Advanced Systems Fall 2015, Cornell University
Dipendra K. Misra (dm644)

SIC: Workload Anticipation for Resource and Budget Optimization in Cloud Computing

Abstract

In this paper, we describe SIC (*Smart Inferface to Cloud*) a middleware that optimizes resources and budget in cloud computing. SIC does this by predicting workload and accordingly planning actions such as launching an instance, uploading a lambda, closing an instance. SIC is meant to be a client side solution that is compatible across different cloud services. Currently SIC supports amazon aws. We argue for useability of SIC for various algorithms specifically AI algorithms which often have predictable workload. We show the promise of SIC by evaluating on a synthetic task, where SIC obtains a compromise between optimizing budget and waiting time.

1 Introduction

Several applications today use cloud computing for its convenience as an on-request service. This has been made possible by several cloud services such as *amazon aws*, *microsoft azure*, *google gcp* etc. A standard way that applications use cloud is to request for instances (atoms of computation) on which the workload of the application is transferred. However, requesting for an instance can take substantial time (upto several minutes for amazon aws) whereas keeping an instance open consumes budget. In this paper, we attempt to solve this dichotomy by proposing workload prediction in order to plan the possible actions that can be taken to reduce the waiting time for an instance and optimizing the budget spent. In the ideal scenario when predictions are perfect the application will never have to wait for an instance and will only pay when an instance is being used for computation (which is also the optimal budget consumption for that instance).

Our solution is applicable to applications where workload can in principle be anticipated. In particular, we are interested in AI algorithms such as machine learning algorithms where this is often true. Example, several learning algorithms perform training epochs where every epoch requires performing inference over a batch. The inference can be performed on cloud either to benefit from specialized hardware such as GPU clusters or to exploit parallelism by opening several instances.

There can be other subtle patterns in workload. Example a house that follows human instruction (Stefanov et al., 2004), can close instances after midnight. An autonomous vehicle (Urmson et al., 2008) can transfer part of its workload to cloud when driving through a congested region.

Our solution is implemented in the form of a middleware called SIC (Smart Interface to Cloud) between the application and cloud vendor. SIC maintains statistics on booting/halting time for an instance, time to upload lambdas, execution time, call patterns etc. It uses polynomial regression to fit the data and predict the next call pattern. The model is used to estimate the time it will take to ready an instance with lambdas for the application and accordingly it schedules a *launch* and *lambda upload* operation. SIC is compatible with different cloud services and supports lambdas in any generic executable format. We now give an overview of our system architecture.

2 Overview of System Architecture

The system architecture consists of a SIC object which is created for every application, which accepts commands from the application and communicates with the cloud via a service class. The service class must implement an interface which defines basic functions such as opening and closing an instance, uploading files etc. FTP operations are performed using Jsch library.

The application can send two type of requests to the SIC object: executing a lambda and updating storage. A lambda for us is any executable image which can be executed on a given parameter without requiring any other dependencies.

SIC maintains datastructures which keeps track of the time when the application made a given request, the time taken to open and close instances of different type, time taken to upload a given lambda to an instance type and time taken to execute the lambda on a given parameter on an instance type.

SIC is oblivious to the algorithmic structure and only attempts to find patterns in the call issued by the application to SIC. Thus, SIC is targeted at optimizing loops in application where there is some periodicity of calls.

This however makes sense since loops are often the performance bottlenecks. In future, programming analysis can help SIC to optimize arbitrary program control flow.

We now review the related work in this area.

3 Related Work

We aim to study workload prediction for optimizing budget and remote resources in cloud. This brings together the following themes: interface to cloud architecture, anticipating workload, optimization of remote resources, optimization budget and target applications. We now review the related work in these areas.

Interface to Cloud Architecture: Cloud computing has been a popular choice for computation in past decade. The main advantage is the ease of sharing easily configurable systems, available as per demand. As such several cloud providers are available such as Amazon aws[1], Microsoft Azure [2], Google GCP [3], IBM SmartCloud[4], Oracle Cloud [5] etc.

A great many of these solutions provide rich command-line interface which can be used by the resource-optimization layer in the proposed system for launching instances, file transfer and calling remote functions. Example, one can launch a remote instance using amazon CLI tool as follows:

```
ec2-run-instances  ami-xxxxxxxx  -t  t1.micro  -k  my-key-pair
-g  my-security-group
```

This command launches a remote instance of type `t1.micro` with specific key-value pairs (such as name of instance) and security permissions. AMI (amazon machine images) contains other permissions and device information. These tools are available as SDK for different programming languages such as Java, C#, python etc.

Some solutions such as Amazon aws provide Lambdas (stateless functions) which can be used in machine learning applications for performing

[1]https://aws.amazon.com/

[2]https://azure.microsoft.com/en-us/

[3]https://cloud.google.com/

[4]http://www.ibm.com/cloud-computing/social/us/en/

[5]https://cloud.oracle.com/home

fast computations by running lambdas in parallel. However, SIC does not depend upon cloud specific solutions.

Workload Anticipation: Workload anticipation is an old optimization technique in systems. A large body of literature exists on algorithms for predicting workload to prevent bottlenecks and to ready the resources for the expected workload. Anticipatory scheduling (Iyer and Druschel, 2001) is a well known algorithm for scheduling disk I/O that anticipates future disk reads; and was the default Linux scheduler(between 2.6.0 and 2.6.18). Machine learning offers several models for predicting time series, with recurrent-neural network (RNN) being a particularly powerful choice. For example, RNN can learn that arrival time has a $O(n^3)$ dependency for load of size n. Bensch et al. (2007) use neural networks (ANN) and support vector regression (SVR) to predict the workload for a mainframe operating system. Several works (Doulamis et al., 2007; Cao et al., 2003) have looked at predicting workload in a grid. However, these works are not focusing on workload anticipation in a cloud.

(Khan et al., 2012) discover workload patterns in cloud for analyzing group-level workload characteristics. For this they cluster VMs that exhibit similar workload patterns. In contrast, SIC is meant to find patterns in volatile instances (even one) over time for optimizing budget and reducing waiting time for applications. In a recent work (LaCurts, 2014) consider more sophisticated optimization in cloud by predicting application workload and learning to place applications on machine in the cloud. In contrast, SIC is meant for optimization of applications outside-the-cloud where the cloud is used intermittently in contrast to placing the entire application on the cloud. Also (LaCurts, 2014) do not seem to take budget into factor.

While not performing workload prediction, CloudProphet Li et al. (2011) tries to predict performance of legacy applications on different cloud infrastructure to find the best performing infrastructure. SIC instead makes optimization decision for a given cloud service infrastructure.

Others (Ganapathi et al., 2009; Akdere et al., 2012) have focused on predicting the execution time of a query. Duggan et al. (2013) in particular look at the problem of workload performance prediction in a cloud. While predicting workload execution time is not a problem when a single application is using the instance, this becomes an important problem when the instances are shared between different applications. Moreover, the prediction algorithms can be used for both predicting job arrival time as well as predicting their execution time.

Remote Resource Optimization: Optimizing remote resources is important to ensure that an application is not starved of resources, prevent adversarial nature and also to indirectly optimize budget. Therefore the natural question arises on optimizing shared resources such as memory, time; in a fair way. Recent work have proposed solutions to this problem. Chaisiri et al. (2012) propose a stochastic programming model formulation called optimal cloud resource provisioning (OCRP) algorithm. Di and Wang (2013) use a cubic complexity dynamic optimal proportional-share (DOPS) resource allocation algorithm. Their key solution involves dynamically scaling the resources proportional to the demand of each tasks. Wei et al. (2010) consider a game-theoretic solution to allocating cloud resources.

Some others have looked at optimizing resources for specific algorithms or class of algorithms. Palanisamy et al. (2015) propose a cloud-service model for MapReduce called Cura by optimizing resources globally(entire cloud). Cura provides a VM-aware scheduler and also handles error in predicting when a job will terminate. Tirapat et al. (2013) optimize scientific workflows for cloud computing, to minimize resource usage and ensure that workflows terminate on time. Their scheduling algorithm is based on GA (Genetic algorithm) and PSO (particle swarm optimization).

Budget Optimization: We are particularly interested in applications that aim to minimize the monetary cost of expensive services. Mao et al. (2010) look at the problem of scaling cloud to satisfy budget and deadline constraint. While this seems as a viable direction, we do not want to only rely on scaling cloud for meeting constraints. Zhu and Agrawal (2010) formulate the problem as an optimization problem using control theory, and show its performance on an experimental cloud.

Van den Bossche et al. (2010) schedule workloads to different cloud-provider in a cost-optimal manner by formulating it as a binary-integer program. In contrast, we are chiefly interested in using a single cloud provider and maintaining budget below a certain threshold. Liu et al. (2010) propose a compromised-time-cost scheduling algorithm for instance-intensive workflows with budget constraint.

Others such as Andrzejak et al. (2010) look at the problem of predicting the bidding price for a given cloud resource using a probabilistic model whereas we are interested in optimizing resource for a given budget.

Target applications: As mentioned in the introduction, our target applications are AI algorithms such as machine learning algorithm that need to perform expensive inference on cloud, for several epochs.

These learning algorithms can involve syntactic parsing of document (Dyer et al., 2015), feed-forwarding batch through a deep neural network, stochastic gradient descent (Mitchell, 1997), POS tagging (Jurafsky and Martin, 2000), classifying documents by its sentiment (Liu, 2012), semantic parsing of a document (Zettlemoyer and Collins, 2007).

Learning algorithms can perform expensive inference over cloud, either to benefit from special hardware such as GPU or to exploit parallelism (batch processing), and then use the results on the local computer. SIC can help these applications by predicting when they will be needing the cloud and prelaunching the instances with required lambdas to reduce waiting time while optimizing budget.

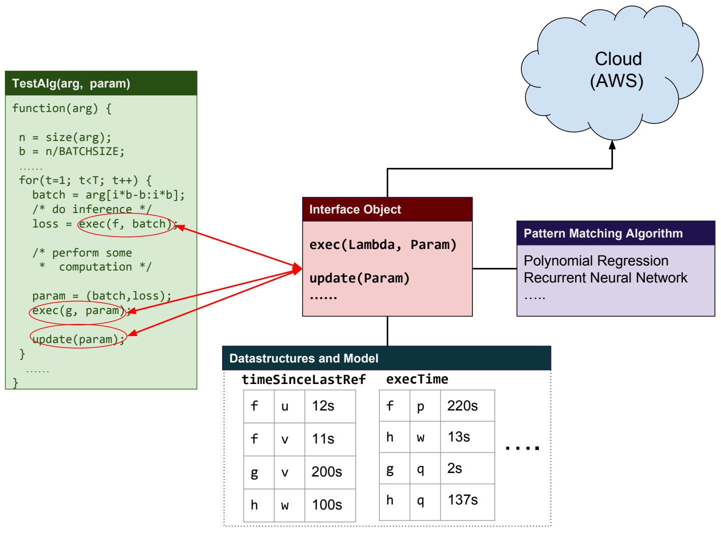
4 Architecture and Implementation

In this section, we describe the architecture and the various components, datastructures, models and their implementation. Figure 1 shows the sketch of the architecture. But before that, we first detail the assumptions we make about the specific problem that we solve in the current implementation.

Assumption: We assume that application calls the cloud for executing lambda functions (stateless) or updating parameters stored in the cloud. The cloud services charges users for every second of use. The cloud is expensive and even few seconds of inactivity is costly. In the first release of SIC, we are only interested in optimizing budget and resources for loops in a single single-threaded application. This also means that application needs only one active instance at a given time. The lambdas however are allowed to be multi-threaded or can spawn new processes. While this may seem limiting, many target applications described before meet this format.

We now present implementation details of different components in SIC.

Init Object: The core of the architecture consists of an interface layer, which handles all communication between a cloud and an application. An object of the interface called *init* is created when the application starts. This object contains information about the cloud specification (e.g., ami images), security key, permission files, session budget in dollars, maximum timeout, number of retries and few additional information. The object kills any threads spawned by it when the application exits.



```
function(arg) {

 n = size(arg);
 b = n/BATCHSIZE;
 ......
 for(t=1; t<T; t++) {
   batch = arg[i*b-b:i*b];
   /* do inference */
   loss = exec(f, batch);

   /* perform some
    *  computation */

   param = (batch,loss);
   exec(g, param);

   update(param);
 }
  ......
}
```

timeSinceLastRef

f	u	12s
f	v	11s
g	v	200s
h	w	100s

execTime

f	p	220s
h	w	13s
g	q	2s
h	q	137s

....

Figure 1: Architecture of the cloud showing various components and sample datstructures, models used.

Types of request: Application can make multiple type of requests to the interface. The two main type of requests are:

- *execute a lambda:* Executes a lambda function (stateless with no side-effect) on some parameters. This is useful for running expensive computations on cloud.

  $\texttt{init.exec}(\texttt{Lambda f}, \texttt{Param arg})$

- *update storage:* Updates some parameter globally. This is useful where several applications are relying on shared parameters and want to benefit from each other's experience.

  $\texttt{init.update}(\texttt{Param newVal})$

Datastructures: The interface object maintains two main table which are continuously updated during the entire session, namely `timeSinceLastRef[Lambda, Param]` and `execTime[Lambda,Param]`. `timeSinceLastRef[f,p]` tells the time since last call was made by the application for running f with parameter p and `execTime[f,p]` gives the time it took for the cloud to execute f

with parameter p. A table also maintains the actions currently allotted to an instance and another table maintains a list of lambda terms that an instance posses. Few other simple tables maintain a list of price of different instance type and time taken to open and close instances, by looking up specifications.

These tables are used to anticipate the time when application will request to run f again and the time it will take for cloud to execute it. SIC keeps updating these tables during the entire session.

Executing Lambda: When `init` receives a request $\texttt{exec(f, p)}$; it searches if there is any active instance already running or in the process of being launched. If there is an instance being launched then it waits for it to be ready for use. Finally, when there is an instance ready; it sees if the instance has a copy of `f` and if not a copy is uploaded. Then the exec command is issued remotely and its results are returned. Jsch library is used for all these operations. Lambda can be any executable such as a jar file, .exe etc.

If there are no instance running, then the algorithm decides on the type of instance to launch (micro, macro etc.) based on the available budget and earlier execution time of f in $\texttt{execTime}[\texttt{f}, \texttt{p}; \texttt{instance\_type}]$. The expected price is computed for a given instance using

$$\texttt{price} = \texttt{execTime}[\texttt{f}, \texttt{p}; \texttt{instance\_type}] * \texttt{instance\_type.price}$$

and if it is in the budget and if $\texttt{execTime}[\texttt{f}, \texttt{p}] < \texttt{timeout}$ then it is added to the list of possible choice. The algorithm then picks the cheapest possible option available.

If no data about f exists then the algorithm always picks the cheapest instance. Once the instance is launched, image of f and p is copied and the exec command is executed and its output returned.

Running generic images requires installing compilers and other language specific libraries. Therefore, instance is launched using a saved image containing these libraries. Amazon aws does this using the Xen hypervisor.

Note that our design easily incorporates the cases where lambda functions are proprietary which reside in the cloud and are not executable on the client's machine. In these cases, there is no need to copy an image of the lambda function.

Workload Prediction: Tables will not contain entries for every (f, p). Hence, we are interested in making predictions for two quantities:

- anticipating when a call to f will be made using a sequence of time between consecutive calls. So if call to f was made at time $t_1, t_2, \cdots t_k$ then pattern of time between consecutive calls f is given by $\{t_2 - t_1, t_3 - t_2, \cdots, t_n - t_{n-1}\}$.

- anticipating the execution cost of $[f, p]$ given a dataset of past observations $\{[f, p_1], [f, p_2], \cdots, [f, p_n]\}$. We make an assumption here that execution call is dependent on the size and value of p. Example, if p is a list of size n then execution cost will be a function of n. This is true for reasonable number of algorithms but in general, special pathological lists (say totally unsorted) etc. may result in more execution time than a list of larger length (which is sorted).

We frame these problems as regression problem and use polynomial regression for fitting the data. For the first task, the data is $\{(i-1, t_i - t_{i-1})\}_{i=2}^{n}$ and for the second task the data is $\{(size(p_i), execTime(f, p_i))\}_{i=1}^{n}$. We briefly review polynomial regression below:
Polynomial Regression: Given a data $(x_i, y_i)_{i=1}^{n}$ with $x_i, y_i \in \mathbb{R}$, polynomial regression learns a polynomial function $y = \sum_{j=0}^{c} a_j x^j + \epsilon$; where $\{a_j\}_{j=0}^{c}$ are constants, c is the degree of polynomial that we are fitting (we use 2 for SIC), and ϵ is noise. The coefficients have closed form solution given by $\vec{a} = (X^T X)^{-1} X^T \vec{y}$ where X is the Vandermonde matrix given by $X_{ij} = x_i^{j-1}$ and $\vec{y}_i = y_i$. For generality and simplicity purposes, SIC solves this by gradient descent with L2 regularization.

We use polynomial regression due to its simplicity and the fact that most practical algorithms have polynomial dependency in size of data. We also experimented with Recurrent Neural Network for the first task, but contrary to expectations we found it difficult to train RNN to learn arithmetic progressions. Modular nature of SIC allows replacing these prediction models by more sophisticated domain-specific models.
Budget Optimization: The interface continuously predicts when next call to a lambda will be made and what will be the execution time of calls. This allows the interface to smartly prelaunch or withhold an instance with particular lambda saving us time and money. This is done using a simple strategy. At the end of every `exec(f,p)` call, the interface predicts the next time the application will ask to execute the lambda f. The interface also computes the loading time given by:

$$\begin{aligned}\texttt{loading\_time} &= \texttt{avg\_Booting\_time} + \texttt{lambda\_upload\_time} \\ &\quad + \texttt{avg\_halting\_time}\end{aligned}$$

this gives us the time it will take to close an instance, open another instance and upload the lambda f. These are computed by simple averaging statistics.

If the interface predicts that the next call to a lambda term f will be made after T seconds, then if $T > \texttt{loading\_time} + \delta$ then SIC schedules the launch and lambda upload instruction after a gap of $T - \texttt{loading\_time}$ and closes the open instance. A $\delta > 0$ term is used as a precaution against uncertainty. We do not want SIC to rely too much on its prediction and close the instance for the benefit of few seconds, since this can keep the application waiting for the instance if predictions are too late.

Note that when the predictions and statistics are exact, then the instance will just be ready by the time the `exec` call is made and the application will not have to wait. Moreover, budget will be spent only when instances are actually computing.

Once the scheduled instance is launched, SIC pushes a close instance event in the dispatcher, so that in case the launched instance is not used within a timeout period then its scheduled to be closed. We found that optimizing $\delta, \mathit{timeout}$ was important to extract benefit from SIC. While manually fixed for now, we want to tailor these hyperparameters in later version of SIC.
Example: We consider the following algorithmic structure given below called `cloudBasedML`, for illuminating our points.

```
cloudBasedML(dataset, model)
for $step = 1; step \leq maxStep; step++$ do
    /* perform inference on the cloud */
    result = exec(inference,(dataset, model));
    /* update the model*/
    ⋮
end
```

Algorithm 1: Prototype algorithm using our interface. An exec call is made to the cloud in a loop. This call is expensive and therefore appropriate use of cloud. After every `exec` call the application updates the model locally, during this time the cloud is not used. SIC predicts workload to decide whether to keep an instance alive in this duration or whether to close the previous one and schedule a new instance request for future.

In Algorithm 1, the application keeps on making expensive exec calls with lambda `inference`, separated by local computation during which instance remains inactive. SIC predicts the time after which the exec calls are made. If SIC sees that the instance is idle for time greater than the time taken to close the current instance and launch a new instance with lambda by a δ margin, then the instance can be released for the duration of the local computation. Since the algorithm runs in loop, this means that total cost saved can be quite a lot even if its small for one iteration. For consideration, some NLP, ML algorithms run for days during which the budget saved can be significant.

5 Experimental Evaluation

As described before, in this version of SIC we are interested in optimizing loops in single single-threaded applications. In order to evaluate the performance of SIC on optimizing budget and waiting time (equivalently resource usage), we consider the algorithm prototype in Algorithm 1. We use a dummy lambda `inference` which sleeps for `a` seconds. Further, we replace the part "update model" by a sleep command for `b` seconds. By varying a, b, changing them with loop index and adding noise; we can simulate a wide range of application behaviour.

We further consider two baselines:

No Waiting: For this baseline, we keep an active instance while the application is alive. This consumes budget every second the application is alive, however the application never has to wait for an instance.
Optimal Budget: We launch an instance whenever the application needs one and then close it afterwards. This ensures that budget is only spent when the instances are computing.

We tested SIC on different values of a, b. Below, we give the results for $a = 20s, b = 60s$. This is a non-trivial example where there is a scope of optimizing the resources. Results are shown in Table 1.

As we can see that SIC saves 1.04$ compared to *No Waiting* while being 33.1s faster than *Optimal Budget* baseline. There is still room for optimizing SIC further, for example by running *close instance* requests in parallel since they are non-blocking.

The exec call pattern for the above example is given in Figure 2. After 2

Algorithm	Time Taken (s)	Budget Spent*
No waiting	338.4s	2.68$
Optimal Budget	550.4s	0.64$
SIC	517.3s	1.03$

Table 1: Performance of SIC against *No Waiting* and *Optimal Budget* baseline. Please see Section 5 for details.(* cloud service rate of 1 cent per second.)

`exec` requests, SIC is able to reasonably predict the next call. Hence, when the actual `exec` requests is made, SIC observes that a prelaunch request is undergoing and hence decides to wait for it to finish. This saves the extra time that would have been spent if a fresh instance launch request would have been issued.

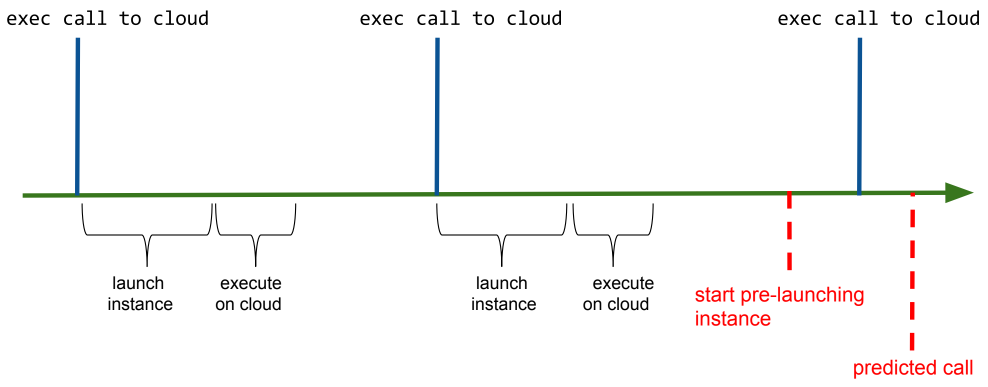


Figure 2: Execution Call for the example described in Section 5. Prediction of call reduces waiting time for the application.

Please see the source code[6] for running these experiments and more examples.

6 Future Work

In future releases of SIC, we want to relax several assumptions and add more powerful optimization tools. Following are a few directions that can be considered:

[6]https://github.com/dkmisra/SIC

Several Multi-threaded Applications: In this version, we tested SIC on single single-threaded application, to test out the potential of SIC. In future, we want to relax this assumption by testing on several multi-threaded applications with shared budget which can potentially require more than one instance.
Optimizing Arbitrary Applications with Program Analysis: In this paper, we focused on optimizing loops which are the bottleneck of several applications. A general optimization was not possible since the program structure was not known. While this allowed us to optimize executables where source code is not known, this made the prediction task harder. In future, we want to use program analysis to help in optimizing arbitrary applications whenever the source code is available.
Evaluation on real-life examples: Our motivation behind developing SIC was to be useful for end-users wanting to save budget and reduce waiting time, while running their applications with commercial cloud services. In this paper, we showed potential of SIC by evaluating on synthetic tasks. In future, we want to evaluate performance of SIC on real-life applications.

Code: Code is available on github at https://github.com/dkmisra/SIC. The README contains instruction on how to run applications with SIC. Please visit the github repository for latest releases, FAQ and more examples.

Acknowledgement: I thank Ken Birman and Theodoros Gkountouvas for their help and suggestion. I thank the anonymous reviewers for their comments and suggestions. I want to thank my advisor Yoav Artzi, for discussing the usability of SIC system for machine learning applications. I also want to thank Praveen Sadhu and Aditya Jami of Zoodig for pointing out the material on Java APIs for Amazon AWS.

References

Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *IEEE 28th International Conference on Data Engineering (ICDE).* IEEE, 390–401.

Artur Andrzejak, Derrick Kondo, and Sangho Yi. 2010. Decision model for cloud computing under sla constraints. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 257–266.

Michael Bensch, Dominik Brugger, Wolfgang Rosenstiel, Martin Bogdan, Wilhelm G Spruth, and Peter Baeuerle. 2007. Self-Learning Prediction System for Optimisation of Workload Management in a Mainframe Operating System.. In *International Conference on Enterprise Information Systems (ICEIS)*. Citeseer, 212–218.

Junwei Cao, Stephen Jarvis, Subhash Saini, Graham R Nudd, and others. 2003. Gridflow: Workflow management for grid computing. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 198–205.

Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. 2012. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing* 5, 2 (2012), 164–177.

Sheng Di and Cho-Li Wang. 2013. Dynamic optimization of multiattribute resource allocation in self-organizing clouds. *IEEE Transactions on Parallel and Distributed Systems* 24, 3 (2013), 464–478.

Nikolaos Doulamis, Anastasios Doulamis, Antonios Litke, Athanasios Panagakis, Theodora Varvarigou, and Emmanuel Varvarigos. 2007. Adjusted fair scheduling and non-linear workload prediction for QoS guarantees in grid computing. *Computer Communications* 30, 3 (2007), 499–515.

Jennie Duggan, Yun Chi, Hakan Hacigumus, Shenghuo Zhu, and Ugur Cetintemel. 2013. Packing light: Portable workload performance prediction for the cloud. In *IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 258–265.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-Based Dependency Parsing with Stack Long Short-Term Memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 334–343.

Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *IEEE 25th International Conference on Data Engineering (ICDE)*. IEEE, 592–603.

Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 117–130.

Dan Jurafsky and James H Martin. 2000. *Speech & language processing*. Pearson Education India.

Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 1287–1294.

Katrina Leigh LaCurts. 2014. *Application workload prediction and placement in cloud computing systems*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. 2011. CloudProphet: towards application performance prediction in cloud. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 426–427.

Bing Liu. 2012. Sentiment analysis and opinion mining. *Synthesis Lectures on Human Language Technologies* 5, 1 (2012), 1–167.

Ke Liu, Hai Jin, Jinjun Chen, Xiao Liu, Dong Yuan, and Yun Yang. 2010. A compromised-time-cost scheduling algorithm in SwinDeW-C for instance-intensive cost-constrained workflows on cloud computing platform. *International Journal of High Performance Computing Applications* (2010).

Ming Mao, Jie Li, and Marty Humphrey. 2010. Cloud auto-scaling with deadline and budget constraints. In *11th IEEE/ACM International Conference on Grid Computing (GRID)*. IEEE, 41–48.

Tom M Mitchell. 1997. *Machine learning*. McGraw-Hill Boston, MA:.

Balaji Palanisamy, Aameek Singh, and Ling Liu. 2015. Cost-effective resource provisioning for mapreduce in a cloud. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2015), 1265–1279.

Dimitar H Stefanov, Zeungnam Bien, and Won-Chul Bang. 2004. The smart house for older persons and persons with physical disabilities: structure, technology arrangements, and perspectives. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 12, 2 (2004), 228–250.

Tanyaporn Tirapat, Orachun Udomkasemsub, Xiaorong Li, and Tiranee Achalakul. 2013. Cost optimization for scientific workflow execution on cloud computing. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 663–668.

Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, and others. 2008. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics* 25, 8 (2008), 425–466.

Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. 2010. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *IEEE 3rd International Conference on Cloud Computing (CLOUD)*. IEEE, 228–235.

Guiyi Wei, Athanasios V Vasilakos, Yao Zheng, and Naixue Xiong. 2010. A game-theoretic method of fair resource allocation for cloud computing services. *The journal of supercomputing* 54, 2 (2010), 252–269.

Luke S Zettlemoyer and Michael Collins. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form.. In *EMNLP-CoNLL*. 678–687.

Qian Zhu and Gagan Agrawal. 2010. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 304–307.