NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM

MASTER THESIS

Application Cost-aware Cloud Provisioning

Alexandros Antoniadis

Supervisor: **Alex Delis**, Professor NKUA

ATHENS

SEPTEMBER 2013

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παροχή Εφαρμογών με Περιορισμούς Κόστους σε Υπολογιστικά Νέφη

Αλέξανδρος Αντωνιάδης

Επιβλέπων: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2013

MASTER THESIS

Application Cost-aware Cloud Provisioning

Alexandros Antoniadis

RN: 1097

SUPERVISOR:

Alex Delis, Professor NKUA

THESIS COMMITTEE:

Alex Delis, Professor NKUA
Mema Roussopoulos, Assistant Professor NKUA

September 2013

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παροχή Εφαρμογών με Περιορισμούς Κόστους σε Υπολογιστικά Νέφη

Αλέξανδρος Αντωνιάδης

ΑΜ: 1097

ΕΠΙΒΛΕΠΩΝ

Αλέξιος Δελής, Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Αλέξιος Δελής, Καθηγητής ΕΚΠΑ
Μέμα Ρουσσοπούλου, Επίκουρη Καθηγήτρια ΕΚΠΑ

Σεπτέμβριος 2013

Περίληψη

Οι πλατφόρμες νέφους επιτρέπουν στους ιδιοκτήτες εφαρμογών την ενοικίαση πόρων, προκειμένου να επεκτείνουν δυναμικά τη συνολική υπολογιστική ισχύ των υποδομών τους. Η ποικιλία των χαρακτηριστικών των πόρων αυτών καθώς και της τιμής μίσθωσης τους είναι συνήθως μεγάλη. Οι πάροχοι νέφους, επίσης, οφείλουν να διασφαλίζουν την ποιότητα της υπηρεσίας (Quality of Service) μέσω εγγυήσεων (Service Layer Agreements) και είναι υποχρεωμένοι να πληρώσουν ποινή κάθε φορά που μια τέτοια εγγύηση παραβιάζεται. Επιπλέον, οι περισσότερες από τις εφαρμογές που βασίζονται στο νέφος προσφέρουν και αυτές τέτοιου είδους εγγυήσεις στους χρήστες.

Σε ένα δυναμικό περιβάλλον, όπου ο χρήστης εκτελεί μια εφαρμογή στο ιδιωτικό νέφος και μπορεί να προσθέσει ή να αφαιρέσει κόμβους από έναν πάροχο νέφους (δημόσιο νέφος) 2 διαφορετικά είδη SLAs υπάρχουν (i) το SLA που προσφέρεται από την εφαρμογή στους τελικούς χρήστες και (ii) το SLA που προσφέρεται από τους παρόχους νέφους στην εφαρμογή. Έτσι, μια ποινή που καταβάλλεται για παραβίαση ενός SLA από την εφαρμογή στους τελικούς χρήστες μπορεί να είναι χαμηλότερη αν ενας πάροχος δημόσιου νέφους πληρώνει ποινή σε περίπτωση που το SLA αυτό παραβιάζεται επίσης. Αυτή η ιδιότητα καθιστά τον υπολογισμό του συνολικού κόστους λειτουργίας περίπλοκο αλλά επεκτείνει και το χώρο αναζήτησης των διαφορετικών επιλογών που μπορεί να έχουν χαμηλότερο συνολικό κόστος.

Σε αυτήν τη διπλωματική παρουσιάζουμε έναν αλγόριθμο παροχής πόρων για NoSQL εφαρμογές, που στοχεύει στην ελαχιστοποίηση του συνολικού κόστους μιας εφαρμογής νέφους λαμβάνοντας υπόψη τις ιδιότητες ελαστικότητας της εφαρμογής αυτής σε ένα ετερογενές περιβάλλον και είναι βασισμένος σε “look-ahead” βελτιστοποίηση.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ : παροχή πόρων, πλατφόρμες νέφους, NoSQL–βασεις δεδομένων, μοντέλο απόδοσης, ελαχιστοποίηση κόστους

Abstract

Cloud computing platforms allow application owners to rent resources in order to expand dynamically the overall computational power of their infrastructure. The variety of the resources and their lease price is usually big. Moreover, cloud providers ensure the high Quality of Service (QoS) through Service Layer Agreements (SLAs) and they are obligated to pay a penalty each time these agreements are violated. In addition, most of the cloud-based applications also offer an SLA to the users.

In a dynamic environment, where a user is running an application on her private cloud and may add or remove nodes from a cloud provider (public cloud), 2 different types of SLAs exist (i) the SLA offered by the application to the end users and (ii) the SLA offered by the cloud providers to the application. Thus, a penalty that is paid for an SLA violation from the application to the end users might be lower than the one stated, if a public cloud provider pays back a penalty in case its SLA is also violated. This property makes the calculation of the total operational cost complex, but also expands the search space of different choices that may have lower total cost.

In this thesis we present an application-cost aware resource provisioning algorithm for NoSQL applications that aims to minimize the total cost of a cloud application by taking into account the elasticity properties of that application in a heterogeneous environment and is based on look-ahead optimization.

SUBJECT AREA: Distributed Systems

KEYWORDS: resource provisioning, cloud platforms, NoSQL–databases, performance model, cost minimization

I dedicate this thesis to my nephew Orestis Antoniadis.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. Alex Delis for his continuous cooperation, constant guidance, encouragement and help through this thesis.

I would also like to thank assistant Prof. Mema Roussopoulos for her useful comments, suggestions and contribution to this thesis.

Contents

Figures

Tables

Algorithms

Chapter 1

Introduction

Cloud-Service Providers (*CSPs*) dynamically offer computational and storage resources so that users can experience timely execution of their applications regardless of the load and queued jobs the infrastructure has to handle [14, 24]. *CSPs* have the freedom to calibrate both type and number of allotted resources at different points in time so that incoming workloads are handled with success. In such settings, *QoS* guarantees regarding performance aspects such as response time, throughput, and service availability can be provided to user applications through the use of *SLAs*. When *SLA* violations occur, monetary penalties are accrued for the *CSP* directly affecting its revenue and more importantly, its reputation [21].

Untimely provisioning by a *CSP* of its own *internal* (or *private*) resources can lead to depressed leasing costs that ultimately prevent application *QoS*-requirements from being met. Moreover, applications demonstrate widely varying and occasionally unpredictable workloads that change over time [18]. Resources needed by an application might change either periodically (i.e., high peak hours or days) or irregularly (i.e., flash crowds that cause sudden, significant depletion of resources [22]). A *CSP* could address internal resource shortages by soliciting additional resources that are available just-in-time from external or *public CSPs*. Dynamic allocation/deallocation of cloud resources might help, but frequent workload changes may lead to deployment thrashing as overheads incurred by the additions/removals of resources may outweigh any short-term benefits gained. To make matters more complicated, pricing for leasing equivalent resources from *public CSPs* continuously fluctuates and must be taken into consideration to identify an allocation with minimum cost. As a result of all of the above factors, resource allocation is not a straightforward task and has thus attracted attention from the research community [6, 18, 19].

In this thesis, we investigate the problem of provisioning a popular class of cloud applications collectively known as *NoSQL*-databases [1, 2, 15]. Their key characteristic is that they can scale their performance as they offer horizontal partitioning of data in a shared-nothing fashion (e.g., sharding) [7]. This feature has propelled their use in databases in computational settings that require on-demand resource allocation including web-portals, big-data processing and *CRM*-systems [23, 25].

NoSQL-databases are typically designed to provide availability and fault tolerance by

replicating their data multiple times on different nodes across a *cluster*. The notion of cluster here is that of a set of network-connected machines possibly with different specifications. As nodes arrive at or depart from the cluster, (e.g., because of energy concerns), replicas have to be respectively expanded or contracted so that availability remains intact. Such "transitions" however expend computational, storage, and network resources and thus, do not occur instantly. The latter is a key aspect that one has to consider when it comes to *NoSQL*-database provisioning and possibly soliciting resources from *public CSPs*.

We present a resource provisioning approach that exploits the pricing models of the available resources as well as the cost imposed by potential movements of shards. We aim to minimize the total cost of a cloud application by using *look-ahead optimization* for a limited time-window. Fig. 1.1 depicts the key operating aspects of our suggested approach. The *private CSP* delegates the selection of resources needed to run an application to the *look-ahead provisioning* algorithm that oversees the minimization of the total cost. As a result, parts of the application may be "tossed out" to *public CSPs* to expedite processing.

Our approach has two main phases. The first phase consists of profiling the application so that a performance model for the execution of the application in specific sample *cluster configurations* is built; a cluster configuration consists of a specific combination of machines that form a cluster. Several executions of the application on different cluster configurations are needed to stress the application and build a performance model for it. The performance model is needed to estimate the behavior of the application on future cluster configurations. Although the creation of a performance model is a costly task in terms of time, it is required just once. The second phase requires the following pieces of information (Fig. 1.1): *i*) the performance model, *ii*) the application *SLAs*, *iii*) the prediction of the upcoming workload, and *iv*) the available resources from the *private* and/or *public CSPs*. Using this information, the resource provisioning algorithm designates which of the available resources should be added and which of the existing ones should be removed so that the cost of the *private CSP* remains at a minimum.

In this thesis, we make the following three contributions:
1) We expose key provisioning factors: We present factors that should be considered in provisioning as they affect the *private CSP* cost either directly or indirectly. We develop a comprehensive cost model to account for all expenses involved. In our cost model, we include penalties that have to be "paid back" by *public CSPs* should the last renege on their own *SLAs*. We also introduce the *transition cost* needed to re-host a portion of an application and examine how this affects the total cost. We ascertain the value of transition costs through experimentation, outline the difficulties for building a performance model for an application that runs on the cloud where there is large diversity in resources, and

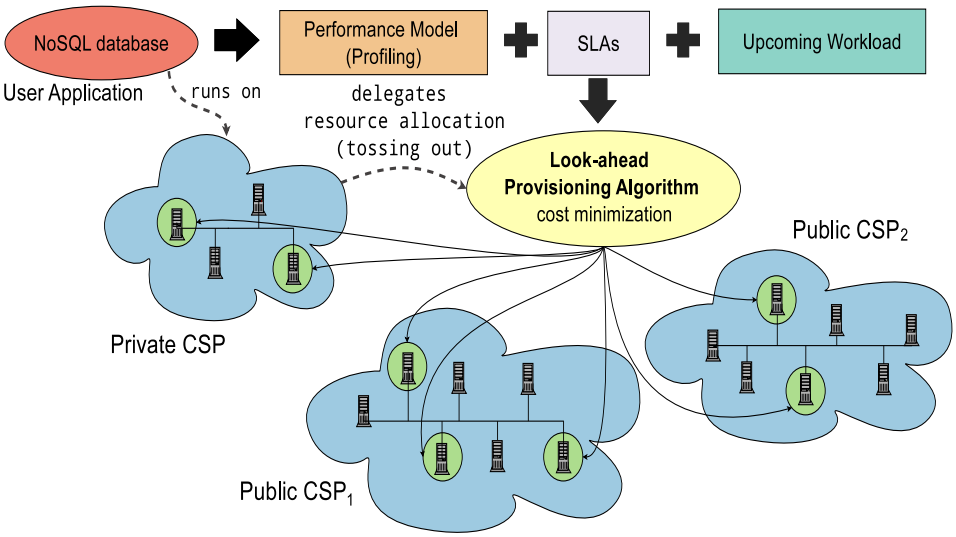

Figure 1.1: Our approach considers application profiling, *SLAs*, and hints on upcoming workload to potentially "toss out" portions of *NoSQL*-database(s) to resources from *public CSPs*.

propose an approach to address this problem.
2) We formulate CSP cost optimization: We address the *NoSQL*-databases provisioning problem from the perspective of the *private CSP* hosting the database. Using our comprehensive cost model, we achieve gains through a collaborative approach in which the just-in-time use of resources from *public CSPs* is exploited. We focus on minimizing the *private CSP*'s cost and we benchmark our method against the conventional and widely used approach of minimizing penalties due to *SLA* violations. Our evaluation shows aggregate gains of approximately 29% for our approach in the conducted experiments.
3) We introduce a look-ahead optimization-based provisioning approach: We study the benefits of using a look-ahead optimization approach on the specific resource provisioning problem compared to other approaches (e.g. [26]), such as thrashing avoidance [18]. We analyze the role the time-window parameter plays in the proposed algorithm, which essentially designates the depth of the search space examined to find an optimal solution. We experimentally evaluate the performance overheads for varying length time-windows and demonstrate that although the complexity of our algorithm is exponential, the execution time is short for reasonably-sized time-windows.

Our thesis is organized as follows: In Section 2, we analyze the salient factors that affect the provisioning problem. In Section 3, we describe profiling and the techniques we use to create a predictive model for the cluster. In Section 4, we present our look-ahead provisioning algorithm. In Section 5 we present a detailed, experimental evaluation of our algorithm. In Section 6 we compare our work with prior related studies and in Section 7, we conclude the thesis.

Chapter 2

Factors in Provisioning

In this section, we discuss the key factors that should be considered in the provisioning of *NoSQL*-databases in a *private/public CSP* context.

2.1 Opportunistic Use of Public Clouds

When *private CSPs* rent additional machines from *public CSPs* to auto-scale *NoSQL*-databases, they form "clusters" of virtual infrastructures that go beyond what they have available locally. This may transparently offer substantial benefits to users as they see their applications "grow" without necessitating the purchase of new machinery but only the occasional leasing of resources. This leasing highly depends on *i*) the specification of the machine(s) needed, and *ii*) the *SLA* that the *public CSP* offers for the request. Differences between nodes that belong to *public* clouds from those in a *private CSP* include the following: *1*) *public CSP* nodes have a rental/lease cost while *private* nodes have an operational cost that entails both energy and maintenance costs, and *2*) a *public* cloud has to compensate the *private* cloud in the form of monetary penalty or *pay-back* anytime an *SLA* is violated. Imposed penalties on *public CSPs* indirectly affect the cost calculation that a *NoSQL*-database host has to pay to a user should *SLA* violations be certified. Thus, the entire amount of penalty is *not* exclusively paid out by the *private CSP* running an application. This is a critical factor that should be taken into account when designing a resource provisioning algorithm. When choosing a *public* cloud, the algorithm should take into account not just the leasing cost but also the *SLA* offered by each public cloud.

2.2 Transitions

Every node addition to or removal from a *NoSQL*-database does not happen instantly. The time it takes for a new node to become operational in a cluster or an operating node to cease operation may range from a few milliseconds to several minutes or hours. A newly instantiated node might need to deploy software artifacts, edit property files and/or start groups of services. Also, in *NoSQL*-databases, a node addition means that data will typically be shipped and replicated over the network. Apart from any requisite data transfer, the cluster may need to update its own internal data structures and indices to

reflect the new state. For example, if record r is replicated from $node_A$ to $node_B$, the cluster has to be made aware that the record now also exists on $node_B$ as well.

The requisite operations described above for a new node to become operational in a cluster or an operating node to cease operation may range from a few milliseconds to several minutes or hours. These operations consume resources from both the new node and the old nodes. In particular, simply replicating data implies heavy use and potentially major overheads in terms of CPU-cycles, network bandwidth, memory and disk. If the above operations are not carefully considered in the provisioning decision, they could move the cluster into an unstable state; instability should be avoided at all costs. In our work, we define the load that a possible transition will incur on the cluster as a set of operations per time unit for a period (i.e., workload). In our experiments, we show that we can model the transition cost as a workflow with certain duration and equal operations per second throughout that duration.

2.3 Cluster Heterogeneity

Cloud infrastructure typically exhibits significant heterogeneity in terms of CPU, memory, disk(s) and NICs of cloud infrastructure nodes [20, 21]. This is due to *private CSPs* incrementally upgrading their internal machinery as well as *public CSPs* competing against each other and frequently changing their rental offerings to better match clients' requests [28]. An adaptive cloud-based application that aims to exploit the best out of the available resources should leverage the heterogeneity of cloud machines accordingly. For example, machines with fast CPUs should be preferred for computationally intensive operations while less powerful machines can be used for applications with ample slack time before they come close to violating their *SLA*(s). This implies that it is harder to create a performance model to estimate future performance outcomes in a heterogeneous environment. In our work, we handle this problem by profiling *NoSQL*-databases such as the ELASTICSEARCH [1] under different cluster configurations. We use *linear regression* and *support vector regression* to predict performance metrics of future deployment outcomes such as estimated throughput and expected percentage of operations with latencies that violate the *SLA* of the application.

Chapter 3

Building an Application Performance Model

To effectively decide which nodes will be part of a cluster, we want to successfully estimate the behavior of the purported configuration once provisioning takes place. We accomplish this by creating a performance model for the *NoSQL*-database under deployment. In this way, we can estimate the cost of a newly introduced configuration as *SLA* violations can be traded off with leasing costs from *public CSPs*.

Predicting the performance of the *NoSQL*-databases with reasonable accuracy is key to our provisioning method. As analytic models based on queueing theoretic techniques [9, 11] are unable to completely capture the operational behavior of applications, we resort to an empirical modeling approach.

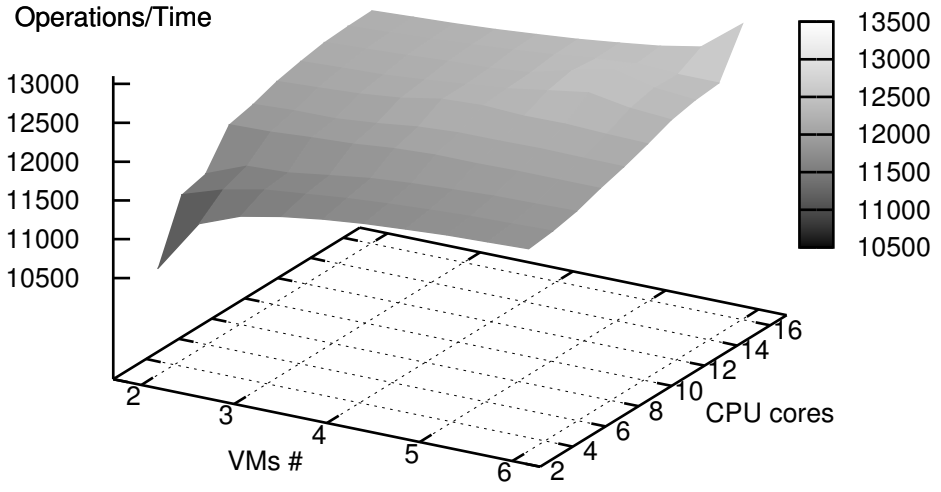

Figure 3.1: Throughput: Operations per second over VMs # and CPU–cores

Our approach has two stages. It first carries out stress-tests on different cluster configurations for the *NoSQL*-database at hand in a similar fashion to that of [17]. This information collection is done offline and imposes no penalties at run time. It then creates

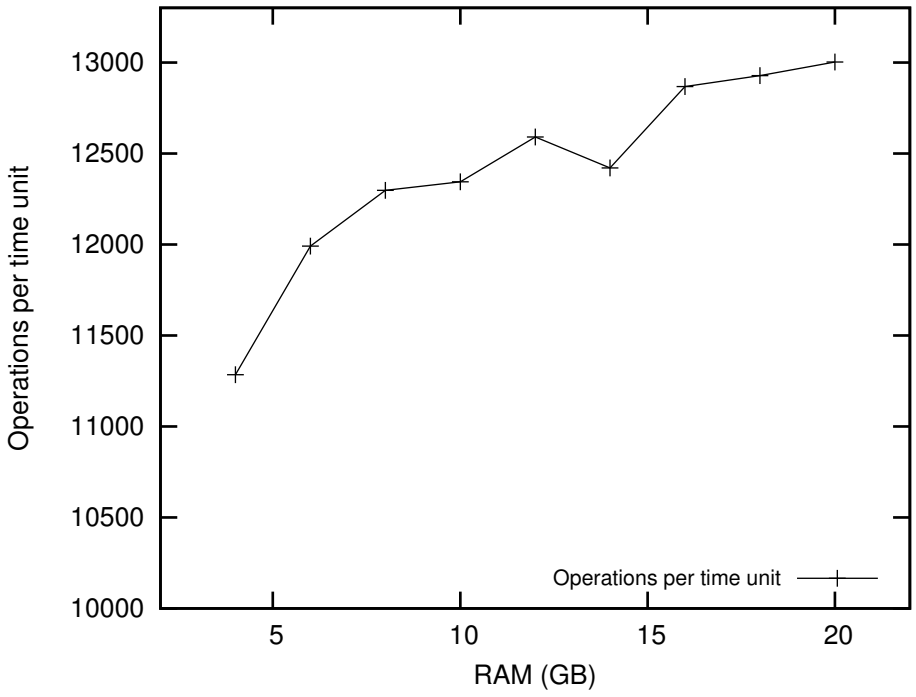

Figure 3.2: Throughput: Operations per second over RAM

a forecasting model to offer suggestions based on data collected and uses linear regression and support vector regression to predict the performance of future cluster configurations to determine whether tossing *NoSQL*-databases out to *public CSPs* is beneficial.

3.1 Profiling Experiments

We use a modified version of the *Yahoo! Cloud Serving Benchmark (YCSB)* [8] to profile ELASTICSEARCH V0.20.6, a popular *NoSQL*-database that uses data sharding [7] to distribute horizontal partitions of data to different *VMs*. ELASTICSEARCH V0.20.6 tends to distribute all the shards equally to all of the nodes that participate in the cluster. Here, we vary the number of CPU-cores, CPU-frequency, memory, and number of *VMs*.

In the standard *YCSB* edition, a client either creates or joins an existing cluster of nodes. Hence, it is likely that at least a portion of the requested data may reside in a shard located on the client's *VM*. This is surely an unusual setting as in cloud environments, the back-end components handling data are often separate from application clients. ELASTICSEARCH features *non-data nodes* that can function as load-balancers. In our modified *YCSB*, a client simply connects to a load-balancer node to access data in all shards dispersed throughout the network. This layout better reflects pragmatic deployments of *NoSQL*-databases.

We perform a number of *YCSB* runs on ELASTICSEARCH with different targeted throughput in clusters with up to 6 nodes and a single load-balancer node while varying the number of CPU–cores, CPU–frequency, memory, and number of *VMs*. We added 3,000,000 records in the ELASTICSEARCH database and then performed 500,000 *GET* operations following a uniform distribution based on the record *ID*. We measured the following:

- throughput of each cluster configuration,
- percentage of operations per time unit that violates *SLAs*; we term this as *DROP*: *delayed response operations percentage*,
- transition cost and delay for data–node addition/removal.

We assume that a request completing in more that *5ms* generates an *SLA* violation. For brevity, we omit showing the profiling experiments for CPU–frequencies. In general, the CPU–frequency profiling results follow similar trends with those of RAM. Below, we outline the key findings from profiling the above *YCSB* database in a *private CSP*.

•*Throughput:* Fig. 3.1 shows that as *VMs* are added, throughput increases almost linearly. The same behavior is observed for CPU–cores as more *GET* operations can be handled. Similar trends are depicted in Fig. 3.2, as far as varying the size of RAM is concerned. It is worth noting here that throughput is affected less by RAM and CPU–frequency as *GET* operations scale out better (i.e., adding more *VMs*) than scaling up (i.e., increasing memory and CPU–frequency).

•*DROP:* Results for requests missing their *SLAs* are not as clear cut as those of throughput. Fig. 3.3 reveals that the resulting *DROP* maintains high margins between the average value and the maximum and minimum values attained as we increase the number of *VMs*. Similar observations hold for *DROP* rates while varying CPU–cores and RAM in Figs. 3.4 and 3.5; they demonstrate behavior with no clear trends. Thus, the above three profiling view-graphs cannot lead to any strong conclusions regarding *DROP* prediction.

•*Transition Cost and Delay:* We have performed experiments where we added or removed data–nodes and monitored how the throughput of the cluster was affected. In these profiling experiments, we ascertained that the transition cost (i.e., transporting shards) is almost independent of the configuration of the nodes that participate in the cluster and mostly depends on the network bandwidth.

Profiling experiments are applicable to both *private* and *public CSPs* and help us derive effective forecasting models for resource provisioning.

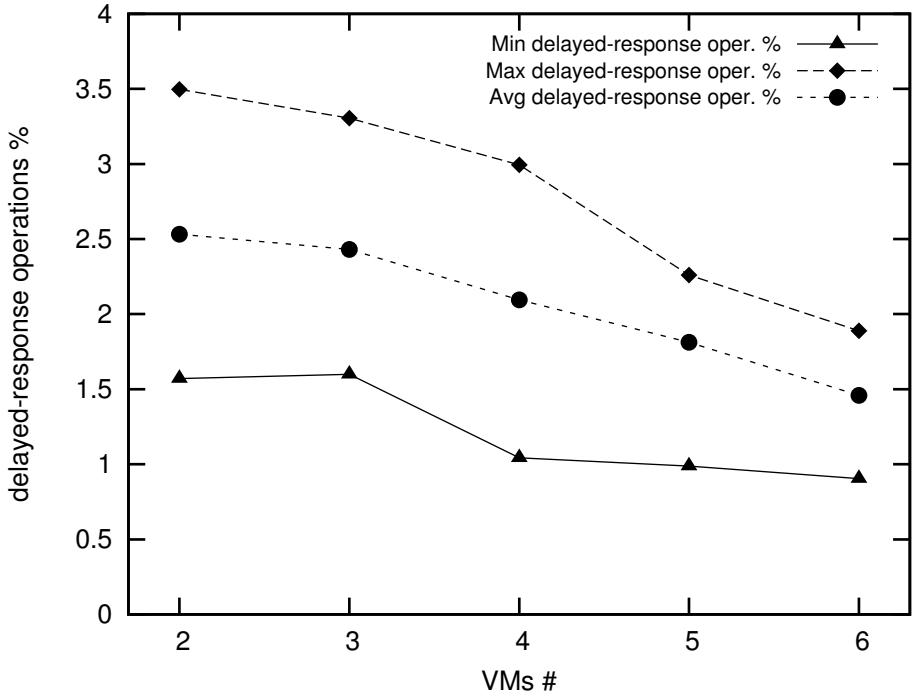

Figure 3.3: *DROP* over VMs #

3.2 Forecasting Models

Our forecasting model takes as *input* a set of *VMs* to be possibly incorporated into the operational cluster along with a number of parameters that include: *i*) *public CSPs* from which to lease the *VMs*, *ii*) CPU–cores, *iii*) CPU–frequency, *iv*) size of RAM, *v*) number of *VM* nodes. The outcome of the forecasting model states how the re-aligned cluster would perform should the additional *VMs* from the *public CSPs* be included as part of the cluster. The *output* of the model consists of the following anticipated rates and/or values: *I)* throughput rate, *II) DROP* rate, as well as *III)* transition cost, duration, and delay. Below, we discuss how we deliver these three rates and costs.

The outcome of our black-box profiling yields selected measurements for specific coordinate values in a multidimensional space. If we knew every possible value in this space, we would be able to derive the best solution for a given provisioning. However, this is infeasible, so we use approximate estimation methods to produce the output rates/values of the model. More specifically, we use predictive techniques [10, 12] to estimate expected rates for throughput and *DROP*. These techniques need an adequate size of training data to be well calibrated. Moreover, the training set has to consist of a representative sample of cluster configurations to both accurately predict the future and successfully remove outliers. We experimented using the *RapidMiner* [3] to ascertain the advantages and

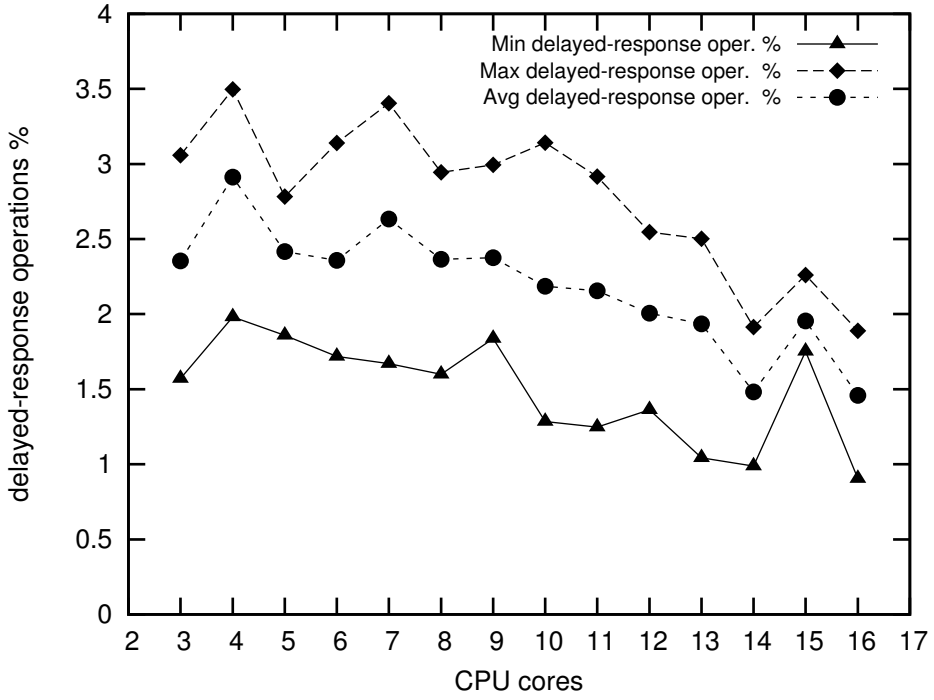

Figure 3.4: *DROP* over CPU–cores

disadvantages of various predictive techniques and we have identified the following two options to respectively estimate throughput and *DROP* rates:

I) Linear Regression for Throughput: Fig. 3.1 clearly shows that the cluster throughput increases linearly with the number of *VMs* and/or CPU–cores. Consequently, using linear regression to estimate throughput rates for cluster configurations that have not been evaluated in the stress-test profiling phase is the apparent choice. In contrast, the addition of RAM in the cluster leads to less discernible gains for throughput (Fig. 3.2). A similar trend to that of RAM occurs with CPU–frequency as well. While using linear regression for throughput estimation, we place less importance on the RAM and CPU–frequency values than the number of *VMs* and CPU–cores used by using appropriate weights; the latter are computed during the fitting process.

II) Support Vector Regression (SVR) for DROP Rate: Fig. **??** collectively reveals that although the average *DROP* rate decreases as the values of input variables increase, the minimum-to-maximum range for resulting *DROP* values remains large. There are undoubtedly complex relationships between the five input variables and the expected *DROP* rate that are impossible to capture using linear estimation techniques. The presence of multidimensional variables along with their complex relationships makes the *Support Vector Regression (SVR)* approach suitable for our case as it can more effectively predict the *DROP* rate.

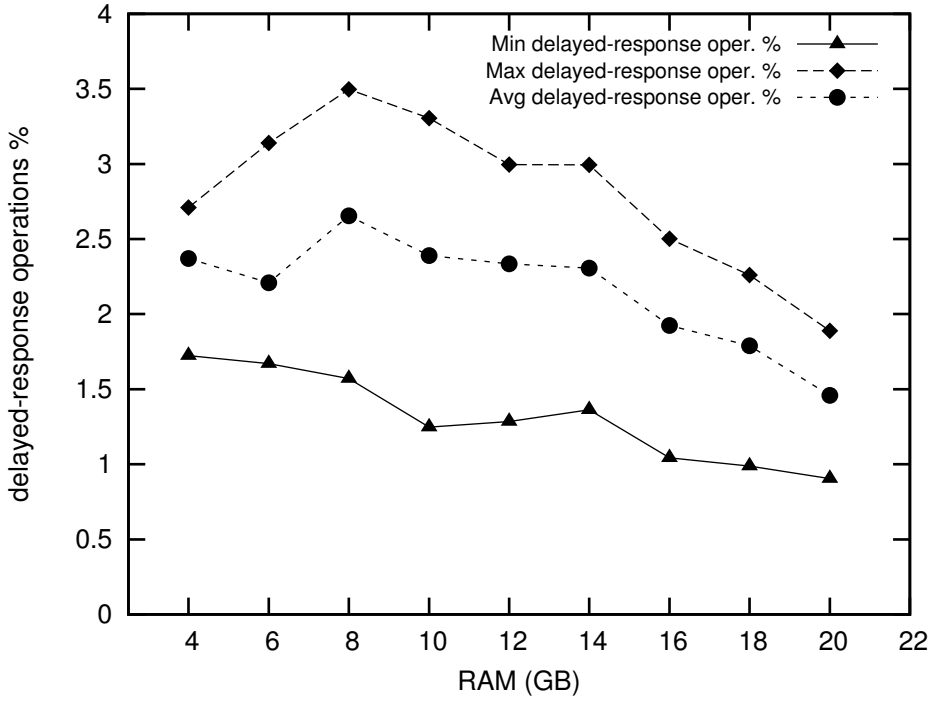

Figure 3.5: *DROP* over RAM

SVR maps data from their own original space into a higher–dimension feature space and then computes an optimal regression function in this new feature space [27]. This data transformation is carried out through the mapping: $v \rightarrow \varphi(v)$, where $v = (v_1, v_2, \ldots, v_n)$ is a vector of independent variables; in our case, v represents the n values of our *input variables*[1] making up a single data point. The mapping is assisted by kernels that essentially bypass the explicit use of $\varphi(.)$ to transform data to the new feature space. Kernels are realized as the dot product of two vectors i and j in the feature space as follows: $k(v_i, v_j) = \varphi(v_i) \cdot \varphi(v_j)$ Among popular non–linear kernels, we use the *Gaussian Radial Basis Function (RBF)* as the *DROP* rate depicts non-linear behavior and *RBF* proves to be the most accurate. The *RBF* kernel is: $k(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j) = e^{-\gamma \|x_i - x_j\|^2}$, where γ is an adjustable positive variable.

III) Using additional *VM*(s) from possibly different *public CSPs* involves a delay that is required to ship a shard to the designated *VM*(s). This transition can be expressed as a function over time as follows:

$$transition(t) = \begin{cases} 0 & t \in [0, delay] \\ tr\_overhead & t \in (delay, T] \end{cases}$$

[1] In particular, five values corresponding to (i) *public CSPs* used, (ii) CPU-{cores, (iii) CPU-{frequency, (iv) size of RAM and (v) number of *VMs*.

Table 3.1: Transition cost values

Variable	**Value**
$D_{startup}$	3 sec
$D_{shutdown}$	2 sec
overhead_per_shard	1200 $^{\text{oper}}/_{\text{sec}}$
duration_per_shard	1.5 sec
total_shards	10

where *delay* = $D_{startup|shutdown}$, *T* = *delay* + *tr_duration*,
tr_duration = *duration_per_shard* * *moved_shards*,
tr_overhead = *overhead_per_shard* * *moved_shards and*
moved_shards = max*(added_nodes* * *(total_shards* / *new_cc_nodes),*
removed_nodes * *(total_shards* / *cc_nodes))*.

In the above, $D_{startup|shutdown}$ represents the fixed time that a *VM* takes to either start up or shut down, *moved_shards* is the number of shards (horizontal partitions of the database) to be moved when the cluster configuration changes, *added_nodes* and *removed_nodes* are the number of the nodes added to or removed from the cluster configuration respectively, *new_cc_nodes* and *cc_nodes* are the number of the nodes in the new and current cluster configurations respectively, *total_shards* is the number of shards involved in the *NoSQL*-database, *duration_per_shard* is the overhead, in seconds, that each moved shard adds and *duration_per_shard* is the overhead in operations per second that each moved shard generates.

Table 3.1 shows average values of key factors as generated during the exploratory profiling process discussed in the previous section (i.e., *Transition Cost and Delay*). Multiple experiments yield invariable values indicating constant overhead behavior.

Chapter 4

Look-Ahead Optimization for CSPs

Our main objective is to identify the least expensive combination of nodes that collectively satisfies the constraints imposed by the cloud applications(s). These constraints can be either strong (i.e., calling for *SLA* penalty minimization) or weak (i.e., seeking to lower the *CSP* expenses). Either way, the selection of *VMs* and the identification of a "cluster" to be used highly depend on the current configuration on which the application runs as well as its anticipated workload characteristics. [20] showed that service provisioning is *NP*-hard and suggested heuristics to prune the solution space by limiting either the depth of the ensued search tree or the time period within which a viable solution is sought.

We employ *Look-Ahead Optimization (LAO)* because it helps identify a (sub)optimal selection of a new cluster configuration by examining all possible paths that are feasible at a specific point in time. Our approach uses the current state of affairs but also seeks to optimize future states. This presents advantages as *LAO* takes a long-term approach that better facilitates the optimization strategy in a dynamic *CSP* environment where factors including cost-changes and application workload variations are the norm. Approaches such as [19] and [26] address the provisioning problem by respectively following an integer linear programming or a constraint satisfaction approach. In doing so, these approaches do not consider valuable information emanating from application workload predictions. As in [19], we assume an accurate predictor for workload characterization.

4.1 Receding Horizon Control (RHC)

Receding Horizon Control (RHC) is a *LAO*–method that iteratively solves an optimization problem for a fixed time interval while taking into account current and future constraints; it has been used for resource provisioning [18] and geographical load–balancing [16, 29]. *RHC* functions in a recurrent fashion as follows:

S1) At time k, find an optimal solution for the specific and fixed–period $[k, k+T]$ while considering current allocations and forthcoming constraints.

S2) Apply only the first element of the above optimal sequence.

S3) Shift time t to $k+1$ and repeat the process for the interval $[k+1, \ldots, k+T+1]$.

Should there be no (other) external factors that affect the cost computation of the solution sought in step *S1* above, the *RHC* finds the optimal solution for the given time-window T.

Let us assume that:

A1) J represents the sum of the operational/leasing cost of the *VM* resources placed in a cluster from both *private* and *public CSPs* the costs of incurred *SLA* violations and *public CSP* pay-backs, and the imposed transition cost for a unit of time,

A2) x_t represents the state of cluster in terms of a set of allotted resources at time t,

A3) u_t entails all feasible transitions to reach a new cluster configuration at time t; this set of transitions involves additions or removals of *VMs*,

A4) $\{x_t\}$ is the sequence of all states generated in the period $k \dots t$,

A5) $\{u_t\}$ is the sequence of all transitions that have taken place within period $k \dots t$,

A6) $x_{i+1} = f(x_i, u_i)$ for $i = k, \dots, k+T$, where f is the function that maps a state x_t to the next x_{t+1} according to u_t input choices available at time t,

A7) $cost(\{x_i\}, \{u_i\}) = \sum_{i=k}^{i=k+T} J(x_i, u_i)$ represents the cumulative cost incurred while following the $\{x_i\}$ sequence with the corresponding $\{u_i\}$ sequence and J is the cost function defined above in *A1*.

In step *S1* of the *RHC*, we identify the optimal solution as the one that provides:

$$cost_{opt} = \min \ cost(\{x_t\}, \{u_t\}) \tag{4.1}$$

The solution of the above optimization problem leads to a sequence of suggested cluster configurations $\{x_k, ..., x_{k+T}\}$ and a respective sequence of transitions $\{u_k, ..., u_{k+T}\}$ that eventually take place. The sequence $\{x_k, ..., x_{k+T}\}$ corresponds to a path having the minimum cumulative cost in the time–window elapsed between $k \dots k+T$.

4.2 Selecting the Time-Window Period

The time-window is a fundamental *RHC* parameter as it designates the depth in which a solution is to be searched and presents a number of trade-offs. On the one hand, a short window might miss a number of good long-term changes if it cannot capture significant future workload changes. On the other hand, a long time-window affects the execution time of the algorithm as it may introduce exponential complexity. In general, a good choice for time-window length should be able to capture at least a few complete transitions in the make up of a cluster as well as pertinent overheads. Any benefits in the operation of a re-aligned cluster will be reaped after the transition eventuates. Hence, it is also imperative that the time-window be a function of the duration of the transitions. An application that appears to have transitions with long durations requires lengthier time-windows than an application that changes its configurations more rapidly. We experimentally assess the impact of this time-window choice in Section 5.

4.3 Resource Provisioning Algorithm

In this section, we present our *RHC*-based resource provisioning approach. Algorithm 4.1 recursively determines the cost as well as the entire sequence of cluster configurations generated within the time-window [*start_time*, *end_time*] that imposes minimum cost for the *private CSP* (*best_configs*). Starting from the initial cluster configuration (*cc*), the algorithm examines all possible configurations that can be reached while trying to identify the next configuration possibly involving *VMs* from *public CSPs* as well. The invocation of POSSIBLE_CLUSTER_CONFIGS() produces feasible configurations by taking into account the replication factor of the *NoSQL*-database. The replication factor designates the number of redundant copies of shards, and so, it limits the number of *VMs* that can be removed from a cluster during a single transition.

Algorithm 4.1 Provisioning Best-Plan

```
procedure BEST_PLAN(cc, start_time, end_time, best_cost, best_config)
    for all cl in POSSIBLE_CLUSTER_CONFIGS(cc) do
        tr_delay, tr_duration, tr_overhead = TRANSITION(cc, cl)
        time = start_time
        if tr_delay + tr_duration + time > end_time then
            cost = 0
            tr_delay = 0
            tr_duration = end_time - start_time
        end if
        cost = PARTIAL_COST(cc, cl, tr_duration, tr_delay, tr_overhead, start_time)
        time += tr_delay + tr_duration
        configs = [ ]
        if time < end_time then
            p_cost, configs = BEST_PLAN(cl, time, end_time, best_cost, best_configs)
            cost += p_cost
        end if
        if cost < best_cost then
            best_cost = cost
            best_configs = cl + configs
        end if
    end for

    return (best_cost, best_configs)
end procedure
```

A possible change in cluster configuration implies transition costs for resource re-alignment that may require non-negligible operations and takes a *duration* interval to unfold. Moreover, the transition may have a latency, termed *delay*, before it actually completes. TRANSITION() computes estimations for transition delay, duration and overhead based on the suggested performance model of Section 3. These three values, along with current and a feasible new cluster configuration, are furnished to Algorithm 4.2 (PARTIAL_COST) to assess the cost of a proposed transition; the latter is essentially the factor J discussed in Section 4.1. Subsequently, Algorithm 4.1 shifts the *start_time* of the time-window by as much as the time required to complete the suggested transition. The

algorithm then moves to compute the rest of the optimal cluster configuration sequence in a recursive manner always using the first element of the remaining sequence as the pivot for its exploration. In this regard, the recursive calls along with the loop over the set produced by POSSIBLE_CLUSTER_CONFIGS(), build a tree with all feasible configuration sequences within the sought time-window. As the recursive calls return, the tree is traversed from the leaves to the root: at every intermediate node the loop keeps the partial path with the minimum cost. As the process continues, the path with the optimal cost from the root to the leafs is found, which is equivalent to the optimal cluster configuration sequence.

Algorithm 4.2 realizes the operation of PARTIAL_COST() and computes the entire cost including transitioning, violation of *SLA*, pay-backs from *public CSPs*, and operational overheads, for a suggested new configuration. PARTIAL_COST() takes as input the current configuration (*old_cl_config*), the proposed new configuration (*cl_config*), estimated transition overhead (*tr_overhead*), duration (*tr_duration*) and delay (*tr_delay*) and returns the total cost of the transition period. The additional work that a *private CSP* has to undertake to bring the cluster to its new suggested state is designated by the *tr_delay* interval. The latter corresponds to the latency of the transition and through this period, the cluster appears as operating its prior configuration (*old_cl_config*). When *VMs* are moved in and out of a configuration, they remain idle during this process –no service is provided– and *tr_delay* accounts for the effort required to accomplish this re-alignment of resources. As soon as *tr_delay* is accounted for, the transition is in progress. In this transition phase, the operating *VMs* of the cluster involve elements from both old and new configurations as: *1*) newly introduced *VMs* become fully functional after the completion of the transition, and *2*) *VMs*–to-be removed are released immediately after *tr_delay*.

For a specific point in time, CLUSTER_CONFIG_COST() computes the operational and penalty costs incurred by possible *SLA* violations. Algorithm 4.3 takes as input the cluster configuration (*cl_config*), the number of *VMs* currently allotted (*op_cl_config*), the expected *workload* at this time instance (expressed in number of operations per time unit) and the transition overhead (*tr_overhead*); CLUSTER_CONFIG_COST() returns the operational cost of the cluster configuration during the time unit in question. To compute potential *SLA* violations, we use *linear* and *support vector regression* to gauge the maximum throughput of a given cluster configuration and the *DROP* rate. The above is accomplished by respectively invoking PREDICT_THROUGHPUT() and PREDICT_DROP(). The fraction of *SLA* violations accorded to *VMs* coming off *public CSPs* yields pay-backs to the *private CSP*. OPER_CL_COST() determines the sum of the rental/operational cost of each node within *op_cl_config* depending on whether the *VMs* in question belong to either an *public* or the *private CSP*.

Algorithm 4.2 Partial Cost

```
procedure PARTIAL_COST(old_cl_config, cl_config, tr_duration, tr_delay, tr_overhead, start_time)
    op_cl_config = UNION(cl_config, old_cl_config)
    // the total nodes allocated during tr_delay

    tr_cl_config = INTERSECT(cl_config, old_cl_config)
    // the fully functional cluster during the transition

    cost = 0
    time = start_time
    tr_end_time = start_time + tr_delay + tr_duration
    while time < tr_end_time do
        wl = workload[time]
        // workload is the array of predicted future workload (operations per time unit)

        if time - start_time < transition_delay then
            p_cost = CLUSTER_CONFIG_COST(op_cl_config, old_cl_config, wl, 0)
        else
            p_cost = CLUSTER_CONFIG_COST(cl_config, tr_cl_config, wl, tr_overhead)
        end if
        cost += p_cost
    end while

    return cost
end procedure
```

Algorithm 4.3 Cluster Configuration Cost

```
procedure CLUSTER_CONFIG_COST(op_cl_config, cl_config, workload, tr_overhead)
    total_workload = tr_overhead + workload
    cluster_throughput = PREDICT_THROUGHPUT(cl_config)
    handled_workload = min(cluster_throughput, total_workload)
    drop = PREDICT_DROP(cluster_config)
    violations = max(total_workload - handled_workload, 0)
    // violations due to throughput

    violations += handled_workload * drop
    // violations due to drop

    violations_per_node = violations / cl_config.nodes_no
    payback = 0
    for node in cl_config do
        if node.belongs_to_public_csp then:
            payback += node.sla.penalty * violations_per_node
        end if
    end for
    total_penalty = app_sla_penalty * violations - payback
    // app_sla_penalty is the penalty for each SLA violation in the application

    return OPER_CL_COST(op_cl_config) + total_penalty
end procedure
```

Although Algorithm 4.1 has exponential complexity, caching of intermediate results –especially for Algorithms 4.2 and 4.3– leads to reduced execution time for our provisioning approach.

Chapter 5

Evaluation

We present key evaluation results based on simulation experiments for our provisioning approach using the models suggested in Section 3. Our simulation package is written in *Python v.2.7.5* and uses the *scikit-learn* library [4] to compute the *SVR*. Table 5.1 outlines the main characteristics of the *VMs* we use in our experiments along with their costs and *SLA* violation penalties as they are advertised by *public CSPs*. We set the penalty for each *SLA* violation of the application running to 0.6 monetary units and set the time-window size to 25 units. In the beginning of every experiment, a cluster consists of 1 *VM* from the *private CSP*. For simplicity, we add/remove 1 *VM* during each transition. We investigate the following:

- the cost model of our approach (*A1* in Section 4.1) vs. that of the conventional *SLA*-cost minimization approach,
- the effect that the transition cost has on provisioning,
- the behavior of our approach during short/long workload spikes, and
- the impact on performance of varying the length of the time-window in our approach.

The workload used was synthetically created using epochs demonstrating periodic behavior; every epoch has length of 1,000 time units, displaying a mean of $9k$ *GET* operations per unit and $0.6k$ operations standard deviation. Within an epoch, short and long spikes occur with a 7:2 proportion and have mean lengths of 32 and 4 time units respectively. Two categories of spikes exist: *i*) high–load spikes with $13.5k$ mean operations per time unit and $0.45k$ standard deviation and *ii*) medium–load spikes with $11.25k$ mean operations per time unit and $0.6k$ standard deviation. The random generators used to produce the workload follow normal distributions. Although, we run experiments for lengthy durations (up to $100k$ time units), we mainly report results in a specific limited range of 200 time units for readability purposes.

5.1 Our Cost Model vs. *SLA*-Cost Minimization

We use *RHC* as the main framework for provisioning and we compare how the cost model we introduced in Section 4.1 fares in comparison with the conventional and widely-used approach of *SLA*-cost minimization [6, 10]. Both techniques are deployed in a simulated

Table 5.1: *VM* specifications

CSP	**CPU cores**	**CPU freq (GHz)**	**RAM (GB)**	**Penalty per *SLA* violation (in monetary units)**	**Cost (per time unit)**
prv	4	3.2	2	–	48
prv	2	3.0	2	–	40
pub1	2	2.4	6	0.3	56
pub1	1	2.4	2	0.1	28
pub2	3	2.4	4	0.3	52
pub2	4	2.4	4	0.3	64

private/public CSP infrastructure and we track the number of allocated *VMs* over time for the execution of a synthetic workload; the latter calls for a diverse number of *GET* operations for specific time units. We also monitor accrued costs including: *1*) the penalty cost of the *SLA* violations, *2*) the operational cost of the *private CSP* *3*) the lease cost of *VMs* rented from a *public CSP* *4*) the penalty payback and *5*) the transition cost. The *SLA*-cost minimization approach involves only the penalty cost due to *SLA* violations and it does not include cluster operational costs, pay-backs from from *public CSPs* as well as transition costs (i.e., *tr_overhead* is 0).

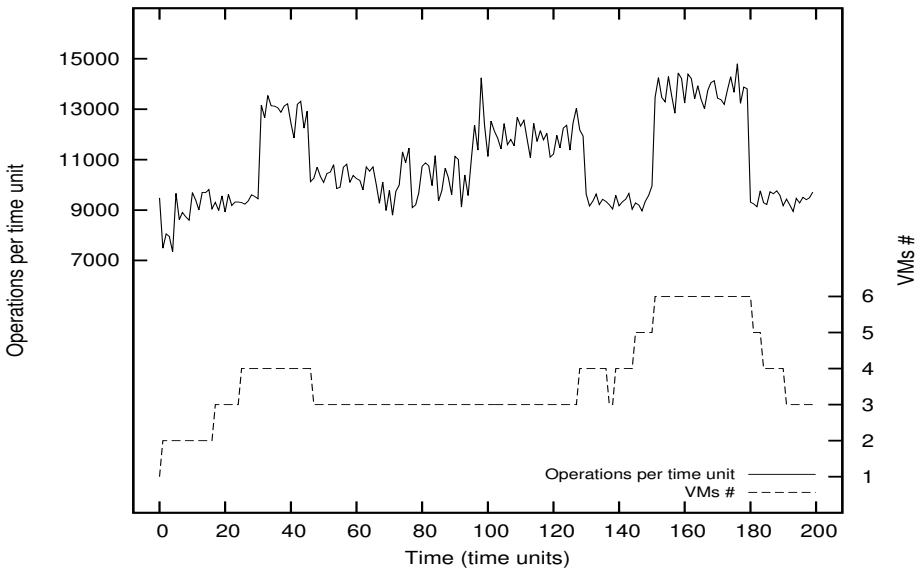

Figure 5.1: *RHC* with our Cost Model

Figs. 5.1 and 5.2 show the number of *VMs* used by our provisioning and the *SLA*--cost minimization approach respectively. Fig. 5.2 shows that the *SLA*-cost minimization approach tends to allocate more *VMs* to handle the workload and to maximize *QoS*. The

conventional *SLA*-cost minimization approach does not take into account the operational cost of the *VMs* in the cluster and thus, often chooses configurations with the highest performance capacity. This approach appears to "encourage" changes in cluster configurations, as there is no consideration for transitional costs. For instance, this is what occurs at time unit 77-78. In contrast, Fig. 5.1 shows that our approach requires fewer

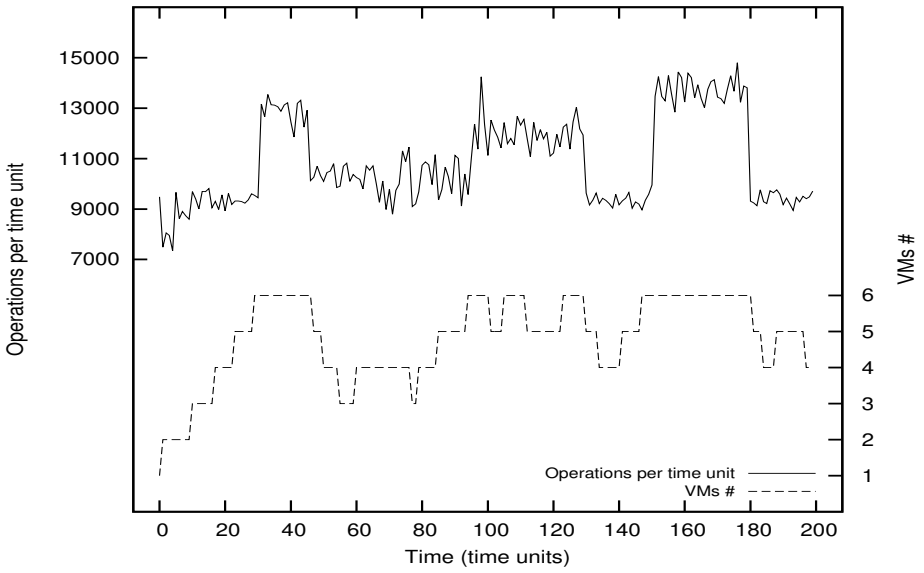

Figure 5.2: *RHC* with *SLA* Cost Minimization

VMs for most of the time and releases them as soon as they are not needed to reduce operational cost. The transition cost makes our approach more conservative to changes; in our approach, there are only 12 encountered configuration changes compared to 24 in the *SLA*-cost minimization method. By considering operational/transition costs as well as pay-backs, our approach tries to balance performance capacity on the one hand, and the investment in new cluster configurations with more/fewer resources on the other. This is why our approach uses only 4 *VMs* to handle the workload in time units 30-45 while the conventional *SLA*-cost minimization method exploits all available *VMs* in the experiment.

When it comes to costs, the *SLA*-cost minimization approach calls for an average of 9% more than our suggested provisioning just for the limited 200 time units of observation for Figs. 5.1 and 5.2. The corresponding result for the $100k$ time units experiment stands at 28.6%. The above clearly point out that the penalty minimization of the *SLA* violations does not lead to the minimization of the total cost. As our approach can better capture the actual costs involved in the execution of workloads, it is of substantial value to *NoSQL*-database owners. As applications grow and load increases, our *RHC*-based approach helps the *private CSP* decide whether it is beneficial to either invest in *QoS* or limit its user capacity.

5.2 The Effect of Transition Cost

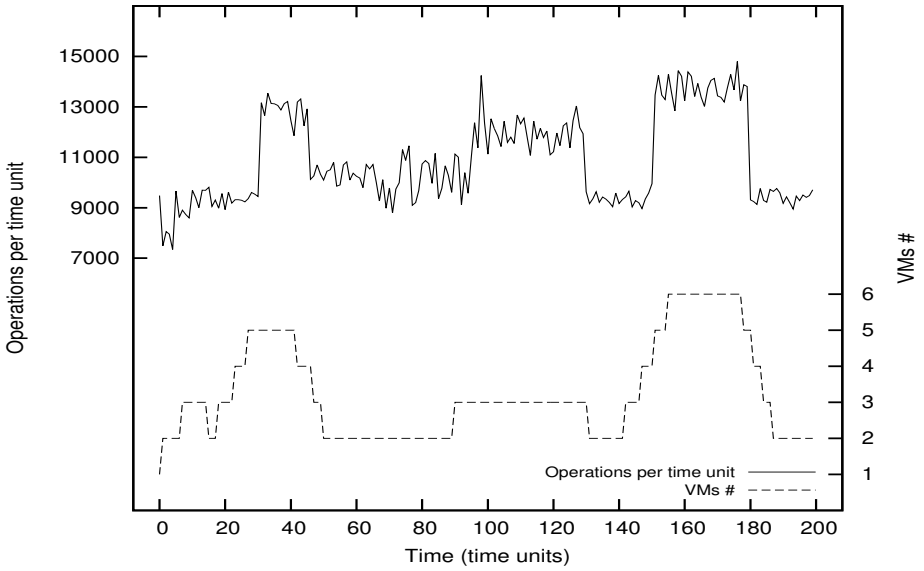

Figure 5.3: Our Provisioning Approach with Lower Transition Cost

We now evaluate how transition cost affects the cluster configuration changes. When a cluster that runs a *NoSQL*–database allocates or deallocates *VMs*, shards need to be transported. The overhead of this process is the transition cost, which is a key factor for provisioning since it indirectly affects the total cost of the *private CSP*. Fig. 5.3 shows the simulation results of the same experiment as that of Fig. 5.1 but with lower transition cost. More specifically, we set the *tr_overhead* and *delay_per_shard* of Table 3.1 at 66% less than the corresponding values of the first experiment. We find that the number of transitions increases from 12 (in Fig. 5.1) to 19 as transition costs decrease. In the original experiment, high-transition configuration changes are not encouraged by our approach, as the potential benefits are less than the incurred cost. With lower transition costs, our *RHC*–based provisioning becomes more aggressive in tracking even rapidly changing workload trends. For an effective transition to occur, the length of the time-window used must be longer than the respective transition cost. The number of allocated *VMs* is higher for Fig. 5.3 than in Fig. 5.1. For instance in the period 30-45, all 6 *VMs* available are now allocated in comparison to 4 for the corresponding time interval of Fig. 5.1. The same applies when it comes to resource deallocation. As transitions become short and less expensive, the *VMs* can be allocated/deallocated faster and in a less costly manner. In this context, abrupt workload changes can be more efficiently handled as *VMs* can be supplied faster. As *NoSQL*-databases occasionally present varying transition costs [1,5,15], systems with such high costs are less effective at handling rapidly changing

workloads. Hence, the transition cost is a key factor when using a *NoSQL*-database as it is equally critical to throughput and *DROP* attained when the workload displays abrupt variations.

5.3 Long vs. Short Workload Spikes

Web-based *NoSQL*-databases may experience significant workload variations for limited periods of time. This is common in e-shop and *CRM* sites that try to overcome surges in workload caused by the sudden announcement of attractive offers followed by numerous user requests (i.e., flash crowds). In this experiment, we investigate how our *RHC*-based approach compares with *SLA*-cost minimization when there are spikes in the workload.

Fig. 5.4 shows a workload featuring a long and a short spike at time intervals $40-65$ and $140-144$ respectively. The figure also depicts how our *RHC*-based provisioning and the *SLA*-cost minimization approaches behave in both instances. Our approach handles

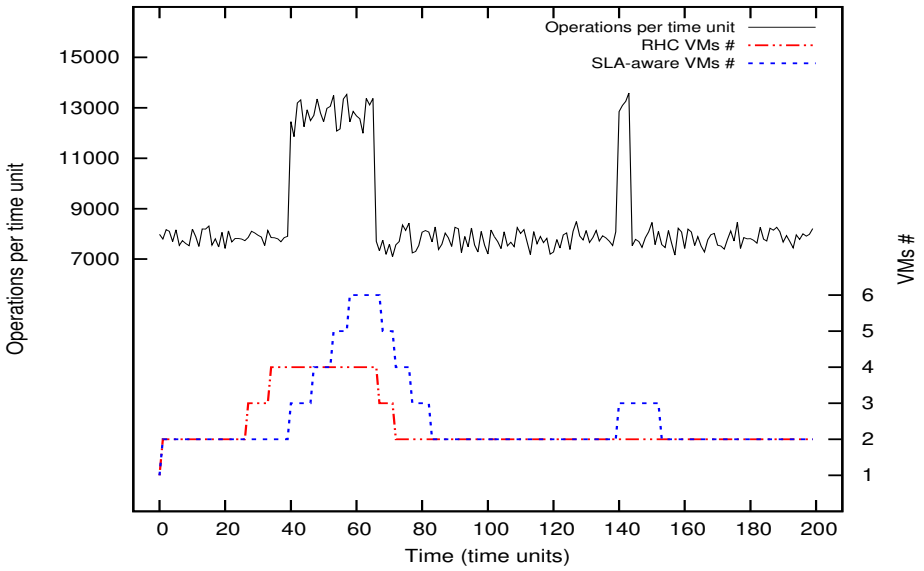

Figure 5.4: Long and Short Workload Spikes

the long spike by adding *VMs* while it essentially ignores the short spike. During the short spike, the cluster does not move to another configuration to handle this short-lived demand as the total cost for a possible transition does not out-benefit inaction. The *SLA*-cost minimization method keeps adding *VMs* while the workload remains high (time interval 40-65). At the end of this spike, the *SLA*-cost minimization method ends up having all available *VMs*, which leads to a long deallocation period as deallocations are not instant. Hence, unnecessary *VMs* continue to be allocated for some time after the spike ceases which results in higher operational cost. During short spikes, the conventional *SLA*-cost

minimization method attempts to allocate additional resources being oblivious of what lies ahead. In this specific instance (i.e., time interval 40-65), the transition costs involved are of similar length to the spike in question. Thus, additional *VMs* become functional beyond the time at which the spike ends yielding a resource thrashing situation. Our *RHC*-based provisioning avoids such thrashing because it continually evaluates the total cost of every feasible sequence of cluster configurations in a time-window and picks the best.

5.4 Time-Window Impact

In this experiment, we measure the effect that the choice of time-window has on the execution time of our algorithm. We ran several simulation experiments with varying time-window sizes on a machine with 2 GHz Intel Core 2 Duo processor and 4GB RAM on *MacOSX 10.8.5*.

Fig. 5.5 shows that the execution time (logarithmic scale) of the algorithm grows exponentially as the length of time-window increases. This growth becomes more pronounced after the 8th time-unit; before this time unit, no opportunities for transition exist. As the window becomes longer, the respective search tree of the approach increases rapidly. We note that setting the time window to a value in the range of 15 to 25 presents a reasonable selection for our experimentation, as the execution was consistently less than 10 seconds.

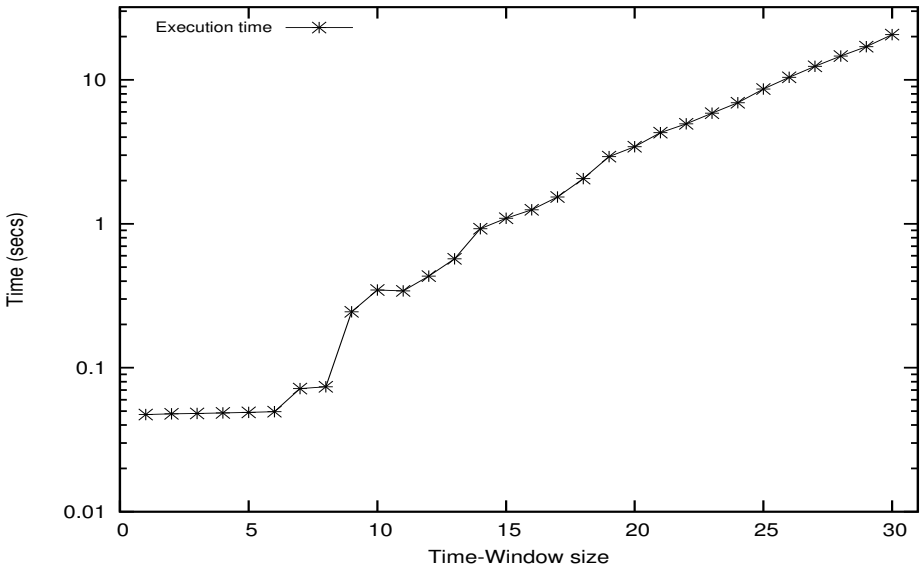

Figure 5.5: Execution Time of our *RHC* Provisioning Algorithm (*log-scale*)

Chapter 6

Related Work

Resource provisioning for cloud-based systems has recently attracted much attention. Efforts in [9, 11] attempt to address the problem through the use of queueing theory and respective model building. As cloud systems are inherently complex, involve parallel and concurrent aspects and are built on heterogeneous environments, such queueing theory models are difficult to extend and quickly become intractable. The extensive use of caching and locking policies further exacerbates matters [5]. Sharma et al. [19] present a system that statically searches for best allocation scenarios and then picks the one that minimizes migration costs. The work also advocates for the adoption of performance model building through profiling. Roy et al. [18] presented an *RHC*-based approach that minimizes the operational cost of a host–cloud while satisfying all *SLA*s. However, price variation for resources is not taken into account in the used price model.

In [13], a classification that designates cloud content as either active or passive based on the frequency of the received read/write operations is proposed. This content model is used to assist server selection strategies to achieve fast and efficient data transfers and processing. A *max-min* algorithm is used to solve the allocation problem at hand. Gourdasi et al. [11] outline an approach to minimize the total energy cost of a cloud computing system while keeping the *SLA*-incurred costs low. They accomplish this by using a heuristic algorithm based on convex optimization and dynamic programming. Although this work appears to incorporate multiple costs into the minimization problem, the use of queueing theory models entails issues similar to those mentioned above. Barker et al. [6] presented a migration approach for multi-tenant databases that utilize a throttling controller; the latter aims to dynamically vary the migration speed to avoid *SLA* violations due to the transition cost imposed by a migration.

To the best of our knowledge there is no work that combines penalty pay-back from *public CSP*s, a critical aspect from the *private CSP*'s point of view, and only a few [6, 18, 19] take into account the transition cost. Our work takes a holistic approach and addresses resource provisioning through the occasional leasing of *public* resources in a way that minimizes *total cost* for the *private CSP*.

Chapter 7

Conclusions & Future Work

In this thesis, we investigate how *NoSQL*-databases running on *private* Cloud-Service Providers (*CSPs*) could be partially "tossed out" to opportunistically exploit resources available from *public CSP* counterparts. Such collaborative auto-scaling helps both minimize total cost for the *private CSP*–hosted application and more flexibly address *QoS*-requirements. We presented a resource provisioning approach based on *look-ahead optimization* that leads to lower *CSP* costs for a limited time-window while considering how to best transform the utilized virtual infrastructure over time. We identify key factors that contribute to the *CSP* aggregate cost and propose a cost model that accounts for both direct and indirect penalties to avoid *SLA* violations for the hosted-application(s). We formalize the transition cost and demonstrate its importance in resource provisioning. Our evaluation demonstrates the benefits of our cost model over the conventional approach of simply minimizing *SLA* cost with reported gains of up to 29% for the conducted experiments. Moreover, we show the benefits of using a look-ahead optimization technique in order to avoid resource allocation thrashing when the workload changes rapidly. We plan to investigate the relaxation of the accuracy of the used predictor, examine the respective ramifications and ascertain the role introduced errors may have in workload estimation. We also plan to develop adaptive time-window provisioning algorithms based on historical data.

Acronyms

Abbreviation	Full Name
CSP	Cloud Service Provider
DROP	Delayed-responce operations percentage
LAO	Look-ahead optimization
NP	Non-deterministic polynomial time
QoS	Quality of Service
RHC	Receding Horizon Control
SLA	Service Layer Agreement
VM	Virtual Machine

References

[1] Elasticsearch. http://www.elasticsearch.org.

[2] Mongoddb. http://www.mongodb.org/.

[3] Rapidminer. http://rapidminer.com/.

[4] scikit-learn. http://scikit-learn.org/.

[5] Couchbase Server Under the Hood: An Architectural Overview. White Paper, 2013.

[6] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. "Cut me Some Slack": Latency-aware Live Migration for Databases. In *Proc. of the 15th Int. Conf. on EDBT*, Berlin, Germany, March 2012.

[7] Rick Cattell. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, 39(4):12–27, May 2011.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM Symp on Cloud Comp. (SoCC'10)*, Indianapolis, IN, June 2010.

[9] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based Resource Provisioning in a Web Service Utility. In *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems (USITS'03)*, Seattle, WA, March 2003.

[10] Saurabh Kumar Garg, Srinivasa K. Gopalaiyengar, and Rajkumar Buyya. SLA-based Resource Provisioning for Heterogeneous Workloads in a Virtualized Cloud Datacenter. In *Proc. of the 11th Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'11)-Vol. Part I*, pages 371–384, Melbourne, Australia, October 2011. Springer-Verlag.

[11] Hadi Goudarzi, Mohammad Ghasemazar, and Massoud Pedram. SLA-based Optimization of Power and Migration Cost in Cloud Computing. In *Proc. of the 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid'12)*, pages 172–179, Ottawa, Canada, May 2012.

[12] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, January 2012.

[13] Debessay Fesehaye Kassa and Klara Nahrstedt. SCDA: SLA-aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval. In *Proc. of the 22nd ACM Int. Symp. on HPDC*, New York, NY, June 2013.

[14] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky Computing. *IEEE Internet Computing*, 13(5):45–51, September 2009.

[15] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[16] Minghong Lin, Zhenhua Liu, Adam Wierman, and Lachlan L. H. Andrew. Online Algorithms for Geographical Load Balancing. In *Proc. of the 2012 Int. Green Computing Conf. (IGCC)*, pages 1–10, Xiangtan, China, November 2012. IEEE Computer Society.

[17] Jennie Rogers, Olga Papaemmanouil, and Ugur Çetintemel. A Generic Auto-provisioning Framework for Cloud Databases. In *Workshops Proc. of the 26th IEEE ICDE*, pages 63–68, Long Beach, CA, March 2010. IEEE.

[18] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proc. of the 4th IEEE Int. Conf. on Cloud Computing (CLOUD'11)*, pages 500–507, Washington, DC, July 2011.

[19] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A Cost-Aware Elasticity Provisioning System for the Cloud. In *Proc. of the 31st IEEE Int. Conf. on Distributed Computing Systems (ICDCS'11)*, pages 559–570, Minneapolis, MN, June 2011.

[20] Sebastian Stein, Nicholas R. Jennings, and Terry R. Payne. Provisioning Heterogeneous and Unreliable Providers for Service Workflows. In *Proc. of the 6th ACM Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'07)*, Honolulu, HI, May 2007.

[21] Christopher Stewart, Terence Kelly, Alex Zhang, and Kai Shen. A Dollar from 15 Cents: Cross-platform Management for Internet Services. In *USENIX 2008 Annual Tech. Conf. (ATC'08)*, pages 199–212, Boston, MA, June 2008.

[22] Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. To Move or not to Move: the Economics of Cloud Computing. In *Proc. of 3rd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud'11)*, Portland, OR, June 2011.

[23] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a Petabyte Scale Data Warehouse Using Hadoop. In *Proc. of the 26th Int. Conf. on Data Engineering (ICDE'10)*, pages 996–1005, March 2010.

[24] 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.

[25] Dimitrios Tsoumakos, Ioannis Konstantinou, Christina Boumpouka, Spyros Sioutas, and Nectarios Koziris. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *Proc. of the 13th IEEE/ACM Int. Symp. on Cluster, Cloud, and Grid Computing (CCGrid'13)*, May 2013.

[26] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. SLA-Aware Virtual Resource Management for Cloud Infrastructures. In *Proc. of the 9th IEEE Int. Conf. on Computer and Information Technology (CIT'09)-vol. 2*, pages 357–362, Xiamen, China, October 2009.

[27] V.N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Germany, 1995.

[28] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, and Gregory R. Ganger. Exertion-based Billing for Cloud Storage Access. In *Proc. of 3rd USENIX Conf. on Hot topics in Cloud Computing (HotCloud'11)*, Portland, OR, June 2011.

[29] Linquan Zhang, Chuan Wu, Zongpeng Li, Chuanxiong Guo, Minghua Chen, and Francis C. M. Lau. Moving Big Data to the Cloud. In *Proc. of IEEE INFOCOM'13*, pages 405–409, Turin, Italy, April 2013. IEEE.