefeli* uses predefined utility functions that correspond to known properties of the task-flows under execution.

Compared to other existing scheduling *VM*–based load-balancing systems [36]–[38], *Nefeli* exhibits two key differences. First, our approach does not examine the execution of specific *VM*s in isolation but considers all task-flows making up the current workload before rearranging the virtual infrastructure. Second, the event-based mechanism that we use to trigger *VM* rearrangements is not based solely on specific usage thresholds of resources. Moreover, *Nefeli* supports the use of any external monitor mechanism available within the physical infrastructure such as *Nagios* [16].

9 Conclusions - Future work

In this paper we present *Nefeli*, a hint-based *VM* scheduler that serves as a *gateway* to *IaaS*-clouds. Users are aware of the flow of tasks executed in their virtual infrastructures and the role each *VM* plays. This information is passed to the cloud provider, as *hints*, and helps drive the placement of *VM*s to hosts. Hints are also employed by the cloud administration to express its own deployment preferences. *Nefeli* combines consumer and administrative hints to handle peak performance, address performance bottlenecks and effectively implement high-level cloud policies such as load balancing and energy savings. An event-based mechanism allows *Nefeli* to reschedule *VM*s to adjust to changes in the workloads served. Our approach is aligned with the separation of concerns *IaaS*-clouds introduce as the users remain unaware of the physical cloud structure and the properties of the *VM* hosting nodes. Our evaluation, using simulated and real private *Iaas*-cloud environments, shows significant gains for *Nefeli* both in terms of performance and power consumption.

In the future, we plan to: *a)* investigate alternative constraint satisfaction approaches to address scalability issues present in large infrastructures, *b)* offer deployment hints that will effectively handle the deployment of virtual infrastructures in the context of real large cloud installations, *c)* extend the support of *Nefeli* to other cloud middleware platforms [4], [5] by providing additional *Cloud Middleware Connectors*.

Acknowledgments

We thank the reviewers for their constructive comments that helped us improve our presentation. We would also like to thank V. Floros for his contribution in the early stages of this work. A preliminary version of the paper appeared in [39]. This work has been partially supported by the *D4Science I & II* EU FP7 projects and ERC Starting Grant # 279237.

References

[1] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, 2005.
[2] "OpenNebula," http://www.opennebula.org, May 2011.
[3] Amazon, "Elastic Cloud," http://aws.amazon.com/ec2/, 2009.
[4] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in *9th IEEE/ACM Int. Symposium on Cluster Computing and the Grid (CCGRID)*, Shanghai, China, May 2009, pp. 124–131.
[5] "OpenStack," http://www.openstack.org/, Feb. 2011.
[6] H. N. Van, F. D. Tran, and J.-M.Menaud, "Autonomic Virtual Resource Management for Service Hosting Platforms," in *Proc. of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, BC, Canada, 2009, pp. 1–8.
[7] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Q.B. Wang, "Appliance-Based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center," in *Proc. of the 4th Int. Conf. on Autonomic Computing*, Washington, DC, 2007, p. 29.
[8] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whalley, and E. Snible, "Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-aware Virtual Machine Placement," in *Proceedings of the 2011 IEEE Int. Conf. on Services Computing (SCC)*, Washington, DCC, USA, July 2011.
[9] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Performance and Availability Aware Regeneration For Cloud Based Multitier Applications," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, Chicago, Illinois, USA, June 2010.
[10] P. deGrandis and G. Valetto, "Elicitation and Utilization of Application-level Utility Functions," in *Proc. of the 6th Int. Conf. on Autonomic Computing*. Chicago, IL, USA: ACM, 2009.
[11] J. O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.
[12] G. Tesauro and J. O. Kephart, "Utility Functions in Autonomic Systems," in *Proc. of the 1st Int. Conf. on Autonomic Computing*. New York, NY, USA: IEEE Computer Society, 2004, pp. 70–77.
[13] D. B. G. Ian J. Taylor, Ewa Deelman and M. Shields, *Workflows for e-Science Scientific Workflows for Grids*. London: Springer, 2007.
[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.
[15] K. Tsakalozos, M. Roussopoulos, and A. Delis, "VM Placement in non-Homogeneous IaaS-Clouds," in *Proc. of the 9th Int. Conf. on Service Oriented Computing*, Paphos, Cyprus, Dec. 2011.
[16] D. Josephsen, *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ: Prentice Hall PTR, 2007.
[17] "sRNA identification protocol using high-throughput technology (SIPHT)," http://newbio.cs.wisc.edu/sRNA/ , Harvard Medical School, Boston, Massachusetts, 2010.
[18] E. Deelman and et al., "Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example," in *Proceedings of the Second IEEE Int. Conf. on e-Science and Grid Computing*, ser. E-SCIENCE '06. Amsterdam, Netherlands: IEEE Computer Society, 2006.
[19] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of Scientific Workflows," in *3rd Workshop on Workflows in Support of Large-Scale Science*, Austin, TX, November 2008, pp. 1–10.
[20] A. Verma, P. Ahuja, and A. Neogi, "pMapper: Power and Migration Cost AwareApplication Placement in Virtualized Systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Leuven, Belgium, 2008, pp. 243–264.
[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. of the 19th ACM Symposium on Operating Systems Principles*. Lake George, NY: ACM, October 2003.
[22] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the Use of Fuzzy Modeling in Virtualized Data Center Management," in *Proc. of the Fourth Int. Conf. on Autonomic Computing*, Jacksonville, Florida, USA, June 2007.

[23] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz, "Guaranteeing High Availability Goals for Virtual Machine Placement," in *Proc. of the 31st Int. Conf. on Distributed Computing Systems*, Minneapolis, Minnesota, 2011.
[24] R. Harper, L. Tomek, O. Biran, and E. Hadad, "A Virtual Resource Placement Service," in *Proc. of the 1st Int. Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV)*, Hong Kong, China, June 2011.
[25] F. Hermenier, J. Lawall, J.-M. Menaud, and G. Muller, "Dynamic Consolidation of Highly Available Web Application," INRIA, Tech. Rep. Research Report RR-7545, 2011.
[26] VMware, "VMware hypervisor," http://www.vmware.com/, 2012.
[27] C. Hyser, B. McKee, R. Gardner, and B. J. Watson, "Autonomic Virtual Machine Placement in the Data Center," http://www.hpl.hp.com/techreports/2007/HPL-2007-189.html, 2008.
[28] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
[29] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky Computing," *IEEE Internet Computing*, vol. 13, September 2009.
[30] K. Lee, N. Paton, R. Sakellariou, E. Deelman, A. Fernandes, and G. Mehta, "Adaptive Workflow Processing and Execution in Pegasus," in *Proc. of 3rd IEEE Int. Conf. on Grid and Pervasive Computing Workshops*, Kunming, PR China, 2008, pp. 99–106.
[31] K. Lee, N. Paton, R. Sakellariou, and A. Fernandes, "Utility Driven Adaptive Workflow Execution," in *Proc. of 9th IEEE/ACM Int. Symposium on Cluster Computing and the Grid*, Shanghai, PR China, 2009, pp. 220–227.
[32] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Centimentel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *Proc. of CIDR*, Asilomar, CA, January 2003.
[33] Y. Ahmad and U. Çetintemel, "Network-Aware Query Processing for Stream-based Applications," in *Proc. of VLDB'04*, Toronto, Canada, Aug. 2004.
[34] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," in *Proc. of ICDE*, Tokyo, Japan, Apr. 2006.
[35] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE–Computer*, vol. 36, no. 1, pp. 41–50, 2003.
[36] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *Proc. of the 17th Int. Symposium on High Performance Distributed Computing*. Boston, USA: ACM, June 2008, pp. 87–96.
[37] C. Weng, M. Li, Z. Wang, and X. Lu, "Automatic Performance Tuning for the Virtualized Cluster System," in *Proc. of the 29th IEEE Int. Conf. on Distributed Computing Systems*, Montreal, Quebec, Canada, June 2009, pp. 183–190.
[38] VMware, "vSphere," http://www.vmware.com/products/vsphere/, Nov. 2011.
[39] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis, "Nefeli: Hint-based Execution of Workloads in Clouds ," in *Proc. of the 30th IEEE Int. Conf. on Distributed Computing Sytems (ICDCS'10)*, Genoa, Italy, June 2010.

Konstantinos Tsakalozos is a PhD candidate at the National and Kapodistrian University of Athens under the supervision of Alex Delis. In 2007 he received his M.Sc. from Department of Informatics and Telecommunications also at the University of Athens. As a member of the "Management of Data Information, & Knowledge Group" he has participated in three EGEE projects, Diligent, D4Science and D4Science II. His research interests include cloud computing, distributed architectures and multidimensional indexing.

Mema Roussopoulos is an Assistant Professor of Computer Science at the National and Kapodistrian University of Athens. Her interests are in the areas of distributed systems, networking, and digital preservation. She is a recipient of the NSF CAREER Award, the ERC Starting Grant Award, and Best Paper Award at ACM SOSP 2003. She received her PhD in Computer Science from Stanford University.

Alex Delis is a Professor of Computer Systems in the Department of Informatics and Telecommunications at the National and Kapodistrian University of Athens. His research interests are in distributed computing and data management. His research work has been supported by agencies and organizations in Australia, US and the European Union. He holds a PhD in Computer Science from the Univ. of Maryland at College Park.