Preliminary version. Final version to appear in 8th IEEE International Conference on Cloud Computing (IEEE Cloud 2015), June 27 - July 2, 2015, New York, USA

Toward Optimal Resource Provisioning for Cloud MapReduce and Hybrid Cloud Applications

Arkaitz Ruiz-Alvarez, In Kee Kim, Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA, USA

Abstract—**Given the variety of resources available in public clouds and locally (hybrid clouds), it can be very difficult to determine the best number and type of resources to allocate (and where) for a given activity. In order to solve this problem we first define the requested computation in terms of an Integer Linear Programming (ILP) problem and then use an efficient ILP solver to make a provisioning decision in a few milliseconds. Our approach is based on the two most important metrics for the user: cost and job execution time. Thus, based on the user's preferences we can favor solutions that optimize speed or cost or a certain combination of both (e.g. cheapest solution that meets a certain deadline). We evaluate our approach with two classes of cloud applications: MapReduce applications, and Monte Carlo simulations. A significant advantage in our approach is that our solution has been proved optimal by the ILP solver; the set of the scheduling decisions based on our model are plotted on a time vs. cost graph that forms a Pareto efficient frontier. This way, we can avoid the pitfalls of a naïve strategy that can lead to a great increase in cost (91%) or job running time (21%) compared to optimal.**

Keywords-cloud computing; ILP; resource provisioning

I. Introduction

Cloud computing [1] has the potential to alter the traditional role of the scheduler, which has been to control access to certain shared resources (e.g. a single CPU or a cluster). The scheduler assigns these resources to processes, threads or parallel applications in such a way that optimizes throughput, latency, fairness and/or other metrics. However, in a cloud environment, applications within a reasonable range of computing requirements operate under the abstraction that computational resources are unlimited and almost immediately attainable. For example, in a local cluster (managed by PBS [2] or another scheduler) an application user may generate a request of 25 machines for 40 minutes for the scheduler and then the user will wait for the application to run. In cloud computing the user may request the application to immediately boot 25 machines in Amazon EC2 [3] and run the application for as long as it needs to. A scheduler that simply replies to requests composed by a number of machines and the job's execution time seems trivial to implement in cloud computing since there is no need to worry about (virtual) resource sharing. In this research we ask the following questions: Is this interface the correct one for cloud computing applications? Should a scheduler optimize some metric instead of blindly processing a request?

We believe that there are new requirements for resource managers for certain cloud applications (e.g.[4][5][6],[7]). Thus, we are not looking for a general scheduler, but for a scheduler that takes advantage of the characteristics of a specific class of applications to produce a scheduling plan that is more cost-effective and/or faster. The first class of applications that we consider in this paper is MapReduce [8] jobs. (Note that we are not researching *where* to place a MapReduce job *within* a given MapReduce cluster). The second one is simulations whose running time can be estimated (similar also to Monte Carlo algorithms). For each, we generate an Integer Linear Programming (ILP) problem based on the information from the cloud providers, the local resources and the application. The solution to this ILP problem will be the optimal assignment of compute resources based on cost or execution time.

To make an optimal scheduling decision for a MapReduce job we need to know: the amount of work to be done [9], the processing capacity of each instance (and its cost) and a constraint on the job's execution time (provided by the user at submission time). The output of our algorithm will be the amount and type of instances to use during the map, the same during the reduce phase, and an estimation of the cost and duration. For example, the output could be "Start 2 High CPU Medium instances and 17 High CPU Extra-large instances and run for 3 hours at a cost of $6.30". As we will show in the next sections, the scheduling decisions made by this strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives (and often both by a margin as high as 2.6x cheaper and 5.46x faster for naïve strategies).

Our second application class is large-scale Monte Carlos in hybrid clouds. Our representative case of this class is our Watershed model calibration system [10][11], which attempts to calibrate a watershed model by comparing the simulation results to actual observations. Each job starts 1,000 different tasks that can be assigned to a local Windows HPC cluster or to instances in Microsoft Azure [12]. Using the same information that we gather for the MapReduce we present our ILP solver with a similar problem; the solutions are also optimal and form a Pareto frontier. We compare our approach with both naïve and incremental strategies. Our approach avoids the potential pitfalls of naïve scheduling where the cost can be as high as

91% more. The incremental strategy uses the same information as our ILP solver but it still can be up to 11% slower or 38% more expensive.

A variation to this scheduling algorithm is the possible replication of the input datasets. Our final evaluation is a watershed model calibration with 5 different models, whose input data may be stored in Microsoft Azure, in the local cluster or in both. We generalize our previous definition to support different tasks and data location. We compare our strategy with a data co-location algorithm (which starts the computation where the data is). Our approach is superior to data co-location even in data co-location's best case (from 5.4% to 18.8% more expensive for the same running time).

In summary, the contributions of this paper are:

- We introduce two ILP problem formulations whose solutions provide optimal scheduling decisions for two classes of cloud applications.
- We show that a MapReduce job using our algorithm can be completed faster and cheaper that a naïve strategy, often significantly (2.6x cheaper and 5.5x faster).
- We show that a Monte Carlo job in a hybrid cloud using our algorithm is provided optimal scheduling solutions that are more cost effective than other advanced strategies (up to 11% slower and 38% more expensive for *incremental*), even comparing it to the other's best case scenario (*data co-location* is 5.4% to 18.8% more expensive).

The rest of the paper is organized as follows: we review related work in Section II. We introduce the ILP problem formulation strategies and solver in Section III. The MapReduce use case is presented in Section IV. We present the results of our research with Monte Carlo simulations in hybrid clouds in Section V before concluding with Section VI.

II. Related Work

Hybrid clouds and "cloudbursting" are logical extensions of the already existing local resources, augmented with the capabilities of the cloud. This type of hybrid computing [13] has been examined as a potential solution for taking advantage of the scalability benefits of the cloud while keeping the performance benefits of the local cluster. These Elastic Clusters are based on software such as OpenNebula [14], Eucalyptus [15] or in application specific middleware [16] (Windows HPC and Windows Azure integration). Interesting algorithms for a cloud bursting scheduler have also been suggested [17]; our approach differs in that we do not consider a job queue but rather make a decision on each job as they come and we are focusing on application specific jobs rather than on a general scheduler (albeit the information required by both approaches is similar). This last approach from S. Kailasam et al. also focuses on data intensive computing, similar to the approach taken by T. Bicer et al. [18] where they suggest middleware to expand the resources available to MapReduce jobs while taking into account the transfer of data for data intensive jobs. In our present work we focus more on the one datacenter approach for MapReduce and the choices related to the computational resources, thus we do not take into account data storage. In our second use case, Watershed, we have not seen watershed models (the input data) of more than a few megabytes; even if the models grow up to 1 GB (which we account for in our work) the computational costs are going to exceed data storage costs by a wide margin. We believe that the data allocation should be planned in advance [19]; however it could be possible to in the future extend our approach to account for multiple datacenters and possible data transfers.

We have also mentioned that our approach uses ILP solvers; this work is based upon and could not have been possible without the improvements made on the efficiency of Boolean satisfiability [21] and ILP solvers [22].

III. ILP Solver

The core of our approach is to generate an Integer Linear Programming (ILP) problem based on the input data and whose solution is the scheduling decision (number and type of instances at job start time and over time). We need to use an integer linear programming model since the number of instances that are used at any given time needs to be an integer number. The main advantage to this approach is that the ILP solver will return an optimal solution; thus no other approach can be better (at most is equal), given the problem model. The two potential issues with this approach are the validity of the problem model and the running time. In the next sections we will present the problem model for each MapReduce and Watershed.

Based on our previous experience we do not consider the running time of the ILP solver to be an issue. Even though the problem is NP hard, recent solvers are remarkably efficient for problems that are small enough. In previous research [19] [20] we used an ILP approach to provide a data management algorithm for cloud storage. In this paper we found that a problem can be solved under a second if we have less than 2,000 variables; in our model that would be equivalent to managing storage data for an application with 3 different datasets where there are 48 different valid storage systems for each dataset. As we will present in the following sections, problem sizes for our scheduler are smaller than this: the number of variables for Watershed are $O(n*m)$, where n is the number of machine types (5 for each Windows Azure datacenter that we consider) and m is the number of time intervals (12 for the examples presented in this paper). The constant factors behind the O notation are also small (lesser than 2). The average time the scheduler spends in the solver phase for Watershed is 9 milliseconds. Thus, we believe that this approach will remain viable even if we scale up the number of datacenters or machine types.

Our scheduler was written in C# with the solver interface being the Microsoft Solver Foundation and the actual solver used is lpsolve [22]. The results based on simulation were run in a desktop machine (Dual Core 2.40 GHz, 2 GB of RAM). The experimental results with Watershed were run on Windows Azure in the Chicago datacenter and our local Windows HPC cluster with 16 cores.

IV. MapReduce

MapReduce is a popular framework for data parallel applications with an open source implementation, Hadoop

[23]. The Amazon cloud offers running Hadoop jobs as a service, which is called Elastic MapReduce (EMR). A user is able to go to the website, select the type and number of instances to run, select the input and output data, and start the job execution. Amazon also offers a programmatic interface that on top of the job configuration and starting commands includes others such as resizing the number of instances being used.

To provide an optimal decision, we need an estimation of the time it takes to process the map and reduce phases on different machines. This estimation comes from benchmarking the application on different instance types. Since the MapReduce jobs are data parallel it should be possible to obtain a good estimation of the running time of an application by executing a subset of it. Aside from this benchmarking information we need a deadline from the user; our solution will find the cheapest way to run the job and meet the deadline.

A. Problem Formulation

We model a MapReduce job as a set of instances of possibly different types during the map phase and another set of instances for the reduce phase. We find a glossary of all the variables used in the following equations and their type in Table 1. In order to find the cheapest solution for the user we generate an ILP problem in which the objective functions is the cost of the job:

$$MIN\colon \sum_{i,j} cost_{i,j} \qquad (1)$$

Here, each variable $cost_{i,j}$ has two possible values: 0 or the cost of running the job using i hours for the map phase and j hours for the reduce phase. Therefore, all the $cost_{i,j}$ will have a value of 0, except the one with the minimum cost, which corresponds to the solution of our problem. Each $cost_{i,j}$ is related to the variables $xm_i xr_j$, which are 0-1 integer variables. Only one variable will be equal to one, and that variable corresponds to the number of hours for the map (i) and reduce (j) phases in the solution. For example, if the duration is 5 billable hours for the map time and 1 billable hour for the reduce time then $xm_5 xr_1$ will be one. Thus, we would like to define each $cost_{i,j}$ as the actual cost multiplied by the corresponding $xm_i xr_j$. This, however, will lead to a non-integer constraint, so we convert the problem by introducing three different constraints instead of one non linear: $cost_{i,j} = (i * cost\ per\ hour\ map + j * cost\ per\ hour\ reduce) * xm_i xr_j$. We accomplish this by using a *MAXMONEY* constant (an amount of money that cannot be spent by the job) and the following (2), (3) and (4):

$$cost_{i,j} \leq MAXMONEY * xm_i xr_j \qquad (2)$$

$$cost_{i,j} - i * cost\ per\ hour\ map - j * cost\ per\ hour\ reduce - MAXMONEY * xm_i xr_j \geq -MAXMONEY \qquad (3)$$

$$cost_{i,j} - i * cost\ per\ hour\ map - j * cost\ per\ hour\ reduce + MAXMONEY * xm_i xr_j \leq MAXMONEY \qquad (4)$$

In our example, every $cost_{i,j}$ will be zero via (2), except $cost_{5,1}$ which will have the correct value (via (3) and (4)). Equations (5) and (6) define the variables *cost per hour map* and *cost per hour reduce* and (7) enforces the restriction that only one $xm_i xr_j$ should not be zero:

$$cost\ per\ hour\ map = \sum_k cost_k * core\ map\ instances_k \qquad (5)$$

$$cost\ per\ hour\ reduce = \sum_k cost_k * core\ reduce\ instances_k \qquad (6)$$

$$\sum_{i,j} xm_i xr_j = 1 \qquad (7)$$

We are only using core instances. If the user wants to take advantage of the lower cost of spot instances we can trivially add the variables *spot map (reduce) instances$_k$*. The only restriction is that spot instances must have a fixed amount of work assigned to them (for example, 20%) to keep the constraints linear. Also, in this case the user has to acknowledge the risk of having instances shut down and the job finishing later than expected. We need also to enforce the following constraints: if we choose to run the job for a certain amount of time ($xm_i xr_j$) the capacity must be there to finish it before the deadline. Thus, we introduce *total work per hour map* and *total work per hour reduce*, which represent the compute capacity for each phase.

$$total\ work\ per\ hour\ map = \sum_k capacity_k * core\ map\ instances_k \qquad (8)$$

Table 1. MapReduce Model Variables

Name	***Description***	***Source***
$cost_{i,j}$	Float, cost of running the job for *i* map hours and *j* reduce hours	Solution
cost per hour map, cost per hour reduce	Floats, hourly job cost during the map and reduce phases	Solution
$cost_k$	Float, hourly cost for instance type k.	Input
core map (reduce) instancesk	Integer, number instances of type k to run during the map (reduce) phase	Solution
$xm_i xr_j$	0-1 variable, 1 if job runs for i billable map hours and j reduce hours	Solution
total work per hour map (reduce)	Float, work done per hour of map (reduce) computation	Solution
capacityk	Float, work done by instance type k per hour	Input
xmi (xrj)	0-1 variable, 1 if job runs for i (j) billable hours during the map (reduce)	Solution
xml (xrm)	0-1 variable, 1 if job runs for l (m) hours during the map (reduce) where l (m) belongs to {0, 0.25, 0.5, …, i (j)}	Solution
MAP WORK, REDUCE WORK	Float, amount of work to be done in each phase	Input
JOB TIME LIMIT	Float, time in hours by which the job should be expected to finish	Input

$$total\ work\ per\ hour\ reduce = \sum_k capacity_k * core\ reduce\ instances_k \quad (9)$$

If we choose to run the map phase for five hours that means that *5 * total work per hour map* must have enough capacity to finish the required work (MAP WORK and REDUCE WORK is the input data from the user). Thus we add the following $i + j$ constraints:

$$i * total\ work\ per\ hour\ map \geq MAP\ WORK * xm_i \quad (10)$$

$$j * total\ work\ per\ hour\ reduce \geq REDUCE\ WORK * xr_j \quad (11)$$

Here we have introduced another 0-1 integer variables xm_i and xr_j. If xm_ixr_j equals 1, then xm_i and xr_j are 1; otherwise they equal 0. These equations will give us the minimum cost for the job; however we cannot be sure about the job duration because these variables represent billable hours of computation. If a computation takes 2.5 hours for map and 0.25 hours for reduce there are 3 billable hours for map and 1 for reduce. The job duration is not 4 hours though, but 2.75. In order to increase the accuracy up to the quarter of hour we introduce new 0-1 variables xm_l and xr_m similar to the ones defined before:

$$l * total\ work\ per\ hour\ map \geq MAP\ WORK * xm_l\ where\ l \in \{0.25, 0.5, \ldots, i, i.25, i.50, i.75\} \quad (12)$$

$$m * total\ work\ per\ hour\ reduce \geq REDUCE\ WORK * xr_l\ where\ m \in \{0.25, 0.5, \ldots, j, j.25, j.50, j.75\} \quad (13)$$

These new variables allow us to introduce the deadline constraint:

$$\sum_l l * xm_l + \sum_m m * xr_m \leq JOB\ TIME\ LIMIT \quad (14)$$

Finally we must take into account that of the x variables in each category to be 1 (and the rest 0), the relationship between xm_ixr_j , xm_i and xr_j ; and the maximum amount of instances running at the same time (Amazon's restriction):

$$\sum_i xm_i = 1\ , \sum_j xr_j = 1\ \ , \sum_l xm_l\ = 1, \sum_m xr_m = 1 \quad (15)$$

$$0.5 * xm_i + \ 0.5 * xr_j \geq xm_ixr_j \quad (16)$$

$$\sum_i core\ map\ instances_i \leq 20\ ,\ \sum_j core\ reduce\ instances_j \leq 20 \quad (17)$$

B. Use Case

Here we present an example of the results produced by our algorithm. We schedule a MapReduce job whose map phase requires 1,000 hours of work (running on Amazon's Small instance, single core). The reduce phase requires 50 hours of work (taking also the Small instance as the reference). One of the requirements is that we have data on how long it takes to run on different instance types, for example an Extra large with 8 compute units will be able to process 8 times more work than a Small instance. We compare our approach (ILP Solver) with the naive strategy of selecting a number and type of instances and running them till job completion. Our results are shown in Figure 1.

Each data point represents a scheduling plan that results in a certain amount of money spent and an estimated job execution time. The naive strategies here start a fixed number of instances of different types. For example, the data points in the *naive 10 instances* strategy will be: use 10 Small instances, use 10 Large Instances, use 10 Extra Large instance, etc. Using 10 High CPU Extra Large instances results in a running time of 5.25 hours and a cost of \$7.2, this is the point (7.2, 5.25) in the graph. For the ILP Solver strategy we give the algorithm a deadline, for example 3 hours, and it returns a scheduling plan (2 High CPU Medium instances and 17 High CPU Extra-large instances for 3 hours for the map and reduce phases) and a cost (\$6.3). If the deadline is not possible to meet then the ILP solver will detect this situation and notify the user. As we can see in this graph, the ability of stopping instances when they are not needed and sizing them properly for the amount of work to be done can greatly reduce the cost and/or the execution time. ILP Solver can complete the job in 45 minutes at a cost of \$9.62; the naïve strategies' solutions go up to 21x slower and up to 2.6x more expensive and often they are both. The ILP solver data points in the graph show the shape of a Pareto frontier.

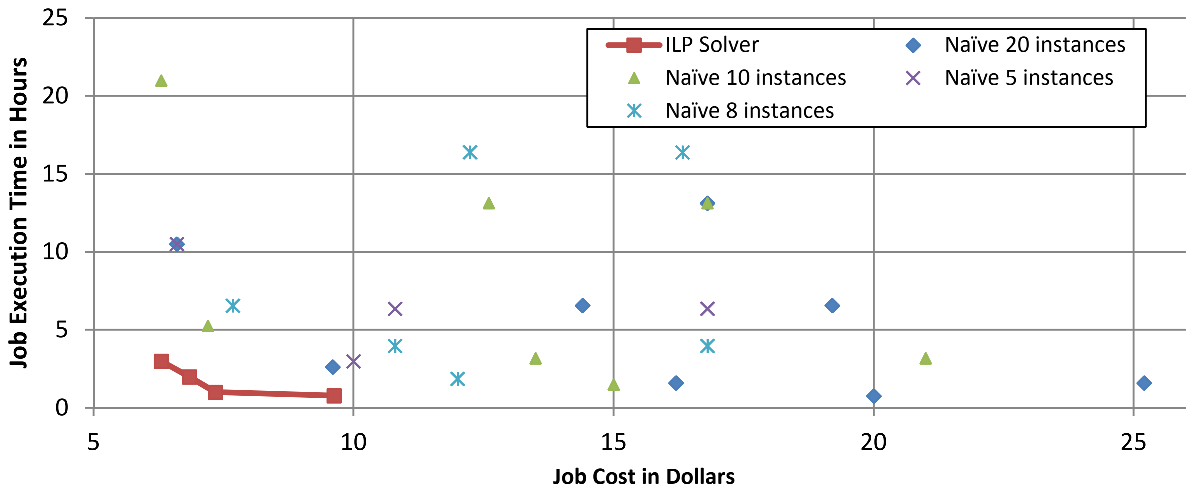

Fig. 1. Job cost and execution time for different MapReduce schedulings. Naïve estrategies start *n* instances of a type (different points in the graph represent a different instance type choice). ILP Solver selects a variable number of instances (and types) for the map and reduce phases.

V. Monte Carlo Simulations in Hybrid Clouds

In this section we present our scheduling algorithm for large-scale Monte Carlos in Hybrid Clouds, specifically our application to perform Watershed model calibration. In essence, the user uploads a watershed model through the Web interface and introduces the parameters for the search. The goal is to calibrate the model by comparing its simulation-based outputs to the actual measurements. Once the job is submitted, the node in charge of the execution distributes the data to the worker nodes and starts 1,000 tasks. Each task analyzes a subset of the search space, which is partitioned in such a way that we can predict each task's duration. Once all the tasks are finished the best result (that is, the one that is closest to reality) is returned to the user. The worker nodes can be located in the local Windows HPC Cluster or in Windows Azure (Chicago datacenter).

A. Problem Formulation

Similar to the problem model in MapReduce, we aim to minimize the cost of the Monte Carlos while trying to finish the job execution before a deadline (a reference of all variables and constants used can be found in Table 2). Our objective function will be simply:

$$MIN: cost \qquad (18)$$

In order to formulate the problem we need to define a series of time intervals. For example, we can use 10 minutes as the duration of our time intervals; if a job needs to finish in less than 1 hour and a half then we will consider 9 time intervals. We should highlight that we are using intervals of equal duration for simplicity; nothing in our model precludes us from using an arbitrary division of time. For each time interval we can then calculate the amount of work that can be done by the different available machines. Thus, we introduce the following variables, $x_{i,j}$ and $work_{i,j}$:

$$capacity_{i,j} * x_{i,j} \geq work_{i,j} \qquad (19)$$

$$For\ every\ i, j: x_{i,j} \leq AVAILABILITY_j \qquad (20)$$

$x_{i,j}$ is a positive integer that represents the number of instances of type j that are active in interval i. $work_{i,j}$ is the amount of tasks that can be processed by all the instances of type j in the same interval. The first intervals will be an special case since we need to take into account the time it takes to boot a virtual machine in Windows Azure (or the waiting time for a local core) and the time it takes to transfer the model data. Thus, the first (or firsts if one interval is not enough to initialize a worker) $capacity_{i,j}$ can be either zero or a lower number than the regular capacity of each instance. Equation (20) adds an upper bound for every variable $x_{i,j}$: this upper bound maybe equal to the number of cores available (for the local cluster) or the maximum number of concurrent instances allowed (for Windows Azure). With all the $work_{i,j}$ defined, we can add the restriction which enforces that all tasks can be processed:

$$\sum_{i,j} work_{i,j} \geq NUMBER\ OF\ TASKS \qquad (21)$$

Table 2. Monte Carlo: Watershed Model Variables

Name	***Description***	***Source***
cost	Float, cost of running the job	Solution
xi,j	Integer, number of instances of type j running concurrently during interval i	Solution
worki,j	Float, amount of done work by all instances of type j during interval i	Solution
billingh,j	Integer, number of billable instances of type j running during hour h	Solution
hourlyCostj	Float, cost in dollars of running an instance of type j for an hour	Input
minuteCostj	Float, cost in dollars of running an instance of type j for a minute	Input
capacityi,j	Float, work done by instance type j during interval i	Input
NUMBER OF TASKS	Integer, work to be done (1,000 by default in watershed)	Input
AVAILABILITYj	Integer, maximum number of concurrent instances of type j	Input

The next step for us is to add a restriction that makes the series over time of the number of instances of a certain time monotonically decreasing. Basically, if at some point during the computation we are going to need 20 small instances in Windows Azure, start them at the beginning. Thus, if the user sets a deadline of 2 hours but the job could be done in an hour and a half (at the same cost), then these linear constraints will make sure that our ILP solver finds the fastest solution:

$$For\ each\ instance\ type\ j, interval\ i: x_{i,j} \geq x_{i+1,j} \qquad (22)$$

It is time now for us to define the *cost* variable. First we will introduce the formulation that takes into account the current hourly billing model in Windows Azure. In order to do so, we introduce new variables $billing_{h,j}$. For every hour of the computation we get billed the maximum number of active instances at any time during the hour. So, we add the following constraints for every machine type j:

$$For\ every\ interval\ i\ in\ hour\ h: billing_{h,j} \geq x_{i,j} \qquad (23)$$

Then, the cost is simply the weighted sum of all billing variables:

$$\sum_{h,j} hourlyCost_j * billing_{h,j} = cost \qquad (24)$$

An alternative formulation may be useful for certain contexts in which billing by the minute is more appropriate. For example, multiple scientists share the same application and residual capacity from other job runs can be used for the current job submission. In this case, the staff in charge of managing the application may choose to bill users by the minute (and maybe include the amortized cost of the wasted capacity). Or it can be the case that each application run lasts for several hours or even days and the wasted cost of the last hour of computation is negligible. In this case, we can skip

(23) and (24) from our problem formulation and directly define cost using the variables $x_{i,j}$:

$$\sum_{i,j} interval\ length_i * minuteCost_j * x_{i,j} = cost \quad (25)$$

In the presentation of our problem formulation we have introduced the definition of intervals instead or allowing an arbitrary amount of time. This is because in order to calculate the cost of running the application we need to multiply the number of instances by the time they are active. If both the number of instances and running time are unknowns this will produce a non linear constraint that cannot be processed by our solver. So, if we divide the time into intervals (it does not matter if they are equal in size) we can calculate the interval cost and capacity before generating the problem. Therefore, interval cost and capacity are constants and the model can be expressed linearly. There are a couple of disadvantages, though. The first one is that if a user asks for a 73 minute deadline our application will process it as a 70 minute deadline. We feel that in our application this is not an issue, and we can always increase the number of intervals to get a 5 minute resolution (or better). The second disadvantage is that the size of the problem increases linearly with the application's running time. We are not concerned about this since it is always possible to define intervals to be one hour or more, get the solution and finally refine the result. For example, an application that has a deadline of 34 hours could be processed with 2 hours intervals. If the resulting scheduling plan takes 30 hours we can a) use this result if it's accurate enough b) rerun the problem by introducing a big 24 hour interval at the beginning and then 30 minute (or less) increment intervals. For our watershed application, however, we have found that 10 minute intervals work well.

The solution to our problem will give us the number of active instances of each type that will be active at each interval ($x_{i,j}$), the number of tasks that can be processed during each interval by each set of machine types ($work_{i,j}$), the cost (*cost*) and execution time (max interval for which any $x_{i,j}$ is greater than zero).

B. NP Hardness

Here we outline a proof of the NP hardness of the problem model just introduced. Thus, we can show that an optimal solution can only be obtained by using an exponential algorithm; although, as we have argued previously, the typical problem size that we encounter can be solved successfully within milliseconds by lpsolve. Other strategies, such as the naive and incremental ones that will be introduced with our simulation and experimental results, could approximate or even equal the optimal solutions in some cases, but not in all of them (since they are polynomial time algorithms).

The basic idea is to perform an efficient reduction from the bounded knapsack problem to our problem formulation (the minute billing version). Thus, the input to the reduction algorithm is the *n* items, the maximum weight *W*, the value of each item v_j, the availability of each item c_j, and the weight of each item w_j. The transformation is simple, the number (and length) of intervals is always 1, the *NUMBER OF TASKS* is *-W*, the $AVAILABILITY_{0,j}$ are the c_j, the $minuteCost_j$ are $-v_j$, and the $capacity_j$ are the $-w_j$. This reduction is trivially done in polynomial time.

The solution to our billing problem is the $x_{0,j}$; that is the number of instances of each type running during the only interval. Each $x_{0,j}$ corresponds to the number of items of each type that will be in the knapsack, x_j. Each $x_{0,j}$ will obey the bounded knapsack problems restrictions since $x_{0,j} \leq AVAILABILITY_{0,j}$ implies $x_j \leq c_j$. The capacity restriction is also taken into account since if we combine equations (19) and (21) we have:

$$\sum_j capacity_j * x_{0,j} \geq NUMBER\ OF\ TASKS \rightarrow$$
$$\sum_j w_j * x_j \leq W \quad (26)$$

since in the reduction we have effectively multiplied by -1 the equation by assigning *-W* and $-w_j$ to *NUMBER OF TASKS* and $capacity_j$. Finally, the solution to the scheduling problem (combining equations (18) and (25)) has to correspond to the solution to the knapsack problem; we can arrive to this conclusion simply by substituting the variables assigned during the reduction:

$$MIN: \sum_{i,j} interval\ lenght_i * minuteCost_j * x_{i,j} \rightarrow$$
$$MIN: \sum_j 1 * (-v_j) * x_{0,j} \rightarrow MAX: \sum_j v_j * x_j \quad (27)$$

Thus, it holds that in order a solver for our scheduling problem formulation (minute billing) will also solve the bounded knapsack problem (Bounded Knapsack $\leq_P$ Scheduling), proving its NP hardness. From this point it is easy to see a further reduction to the hourly billing version from the minute billing version of the problem in which we divide the capacity and cost arrays by 60 to convert hours into minutes and then adjust the number of intervals accordingly. We do not want to imply that there are no good polynomial-time algorithms for schedulers to implement. Indeed, as we will see in the next sections algorithms that are not naive and take into account the same information available to our ILP solver can do well under certain circumstances. Our argument is that, given that an optimal algorithm which is $O(2^n)$ in theory but it is fast in practice for the size of problems we are interested in solving, this algorithm should be the preferred choice.

C. Use Case with Hourly Billing

We present the results of our approach with the hourly billing model in Figure 2. For this use case the watershed model calibration requires running 1,000 tasks, where each task takes 90 seconds to run on a single core and the input data size is 1 GB. We compare our approach with two different strategies, *naïve* and *incremental*. The *naïve* approach is simple: the scientist selects the number of instances to run for the complete duration of the job, for example, 20 instances. The scheduler selects all the available local nodes and if that's not enough it starts several instances in Windows Azure to reach that number. In our example we always use 16 local cores and start anywhere from 0 to 44 small instances in Windows Azure. The dotted-square line in the graph plots the duration of each job against its cost (local instances are considered free). This line makes a Z type curve around the 1 hour limit; even if we select other instance types we observe the same phenomena. The cause of this shape is the billing model in Windows Azure, where the user pays by the hour,

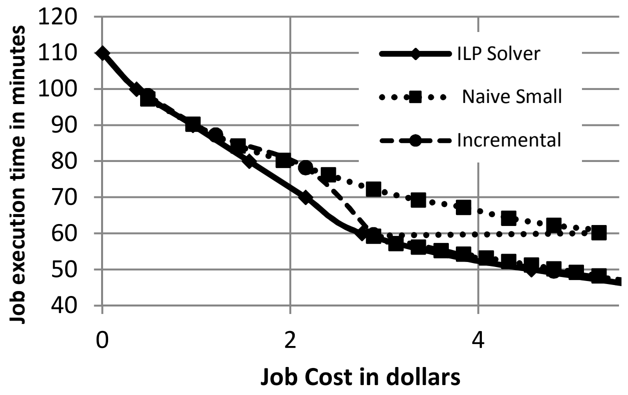

Fig. 2. Job cost and execution time for different Watershed schedulings. Naïve estrategies use 16 local cores and additional instances of a given type in Windows Azure. ILP Solver selects a variable number of instances (and types) over time.

regardless of where she uses 1 or 59 minutes of time. A job that with the help of 20 small instances lasts for 63 minutes costs \$4.8 since we are paying for 40 compute hours. However, had the user selected 24 small instances the job would have finished in just under one hour and the bill would have been \$2.88 (24 billable compute hours). Thus, we can see here the two main problems with the naïve approach: the user interface and the suboptimal choices. The user interface asks the user for the number of instances but that is not a metric important to her; the metrics the scientist cares about are cost and execution time. There is no way for the user to relate cost and time to number of instances unless she has previous experiences with the system. The other problem is that we can clearly see the suboptimal choices that are made because of the hourly billing model in cloud computing. The worst case scenario for the naïve strategy is at the 61 minutes mark, where it costs \$5.28 to run the job, compared to the \$2.76 cost using our strategy.

We introduce another approach, *Incremental*, and compare both of them to our ILP solver. The incremental approach uses benchmarking information to predict the cost and execution time of each job given a set of instances of different types. If we have information about how long each task usually takes on each type of instances, the time it takes to initialize a node and stage in the data then we can calculate the cost and time of each job before it is submitted with a certain configuration. This is the same type of information available to our ILP solver. For this approach the input from the user is the expected execution time. The incremental algorithm starts with the local machines as the starting configuration and calculates cost/time. Then it incrementally adds instances from Windows Azure (mixing different types) up to a limit, and returns the best solution. The best solution is the cheapest one whose completion time is below the user's limit. We can see this approach as simulating every possible naïve strategy and selecting the best one. This way we can avoid the two main problems of the naïve strategy: the interface is now presented in terms of what the user cares about, cost and time; and there are no choices which are clearly wrong (more expensive and slower than other possible configurations). In the graph the dotted-circle line represents this strategy.

Even though the choices for the incremental algorithm are good, we can do better. Remember that in both naïve and incremental we select some instances and run them till job completion time. It is possible to stop some instances before the job completion time to avoid being billed an additional hour. For example, we can start 18 small instances in Windows Azure for one hour and use the 16 local cores for the whole job. This way the job can be completed in 70 minutes and it will cost \$2.16. A comparable data point for the incremental algorithm is 78 minutes and \$2.16; that is, 11.4% slower for the same amount of money. Our strategy (ILP solver) is represented by the solid black line.

As we can see in the graph, both *Incremental* and ILP lines get much closer when the job duration is less than one hour. This is expected, since the main advantage for the ILP strategy (stopping instances at hour intervals to avoid incurring charges because of partial hours) is lost. However, we still consider it to be the best mechanism since it gives the best solution in all instances. *Incremental* may get close to the best solution for jobs under one hour, but these jobs may not be that common in practice since they are twice or thrice more expensive to run in exchange for a 25% or 33% (15 or 20 minutes) decrease in job execution time.

D. Multiple Watershed Models

In our last use case the computational job is divided into 1,000 tasks, where the only difference between tasks is some input parameters (each tasks runs the same watershed model). In this section we present the modifications to our model in order to support a more complex scenario:

- The job is composed by k different models where each model is composed by $NUMBER\_OF\_TASKS_k$ tasks. The execution time varies for each model; all tasks of one model last the same amount of time.
- The input data may be stored locally, in Windows Azure or in both.

The objective function is the same one as in the last use case, equation (18). In order to account now for the different watershed models we have to modify existing variables. Basically we add need the k suffix in order to account for the k different models. The new variables $work_{i,j,k}$ and $x_{i,j,k}$ are defined in:

$$capacity_{i,j,k} * x_{i,j,k} \geq work_{i,j,k} \qquad (28)$$

$$\forall\, i,j\colon \textstyle\sum_k x_{i,j,k} \leq AVAILABILITY_j \qquad (29)$$

The following constraint enforces that the number of machines (local or in Windows Azure) used for this job decreases monotonically. Thus, if we need to ask for 16 local cores at some point during the job execution, do it at the beginning:

$$For\ each\ instance\ type\ j, interval\ i:$$
$$\sum_k x_{i,j,k} \geq \sum_k x_{i,j,k} \quad (30)$$

In our last use case we knew that at the start of every job every machine that was participating will be staging in the necessary data. Here we do not have that advantage since a machine could be processing job 1 for the first half an hour and then switch to job 2. Thus, the initialization can happen at any time and we cannot zero out (or decrease) the *capacity*$_{i,j,K}$ constants, which is what we did before. Instead we add the initialization time as negative work done; this is used in every machine so we have to multiply this quantity by the maximum number of machines that participate processing watershed model k:

$$For\ each\ instance\ type\ j, watershed\ model\ k:$$
$$\sum_i work_{i,j,k} - InitPenalty * max\ machines_{j,k} \geq total\_work_{j,k} \quad (31)$$
$$For\ each\ interval\ i, instance\ typej, watershed\ model\ k:$$
$$max\ machines_{j,k} \geq x_{i,j,k} \quad (32)$$

We also need to add a constraint to make sure that all tasks for each watershed model are completed:

$$\forall\ k: \sum_j total\ work_{j,k} = NUMBER\ OF\ TASKS_k \quad (33)$$

Finally, we define the new *cost* variable. In this case it will be composed by the compute costs and possibly data transfer costs. In order to define the transfer costs we need to introduce new binary variables, *transfer*$_{k,l,m}$, which are one in case we run a watershed model in a location where the input data needs to be transferred. For example, running a model locally but the input data is located only on Windows Azure. The definition of these new variables is:

$$For\ each\ instance\ type\ j, watershed\ model\ k:$$
$$\sum_i x_{i,j,k} \leq MAX\ TASKS * t_{k,l,m} \quad (34)$$
$$transfer\ costs = \sum_{k,l,m} input\ data\ size_k * (costGBOut_l + costGBIn_m) * t_{k,l,m} \quad (35)$$
$$compute\ costs = \sum_{i,j,k} interval\ length_i * minuteCost_j * x_{i,j,k} \quad (36)$$
$$cost = transfer\ costs + compute\ costs \quad (37)$$

We compare our approach with a data co-location strategy. In this case we are submitting for execution 5 different watershed models, each one composed of 200 tasks (1,000 tasks total for calibration). The execution time varies from 1 minute to 7.5 minutes (all tasks of one model last the same amount of time). In scenario A the inputs for models 1, 3 and 4 are stored locally and the ones for 2, 3 and 5 are stored in Windows Azure. In scenario B the inputs for models 1, 2, 3 and 4 are stored locally and the ones for 3 and 5 are stored in Windows Azure. For scenario A models 2, 3 and 5 run on Windows Azure and in scenario B models 3 and 5 do. We present our experimental results on Figure 3. In the graph each data point in the data co-location strategy's curve represents a job execution with a fix number of cloud instances of the medium size (from 1 to 32). For the ILP solver, on the other hand, it represents a configuration (the solution to the ILP problem) where the

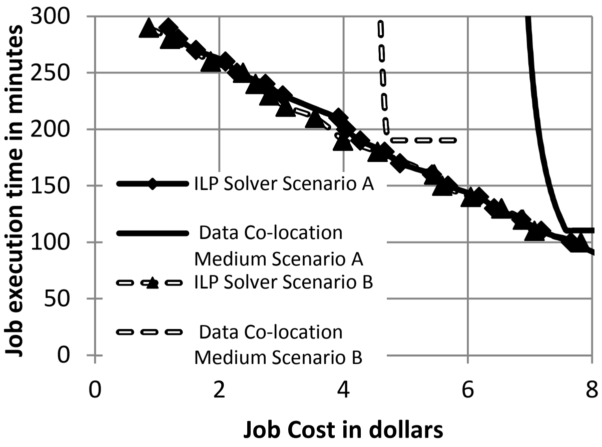

Fig. 3: Scheduling cost and turnaround time for multiple watershed models.

maximum number of intervals has been set. The data co-location strategy works well when you have the right amount of instances on Windows Azure so that both the local and cloud instances finish around the same time. In the graph this is represented by the data point at the elbow. The advantages of the ILP strategy are two-fold. First, the ILP solver's ability to move the data and tasks around gives the user a lot more options for balancing the cost and execution time. Second, even in the best case data co-location for scenario A is 5.4% more expensive than ILP solver and data co-locations for scenario B is 18.8% more expensive (than ILP solver).

VI. CONCLUSION

New algorithms are needed for cloud schedulers. For both MapReduce and Monte Carlos we have proposed two problem formulations based on ILP. Scheduling decisions made by our strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives; in MapReduce naïve alternatives can often be both by a margin as high as 2.6x cheaper and 5.46x faster. For Watershed we find similar results for naïve strategies, although existing algorithms can still be up to 11% slower and up to 38% more expensive.

REFERENCES

[1] M. Armbrust et al., "A view of cloud computing." Commun. ACM, vol. 53, no. 4, pp. 50-58, Apr. 2010.

[2] R. Henderson, "Job scheduling under the Portable Batch System," in Job Scheduling Strategies for Parallel Processing, vol. 949, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 1995, pp. 279-294.

[3] Amazon, "Amazon Web Services," 2015.

[4] J. Li, Q. Wang, Y. Kanemasa, D. Jayasinghe, S. Malkowski, P. Xiong, M. Kawaba and C. Pu."Profit-Based Experimental Analysis of IaaS Cloud Performance: Impact of Software Resource Allocation", Services Computing (SCC), 2012 IEEE Ninth International Conference on. IEEE, 2012.

[5] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu and H. Hacigumus, "ActiveSLA: A Profit-Oriented Admission Control Framework for Database-as-a-Service Providers", In Proceedings of ACM Symposium on Cloud Computing (SOCC'11), Oct. 2011.

[6] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe and C. Pu. "Economical and Robust Provisioning of N-Tier Cloud Workloads: A Multi-level Control Approach", In Proceedings of IEEE International Conference On Distributed Computing Systems (ICDCS) (ICDCS'11), June 2011.

[7] I.K. Kim, J. Steele, Y. Qi, and M. Humphrey. "Comprehensive Elastic Resource Management to Ensure Predictable Performance for Scientific Applications on Public IaaS Clouds." Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2014). Dec 2014.

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp. 107-113, Jan. 2008.

[9] J. O'Loughlin & L. Gillam. Performance Evaluation for Cost-Efficient Public Infrastructure Cloud Use.. Economics of Grids, Clouds, Systems, and Services. Lecture Notes in Computer Science Volume 8914, 2014, pp 133-145

[10] M. Ercan, J. Goodall, A. Castronova, M. Humphrey, and N. Beekwilder. Calibration of SWAT models using the cloud. Environmental Modelling & Software. 62, 188-196

[11] M. Humphrey, N. Beekwilder, J. Goodall, and M. Ercan. Calibration of Watershed Models using Cloud Computing. Proceedings of the 8th IEEE International Conference on eScience (eScience 2012). Oct 8-12 2012.

[12] D. Chappell, "Introducing the Windows Azure Platform," 2009.

[13] G. Mateescu, W. Gentzsch, and C. J. Ribbens, "Hybrid Computing—Where HPC meets grid and Cloud Computing," Future Generation Computer Systems, vol. 27, no. 5, pp. 440-453, May 2011.

[14] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Elastic management of cluster-based services in the cloud," International Conference on Autonomic Computing, pp. 19-24, 2009.

[15] D. Nurmi et al., "The Eucalyptus Open-Source Cloud-Computing System," Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on. IEEE, 2009.

[16] M. Humphrey, Z. Hill, K. Jackson, C. van Ingen, and Y. Ryu, "Assessing the Value of Cloudbursting: A Case Study of Satellite Image Processing on Windows Azure," in 7th IEEE International Conference on e-Science (escience 2011), 2011.

[17] S. Kailasam, N. Gnanasambandam, J. Dharanipragada, and N. Sharma, Optimizing Service Level Agreements for Autonomic Cloud Bursting Schedulers. IEEE, 2010, pp. 285-294.

[18] T. Bicer, D. Chiu, and G. Agrawal, "A Framework for Data-Intensive Computing with Cloud Bursting," in 2011 IEEE International Conference on Cluster Computing, 2011, pp. 169-177.

[19] A. Ruiz-Alvarez and M. Humphrey, "A Model and Decision Procedure for Data Storage in Cloud Computing," in The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012.

[20] A. Ruiz-Alvarez and M. Humphrey. "An Automated Approach to Cloud Storage Service Selection." Proceedings of the 2nd Workshop on Scientific Cloud Computing (ScienceCloud 2011). June 8, 2011

[21] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers ," Computer Aided Verification, vol. 2404, pp. 641-653, Sep. 2002.

[22] M. Berkelaar, K. Eikland, and P. Notebaert, "lpsolve: Open source (mixed-integer) linear programming system," 2011. [Online]. Available: http://lpsolve.sourceforge.net/.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, 2010, pp. 1-10.