City Research Online

City St George's, University of London

Citation: Mai, L., Kalyvianaki, E. & Costa, P. (2013). Exploiting Time-Malleability in Cloud-based Batch Processing Systems. Paper presented at the LADIS '13: 7th Workshop on Large-Scale Distributed Systems and Middleware, 02-11-2013 - 03-11-2013, Pennsylvania, USA.

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/8179/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Exploiting Time-Malleability in Cloud-based Batch Processing Systems

Luo Mai
Imperial College London

Evangelia Kalyvianaki
City University London

Paolo Costa
Microsoft Research

Abstract

Existing cloud provisioning schemes allocate resources to batch processing systems at deployment time and only change this allocation at run-time due to unexpected events such as server failures.

We observe that MapReduce-like jobs are *time-malleable*, i.e., at runtime it is possible to dynamically vary the number of resources allocated to a job and, hence, its completion time.

In this paper, we propose a novel approach based on time-malleability to opportunistically update job resources in order to increase overall utilization and revenue. To set the right incentives for both providers and tenants, we introduce a novel pricing model that charges tenants according to job completion times. Using this model, we formulate an optimization problem for revenue maximization.

Preliminary results show that compared to today's practices our solution can increase revenue by up to 69.7% and can accept up to 57% more jobs.

1 Introduction

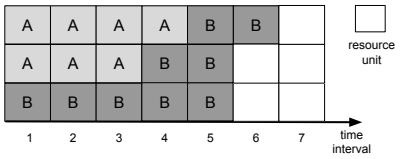
MapReduce and its derivatives such as Dryad and Spark are the de-facto standard to execute batch jobs in a cloud environment. They support a simple programming model that limits the dependencies among their sub-tasks, leading to a flexible mapping between tasks and resources. This allows jobs to scale out to run on an arbitrarily large number of servers, without requiring any additional effort from programmers. The constrained programming model and fixed dependencies among tasks also simplify the estimation of the job execution time and several estimation models, which take into account job characteristics, input data size, and the resources allocated, have recently appeared in literature [5–7, 10, 11].

Several systems build upon this predictability at deployment time to derive the number of resources needed to meet user specified deadlines [6, 7, 11]. Small allocation adjustments can be made at runtime in order to cope with unexpected and not-so-common events such as server failures [5]. A major drawback of these approaches, however, is that the provider is bound to a *fixed* allocation plan. This is at odds with the elastic nature of cloud where available resources fluctuate over time.

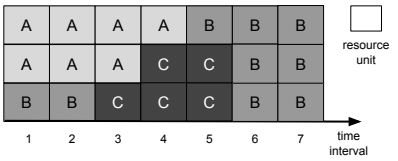
In contrast, we propose an approach that deliberately varies the amount of resources allocated to jobs over time in order to control their completion time and increase cloud utilization. The intuition is that if spare resources are available, they should be allocated to running jobs to reduce their completion time. However, if new jobs are submitted, these resources should be claimed back to accommodate the new jobs. Our approach builds upon the observation that most (if not all) MapReduce-like jobs are *time-malleable*, i.e., it is possible to change their resource allocation at runtime, without affecting the correctness of the results [2, 7].

A key challenge of our approach is that under the current, pay-as-you-go, pricing model, there is no incentive for the provider to reduce execution time. Even with the recently proposed *deadline-based* models, e.g., [7, 8], users can only specify a single desired completion time. To address these shortcomings, we combine our solution with a novel pricing model in which the later a job is completed, the lower users pay. However, to avoid unbounded completion time, users also specify the longest acceptable deadline of their jobs. Dually, they also indicate the maximum price they are willing to pay.

We believe that this model is advantageous to both



(a) Allocation plan for jobs A and B



(b) Runtime reallocation for new job C

Figure 1: Example of dynamic resource allocation

users and the provider. It provides great incentives users to define a long deadline in the exchange of a reduced price. At the same time, this model also benefits the provider to devise flexible planning decisions among all jobs.

In this paper, we take a first step in this direction by formulating the allocation problem as a mixed integer program to maximize provider's revenue (Section 3.2). Our work relies on existing work to derive the initial resource requirements for jobs at deployment and we extend our formulation to consider related errors at runtime (Section 3.3). Simulation results show that our approach significantly improves both provider revenue and jobs acceptance ratio. (Section 4).

2 Overview

We illustrate our approach by means of the example in Figure 1. We assume a cloud data center with three resource units (e.g., a VM or a MapReduce slot) at any given time. Consider two jobs, A and B, with the following requirements: ($r_A = 7$, $d_A = 6$) and ($r_B = 9$, $d_B = 7$), where r_j denotes the total resource units for job j and d_j shows the completion time deadline of j counting from its submission time. We assume an initial *feasible* allocation plan as depicted in Figure 1(a); an allocation plan shows the number of units allocated to a job at any given time. A plan is feasible when the total allocation of resource units per interval does not exceed the cloud capacity.

Let us now suppose that a new job C is submitted at time $t = 2$ with the requirements ($r_C = 5$, $d_C = 3$). Under today's *rigid* allocation scheme, job C could not be accommodated, since by the time B completes at $t = 6$ and resources become free, C's deadline has already expired. However, by exploiting job time-malleability, it is possible to *dynamically* change the allocation plan, exploiting the loose deadline of B to accommodate C and potentially increase its revenue. Figure 1(b) shows such a reconfiguration that leads to feasible allocation plans for jobs B and C.

Besides meeting the deadlines of current jobs, the provider also aims at maximizing its revenue. Therefore, while reallocating resources, it must ensure that the total revenue does not decrease as a consequence of accepting a new job and delaying an already running one. The next section presents our solution to this problem.

3 Allocation for time-malleable jobs

In this section, we formulate the problem of cloud revenue maximization. We first present a basic formulation, followed by an adjustment that takes into account errors in the estimations of jobs resource requirements. Overall a new job is accepted if and only if the solution to the maximization problem provides a feasible allocation plan with an estimated revenue higher than the revenue coming from the current allocation without the new job. A user with a new job j communicates to the provider the longest acceptable deadline d_j and the number of resource units required r_j. The user and the provider also agree upon a pricing function described below.

3.1 Pricing function on completion time

A pricing function $P(t)$ describes the relationship between the job completion time t and the corresponding price to be paid to the provider. To build such a function, the user indicates the longest acceptable deadline (hereafter referred to as the deadline) of her job d. The user and the provider also agree upon a maximum affordable price $p_{\max}$ paid if the job finishes at an earliest possible completion time s. A pricing function that captures the fact that a job should cost less for longer completion times is:

$$P(t) = \begin{cases} p_{\max} & \text{if} \quad t < s \\ f(t) & \text{if} \quad s \leq t \leq d \\ 0 & \text{if} \quad t > d. \end{cases}$$

Figure 2 provides an example $P(t)$. In the range where ($s \leq t \leq d$), the price $f(t)$ of a job completion time t should be monotonically decreasing to t and its values never exceed $p_{\max}$. Figure 2 shows a convex function; but different shapes can be considered as well. Note that if ($t > d$) then the job fails to meet its deadline and no payment is issued.

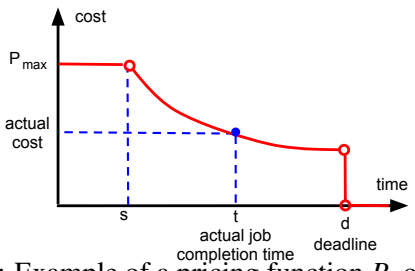


Figure 2: Example of a pricing function P_j of job j.

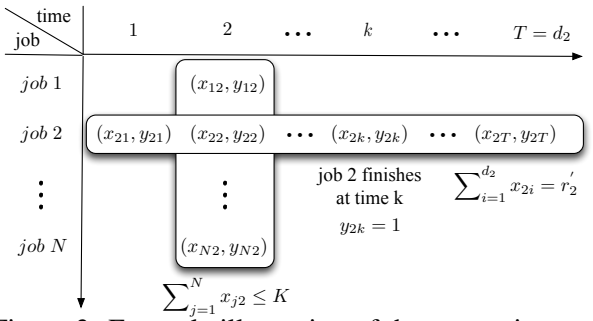


Figure 3: Example illustration of the constraints and the decision variables x and y.

3.2 Cloud revenue maximization

Consider $N-1$ jobs running under certain allocation plans and a new job arriving. Our goal is to find a feasible allocation for the new job that maximizes the total revenue for all N jobs over time. To find a set of feasible allocation plans we formulate the mixed integer programming (MIP) problem with the objective (1) subject to constraints (2)- (6). We solve the maximization problem for a duration of $T = \arg\max_j d_j$ intervals to include the longest deadline among all N jobs. We use t to index intervals for the next T intervals starting from the current one pointed by $t = 1$, i.e., $t \in \{1, \ldots, T\}$. We ensure that the solution includes feasible plans via the constraints explained below. We also use Figure 3 to illustrate key points of the constraints.

$$\underset{x,y,p}{\text{maximize}} \sum_{j=1}^{N} p_j \tag{1}$$

$$\text{subject to } \sum_{i=1}^{d_j} y_{ji} = 1 \quad \forall j, \tag{2}$$

$$\sum_{i=1}^{d_j} x_{ji} = r'_j \quad \forall j, \tag{3}$$

$$\sum_{j=1}^{N} x_{jt} \leq K \quad \forall t, \tag{4}$$

$$0 \leq x_{jt} \leq K \times \sum_{i=t}^{T} y_{ji} \quad \forall j,t, \tag{5}$$

$$p_j \leq P_j\left(\tilde{t}_j + \sum_{i=1}^{d_j} i \times y_{ji}\right) \quad \forall j, \tag{6}$$

$$y_{ji} \in \mathbb{B}^{N\times T}, \; x_{ji} \in \mathbb{N}^{N\times T}, \; p_j \in \mathbb{R}_+, \tag{7}$$

$$j \in \{1, \ldots, N\}, \; t \in \{1, \ldots, T\}. \tag{8}$$

Deadline feasibility constraint. All jobs should finish before their deadlines shown by (2) where the binary variable $y_{ji} \in \mathbb{B}^{N\times T}$ is assigned to 1 if job j completes at time i, otherwise is set to 0, also shown in Figure 3. For every job j there should be only one $y_{ji} = 1$ at a time before its deadline d_j.

Resource feasibility constraints. All jobs should be given resource units according to their requirements shown by (3) and exemplified in Figure 3. The decision variable $x_{ji} \in \mathbb{N}^{N\times T}$ denotes the allocation of resource units per job j at interval i. The number of resource units before the deadline, i.e., $\sum_{i=1}^{d_j} x_{ji}$, must be equal to r'_j; where $r'_j \leq r_j$ denotes the remaining resource units needed to allocate for job j given that j might be an existing running job and has already used some of its resource units. The sum of allocations across jobs per interval must not exceed the total cloud capacity denoted by K and shown by (4). Furthermore, the resource allocation x_{ji} itself cannot exceed the total capacity of the cloud and is shown by (5).

Finally, constraint (6) denotes the payment of a job given its completion time and where $\tilde{t}_j$ denotes the previous running time of job j. This payment is bounded by function $P_j(t)$.

The problem to find the optimal solution for a MIP formulation has shown to be NP-complete. Therefore, we use the CPLEX state-of-the-art optimization solver [13] to find approximated solutions within short times.

3.3 Discrepancies in resource estimation

Our model uses an estimate of the number of resource units required by a job. Recent work on offline profiling and analytic performance models derive the resource units required for MapReduce jobs e.g., [5, 7, 10, 11]. Further, some workloads exhibit a large fraction of recurring jobs. For example, in Bing's production clusters recurring jobs account for 40.32% of the total [1]. This allows to improve the prediction by taking into account prior executions.

Nevertheless, despite the accuracy of these initial estimations, it is still possible that discrepancies arise during job execution, e.g., due to stragglers [3, 12]. Therefore, we extend our basic formulation to accommodate for estimation errors.

We insert a random parameter e_j, referred to as the

job size estimation error, into constraint (3) to introduce the job execution variability in planning, i.e., $\frac{r_j}{e_j}$. For example, when $e_j \in (0,1)$ then the size of the job is underestimated (resp. overestimated if $e_j > 1$). Hence, constraint (3) can be transformed into a probability constraint: $\text{Prob}_j \left(\sum_{i=1}^{d_j} x_{ji} = \frac{r'_j}{e_j} \right) \geq 1-\alpha, \forall j$.

This rewritten constraint ensures that, with a $(1-a)$ probability guarantee, the accumulated number of resource units allocated to a job before its deadline equals its remaining job size considering any estimation errors. Using a probability constraint makes the program stochastic and cannot be solved by standard solvers like CPLEX. To calculate the solution to it, we transform it into a deterministic equivalent by assuming that e_j is subject to a normal distribution, i.e., $e_j \sim N(\mu_j, \sigma_j^2)$. Due to space limitations we omit the details of the transformation. At the end, the transformed formulation can be solved with CPLEX.

4 Preliminary Results

To quantify the benefits achieved by the formulation approach outlined in Section 3 (hereafter, referred to as GEARBOX), we use a discrete event simulator and CPLEX to solve the maximization problems. While preliminary, our results indicate that by allowing users to specify long deadlines and by exploiting time-malleability our approach increases both provider revenue and acceptance ratio.

4.1 Setup

Given the unavailability of a reference workload that would fit our pricing model, we adopt a synthetic workload. While admittedly simplistic, we believe that this workload is a good approximation of what a real workload might be.

We assume a cloud with a total capacity of $K = 1,000$ resource units. We denote r_j to be the initial number of units required by job j; $d^j_{min} = \frac{r_j}{K}$ is the shortest completion time possible when *all* cloud resources are assigned to j. We consider two different scenarios. In the former (Figure 4), we assign the initial resource requirements r_j of job j to be a randomly chosen integer between 2,500 and 7,500 resource units. This ensures that all jobs take at least more than two units of time to complete (i.e., $d^j_{min} > 2$ for all jobs). In the latter scenario (Figure 5), we set $r_j = 50 \cdot (1+b)$, where b is randomly selected using a half-normal distribution with a mean of 0 and standard deviation of 50 modeled after the analysis from Google cluster traces [9]. The parameters s_j and d_j of the pricing function in Figure 2 are randomly chosen and $s_j \in [d^j_{min}, 5 \cdot d^j_{min}]$ and $d_j \in [s_j, 3 \cdot s_j]$. Note that this setup implies that some jobs may have no flexibility at all, i.e., $s_j = d_j$.

For simplicity we consider a linear pricing function P_j. We also experimented with different function shapes and different ranges for s_j and d_j, observing similar trends to the results reported below.

We simulate job requests arriving over time for a total duration of 10,000 seconds. We assume Poisson job arrivals with a mean arrival rate λ varying between 0.12 and 0.26 jobs/s (resp. 0.30 and 0.65 jobs/s for the Google-inspired workload). These values yield a data center utilization between 60% and 100% when we allocate resources.

Baselines. We compare our approach against three baselines, representative of common techniques used in today's systems. **EARLY**: a fixed set of resources is reserved for a job to meet its most demanding deadline s_j. This closely resembles the existing approach in deadline-based systems where a single deadline is considered. **LATE**: a fixed set of resources is reserved for a job to meet its deadline d_j; **EDF**: this baseline implements an earliest deadline first (EDF) approach where we allow runtime modifications in jobs allocation plans; we consider the deadline d_j.

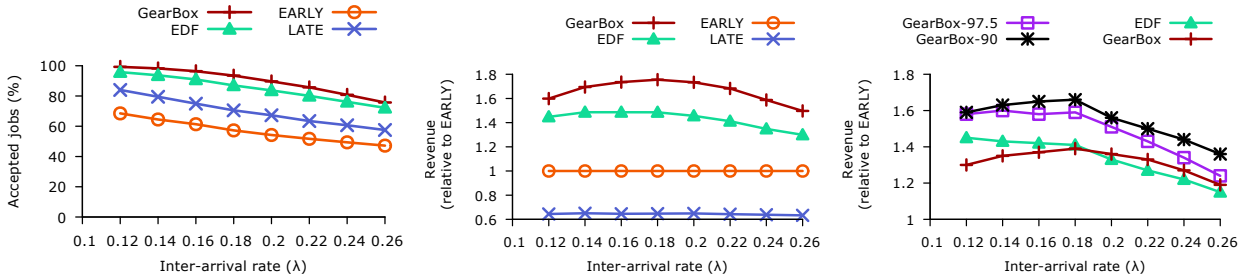
CPLEX execution time. We run our experiments on a server with 16 Intel Xeon E5-2690 cores and 32 GB of RAM. We set the upper bound of the CPLEX execution time per job to 100 s and the error rate to 1%. Across all runs[1], the median CPLEX execution time is 2 s and the 95^{th} percentile is 6 s. Since our algorithm needs to be run only when a job is submitted, the overhead introduced is negligible.

4.2 Basic formulation results

We begin our analysis by considering the case with no estimation errors. Figure 4(a) shows the percentage of jobs accepted by each method. We observe that the dynamic allocation plans (GEARBOX and EDF) accept significantly more jobs than the static ones (EARLY and LATE). The reason is that, by being unable to reallocate resources as new jobs come in, or existing jobs terminate, the performance of static allocation plans is drastically reduced. This highlights the problem with today's setup and motivates our effort to explore alternative solutions.

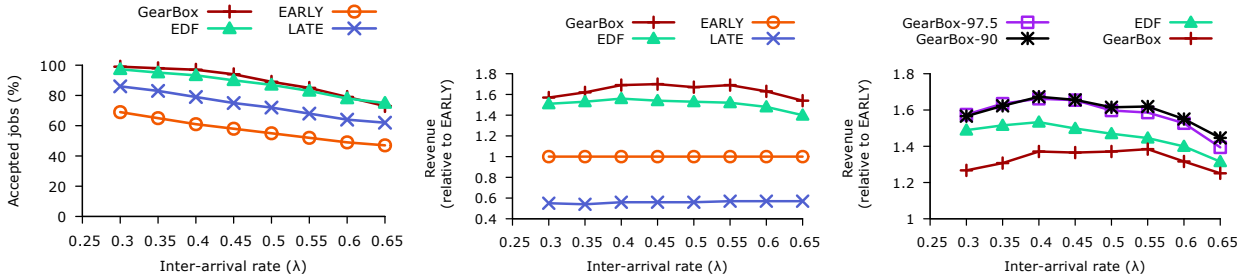
The second metric used in our evaluation is the

[1] We average the results across 5 runs and the standard deviation for all experiments in this section is within 3%.



(a) Job acceptance ratio (b) Total revenue (c) Total revenue with estimation errors

Figure 4: Simulation results with the job sizes generated using a uniform distribution



(a) Job acceptance ratio (b) Total revenue (c) Total revenue with estimation errors

Figure 5: Simulation results with the job sizes generated using a half-normal distribution [9]

total revenue, shown in Figure 4(b), normalized against EARLY. GEARBOX achieves much higher revenue than EARLY (between 53.7% and 69.7%) and LATE (between 170% and 205%). This is because of GEARBOX's ability to dynamically reassign resources and, hence, accept more jobs. Interestingly, GEARBOX also outperforms EDF (between 5.6% and 17.5% higher revenue), although their fraction of accepted jobs is close. The reason is that GEARBOX admits a new job only if this increases the total revenue. This means that in some cases, even if there would be enough idle resources to accept a job, GEARBOX can still decide to reject it, if this is not cost-effective. In contrast, if resources are available, EDF always accepts a new job, although this might be detrimental in the long term. This shows the importance of considering the total revenue as a first-class citizen in the admission control. Similar trends are also observed when using the Google-inspired workload as shown in Figure 5.

We also experimented with a flat pricing function, obtaining a factor of 1.8x improvement in the revenue of GEARBOX over EARLY (resp. 1.1x over EDF). This shows that even when there is no additional premium for the provider to complete jobs early, it can still increase its revenue by exploiting their malleability. This is because GEARBOX fully utilizes resources as soon as they become available and, hence, it is more prepared to accommodate future bursts of jobs. Furthermore, and in contrast to EARLY, it also has the ability to reclaim resources back from running jobs as appropriate for new ones.

Interestingly, at low load ($\lambda = 0.12$), the median job completion time increase (relative to s_j) is less than 1% (95^{th} percentile is 46%). At high load ($\lambda = 0.26$), instead, the median is 26% (95^{th} percentile is 177%). This indicates that GEARBOX exploits time-malleability only for a few jobs, while for the vast majority of them, it strives to minimize the completion times. This is a consequence of the incentives set by our pricing model. The provider has a strong incentive to finish the job as soon as possible. At the same time the provider can delay a few jobs if this increases the overall revenue.

4.3 Estimation errors

We now consider the impact of estimation errors on our base solution, GEARBOX, and show how the extension detailed in Section 3.3 can mitigate these. We refer to our extended solution as GEARBOX-p, where p is the probability in the transformed constraint.

We assume that the estimation error e_j for a job j follows the normal distribution $N(1,\ 0.1)$. This is consistent with the results presented in [6,7]. We also experiment with other values of standard deviation

$\sigma \in [0.05, 0.15]$, obtaining similar results.

Figure 4(c) and 5(c) show the revenue for different job inter-arrival rates λ in the two workloads considered. GEARBOX-97.5 and GEARBOX-90 achieve higher revenue than both EDF and GEARBOX. In this scenario, EDF outperforms GEARBOX because as it recomputes the deadlines at runtime, it naturally shifts resources to jobs that are close to deadlines, thus implicitly accounting for estimation errors. However, the GEARBOX-p solutions achieve the best performance because they account for estimation errors in their allocation plans.

This is also reflected in the fraction of accepted jobs that miss the deadline. While EDF and GEARBOX exhibit a miss ratio of 6.05% and 13.75% respectively, GEARBOX-90 achieves a miss ratio of 0.7% (respectively 0.07% for GEARBOX-97.5).

5 Discussion

We are implementing our approach in the Apache Hadoop framework. Our prototype currently supports dynamic reallocation in the map phase. Since map tasks are short, independent and run in multiple waves, we can vary the number of resources between each wave. Reduce tasks are more complex as they are typically long-running and they run in a single wave. To support dynamic reallocation in the reduce phase, we are currently integrating our prototype with the Sailfish project [2, 14], which supports suspend and resume of reduce tasks.

The current version of the model makes some simplifying assumptions. In particular, we assume perfect scalability and ignore data dependencies. For a real deployment, our model needs be extended to include constraints such as job barriers, which can reduce parallelism, and data locality, which can have an impact on the total running time. Addressing these limitations as well as including network constraints (possibly leveraging our prior work [4, 7]) is part of our current research agenda.

We also intend to explore more advanced pricing functions. In the results presented in the previous section, we assume *static* pricing, i.e., the shape and values of the function depend only on the job type and size. We are currently investigating the benefits of using *dynamic* pricing functions similar to the Amazon spot instances model [15], in which prices changes based on the current utilization. We believe that combining dynamic pricing with job malleability would allow to further increase utilization and revenue while providing more flexibility to tenants.

6 Conclusions

Current batch processing systems support fixed allocation plans, in which the resources allocated to a job are never changed (unless in cases of unexpected events such as failures or stragglers). Rather, we argue that the time-malleability property of batch jobs should be exploited to opportunistically vary the allocation plans at runtime. Our approach allows users to specify the longest acceptable deadline for their jobs along with the maximum price they are willing to pay. Providers then use this information to *dynamically* allocate resources to jobs in order to improve utilization and revenue. Preliminary results show that our approach can significantly increase revenue and acceptance rate, by only marginally affecting job execution time.

Acknowledgements. The authors wish to thank Wolfram Wiesemann for his helpful comments and suggestions, and Bolun Dong, James Simpson, and Jiawei Yu for their help on building a preliminary prototype of our system. Luo Mai is a recipient of the Google Europe Fellowship in Cloud Computing, and this research is supported in part by this Google Fellowship.

References

[1] AGARWAL, S., KANDULA, S., BRUNO, N., WU, M.-C., STOICA, I., AND ZHOU, J. Re-Optimizing Data-Parallel Computing. In *NSDI* (2012).

[2] ANANTHANARAYANAN, G., DOUGLAS, C., RAMAKRISHNAN, R., RAO, S., AND STOICA, I. True Elasticity in Multi-Tenant Data-Intensive Compute Clusters. In *ACM SoCC* (2012).

[3] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using Mantri. In *OSDI* (2010).

[4] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *SIGCOMM* (2011).

[5] FERGUSON, A., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys* (2012).

[6] HERODOTOU, H., DONG, F., AND BABU, S. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *ACM SoCC* (2011).

[7] JALAPARTI, V., BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Bridging the Tenant-Provider Gap in Cloud Services. In *ACM SoCC* (2012).

[8] LUCIER, B., MENACHE, I., NAOR, J., AND YANIV, J. Efficient Online Scheduling for Deadline-Sensitive Batch Computing. In *SPAA* (2013).

[9] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute cluster. In *Eurosys* (2013).

[10] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *ACM ICAC* (2011).

[11] WIEDER, A., BHATOTIA, P., POST, A., AND RODRIGUES, R. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *NSDI* (2012).

[12] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce performance in heterogeneous environments. In *OSDI* (2008).

[13] IBM, ILOG CPLEX. `http://www.ibm.com`.

[14] Sailfish Project. `http://code.google.com/p/sailfish/`.

[15] Amazon EC2 Spot Instances. `http://aws.amazon.com/ec2/spot-instances/`.