Forecasting for Grid and Cloud Computing On-Demand Resources Based on Pattern Matching

Eddy Caron Frédéric Desprez
Adrian Muresan
University of Lyon - CNRS - ENS Lyon - UCB Lyon 1 - INRIA
LIP Laboratory, UMR 5668. Lyon, France
{Eddy.Caron, Frederic.Desprez, Adrian.Muresan}@ens-lyon.fr

Abstract

The Cloud phenomenon brings along the cost-saving benefit of dynamic scaling. As a result, the question of efficient resource scaling arises. Prediction is necessary as the virtual resources that Cloud computing uses have a setup time that is not negligible. We propose an approach to the problem of workload prediction based on identifying similar past occurrences of the current short-term workload history.

We present in detail the Cloud client resource auto-scaling algorithm that uses the above approach to help when scaling decisions are made, as well as experimental results by using real-world traces from Cloud and Grid platforms. We also present an overall evaluation of this approach, its potential and usefulness for enabling efficient auto-scaling of Cloud user resources.

1 Introduction

The evolution of IT software services in the direction of Cloud Computing took a step forward in the efficient use of hardware resources through virtualization. In classical Grids or service oriented platforms, users receive a static amount of hardware resources that they make use of. In contrast to this, the Cloud approach consists in offering on-demand virtualized resources to its users. Because virtual resources can be added or removed at any time during the lifetime of the application hosted on a Cloud, the possibility of dynamic scaling arises. Even more, dynamic application resource scaling can be easily automated either at Cloud provider level or at Cloud client level through the use of the Cloud provider's APIs.

When a Cloud client achieves a more efficient use of his resources through dynamic provisioning, he saves expenses. It is the cost reduction that drives Cloud clients towards automatic strategies for scaling the number of resources up or down depending on the platform usage and current demands.

The main problem with dynamic provisioning is the fact that new resources are not obtained instantaneously, they have a start-up time that is not negligible. In essence, the Cloud client has two possibilities of addressing this problem if on-demand provisioning is applied: either delay handling client requests until resources become available or simply drop client requests. In either case, the quality of service of the Cloud client is reduced significantly. To address this problem, current strategies use a predictive approach in hopes of getting an insight into future platform usage and, as a consequence, being able to make scaling decisions ahead of time, compensating for the resource start-up time.

The idea of self-similarity in web traffic is not new [3]. Based on this, a new Cloud client application resource auto-scaling strategy can be elaborated. By identifying resource usage patterns that have occurred in the past and have a high similarity to the present resource usage pattern, a decision can be made as to the necessity and/or direction of scaling for the present situation. This paper presents an approach to the resource usage prediction problem based on identifying past resource usage patterns that are similar to the present use of the system. We present an efficient algorithm for identifying the patterns by using an approximate matching approach.

In Figure 1 we have a generic Cloud system usage model to have a top-level view on the role of the prediction model. As part of a Cloud client's resource management module, the prediction module uses the Client's resource usage history to try and make an intelligent guess on short-term resource demands. This alone does not constitute the Client's scaling decision as there are a number of other relevant factors that should be taken into consideration like the migration of currently running tasks from virtual resources that need to be terminated. In our current work, we are focusing

978-0-7695-4302-4/10 $26.00 © 2010 IEEE
DOI 10.1109/CloudCom.2010.65

only on resource usage prediction. The impact that other factors have on the scaling decisions of a Cloud client is an interesting topic of research, yet it is beyond the scope of our current work. The main contribution of this paper is the elaboration of a resource usage prediction algorithm based on pattern matching with the goal of aiding Cloud clients in making automatic application resource scaling decisions.

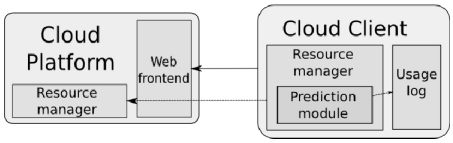

Figure 1. The role of the prediction component in a generic model of a Cloud system usage scenario.

The rest of this paper is organized as follows. Next section presents an overview of existing approach given in the literature. Then, Section 3 presents our algorithm and its key design principles. Finally, before a conclusion and a description of future work, Section 4 presents our experimental results using actual Cloud and Grid traces.

2 Related Work

There are currently two main approaches to facilitate the auto-scaling decisions of Cloud clients as a result of resource usage. The first approach treats the past server usage as a predictable sequence and constructs a mathematical model around it. As a result, the next value of the request sequence is obtained by evaluating the obtained model at the next time point. In other words, a prediction model is built by considering past resource usage. The second approach is a reactive one, based on the current server load and auto-scaling rules that are set up by a human operator (usually a Cloud client). This approach has been often referred to as the "Elasticity rules" approach or the "SLA" approach [5].

In [10], a description and comparison of three different auto-scaling algorithms is given: Auto-Regression of order 1 (AR1), Linear Regression, and the Rightscale algorithm. The AR1 algorithm belongs to the first category of auto-scaling algorithms. Its approach consists in using a finite history window and identifying appropriate parameters so that a recurring sequence can be obtained and therefore used to calculate the next values. The parameters obtained are adapted as the window slides along the time axis. The Linear Regression algorithm belongs to the first category and calculates a polynomial approximation of the history of requests. The predicted value is then obtained by evaluating the polynomial at a higher point along the time axis. The Rightscale algorithm belongs to the second category, being a version of threshold-based auto-scaling. Its approach consists in using a democratic voting system based on the current server load. Each virtual machine owned by the Cloud client has a vote based on its current load level and two thresholds: low threshold that corresponds to a "scale down" vote and a high threshold that corresponds to a "scale up" vote. The votes are collected by a central machine and the majority decides the scaling decision for the whole platform. The three algorithms have been put side-by-side and compared by a metric proposed in the same article. Their performance is considerably high.

A more complex form of SLA-based dynamic provisioning can be described by using elasticity rules that dictate what part of the Cloud client needs to scale, in which direction and by how much. In [5], we find such an example with threshold-based rules. This is done by means of an extension to the OVF (Open Virtualization Format), an interoperable, platform and vendor neutral, open format that is used to describe VAs (Virtual Applications). VAs are preconfigured software stacks consisting of one or several Virtual Machines with the purpose of offering self-contained services. The OVF documents have three components: an associated name, a trigger condition based on the defined key performance indicators and an associated action that represents the implementation of the rule in the form of instantiating new components of the VA or removing existing component instances. Like the Rightscale algorithm, this approach is also a reactive one. Scalability rules have the benefits of combining the high performance of threshold-based algorithms such as Rightscale with tune-ability and therefore have been widely used in practice in commercial Clouds.

In [13], a Decentralized Online Clustering model is described and proposed for automatic workload provisioning for enterprise Grids and Clouds and addresses their distributed nature. In this approach a workload prediction algorithm is used and integrated into the system to model the application dynamics. More specifically, a quadratic response surface model is used.

The ideas of workload prediction and workload modeling are by no means new, in fact they have been active areas or research in the field of Grid computing. In [8], we find a fine-detailed study on the topic of Grid performance evaluation by using synthetic workloads obtained from the modeling of Grid workloads. The work describes performance metrics useful to evaluate Grid environments. These are composed of traditional performance metrics that are time, resource or system related and Grid-specific related to workload completion or failure metrics. The paper continues by describing the specifics of Grid workload modeling. These include user group modeling that underlines the importance of taking into consideration statistics for all jobs on one hand and statistics for each user in particular on the other hand, based on his (or her) past actions. The paper also describes submission patterns that arise in Grid en-

vironments and enumerate some of the current approaches of modeling them that include combining Poisson distributions for daily patterns or by using a polynomial function of degree eight. The authors argue that these pattern modeling approaches may not hold as they are indifferent to workload inter-dependency. The authors continue by presenting the GRENCHMARK synthetic Grid workload generation, execution and analysis framework [6].

In [7], we find an integration effort of a Grid application development toolkit named Ibis [15], a Grid co-scheduler named Koala [11] and the GRENCHMARK [6] synthetic Grid workload generator with the purpose of providing an end-to-end workload generation and testing framework. The authors argue about the benefits that experimental testing of Grid systems has over an analytical or simulated test model. The authors also argue in favor of using synthetic Grid workloads over real Grid workloads or benchmarking approaches. Next the authors describe their integration proposal of building applications with the Ibis toolkit, generating and submitting synthetic workloads with GRENCHMARK, and then scheduling them with Koala so that the results can be analyzed with GRENCHMARK again.

A non-linear model for Grid workload prediction can be found in [4]. The authors propose a prediction model as a series of finite known functional components, usually taken from the sigmoid function class, with unknown coefficients. The coefficients are determined by using using the least square approximation method on a training set. Their model has been tested on a 3D image rendering set of tasks based on the Blue Moon Rendering Tool. The error of their prediction is less than 14% with an average of 7.5%.

In [12], we find a real-time resource provisioning system for massive multiplayer online games based on a predictive usage model. The application is dynamically provisioned on a Grid environment. The authors propose a predictive model based on a neural network. It is prepared with two offline phases that include gathering of training samples and using them to train the neural network. As results of experimentation, the neural network approach has prover to have a greater accuracy when compared to the other tested prediction methods: average, moving average, last value and exponential smoothing. The obtained prediction error during the experiments has a maximum value of 33% and a minimum value of 4.94%.

A shortcoming of existing reactive approaches is the fact that they are blind relative to the trend of the workload they are predicting as they do not consider the recent workload states for their results. In contrast, predictive approaches do consider trend, but the presented approaches that are using mathematical models do not consider arbitrarily-repetitive self-similarities. It is this motivation that led us to the current work.

3 String Matching based Scaling Algorithm

3.1 The String Matching concept

The usage of a Cloud client can sometimes have a repetitive behavior. This can be caused by the similarities between tasks that the Cloud client is running or the repetitive nature of human behavior. Given the self-similar nature of web traffic, it follows that current usage patterns of online services have a probability of having already occurred in the past in a very similar form. Therefore we can infer what the system usage will be for a Cloud client by examining its past usage and extracting similar usages.

The pattern strategy has two inputs: a set of past Cloud client usage traces and the present usage pattern that consists of the last usage measures of the Cloud client. Cloud clients working in the same application domain have a higher similarity in resource usages. Due to this similarity it follows that the most relevant historic resource usage data that can be used comes from Cloud clients working in the same application domain. Therefore it would make sense to isolate historical data based on application domains before usage.

The present usage pattern of the Cloud client is used to identify a number of patterns in the historical set that are close to the present pattern itself. Identified patterns should not be dependent on their scale, just on the relation between the elements of the identified pattern and the pattern we are looking for. The resulting closest patterns will be interpolated by using a weighted interpolation (the found pattern that is closest to the present pattern will have a greater weight) and will have as result an approximation of the values that will follow after the present pattern. In essence, the usage of the Cloud client is predicted by finding similar usage patterns in the past or in other usage traces.

The problem of finding a pattern inside an array of data that is very similar to a given pattern is close to the problem of string matching. The approximate string matching problem has been widely studied especially in its relation to bio-informatics problems, yet it is considerably different from the problem we are addressing.

One definition for the approximate string matching problem is the following. Given a text string $T = t_0 t_1 ... t_n$ and a pattern $P = p_0 p_1 ... p_m$ find a substring of consecutive characters from T call it $T_{i,j}$ that has the smallest edit (or Levenshtein) distance as possible [1].

The edit distance is defined as the number of simple string operations (insert, delete, replace and sometimes exchange) that need to be performed on the identified text substring to have equality to the pattern. The operations can have the same or different weights, depending on the problem needs. The identified match can have any length because of the possible insert and delete operations.

For the problem that we are addressing, the edit distance cannot be applied as we are not comparing string character values, but floating point values. We are interested in identifying sub-arrays of the same, or very close, length and whose floating point absolute value difference is as close as possible to zero. An insertion into or deletion from the identified sub-array would have a great impact on the floating-point difference.

We shall now describe the problem of string matching and its relation to the problem that the current paper addresses, as well as our proposal for the approximate variant that is relevant to our problem.

3.2 The String Matching Algorithm

There are several solutions to the string matching problem. We have chosen the Knuth-Morris-Pratt (abbreviated KMP) algorithm as its performance are good (as described in [2]). Despite the great similarities, our own pattern matching problem has some particularities of its own. An approximate matching is needed since the odds of finding an identical pattern to the one we are looking for are considerably low. The found candidate matches which are too dissimilar either on small intervals or as a whole need to be discarded. The resulting matches should then be interpolated having different weights on the final result, based on their similarity to the identified pattern.

In order to do an approximate matching, the original KMP algorithm needs to be changed in the content of both functions, therefore they need to be modified accordingly.

Two types of approximation errors are used for the matching:

1. an instant error which dictates the amount by which the current match is allowed to differ from the pattern by comparing in smallest possible units. In our pseudocode, this is returned by the $Distance()$ function.

2. a cumulative error that characterizes the amount by which the current match is allowed to differ from the pattern as a whole. This is basically a sum of the instant errors of the whole matching and is returned by the $CumulativeDistance()$ function in our pseudocode.

The distance between the pattern we are trying to match and a candidate pattern should be computed in a scale-independent manner by first normalizing the two pattern values to a common scale. To decrease floating point approximation errors, one can choose a distance computation that does not use divisions and therefore calculating only on integer values.

3.3 KMP Modification

Algorithm 1 Calculate-prefix(P, ER_INST)

```
1:  m ← length(P)
2:  π[0] ← -1
3:  k ← -1
4:  scaleK = P[0]
5:  scaleQ = P[1]
6:  for q ← 1 to m - 1 do
7:     dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
8:     maxDistance ← ER_INST × scaleQ × P[k+1]
9:     while k > -1 and dist > maxDistance do
10:       k ← π[k]
11:       dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
12:       scaleQ = P[q - (k+1)]
13:    end while
14:    if dist ≤ ER_INST × scaleQ × P[k+1] then
15:       k ← k+1
16:    end if
17:    π[q] ← k
18: end for
19: return π
```

The prefix calculation function is changed as described in Algorithm 1. The scales of the two components compared are represented by the first value of each component. This is arguable, but in practice we have achieved good results with this approach. In the function, $scaleK$ represents the scale of the prefix and $scaleQ$ represents the scale of the post-fix of the pattern. The $Distance()$ function returns an appreciation of the distance between two different pattern instances, each having a different scale which is passed as parameter. The comparisons on lines 9 and 14 guarantees that the current instant distance does not differ by more then the acceptable error (in percentage) from the actual pattern that we are matching. In the comparison on line 14, the $scaleQ$ term represents the scale of the data. It is needed in order to bring the pattern to the same scale as the data.

The matching algorithm is changed as described in Algorithm 2. Compared to the original KMP algorithm, the main difference is the use of the instant and cumulative distances as a mean of filtering out potential matches that are too different either on small time intervals or as a whole.

On lines 10 and 16 we guarantee that the instant distance between the identified candidate and the pattern is no more than what the acceptable error permits. In order to guarantee a correct comparison, the pattern term needs to be scaled to the same size as the data, hence the $scaleT$ term is used in the comparison. Filtering by cumulative distance is done in lines 20 to 24. The $CumulativeDistance()$ function returns a sum of instant distances for every instant of the two compared arrays. The running time of this function is $\Theta(m)$ where m is the length of the arrays, which in our case is always equal to the length of P. Line 22 of the algo-

Algorithm 2 KMP-approx(T, P, ER_INST, ER_CUMUL)

```
1: n ← length(T)
2: m ← length(P)
3: π ← Calculate-prefix(P)
4: q ← -1
5: scaleP = P[0]
6: scaleT = T[0]
7: for i ← 0 to n - 1 do
8:    dist ← Distance(P[q+1], scaleP, T[i], scaleT)
9:    maxDist ← ER_INST × scaleT × P[q+1]
10:   while q > -1 and dist > maxDist do
11:      dist ← Distance(P[q+1], scaleP, T[i], scaleT)
12:      q ← π[q]
13:      scaleT = T[i - (q+1)]
14:      maxDist ← ER_INST × scaleT × P[q+1]
15:   end while
16:   if dist ≤ maxDist then
17:      q ← q+1
18:   end if
19:   if q = m-1 then
20:      dist ← CumulativeDistance(P, T, i - m + 1)
21:      maxDist ← ER_CUMUL × patternSum × scaleT
22:      if dist ≤ maxDist then
23:         StoreSolution(dist / scaleT, i - m + 1)
24:      end if
25:      q ← π[q]
26:      scaleP = P[q+1]
27:      scaleT = T[i - (q+1)]
28:   end if
29: end for
```

rithm assures that the cumulative distance of the candidate does not differ more than is accepted by the cumulative error from the pattern itself. The pattern itself is represented by the $patternSum$ term in the comparison. This is a sum of all the terms in the pattern and should be calculated only once, at the beginning of the algorithm. The pattern sum needs to be brought to the same scale as the candidate sequence and therefore the $scaleT$ term is used. Filtering by an acceptable cumulative error that is smaller or equal to the acceptable instant error is useless. This conclusion is trivial when taking into consideration that the cumulative error is a sum of all the instant errors. The use of the cumulative error changes the running time of the matching algorithm to $\Theta(n \times m)$ in the worst case, where n is the length of the string to match against and m is the length of the input pattern.

Once approximate matches have been found, the problem of obtaining a relevant result from those matches is raised. Each match should have a contribution to the final result that is proportional to its relative distance to the pattern with respect to the other identified patterns. This corresponds to a weighted sum of the identified matches, where weights are calculated by considering the distance of the current match to the pattern and to the rest of the matches. Once the weights are calculated, the interpolation is performed between the following L elements after each approximate match. The result is a predicted sequence of length L.

3.4 Algorithm parameters

The algorithm accepts a number of parameters used for fine-tuning in accordance to each use-case. These parameters are:

- The maximum number of matches (called closest neighbors) to take into consideration (denoted K).
- The length of the predicted sequence (denoted L).
- The acceptable instant error representing the amount by which the identified sequence is allowed to differ on the smallest possible interval lengths from the pattern we are looking for.
- The acceptable cumulative error which represents the amount by which the identified sequence is allowed to differ as a whole from the pattern we are looking for.
- The input set of data representing the database of past requests.
- The input pattern representing a sequence with the last period of requests received.

The first parameter is actually influenced considerably by the acceptable errors: the larger the acceptable error, the more matches the algorithm identifies, but the more irrelevant they will be.

The value of the acceptable errors can be calculated based on the maximum number of neighbors that we wish to find. The approach for this is to use a binary search to zone in on the appropriate values for the acceptable errors. By using the binary search approach, we have obtained values that have proved to be good in practice. We have used a lower bound of 20% of K for a minimum of identified neighbors and 90% of K as the upper bound for maximum number of identified neighbors.

The length of the pattern that represents the last traces of server usage has a great impact on the results of the algorithm. Finding the appropriate length is a problem in its own as we have a trade-off between patterns of big lengths that yield a small number of similar candidates, that might be too small in order to be usable, and patterns of small lengths, that find more candidates but they tend to be more irrelevant to our current situation. We have used with good results, pattern lengths that are close to the average or the median length of jobs found in the historic workloads.

4 Experimental Results

To validate our model we have used real-world traces from one Cloud client platform and as a supplement we have also used traces from production Grids as these are the environments that are closest to actual Clouds from the point of view of resource allocations. Production Grids, like Clouds, offer resources on-demand as a primary resource allocation mechanism. The reason for including Grid traces in our experimentation is the fact that there is no public archive of traces from Cloud platforms and this makes obtaining real-world trace a difficult task. This is also the reason for a disproportionate presence of Cloud data in our experimentation. The only possible source then is the commercial sector that is subjected to severe constraints as the traces reflect their customer's private platform usage information.

In all our experiments, we have used a time unit of 100 seconds as a discrete step. The predicted traces represent the total number of CPUs used by different jobs running in parallel in the time unit of 100 seconds. We have focused only on CPU usage as the information of memory usage was not available. Nevertheless, should the information of memory usage be available our approach can also be applied for its prediction.

4.1 Data Sources

We have tested our auto-scaling approach with traces from one Cloud client and three different Grids, each having its own usage particularities, with main differences in the frequency and amplitudes of changes in their overall usages.

Animoto [1] is a Cloud client application that specializes in automatically-orchestrated videos starting from user-generated content. Their platform usage represents oscillations as per user activity.

LCG [2] **(Large Hadron Collider Computing Grid)** contains traces from several nodes from the computing Grid associated to the Large Hadron Collider. Its behavior is mildly oscillatory.

NorduGrid [3] has higher amplitudes for oscillations as the Grid is more heterogeneous than the previous.

SHARCNET [4] has been described as a "cluster of clusters". Its volatility is very high and its amplitudes can reach surprising peaks.

4.2 Analyzing the data sources

We need to have a better way to choose the pattern length, that would give more relevant results and avoid pollution as much as possible. The length of the pattern should be influenced by the time it takes to service a request on the server.

	Animoto	LCG	Nordugrid	Sharcnet
Avg	1296	8970	91893	33516
Min	4	0	0	0
Median	283	255	3861	12165
Max	22452	586702	1452763	7449415

Table 1. Job length statistics for the data sources. Values represent time in seconds based on the running time of the recorded jobs.

By analyzing the data sources from Animoto and the workload archive of TUDelft University [14] we have obtained the running time in seconds of each job with the results given in Table 1. The conclusions here are that, for all practical purposes, a pattern length that is a minimum or even a median of the time it takes for a job to be run, is unusable when dealing with servers that have a similar usage to the Cloud application or research Grids described above. In practice we have used the average of the request service time and have obtained good results.

4.3 Experiment setup

All the experiments use the server traces with the same form of input data as described above with time units of 100 sec., and resource usage value consisting of the total number of CPUs used across the 100 sec. A pattern length of 100 time units has been used for all the experiments (this is 100 $\times$ 100 seconds - approximately 2.7 hours of server time) and predictions are made for one time unit, this is 100 seconds, which is a little over 1 min 30 seconds.

The results are displayed under the form of a set of standard metrics that include minimum, maximum, median, and average percentage and value difference between the prediction and the actual value.

A second set of metrics has also been used that allows the comparison to other existing auto-scaling algorithms. This metric was proposed and used by UCSB to compare the performance of three existing auto-scaling algorithms [10]: AR1, Linear Regression and the Rightscale democratic voting algorithm.

We have also measured the average running time necessary for calculating one prediction. This has an impact on the practical usefulness of the prediction since it needs to be subtracted from the prediction time, which is 100 seconds, to calculate the effective prediction time.

The metric proposed by UCSB is influenced by platform availability and cost by the following formula:

$$\frac{(A_{log})^{\alpha}}{C} - \frac{\gamma C}{A_{log}} + \beta \qquad (1)$$

where: $A = \#serviced\_requests/\#of\_requests$ repre-

[1] http://animoto.com
[2] http://lcg.web.cern.ch/LCG/
[3] http://www.nordugrid.org
[4] http://www.sharcnet.ca

sents the availability of the platform,

$$A_{log} = -log(1 + \delta_a - A), \delta_a < 1 \quad (2)$$

represents the availability in logarithmic scale and $C = \#CPU/(hours \times 0.10)$ represents the cost. The constants α, β, γ and δ_a have been chosen through experimentation.

We have used two versions of the metric proposed by the UCSB team: an instant score where we considered resource cost as being charged per fraction of an hour, although this is not the case in current Cloud providers and a second score where we take the maximum prediction over the course of an hour and use that as static provisioning for the whole hour.

4.4 Results

We have done two types of tests: self-prediction in which a resource trace is used as historic data to predict itself, ignoring exact matches and cross-prediction tests in which the resource trace of one source is used to predict the resource trace of another source. The results can be seen in Table 2.

For the sake of comparison we can consider results of evaluating three other algorithms obtained in [10]. These values are taken from experimentation against a randomly generated usage pattern and feature the variants of all three algorithms that implement the Smartkill strategy, thus making them more effective. Only measures of the UCSB metric (instant) were available:

- RightScale: 11.11
- Autoregression of order 1: 17.3
- Linear Regression: 10.8

It is worth noting that although the maximum prediction error is high, the median error is considerably low (between 0.9% and 4.08% in all tests). This leads us to believe that this is a feasible approach to auto-scaling. It is clear that the algorithm yields better results when the source of the historic data that is used has a similarity to that which is being predicted. This similarity is influenced by several parameters that constitute the domain of the server whose load is being predicted. It follows from the obtained results that data from the same domain can easily be used to predict one-another.

When compared to other approaches, we have obtained both better and worst results, depending on the algorithm parameters. This leads us to believe that the algorithm itself can yield better results in practice if its parameters are calibrated accordingly. Of most importance are the pattern length and the relevance of the domain of the historic data with respect to the domain of the platform whose usage is being predicted.

According to [9], the total average time necessary to obtain an Amazon EC2 instance of type `m1.small` is 82 seconds. In combination with the prediction look-ahead of 100 seconds, the running time of the algorithm obtained in practice can be neglected (between 28 ms and 518 ms).

We have experimented with various lengths of the historic data set and of the pattern that is considered for input. The results with the prediction error in each case can be seen in Table 3. Although this does not show that the algorithm yields the best possible results, it does show that there is a clear tendency for the accuracy of the prediction to improve as we increase the size of the historic data and as we find the best pattern length to take into consideration when predicting. The results table illustrates results when varying the pattern length and the length of the historic data used for prediction. We have varied the historic data from 100% - the full set, to 50%, 25% and 12.5% of the set. The pattern length has also been varied from 1000 time units to 500, 100, 50, 25, 12 and 2 time units.

Pattern length	**Data length**			
	100.0%	50.0%	25.0%	12.5%
1000	5.3%	9.5%	19.7%	100%
500	3.7%	6.0%	8.6%	18.7%
100	1.0%	1.2%	1.3%	2.0%
50	0.6%	0.5%	0.9%	1.3%
25	0.3%	0.3%	0.4%	0.5%
12	0.2%	0.2%	0.2%	0.3%
2	98%	100%	100%	82%

Table 3. The prediction error obtained for various lengths of historic data and pattern lengths for the LCG platform.

The reader will note that in our experiments we have considered only CPU usage as measure and prediction target. In a Cloud environment, a virtual resource usually has more characteristics associated to it than just CPU power. In particular, memory usage is one of the most notable characteristics. Our approach can also be used to have a prediction of the memory usage if the server traces also contain information about past memory usage. With predictions for both memory and CPU usages, the scaling component of the Cloud client should be able to more accurately decide the characteristics of the virtual resources that are to be instantiated or released. The topic of making a good scaling decision both in direction and in virtual machine characteristics is an interesting topic of research, yet it is beyond the scope of the current work.

5 Conclusions and future work

One of the most important benefits of Cloud Computing is the ability for Cloud clients to adapt the number of resources used based on their actual use. This has great implications on cost saving as resources are not paid for when they are not used. Dynamic scalability is achieved through virtualization. The downside of virtualization is that it has a

Metric	A w/ A	A w/ L	L w/ L	L w/ N	N w/ L	S w/ N
Min prediction error (%)	0.0	0.0	0.0	0.0	0.0	0.0
Max prediction error (%)	100	856.87	53.4	100.0	1146.00	528.03
Med prediction error (%)	2.69	4.08	1.0	1.2	1.74	0.9
Avg prediction error (%)	5.42	7.4	1.749	7.32	35.38	375.65
UCSB metric (max per hour)	-1.39	-15.95	10.66	3.43	30.64	-3.23
UCSB metric (instant)	-18.38	-38.75	-2.68	-10.71	27.27	-2.06
Avg runtime per instance (ms)	186.625	27.63	41.734	514.956	162.949	528.418

Table 2. Results of prediction experiments with traces from the four data sources: Animoto, LCG, Nordugrid and SHARCNET. Experiments consist of predicting one platform's usage with another platform's traces as historic data, across time slices of 100 seconds. The naming of the columns is done after the following convention **A w/ B** where A is the platform whose usage is being predicted and B is the platform whose trace is being used as historic data.

non-zero setup time that has to be taken into consideration for an efficient use of the platform. It follows that an accurate prediction method would greatly aid a Cloud client in making its auto-scaling decisions.

In this paper, a new resource usage prediction algorithm is presented. It uses a set of historic data to identify similar usage patterns to a current window of records that occurred in the past. The algorithm then predicts the system usage by interpolating what follows after the identified patterns from the historical data. Experiments have shown that the algorithm has good results when presented with relevant input data and, more importantly, that the quality of results can improve by increasing the historic data size. The running time of the algorithm has proved negligible in our experiments, which makes it a good candidate for practical use.

As future work directions, we will be looking into ways that a relevant set of historic data can be composed for a particular application domain.

Acknowledgments: This work was developed with financial support from the ANR (Agence Nationale de la Recherche) through the SPADES project referenced 08-ANR-SEGI-025. The authors would like to thank Animoto and The Grid Workload Archive for the high-quality workload traces that they provided.

References

[1] W. I. Chang and T. G. Marr. Approximate String Matching and Local Similarity. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 259–273, London, UK, 1994. Springer-Verlag.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms, Chapter 32: String Matching*. McGraw-Hill Higher Education, 2001.

[3] M. Crovella and A. Bestavros. Explaining World Wide Web Traffic Self-Similarity. Technical Report TR-95-015, Boston, MA, USA, 1995.

[4] N. Doulamis, A. Doulamis, A. Litke, A. Panagakis, T. Varvarigou, and E. Varvarigos. Adjusted fair scheduling and non-linear workload prediction for QoS guarantees in grid computing. *Computer Communications*, 30(3):499 – 515, 2007. Special Issue: Emerging Middleware for Next Generation Networks.

[5] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, and L. M. Vaquero. Service specification in cloud environments based on extensions to open standards. In *COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, pages 1–12, New York, NY, USA, 2009. ACM.

[6] GrenchMark. http://grenchmark.st.ewi.tudelft.nl.

[7] A. Iosup, , R. Iosup, D. H. J. Epema, J. Maassen, and R. V. Nieuwpoort. Synthetic grid workloads with ibis, koala, and grenchmark. In *In Proceedigs of the CoreGRID Integrated Research in Grid Computing*, 2005.

[8] A. Iosup, D. H. J. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, and R. Yahyapour. On grid performance evaluation using synthetic workloads. In *JSSPP'06: Proceedings of the 12th international conference on Job scheduling strategies for parallel processing*, pages 232–255, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE TPDS (in print)*, November 2010.

[10] J. Kupferman, J. Silverman, P. Jara, and J. Browne. Scaling Into The Cloud. `http://cs.ucsb.edu/~jkupferman/docs/ScalingIntoTheClouds.pdf`, 2009.

[11] H. Mohamed and D. Epema. Koala: a co-allocating grid scheduler. *Concurr. Comput. : Pract. Exper.*, 20(16):1851–1876, 2008.

[12] R. Prodan and V. Nae. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems*, 25(7):785 – 793, 2009.

[13] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards Autonomic Workload Provisioning for Enterprise Grids and Clouds. In *Proceedings of the 10 th IEEE/ACM International Conference on Grid Computing (Grid 2009)*, pages 50 – 57, 2009.

[14] T. U. The Grid Workloads Archive. http://gwa.ewi.tudelft.nl.

[15] R. V. van Nieuwpoort, J. Maassen, G. Wrzesińska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):1079–1107, 2005.