Nefeli: Hint-based Execution of Workloads in Clouds

Konstantinos Tsakalozos #1, Mema Roussopoulos #2, Vangelis Floros * and Alex Delis #3

Univ. of Athens, Athens, 15748, Greece {*k.tsakalozos*[1], *mema*[2], *ad*[3]}*@di.uoa.gr*

* *Greek Research & Technology Network, Athens, 11527, Greece* *efloros@grnet.gr*

***Abstract*—Virtualization of computer systems has made feasible the provision of entire distributed infrastructures in the form of services. Such services do not expose the internal operational and physical characteristics of the underlying machinery to either users or applications. In this way, infrastructures including computers in data-centers, clusters of workstations, and networks of machines are shrouded in "clouds". Mainly through the deployment of virtual machines, such networks of computing nodes become cloud-computing environments. In this paper, we propose *Nefeli*, a virtual infrastructure gateway that is capable of effectively handling diverse workloads of jobs in cloud environments. By and large, users and their workloads remain agnostic to the internal features of clouds at all times. Exploiting execution patterns as well as logistical constraints, users provide *Nefeli* with hints for the handling of their jobs. Hints provide no hard requirements for application deployment in terms of pairing virtual-machines to specific physical cloud elements. *Nefeli* helps avoid bottlenecks within the cloud through the realization of viable virtual machine deployment mappings. As the types of jobs change over time, deployment mappings must follow suit. To this end, *Nefeli* offers mechanisms to migrate virtual machines as needed to adapt to changing performance needs. Using our prototype system, we show significant improvements in overall time needed and energy consumed for the execution of workloads in both simulated and real cloud computing environments.**

I. Introduction

Computing "clouds" allow for the transparent access to diverse physical resources which are made available in the form of services. In general, cloud services can be classified according to the level at which they function as [1]: **a)** Software as a Service (*SaaS*), **b)** Platform as a Service (*PaaS*) and **c)** Infrastructure as a Service (*IaaS*). In this paper, we focus on *IaaS*-clouds [2] that exploit the use of virtual machines (*VMs*) to deploy computing systems on-demand [3], [4], [5]. We examine the effective deployment of *VM*s so that multiple and diverse workloads can be efficiently handled by the physical infrastructure. 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 virtual machines and effective options for node recovery in light of failure(s). However, sharing of physical resources may yield peak performance rates that are below expectation due to *VM* contention on particular physical nodes.

In this paper, we describe the design, implementation, and evaluation of *Nefeli*, a cloud gateway, that seeks to overcome contention of *VM*s for workloads consisting of diverse tasks executing in a cloud. *Nefeli* accepts requests from users for execution of particular workloads in the cloud and deploys these workloads within the cloud. *Nefeli* performs intelligent placement of *VM*s onto physical nodes using user-provided *hints*. Users model workloads as patterns of flows of data, computations, control/synchronization points and necessary network connections. We refer to these patterns as *task-flows* to distinguish these from the traditional workflow concept. Specifically, task-flows illustrate distinct computational phases meant to be executed on (distinct) virtual machines. Users provide hints to *Nefeli* by highlighting points of possible resource contention in their task-flows. *Nefeli* uses these hints to (re-)deploy *VM*s in the cloud in ways that achieve more efficient execution.

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. Moreover, user hints must not be allowed to violate the cloud abstraction and refer to internals of the physical hardware layout. *Nefeli* addresses both of these challenges.

To this date, there have been a number of efforts that attempt to fine-tune virtual infrastructures for executing specific types of jobs [6], [7]. In those, users "evaluate" the quality of the mapping of computational resources to *VM*s [8], [9], [10] by using either fixed service-level agreements (SLAs) or high-level conditions. In general, producing an "evaluation function" is a nontrivial task for it requires expertise of both the application at hand and the policies regulating resource sharing within the physical infrastructure [8].

Our key objective in building *Nefeli* is to free users from the task of creating, maintaining and tuning the operation of *VM*s in such environments. *Nefeli* offers a set of predefined utility functions that help users express task-flows in their workloads. These utility functions communicate to *Nefeli* possible favorable virtual resource layouts for the user's task-flows. In response to user requests, changing workloads, and events monitored within the cloud, *Nefeli* produces *VM*s to physical node mappings that are near-optimal for the execution

This work has been partially supported by the *D4Science I & II* EU FP7 projects.

of admitted task-flows. *Nefeli* deploys these mappings within the cloud by issuing the appropriate *VM* migration calls to the underlying cloud middleware. In this entire process, users have no direct contact with the physical infrastructure.

We have created a detailed *Nefeli* prototype and experimented with both simulated and real cloud environments. Our approach consistently displays significant performance improvements when compared with a variety of *VM* scheduling policies. In video transcoding, *Nefeli* achieves 17% reduction in processing times while in scientific task-flows *Nefeli* functioning above a simulated cloud demonstrates up to sixteen times higher throughput than other *VM* scheduling policies. Significant savings in terms of power consumption are reported as well. The rest of this paper is organized as follows: Section II states the problem we address and Section III outlines earlier related work. Sections IV–VII present in detail all the architectural elements of *Nefeli*. Section VIII discusses our experimental findings and finally, Section IX offers concluding remarks.

II. Managing *IaaS*–Cloud Virtual Resources

IaaS-clouds provide for their users a separation of concerns at the level of hardware as their respective services are confined in 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 in order to influence the performance of the infrastructure by indicating how *VM*s are to be actually deployed on the physical resources [3]. On the other hand, the clouds or service providers undertake all administrative actions on physical computing nodes including setting the policy with which consumer requests are to be handled.

Our conjecture is that 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 more effective resource sharing. As the cloud-contract does "prevent" the physical substrate from revealing most of its organizational features, user preferences and desired operational conditions can be mainly routed from the *IaaS* consumer to the provider. Perhaps, the most critical parameter that users have to alert the cloud about is the nature of the task-flows submitted. In this paper, we take the view that consumers may communicate this information in the form of hints. The latter could be used while trying to appropriately deploy *VM*s. For instance, should a user request *VM*s with the intention of deploying mirrors of a database, this would be of much importance. *VM* mirrors should be placed on different physical nodes so as the system can successfully handle failures. In similar spirit, *VM*s that are to perform parallel jobs –very much in the *MapReduce* fashion– should be also spread across different nodes [1]. Of course, this deployment pattern is not the only one that users may ask for. Favoring specific *VM*s or co-deploying others may also result into enhanced performance.

Apart from deployment decisions, *IaaS* consumers have also 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. It would be desirable that the actual placement of *VM*s to nodes changes to better address the needs of changing workloads. For instance in a video-encoding application, it might seem beneficial to use a highly distributed setup for *VM*s across various physical nodes in order to harness as many CPU-cycles as possible. Occasionally however, the aforementioned layout might generate significant network traffic calling for opportunistic collocation of *VM*s. It would be therefore necessary that the cloud should undertake actions to dynamically redeploy *VM*s to better serve workloads whose nature may continually change. Overall, the challenge *IaaS*-clouds face is how to permit more sophisticated interaction with the users while keeping the latter agnostic of the cloud internals. Contemporary clouds allow for transparent operations at the expense of depriving users from the option of using key virtualization features. By accepting user hints, *Nefeli* plays a major role in helping attain favorable *VM* deployments. The user remains agnostic of the clouds internals as any piece of his information arriving at the cloud gateway strictly refers to the type of the workload(s) the virtual infrastructure is to serve.

III. Related Work

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 [11]. *Nefeli* offers the means to state the existence of such patterns as task-flows and exploits these patterns to attain better *VM* deployment. We have chosen the term task-flow to illustrate distinct computational phases that take place on virtualized machines.

The allocation of resources in distributed environments requires adaptive policies. In [12], [13], such resource sharing policies are proposed for the execution of jobs on the GRID. Utility functions [13] are proposed to help quantify the efficient execution of jobs in light of different resource sharing disciplines. *GRID*-jobs are frequently form large *DAG*s and are often split before they are dispatched for execution. This flow splitting allows for re-adjustment in the management of resources. *Nefeli* also provides dynamic management of (virtual) resources but operates in a much different way, through migration of *VM*s.

In many respects, *Nefeli* realizes a number of features envisaged by autonomic computing [14]. Autonomic systems

[1]Cloud providers such as Amazon [4] allow users to ask for *VM*s deployed on different sites. Yet, such ad-hoc engineering solutions cover only portion of the needs of a user and even worse, they do disclose information on the cloud's internal structure.

attempt to self-adjust according to the needs of the applications they process. Here, 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 level in which an SLA is satisfied is quantified through user-furnished utility functions [9], [10]. Although, the stated objective of SLAs is to make applications agnostic of the system they are running 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 [8]. In contrast, *Nefeli* uses predefined utility functions that correspond to known attributes and patterns of all task-flows under execution. The user chooses from among the predefined utility functions and uses them to indicate possible points of contention in the infrastructure. This simplifies the user's responsibilities as *1)* the user need not create his own utility functions from scratch and *2)* the user remains unaware of the cloud internals.

In pure virtualized environments Van et al. [6] and Wang et al. [7] examine the use of SLA utility functions for workload execution. Each function provides feedback to a global, system wide optimization mechanism that decides on *VM* deployment policies. With *Nefeli*, we target private clouds on which we process all workloads as if they were served by a single virtual infrastructure featuring multiple task-flows. To this end, our approach uses a single optimization function in which all deployment preferences are accounted for.

Compared to other existing scheduling *VM*–based load-balancing systems [15], [16], [17], *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 used by our approach to trigger *VM* rearrangements is not bound solely on specific usage thresholds of resources. Instead, *VM*-migration is the outcome of external and/or internal perturbations in our infrastructure. In this regard, we support both resource depletion thresholds but also any monitoring mechanism that users may desire.

Finally, data stream processing systems [18], [19], [20] aim to produce the most efficient placement of operators in the network for processing of data flowing from data sources to interested data consumers. These systems assume a distributed environment where data sources and data consumers are widely distributed (geographically or across the Internet) and task-flows can overlap in terms of operators used and data streams processed. *Nefeli* aims to support virtual machine placement in a cloud computing environment where data sources are locally stored within the cloud. Moreover, task-flows from different users are assumed to be independent of each other and hence the instantiation of a *VM* for one user's task-flow and the specific actions performed by that *VM* are disjoint from other *VM*s invoked for the task-flows of other users.

IV. Overview of Nefeli

Nefeli adds a layer between the user and the infrastructure providing *IaaS*-cloud services, shown in Figure 1. *Nefeli* must interface with the lower level cloud services that handle the *VM* lifecycle and perform fundamental administrative tasks. This interface, denoted as 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*

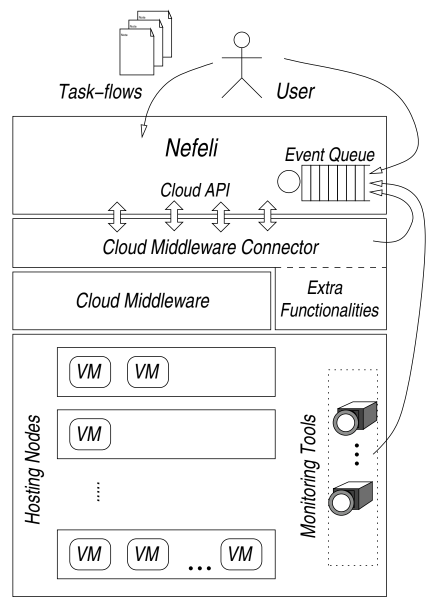

Fig. 1. *Nefeli*'s structured layout and interaction model

needs to obtain the following information:

- *Physical node properties:* these properties include free memory, total memory, CPU utilization, the name/ID of each hosting node and the amount of free disk space.
- *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 hosting nodes; *VM* properties include the memory usage and the disk space reserved in each virtual machine. Also the *IP*-address of each *VM* should be provided by the cloud *API* so as to be forwarded to the user as a *VM* access reference point.

VM deployment operations are handled through the cloud *API* of Figure 1 and include:

- Spawn a new *VM*.
- Shutdown a *VM*.
- Migrate a *VM*. For this operation, the names/IDs of the hosting nodes are needed.

While part of the interaction *Nefeli* has with the physical infrastructure can be provided by a cloud middleware[3], [5], [21], there are cases where additional functionality is need.

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 (Figure 1).

Nefeli has the role of an *IaaS*-cloud gateway. Users contacting *Nefeli* request virtual infrastructures created by *instantiating sets of VMs*. Two sample graphs of task-flows executed in such an infrastructure are displayed in Figure 2. Here, each

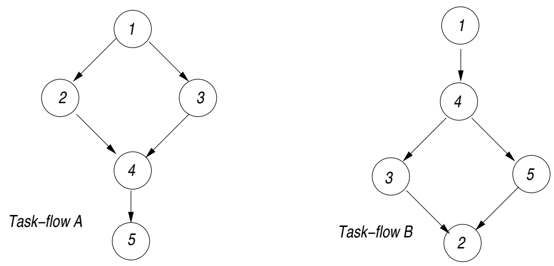

Fig. 2. Task-flow sample graphs

node represents a single *VM* while edges indicate control and data flows. The *VM* specifications are accompanied with 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 also 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. In the *task-flow A* of Figure 2 some deployment hints might be that a) *VM*s 1 and 2 would preferably be deployed on different hosting nodes and b) *VM* 4 should be favored by deploying it in a host without any other *VM*s. The latter constraint points out a possible CPU performance bottleneck of the task-flow at hand. Table I presents those constraints that we frequently found applicable in most task-flows tested with *Nefeli*. Additional, cloud specific, constraints may allow for enhanced Internet connectivity or give data locality hints.

TABLE I
COMMONLY USED CONSTRAINTS SUPPORTED BY *Nefeli*.

`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 physical network connections.
`ParVMs`	Try to deploy a set of *VM*s in separate physical nodes so as not to compete over the same resources
`PowerSave`	Reduce the number of hosting nodes used for *VM* deployment
`EmptyNode`	Offload a specific physical node

A single *XML* document can be used to contain all user-provided information. Figure 3 presents all aspects related to *task-flow A*. In the first section, the *VM* specifications are provided. 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 this 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 their weights is questioned. We trust the user has knowledge of the performance bottlenecks in his task flows. In what follows, we

```
<Task-flow>
  <Virtual Machines>
    <VM><ID>1</ID> <RAM>512</RAM> <Disk>VM1.img</Disk> </VM>
    <VM> <ID>2</ID> <RAM>512</RAM> <Disk>VM2.img</Disk> </VM>
    <VM> <ID>3</ID> <RAM>512</RAM> <Disk>VM3.img</Disk> </VM>
    <VM> <ID>4</ID> <RAM>512</RAM> <Disk>VM4.img</Disk> </VM>
    <VM> <ID>5</ID> <RAM>512</RAM> <Disk>VM5.img</Disk> </VM>
  </Virtual Machines>
  <Constraints>
    <ParVMs> <ID>1</ID> <VMID>2</VMID> <VMID>3</VMID> </ParVMs>
    <FavorVM> <ID>2</ID> <VMID>4</VMID> </FavorVM>
  </Constraints>
  <Profiles>
    <Profile> <ID>1</ID>
      <Weights>
        <ConstrID>1</ConstrID> <W>0.4</W>
        <ConstrID>2</ConstrID> <W>0.3</W>
      </Weights>
    </Profile>
  </Profiles>
</Task-flow>
```

Fig. 3. *Nefeli* input derived from sample *task-flow A*

discuss how *Nefeli* handles this type of single task-flow input and then we look at how our approach offers simultaneous execution of multiple task-flows in more than one virtual infrastructures, running on the same physical nodes.

V. SINGLE TASK-FLOW EXECUTION

Figure 4 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 uses information gathered not only from the user hints but also from the cloud administrator and the physical infrastructure. Combining the user-provided constraints with the *VM* specifications, as described in the *XML*-document of Figure 3, results in deployment patterns. These patterns are the outcome of examining user preferences alone and our approach uses them to match *VM* requirements to physical node resources. This matching process creates the actual deployment profiles once preferences from the cloud administration have been taken into account.

A. Constraints

Constraints express user and administration preferences. A constraint is realized as a utility function $F : M_{all} \mapsto [0, 1]$

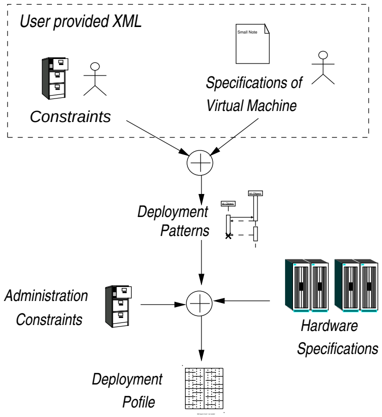

Fig. 4. *Nefeli*'s operational model

that evaluates a single deployment profile. In the context of *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 `FavorVM` is presented by Algorithm 1. Here, we measure the success of a profile in favoring a specific *VM*. From the perspective of the physical infrastructure, favoring a *VM* means that it shares the same hosting node with *as few as* possible *VM*s. To evaluate the success of a deployment profile in satisfying this constraint, we first count all *VM*s co-located with the one we favor and then we use the total number of *VM*s to normalize the result. We assume that all resources are equally shared among co-located *VM*s. Given that all functions are *Nefeli*-provided, we ensure that such assumptions are consistent across the board for all constraints. If and when one or more assumptions regarding resources becomes inaccurate [2] then, we will have to provide new implementations for affected utility functions. The latter are expected to work in a plug-and-play fashion. Clearly, all utility functions of Table I are realized in similar fashion but for brevity we omit their detailed discussion here.

Algorithm 1 `FavorVM` Utility Function

Input: `VM_ID`: ID of the *VM* to favor

V: Set of virtual machines

$M()$: Deployment profile function

Output: Satisfaction degree of "Favor Virtual Machine" constraint

Begin

```
host_ID := M(VM_ID)
collocated := 0
all_VMs := 0
for all v ∈ V do
  if (host_ID = M( ID of v )) AND (ID of v != VM_ID) then
    collocated++
  end if
  all_VMs++
end for
return (all_VMs - collocated)/all_VMs
```

End

[2]perhaps due to major changes in the infrastructure

B. 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. In the example of Figure 3 where the two constraints `ParVMs` and `FavorVM` with weights 0.3 and 0.4 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 deployment profiles.

Finding optimal deployment profiles is *NP*-hard so we employ simulated annealing [22] to attain plausible approximations[3]. 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 usually longer paths between successively visited neighbors. 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: $e^{D/T} >$ 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, we may reach a different near-optimal solution.

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.

[3]For relatively small infrastructures, simulated annealing performs well. However, in large infrastructures other, more efficient, methods should be considered.

Algorithm 2 Simulated-Annealing-based 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

```
same = 0
best_profile = current_profile = GetRandomProfile()
while same <same_iterations do
    new_profile = GetNeightborOf(current_profile)
    D = Score(new_profile) - Score(current_profile)
    if ( T > 10^-5 AND e^(D/T) > Random()) OR
          (T < 10^-5 AND D > 0) then
        current_profile = new_profile
    end if
    if Score(new_profile) > Score(best_profile) then
        best_profile = new_profile
        same = 0
    end if
    same++
    T = 0.99 * T
end while
return best_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.

VI. Handling Increasing Numbers of 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 appears 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.

Figure 5 shows the *XML*-description for the second task-flow of Figure 2. For this task-flow there are two constraints: *a)* nodes 1 and 4 would better be co-located since they will be producing too much network traffic and *b)* nodes 3 and 5 are to be deployed on different hosting nodes. *VM* IDs are system-wide identifiers thus both task-flows of Figure 2 make use of the same *VM*s; the two graphs describe different flows within the same virtual infrastructure. *Nefeli* never reveals the set of *VM* identifiers to users. Collaborative environments where users share *VM*s, thus *VM* IDs, in producing task-flows have to be realized by frameworks in a higher level.

```
<Task-flow>
  <Virtual Machines>
    <VM><ID>1</ID> <RAM>512</RAM> <Disk>VM1.img</Disk> </VM>
    <VM> <ID>2</ID> <RAM>512</RAM> <Disk>VM2.img</Disk> </VM>
    <VM> <ID>3</ID> <RAM>512</RAM> <Disk>VM3.img</Disk> </VM>
    <VM> <ID>4</ID> <RAM>512</RAM> <Disk>VM4.img</Disk> </VM>
    <VM> <ID>5</ID> <RAM>512</RAM> <Disk>VM5.img</Disk> </VM>
  </Virtual Machines>
  <Constraints>
    <MinTraf> <ID>1</ID> <VMID>1</VMID> <VMID>4</VMID> </MinTraf>
    <ParVMs> <ID>2</ID> <VMID>3</VMID><VMID>5</VMID> </ParVMs>
  </Constraints>
  <Profiles>
    <Profile> <ID>1</ID>
      <Weights>
        <ConstrID>1</ConstrID> <W>0.4</W>
        <ConstrID>2</ConstrID> <W>0.3</W>
      </Weights>
    </Profile>
  </Profiles>
</Task-flow>
```

Fig. 5. *Nefeli* input derived from sample *task-flow B*

Producing a deployment profile for both task-flows of Figure 2, is done by combining the respective descriptions of Figures 3 and 5. In this case, the set of *VM*s is the same in both descriptions but 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. However, such policies are out of the scope of this paper. We expect them to be enforced at a higher level also. The score function for a deployment profile m becomes:

$$\begin{aligned} \textit{Score}(m) \quad &= \quad 0.4 * \texttt{ParVMs}_1(m) + 0.3 * \texttt{FavorVM}(m) + \\ &+ \quad 0.4 * \texttt{MinTraf}(m) + 0.3 * \texttt{ParVMs}_2(m) \end{aligned}$$

where $\texttt{ParVMs}_1$ and $\texttt{ParVMs}_2$ 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 a live migration feature result in some down time of the virtual infrastructures. In this case, *VM*s have to be suspended and copied to other computing nodes where they can resume their normal operation. To tackle such inefficiency, the profile creation procedure may trade profile quality for swifter transitions. 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 $M_1, M_2 \in M_{all}$ is:

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

Given an initial deployment profile m_s, to reduce *VM* migration overheads *Nefeli* first produces k high scoring profiles and then picks the one whose transition from m_s requires migrating fewer *VM*s. Let M_b be the set of the k-top scoring profiles produced, the one (m_q) that will be used is:

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

With k regulating the tradeoff between migration overhead and profile quality, we are able to express the virtual infrastructures' sensitivity to downtimes.

VII. Nefeli's Components and Application Interaction

In Figure 6, we present the environment in which the main components of *Nefeli* operate. The *Deployer* is the component that keeps track of all active task-flows. This component updates a list of all submitted task-flow descriptions upon arrival of external events. Event arrival also causes the interaction between *Deployer* and *Planner*. The *Deployer* provides a list of task-flows, constraints and the respective weights, and the *Planner* produces a single deployment profile as if it were a single task-flow. The profile produced is applied using calls to the cloud *API*.

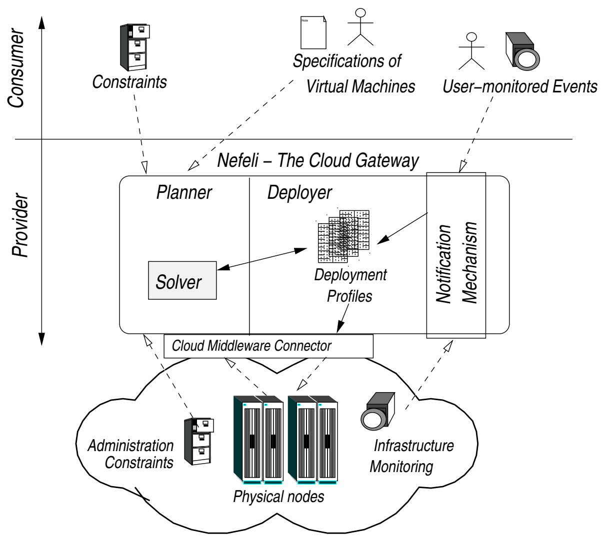

Fig. 6. The environment *Nefeli* operates in.

The event-based mechanism that triggers the *Deployer*'s operation is not limited to events caused by new task-flow submission. This user generated event is not the only one required to handle task-flows, *Nefeli* must also be informed of a task-flow ending. Task-flow termination events come from either the user or some component of the task-flow itself.

The overall goal of *Nefeli* is to make choices regarding the deployment profile based on the user's needs and the system's performance. To this end, we have extended the capabilities of the notification mechanism so that *Nefeli* receives any kind of signal that will assist in achieving its goal. We group events into two classes according to their origin:

• *Events activated by direct human intervention:* events of this class include the submission or removal of any number of task-flows served. This class also includes events that help users gain full control over the cloud's and *Nefeli*'s operation. Consider for example node maintenance tasks that require specific parts of the hardware infrastructure to be shutdown. 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 the cluster, the virtual infrastructure or an authorized third party component:* typically, *VM* redeployment takes 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 like network traffic are monitored through third party monitoring tools (Figure 1) that have to be installed within the infrastructure. Receiving this type of event 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 stay 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 not only from the physical infrastructure but from the virtual as well. 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 them agnostic of the infrastructure on which they are being executed.

A. Applications-Driven Operation

Users contacting an *IaaS*-cloud in search of a virtual infrastructure are well aware of the task-flows comprising the workload they need to serve. In such cases, *Nefeli* allows users to have control over which specific task-flows are to be favored.

Figure 7 presents the description of a workload that furnishes more than one task-flow within the same infrastructure. As in the case of a single task-flow submission the `Virtual Machines` section describes all *VM*s of the virtual infrastructure. The `Constraints` section describes all constraints regardless of the task-flow they are referring to. What differentiates task-flows is the weight values with which constraints are accounted for during deployment profile production. Note here that if the user wants to optimize the *VM* deployment for simultaneous execution of more than one task-flow she must consider the combination of the respective task-flows in a new profile and thus provide a combined set of weights. In the `Profiles` section the number of weight sets provided have to be equal to the number of deployment profiles to be produced. In the workload of Figure 7, there are two *VM*s with IDs 1 and 2. Each binds to a `FavorVM` constraint with IDs 1 and 2. The user's intention is to have two deployment profiles each one with the respective constraint active. Therefore, there are two deployment profiles in the `Profiles` section, each one assigning 0.9 weight to the constraint to be active and 0.0 to the other one.

The next two sections of the input *XML* of Figure 7 refer to the transition between the deployment profiles. The `Events`

```
<Multiple Task-flows>
  <Virtual Machines>
    <VM>
      <ID>1</ID> <RAM>512</RAM> <Disk>VM1.img</Disk>
    </VM>
    <VM>
      <ID>2</ID> <RAM>512</RAM> <Disk>VM2.img</Disk>
    </VM>
  </Virtual Machines>
  <Constraints>
    <FavorVM> <ID>1</ID> <VMID>1</VMID> </FavorVM>
    <FavorVM> <ID>2</ID> <VMID>2</VMID> </FavorVM>
  </Constraints>
  <Profiles>
    <Profile> <ID>1</ID>
      <Weights>
        <ConstrID>1</ConstrID> <W>0.9</W>
        <ConstrID>2</ConstrID> <W>0.0</W>
      </Weights>
    </Profile>
    <Profile> <ID>2</ID>
      <Weights>
        <ConstrID>1</ConstrID> <W>0.0</W>
        <ConstrID>2</ConstrID> <W>0.9</W>
      </Weights>
    </Profile>
  </Profiles>
  <Events>
    <Time>
      <ID>1</ID> <Period>1000</Period>
    </Time>
    <Net>
      <ID>2</ID> <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>
</Multiple Task-flows>
```

Fig. 7. *Nefeli* input example.

section points out events whose signal will cause a change in the deployment followed. *Nefeli* can be extended with user-provided, application-specific events since their implementation does not require exposing any information about cloud internals. Currently *Nefeli* provides two event types (of the second category of the previews subsection), also used during our evaluation:

- A time-based event that periodically sends a signal.
- A network-based event that starts a server listening for a predefined message to arrive.

Both event types are demonstrated in Figure 7. Here, the first event, with ID 1, will be triggered every 1000 seconds as defined within the `Time` tag. The second event (`Net` tag) will have *Nefeli* listen for messages coming in to port 2324. If the message received is the string `Change`, the event will be triggered.

All events can be used in boolean expressions formed using *AND*, *OR* and *NOT* aggregation events. Figure 8 presents one such expression. The event with ID 6 is triggered when the expression: *A OR (B AND (NOT C))* evaluates to true. A, B and C are events with IDs 1, 2 and 3 respectively. The boolean event operators *OR, AND* and *NOT* make use of other event IDs to point to their operands.

Events cause transitions between deployment profiles. These

```
<Events>
  <Event_A> <ID>1</ID> ... </Event_A>
  <Event_B> <ID>2</ID> ... </Event_B>
  <Event_C> <ID>3</ID> ... </Event_C>
  <NOT> <ID>4</ID> <Invert_ID>3</Invert_ID> </NOT>
  <AND> <ID>5</ID> <EventID>2</EventID> <EventID>4</EventID> </AND>
  <OR> <ID>6</ID> <EventID>1</EventID> <EventID>5</EventID> </OR>
</Events>
```

Fig. 8. Combining events into boolean expressions.

transitions are described in the final section of the input *XML* description. In the example of Figure 7, the event with ID 1 will cause a transition from the deployment profile 1 to profile 2, while activation of the event 2 will have the opposite effect. The outcome of this "Multiple 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 *VM* favoring loop will continue as long as the virtual infrastructure remains on-line.

To serve a workload request indicating execution of multiple task-flows *Nefeli* starts with the creation of the first profile in the respective description document section. As soon this profile is applied, the *Deployer* of *Nefeli* goes over the profile transitions and registers for receiving those events that will cause a transition to a next profile. Upon receiving such an event, the *Deployer* produces and applies the transition target profile. Immediately after that, it unregisters the old events and registers any new events listed. This profile swapping continues until the task-flows are removed. Of course, serving workloads with multiple task-flows does not prevent *Nefeli* from serving separate task-flows at the same time. The key difference in this kind of task-flow description is that each time only one profile from the `Profiles` section is used for the creation of new deployment profiles.

VIII. Evaluation

We have implemented *Nefeli* as a Java library to more readily have it embedded in cloud management systems. Our experimental evaluation that involves at all times our *Nefeli* prototype has a number of key objectives which are to:

- Examine the efficiency of our system as compared to existing job scheduling alternatives as far as CPU utilization and throughput rates achieved for diverse sets of task-flows.
- Investigate the behavior of *Nefeli* as the number and features of virtual resources available for processing change over time.
- Evaluate the overheads involved in deploying and using *Nefeli* for interacting with cloud computing systems.

Our evaluation process includes experimentation: with **a)** diverse scientific task-flows executed on simulated infrastructures and **b)** applications executed in the *Iaas*-cloud environment of our laboratory. The difference between simulation and real application evaluation is in the infrastructure used – we have done this mainly to maximize flexibility during our evaluation. We have implemented two cloud middleware connectors: the first simulates an infrastructure and the second interacts with *OpenNebula* [3] through *XML-RPC*. At this time, *OpenNebula* along with *Eucalyptus* [5] and *Nimbus* [21] are all key open-source *IaaS*-cloud middleware projects with similar if not identical objectives. In what follows, we first examine scalability and performance issues using the simulated infrastructure and subsequently present the gains of using *Nefeli* in a real application.

A. 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 [4] and consume CPU-cycles to transform input data into output. The amount of available cycles per second to be shared among hosted *VM*s allow us to designate the CPU performance rate. We set physical nodes to have 8 GB of RAM and virtual ones 512 MB. The behavior of each *VM* is designated by the *input-to-output* size ratio (input *KBytes*/output *KBytes*) and the CPU-cycles required to produce a single output unit (cycles / output *KBytes*). The *input-to-output* ratio quantifies how much of the input must be consumed in order to produce a single unit of the output data. Similarly, the *cycles-to-output* ratio indicates how many cycles have to be expended in order to produce a single unit of output (i.e., *Byte*). Output bytes are forwarded to other virtual machines consuming network bandwidth. Using the above abstraction, a task-flow creation requires the following: firstly, setting up characteristics of each *VM* with both *input-to-output* and *cycles-to-output* rates and secondly, defining the network connections designated by the data paths of the specific task-flow(s) at hand.

The workload we present here is highly influenced by the *Montage*-engine that generates sky mosaics [23]. This engine is used in scientific applications as the wrapper to access telescope data. It is used by several groups of astronomers including those in *NASA* for processing images taken from space. During operation, numerous input images are scaled, rotated and filtered using specialized algorithms so that the images produced correctly map sky-areas. This application has been split into well defined tasks so that it can be conventionally executed on cluster environments. Figure 9 depicts processing nodes and the network topology used by this task-flow. The *input-to-output* and *cycles-to-output* ratios for all *VM*s, as extracted in [24], are shown in Table II. The way nodes have been "networked" as well as the data flow ratios of Table II readily point out potential bottlenecks of this task-flow and thus, we can easily formulate a number of user hints to be passed to *Nefeli*. For example, the *VM*s in sets $\{1, 2, 3, 4\}$, $\{5, 6, 7, 8, 9, 10\}$ and $\{13, 14, 15, 16\}$ have to be deployed in different hosting nodes as they operate *in parallel*, while nodes $\{17, 18, 19, 20\}$ are better placed on the same hosting node since they operate in *sequence* and may consume considerable network bandwidth (as data-flows in Table II indicate). Table III presents our choice for "relaxed"

[4] *VM* deployment may fail due to resource shortage.

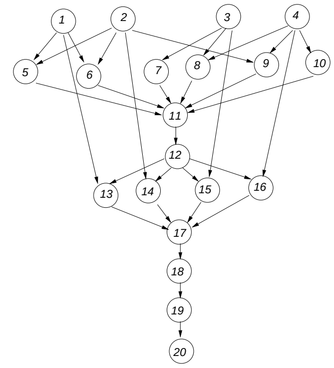

Fig. 9. *Montage*-like task-flow

TABLE II
VM CHARACTERISTICS FOR THE *Montage*-LIKE TASK-FLOW.

VM IDs	Input-to-Output	Cycles-to-Output
$\{1, 2, 3, 4\}$	0.5070	0.5830
$\{5, 6, 7, 8, 9, 10\}$	29.4070	0.0500
11	1.5000	0.0014
12	30.0000	9.1910
$\{13, 14, 15, 16\}$	1.0010	0.7390
17	0.0200	33.4210
18	1.0790	10.9050
19	25.0650	0.5370
20	20.6250	0.0290

weights for each of the constraints used in order to generate the deployment profiles. The weights in question highlight the likely presence of bottlenecks and offer *Nefeli flexibility* while placing various *VM*s in different computing nodes. Table III depicts two sets of constraints: the first is mostly concerned with throughput attained by the infrastructure and is simply termed *Nefeli*, while the second includes the restriction that the lowest number possible of hosting nodes should be active. The latter essentially has to do with the consumption of power, often of very high-concern in computing installations, and so this set is termed *Nefeli-power*. On a *Core(TM)2 Duo CPU T7100 at 1.80GHz* producing a deployment profile for any of the two *Nefeli* configuration adds an overhead of less than 6.9 *Seconds*.

The simulated environment allows us to perform two types of experiments that would have been otherwise impossible to perform using a physical infrastructure. Here, we can: *a)* selectively "increase" the CPU performance, *b)* offer additional hosting nodes. In all cases, we measure the throughput of the entire flow by measuring the outcome of the trailing node; in

TABLE III
TWO SETS OF USER WEIGHTED CONSTRAINTS FOR *Montage*

Constraints	*Nefeli*	*Nefeli-power*
`ParVMs` on *VM*s $\{1, 2, 3, 4\}$	0.30	0.30
`ParVMs` on *VM*s $\{5, 6, 7, 8, 9, 10\}$	0.30	0.30
`ParVMs` on *VM*s $\{13, 14, 15, 16\}$	0.30	0.30
`MinTraf` on *VM*s $\{17, 18, 19, 20\}$	0.50	0.50
`PowerSave`	0.0	0.80

our *Montage*-like task-flow this is the *VM* with *ID 20*. Our two configurations –*Nefeli* and *Nefeli-power*– are compared against our own implementations of the following scheduling policies:

- *Power Saving:* in instantiating *VM*s exclusively use the clause that the number of active hosting nodes be the smallest possible.
- *Random:* schedule *VM*s randomly. The discipline is straightforward to implement and bears minimal overheads.
- *Balance VMs:* attempt to distributed *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 200 times. Figure 10 depicts the performance gains obtained while using the configuration *Nefeli* of Table III. The *Random* and *Balance VMs* schedulers demonstrate approximately the same performance. As the *Balance VMs* iterates over the hosting nodes for placing newly instantiated *VM*s, it would appear that it should be more beneficial than its *Random* counterpart. However, it is not. The reason for this is that it does not discriminate between which *VM*s to be deployed on the hosts. *VM*s are chosen randomly, though evenly distributed to hosts. The *Power Saving* produces less network traffic than *Balance VMs* and *Random* thus it is favored by high performance CPUs. *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. Our approach achieves a factor of 2 to 16 times in throughput increase.

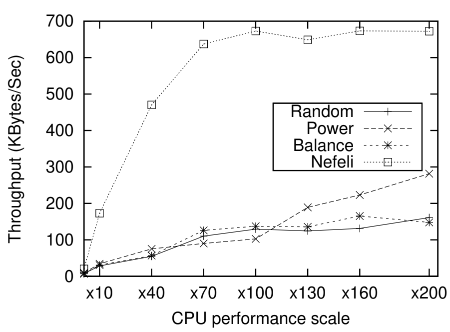

Fig. 10. *Montage* throughput under increasing CPU performance

Provide additional physical nodes: increasing the number of physical nodes results in having *a)* more CPU-cycles available, *b)* increased overall network bandwidth capacity and *c)* increased power consumption. We start with 3 hosting nodes and we gradually reach an infrastructure consisting of 10 nodes. In Figure 11, we present the mean throughput delivered by the *Montage* workload when the three schedulers and the two *Nefeli* configurations are used. Even in an infrastructure with very few nodes the performance gains achieved with *Nefeli* are noteworthy. As the number of nodes increases, all schedulers except the *Power Saving* display minor performance improvements. The *Power Saving* scheduler always

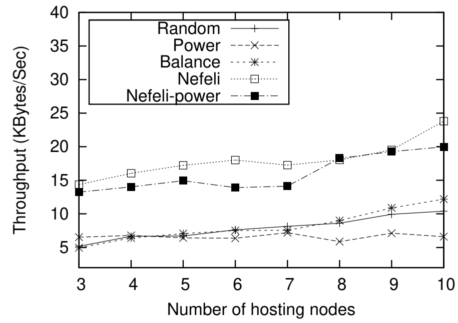

Fig. 11. *Montage*-throughput under increasing number of hosting nodes

uses a fixed number of hosting nodes, in our case two, thus it displays no improvement.

Power saving schedulers are popular among cloud operators for they reduce the maintenance cost of the physical plant. 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, Figure 12 shows the task-flow throughput achieved normalized by the number of active nodes. This average throughput rate per active host captures the power efficiency of the physical substrate. Lower values indicate that nodes have to remain on-line longer periods for producing the same amount of output. The *Power Saving* scheduler always uses exactly two nodes to deploy all *VM*s and therefore, it remains largely unaffected by the addition of extra nodes; it provides an average of 3.1 *KBytes* per active node. *Random*, *Balance VMs* and *Nefeli* use as many hosting nodes as possible. The trend displayed by those three policies is a decrease in the average throughput as nodes are added. As Figure 12 shows, *Nefeli-power* offers an improvement over *Nefeli* in terms of average throughput achieved per active node. *Nefeli-power* also outperforms the *Power Saving* scheduler in settings with few hosting nodes, while both policies perform equally well for more than six nodes. The reason for this is that *Nefeli-power* "settles" for solutions that use more nodes in order to satisfy the `ParVM` deployment constraints. *Nefeli-power* presents a compromise between the high throughput rates achieved by *Nefeli* and the number of active nodes. Figure 11, combined with Figure 12, completes our view of the power saving constraint used as it depicts the impact of compromising between overall performance and number of active nodes.

The outcome of experimenting with simulated cloud environments points out the potential of our approach. While outlining likely bottlenecks using "hints" or constraints *Nefeli* may drastically enhance the overall performance of the equipment used. Moreover, the user does not violate in any way the cloud contract as she cannot directly influence the inner working of what is used beneath the cloud middleware. In

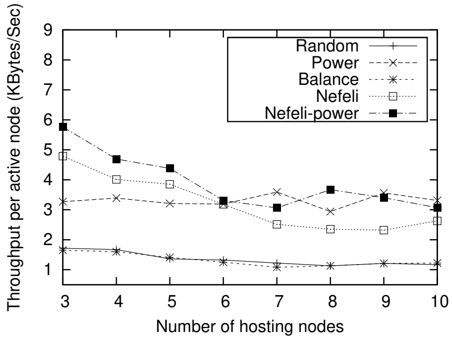

Fig. 12. Average task-flow throughput per active node when increasing hosting nodes

the next subsection, we outline our evaluation using a private-cloud environment running *Nefeli* and show the gains obtained while competing with the scheduler of the open-source cloud middleware used [3].

B. *Nefeli in a Private Cloud Environment*

We have created a cloud-enabled application that uses *Nefeli* to perform video and audio transcoding. Such applications are very well suited for cloud execution as many *VM*s can be simultaneously utilized, each one operating on a separate fragment of the input media. In addition, the elongated processing time ameliorates the *VM* scheduling and deployment delays. In our implementation of the application, the output file format may be any of the following: DVD, SVCD or VCD. For each of those formats the user is allowed to set an extra video encoding property regarding regional settings (i.e., if the encoding is going to produce video according to either PAL or NTSC standards). Also, we first transcode video and then we separately work with audio. As soon as the output format (DVD, SVCD or VCD) and the regional settings (PAL or NTSC) sought are defined, the transformation commences. A four step procedure is followed: *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 this setting). *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 transcoded video.

All valid combinations of the three output formats and the two regional settings for audio and video transformation yield 12 distinct tasks. Each of these tasks is carried out by 3 *VM*s so that the input file split into three parts and processed in parallel. In our virtual infrastructure, there is a total of 6 *VM*s deployed on 3 physical nodes. Table IV presents the transformational tasks and the corresponding *VM*s capable of serving them. The mapping of transformational tasks-to-*VM*s is fixed during the the application's operation whereas the *VM*-to-host mapping is performed dynamically by *Nefeli*. In an optimal *VM* deployment virtual machines of video and audio transcoding tasks are distributed among different hosting nodes so that they do not compete for CPU cycles. The lines of Table IV show all allowed transcoding operations each one requiring its own optimal deployment profile.

TABLE IV
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

The constraints expressed to *Nefeli* for each of the profiles is to have those *VM*s operating simultaneously –processing different parts of the input file– deployed in separate physical nodes since they will be consuming considerable CPU resources. For example in the optimal mapping of DVD/PAL, *VM*s *1, 2, 3* are distributed among different nodes as they perform the video transformations. Along these lines, *VM*s *1, 4, 5* have also to be placed in different nodes.

When a user commences the procedure of transcoding, the application issues its constraints so that *Nefeli* finds proper deployment profiles. The communication between the application and *Nefeli* is based on the events mechanism of the *Deployer*. We assign one event for each of the output formats (DVD, VCD, SVCD) and one for the region property (PAL or NTSC). These four events are triggered by sending a signal to ports *Nefeli* listens to. Format and region related events are combined into boolean expressions before invoking the *Deployer* to take appropriate action. For instance, requesting a DVD/PAL transformation will trigger events "DVD" and "PAL".

The transcoding application is setup on a rack whose three dedicated nodes work under the supervision of *Nefeli*. All systems are connected through a 1 $GBps$ Ethernet switch. Each physical node is equipped with 8 GB of RAM and a Intel(R) Core(TM)2 CPU 6600 at 2.40 GHz CPU. Live migration is not available and *VM*s images are fetched from a file server. The *VM* hypervisor we use is Xen 3.2-1 [25] and the cloud middleware is *OpenNebula v.1.2.0* [3]. *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. To achieve this the application splits the video and audio parts into even smaller parts that are processed in parallel by all CPU cores. To this end, we make use of the *Xen* VCPU-option in order to simultaneously utilize all cores available at the CPU.

Figure 13 presents the results of video encoding in our cloud infrastructure using either *Nefeli* or simply employing the default *OpenNebula VM* scheduler. *OpenNebula* uses a match making mechanism based on the free memory and CPU each hosting node shows and respective *VM* requirements. Each *VM* occupies a percentage of the hosting CPU and uses portion of its memory. The *OpenNebula* match-making approach proves ineffective for our application as *VM* CPU requirements are known during only runtime and not at

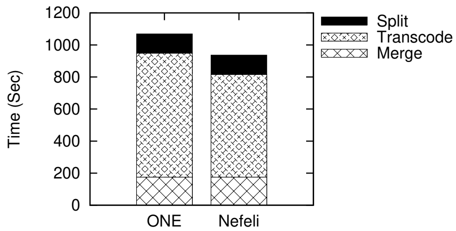

Fig. 13. Comparing *Nefeli* to *OpenNebula* in the three phases of the encoding.

deployment time. *Nefeli* achieves a 17% improvement on the time required to have video and audio transcoding complete. File splitting and merging displays no gains from an optimal deployment since both operations are performed in 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 if compared to *VM staging* operations; it only required less than 5 *Seconds* to complete.

IX. Conclusions - Future work

This paper presents *Nefeli*, a flexible gateway that allows for the effective use of virtual infrastructures. *Nefeli* accepts user "hints" regarding the nature of the workload under execution and exploits conditions that concern the physical infrastructure to better schedule and utilize instantiated *VM*s. These constraints essentially designate fundamental deployment patterns which should they be followed in the actual *VM* deployment, they could assist in avoiding potential bottlenecks. While experimenting with a prototype on both simulated and real private *Iaas*-cloud environments, we established significant gains for *Nefeli* both in terms of performance and power consumption. In offering a comprehensive *VM* management in *IaaS*-clouds, our approach displays a number of advantages: firstly, it considers the provision of virtual infrastructures as a whole and does not only deal with the handling of individual *VM*s. Secondly, users remain at all time unaware of the inner structure and/or architecture of the physical nodes. Thus, *Nefeli* does offer a true separation of concerns in *IaaS*-clouds. Lastly, our approach gracefully adapts to the changing workload needs as *VM* migration do occur in order to offset unfavorable deployments currently in place.

In the future, we plan to: *a)* integrate the event mechanism provided by *Nefeli* with other existing monitoring tools like Nagios[26] and create connectors for other cloud middlewares including *Eucalyptus* [5] and *Nimbus* [21], *b)* examine the use of alternative scheduling options and adopt rigorous approaches in selecting deployment profiles, *c)* investigate ways to organize and better manage virtual resources for applications that necessitate the use of massive data sets.

References

[1] M. T. Jones, "Cloud Computing with Linux," http://www.ibm.com/developerworks/linux/library/l-cloud-computing/index.html, February 2008.

[2] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[3] "Opennebula," http://www.opennebula.org, May 2009.

[4] Amazon, "Elastic Cloud," http://aws.amazon.com/ec2/, 2009.

[5] 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.

[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] 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, pp. 107–116.

[9] J. O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[10] 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.

[11] 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.

[12] 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.

[13] K. Lee, N. Paton, R. Sakellariou, and A. Fernandes, "Utility Driven Adaptive Workflow Execution," in *Proc. of the 2009 9th IEEE/ACM Int. Symposium on Cluster Computing and the Grid*, Shanghai, PR China, 2009, pp. 220–227.

[14] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE–Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[15] 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.

[16] C. Weng, M. Li, Z. Wang, and X. Lu, "Automatic Performance Tuning for the Virtualized Cluster System," Montreal, Quebec, Canada, 2009, pp. 183–190.

[17] VMware, "vSphere," http://www.vmware.com/products/vsphere/, Nov. 2009.

[18] 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.

[19] Y. Ahmad and U. Çetintemel, "Network-Aware Query Processing for Stream-based Applications," in *Proc. of VLDB'04*, Toronto, Canada, Aug. 2004.

[20] 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.

[21] "Nimbus," http://workspace.globus.org/, Nov. 2009.

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.4175

[23] "Montage: An Astronomical Image Mosaic Engine," California Institute of Technology, http://montage.ipac.caltech.edu, Pasadena, CA, 2009.

[24] 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.

[25] 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, pp. 164–177.

[26] D. Josephsen, *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ: Prentice Hall PTR, 2007.