Elastic Virtual Machine Scheduling for Continuous Air Traffic Optimization

Shigeru Imai, Stacy Patterson, and Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
{*imais,sep,cvarela*}*@cs.rpi.edu*

***Abstract*—As we are facing ever increasing air traffic demand, it is critical to enhance air traffic capacity and alleviate human controllers' workload by viewing air traffic optimization as a continuous/online streaming problem. Air traffic optimization is commonly formulated as an integer linear programming (ILP) problem. Since ILP is NP-hard, it is computationally intractable. Moreover, a fluctuating number of flights changes computational demand dynamically. In this paper, we present an elastic middleware framework that is specifically designed to solve ILP problems generated from continuous air traffic streams using a Lagrangean approximation over an IaaS cloud. We propose a model-based speculative VM scheduling algorithm: it implements a time series prediction model to decide when to allocate/deallocate VMs, and it also uses a resource prediction model to estimate how many VMs to allocate/deallocate. Experiments show that our speculative VM scheduling algorithm can achieve a similar performance to a static schedule while using 49% less VM hours for a smoothly changing air traffic. However, for a sharply changing air traffic, our speculative VM scheduling algorithm costs slightly more VM hours to achieve the same performance. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy.**

1. Introduction

The number of flight passengers is expected to reach 7.3 billion by 2034 globally, which requires a 4.1% average growth in flight capacity in every year from 2014 on [1]. Air traffic optimization is crucial to enhance flight capacity and also alleviate human controllers' workload. Air traffic management problems are commonly formulated as integer linear programs (ILP), which are known to be NP-hard [2]. Therefore, large-scale ILP problems are computationally intractable. Moreover, since the number of flights fluctuates a lot in practice, computational demands for air traffic optimization also change dynamically. For example, Figure 1 shows how the number of commercial flights in the U.S. changed over 24 hours from 4am EST on January 18th, 2014. Once air traffic hits the peak at around 1pm, it gradually drops and eventually reaches 200 at around 3am. To keep up with the fluctuating computing demands in a cost-efficient way, we can dynamically allocate and deallocate virtual machines (VMs) from Infrastructure-as-a-Service (IaaS) cloud computing providers. The challenge is dynamically choosing the right number of VMs that satisfies computational demands at the lowest possible cost.

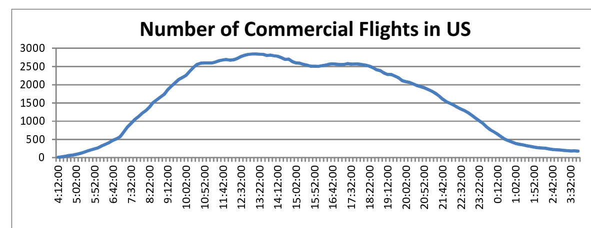


Figure 1. Example of U.S. flights on January 18th, 2014 (created from data available on [3]).

ILP has a lot of practical applications, and some of them have a strong time-dependency. Examples of such applications include: public transportation routing [4], investment portfolio optimization [5], and marketing budget optimization [6], [7]. Just as air traffic management, public transportation routing has similar characteristics: *e.g.*, the number of buses changes depending on time of the day. Investment portfolios must be evaluated periodically to keep up with stock markets. Advertisers look for an optimal way of spending their money across both traditional and online media; however, advertisement costs always change especially for online advertisements such as Google AdWords [8] because of its bidding mechanism [7], [9]. These time-dependent factors can fluctuate the computational cost of optimization to a large degree. Elastic ILP middleware is potentially useful for many application areas considering these applications' time-dependent behaviors.

To implement such an elastic ILP middleware, we can either adaptively adjust the number of VMs at runtime without any prior knowledge of the application or proactively predict the number of VMs using a resource prediction model. The former includes a threshold-based approach, as used in Amazon's Auto Scaling [10], and reinforcement learning [11], [12]. In this paper, we take the latter approach with the intention to improve the resource utilization, cost, and latency violations. That is, we model required computational resources to solve an ILP problem formulated for air traffic

management. Moreover, we use an autoregressive time series prediction model to decide when to allocate/deallocate VMs in a speculative manner.

To the best of our knowledge, this is the first attempt to present an elastic middleware framework specifically designed to solve ILP problems created from continuous air traffic streams. Here is the summary of our contributions:

- We develop an elastic middleware framework (Section 3) to solve optimization problems created from continuously incoming air traffic streams (Section 2) over IaaS clouds. The framework obtains an approximate solution to ILP problems using a two-level optimization technique based on Lagrangean decomposition [13].
- We design VM scheduling algorithms that are specifically designed to solve ILP problems generated from continuous air traffic streams (Section 4). We use a time series prediction model to decide when to allocate VMs and we also use a resource prediction model to estimate how many VMs to allocate. The resource prediction model estimates required VM resources given the number of flight routes and target processing latency by using linear regression.
- Experiments show that our speculative VM scheduling algorithm can achieve a similar performance to a static schedule while using 49% less VM hours for a smoothly changing air traffic. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy.

The rest of the paper is organized as follows. First, in Section 2, we formulate an air traffic management problem as an ILP problem and describe how we apply Lagrangean decomposition to the defined ILP problem. Next, we present the design of our elastic air traffic management middleware in Section 3 and VM scheduling algorithms in Section 4. Then, in Section 5, we present evaluation results for the proposed VM scheduling method. Finally, we present related work in Section 6 and conclude the paper in Section 7.

2. Air Traffic Management Problem

2.1. Problem Formulation

The Link Transmission Model (LTM) [14] is an air traffic flow management model that optimizes nationwide air traffic by formulating it as an ILP problem. Cao and Sun decompose the original LTM problem into multiple subproblems using Lagrangean decomposition and use MapReduce [15] to approximate the solution to large scale LTM problems in parallel [16].

Our work is inspired by their approach. We formulate a simplified version of the LTM problem that captures the computationally intensive nature of the original LTM problem. Figure 2 shows an example of the simplified air traffic management problem. A *route* connects a departure airport and an arrival airport, and it consists of multiple *links* that are distributed over multiple *sectors*. As illustrated in the example, the same sector at the center of the grid is shared by multiple routes, therefore congestion must be controlled.

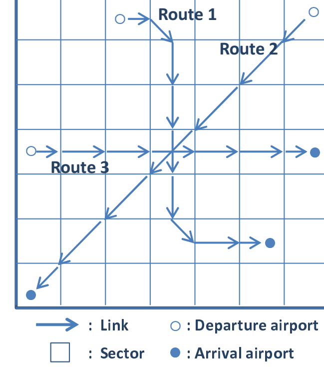


Figure 2. Example of the simplified air traffic management problem.

We can formalize the simplified air traffic management problem as an ILP problem as follows:

$$\text{maximize} \quad \sum_{i=1}^{K} \mathbf{c}_i^{\intercal} \mathbf{x}_i \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{K} A_i \mathbf{x}_i \leqslant \mathbf{b} \tag{2}$$

$$\mathbf{0} \leqslant \mathbf{x}_i \quad \forall i \in [1, K], \tag{3}$$

$$\sum_{j=1}^{N_i} x_i^j \leqslant d_i \quad \forall i \in [1, K], \tag{4}$$

$$\text{where} \quad x_i^j \in \mathbb{Z} \geqslant 0, \quad A_i \in \{0,1\}^{S,N_i}, \quad \mathbf{b} \in \mathbb{Z}^S \geqslant 0, \quad \mathbf{c}_i \in \mathbb{R}^{N_i} \geqslant 0, \quad d_i \in \mathbb{Z} \geqslant 0.$$

The objective of this optimization problem is to assign an ideal number of flights to each sector so that we can maximize the air traffic capacity while satisfying capacity of each sector. Given constants are: the number of routes K, the number of links of the i-th route $N_i (i = 1, ..., K)$, and the number of sectors S. The number of flights for a route i is expressed as a vector $\mathbf{x}_i = [x_i^1, x_i^2, ..., x_i^{N_i}]^{\intercal}$, where $x_i^j \in \mathbb{Z} \geqslant 0$ is the number of flights at link j of the route i. $\mathbf{x}_i (i = 1, ..., K)$ are the variables to be optimized subject to the following capacity constraints:

- **Sector capacity:** Total number of flights in a sector s must be less than or equal to $b_s \in \mathbb{Z} \geqslant 0$ $(s = 1, ..., S)$ (Inequality (2)).
- **Route capacity:** Total number of flights on a route i must be less than or equal to $d_i \in \mathbb{Z} \geqslant 0$ $(i = 1, ..., K)$ (Inequality (4)).

The objective function $\sum_{i=1}^{K} \mathbf{c}_i^{\intercal} \mathbf{x}_i$ is defined with vectors $\mathbf{c}_i = [c_i^1, c_i^2, ..., c_i^{N_i}]^{\intercal} (i = 1, ..., K)$, where $c_i^j \in \mathbb{R} \geqslant 0$ determines the degree of preference of assigning flights

on link j of route i. We can set higher values for less congested sectors and lower values for highly congested sectors. The sector capacity constraint is defined by Inequality (2) using $S \times N_i$ matrices $A_i(i = 1, ..., K)$ and a vector $\mathbf{b} = [b_1, b_2, ..., b_S]$, where $b_s \in \mathbb{Z} \geqslant 0$. For a route i, the mapping of links to sectors naturally dictates the construction of A_i; each element a_{sj} takes the value of 1 if link j is on sector s, otherwise 0. Each element of $\mathbf{b}$ determines the sector capacity of a corresponding sector.

A solution to this problem captures the two important properties of the original LTM problem that affect the computational workload. First, adding a new route increases the number of variables in proportion to the number of links on the route. Second, each A_i matrix is extremely sparse, which significantly affects the difficulty of satisfying constraints because there are very few number of variables in each constraint.

2.2. Lagrangean Decomposition

ILP is NP-hard. Thus, it is common to use algorithms that find an approximate solution in a reasonable amount of time. Lagrangean decomposition is a popular technique to obtain an approximate solution to ILP problems, and it was used for LTM in [16]. Lagrangean decomposition offers a way to split a larger linear integer problem into multiple smaller sub-problems by relaxing *complicating constraints*. For our air traffic management problem described in Section 2.1, the complicating constraint is the sector capacity constraint (Inequality (2)), which prohibits us from separating the original problem into K sub-problems with Inequality (3) and (4). By constructing a Lagrangean relaxation of the original problem, we can bring the complicating constraints to the objective function as a penalty term as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{K} \mathbf{c}_i^\intercal \mathbf{x}_i - \boldsymbol{\lambda}^\intercal \Big(\sum_{i=1}^{K} A_i \mathbf{x}_i - \mathbf{b} \Big) \quad (5) \\ \text{subject to} \quad & \mathbf{0} \leqslant \mathbf{x}_i, \ \sum_{j=1}^{N_i} x_i^j \leqslant d_i \ \ \forall i, \end{aligned}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^S \geqslant \mathbf{0}$ is a vector of Lagrange multipliers. Now, we can decompose the problem (5) into a smaller sub-problems, one for each route i:

$$\begin{aligned} \text{maximize} \quad & \Big(\mathbf{c}_i^\intercal - \boldsymbol{\lambda}^\intercal A_i \Big) \mathbf{x}_i \quad (6) \\ \text{subject to} \quad & \mathbf{0} \leqslant \mathbf{x}_i, \ \sum_{j=1}^{N_i} x_i^j \leqslant d_i. \end{aligned}$$

Next, we define the master dual problem of the Lagrangean (5), which is responsible for updating $\boldsymbol{\lambda}$:

$$\begin{aligned} \text{minimize} \quad & g(\boldsymbol{\lambda}) = \sum_{i=1}^{K} (\mathbf{c}_i^\intercal - \boldsymbol{\lambda}^\intercal A_i) \mathbf{x}_i^* + \boldsymbol{\lambda}^\intercal \mathbf{b} \quad (7) \\ \text{subject to} \quad & \mathbf{0} \leqslant \boldsymbol{\lambda}, \end{aligned}$$

Algorithm 1: Two-level ILP optimization

input : $A_i, \mathbf{c}_i, d_i (i = 1, ..., K), \mathbf{b}, \boldsymbol{\lambda}_{\text{init}}, \delta, I$
output: $\mathbf{x}_i (i = 1, ..., K), minObj$

1 $t \leftarrow 0$;
2 $\boldsymbol{\lambda}(t) \leftarrow \boldsymbol{\lambda}_{\text{init}}$;
3 $minObj \leftarrow Double.MAX_VALUE$;
4 **while** *Iterations* $minObj$ *not improved more than* $\delta\% < I$ **do**
5   // Solve K sub-problems
6   **for** $i = 1$ **to** K **do**
7     $\mathbf{x}_i \leftarrow solveILP(A_i, \mathbf{c}_i, d_i, \boldsymbol{\lambda}(t))$;
8   **end**
9   // Update master objective
10   $obj \leftarrow compObj(A_1, ..., A_K, \mathbf{b}, \mathbf{c}_1, ..., \mathbf{c}_K, \mathbf{x}_1, ..., \mathbf{x}_K, \boldsymbol{\lambda})$;
11   **if** $obj < minObj$ **then**
12     $minObj \leftarrow obj$;
13   **end**
14   // Update λ for the next iteration
15   $\alpha = \frac{1}{t}$;
16   $\boldsymbol{\lambda}(t+1) \leftarrow \boldsymbol{\lambda}(t) - \alpha \cdot gradient(A_i, \mathbf{b}, \mathbf{x}_i)$;
17   $t \leftarrow t + 1$;
18 **end**
19 **return** $\mathbf{x}_i (i = 1, ..., K), minObj$;

where $\mathbf{x}_i^*$ is the optimal solution to the sub-problem (6) for a route i. To solve $\boldsymbol{\lambda}$ for the dual problem (7), we use the gradient method:

$$\begin{aligned} \frac{\partial g}{\partial \boldsymbol{\lambda}} &= \mathbf{b} - \sum_{i=1}^{K} A_i \mathbf{x}_i^* \quad (8) \\ \boldsymbol{\lambda}(t+1) &= \boldsymbol{\lambda}(t) - \alpha \frac{\partial g}{\partial \boldsymbol{\lambda}}, \quad (9) \end{aligned}$$

where α is a small positive step-size.

As shown in Algorithm 1, we iteratively solve the K sub ILP problems to find optimal $\mathbf{x}_i (i = 1, ..., K)$ for a specific $\boldsymbol{\lambda}$ (Line 7) and update $\boldsymbol{\lambda}$ for the master dual problem (Line 16). We keep track of the minimum value of objective ($minObj$), and if it is not improved by more than $\delta\%$ for I iterations, we go out from the while loop and return the final results.

3. Elastic Air Traffic Management Middleware

3.1. Background

System interaction. We assume that the user of the middleware is a human air traffic controller who uses output of our middleware for air traffic control activity. We also assume that some flight information providers (*e.g.*, FlightAware [17]) or airplanes directly send the latest flight status information to the middleware (see Figure 3). Since air traffic management is time critical, the middleware tries to schedule VMs so that the optimization result can be used by the user in a timely manner. Hence, the user can configure *latency* to request how quickly the application should return the result.

Cloud deployment. The middleware is designed to work on an IaaS cloud. The IaaS cloud can be private, public, or hybrid; however, the scheduling algorithm presented in Section 4.3 is optimized for public IaaS clouds due to its *billing cycle* aware scheduling. The billing cycle is the unit of monetary charge (*e.g.*, 1 hour for Amazon EC2 [18]). The scheduler only terminates VMs just before their billing cycle so that the application can use the VMs' computing power until the last minute.

3.2. Application Implementation

We use Spark 1.5.1 [19], a general cluster computing engine, to implement Algorithm 1. Spark's high-level abstractions for distributed programming and in-memory data processing features are suitable for the iterative ILP problem solving process. Spark applications run on a cluster consisting of a master node and multiple worker nodes. In Algorithm 1, *executors* running on the worker nodes execute Line 7 to solve K sub-problems in parallel, and the rest of the code is executed on the master node. While Spark allows us to cache parameters $A_i, \mathbf{c}_i, d_i$ for sub-problems on each worker node, the master needs to broadcast the updated value of $\boldsymbol{\lambda}$ to the workers in each iteration.

When executors solve the sub-problems, we use lp_solve [20] since it is open-source and thread-safe. Since Spark runs multiple threads in one executor process in parallel, thread safety is a required property for the ILP problem solver.

3.3. Middleware Architecture

Figure 3 illustrates the architecture of the proposed middleware framework. We describe how the middleware works, step by step, as follows:

- **Step 1**: The *Controller* periodically pulls (*e.g.*, every 5 minutes) flight status information in the queue such as airplane positions and flights' departure and arrivals.
- **Step 2**: The Controller creates an ILP problem instance from the obtained flight status information and then pushes it to the *VM Scheduler* with requested processing latency (*e.g.*, 4 minutes).
- **Step 3**: The VM Scheduler uses a time series prediction model and a resource prediction model to estimate the required number of VMs to finish the optimization within the requested processing latency.
- **Step 4**: The VM Scheduler allocates or deallocates VMs accordingly by calling cloud APIs such as the ones provided by Amazon EC2 [21].
- **Step 5**: The Controller requests the *Application Launcher* to run the ILP application.

Even though flight status information flows into the middleware continuously, the middleware processes the information collected within a sliding time window. We can see this as a *discretized stream* processing model just as used in Spark streaming [19].

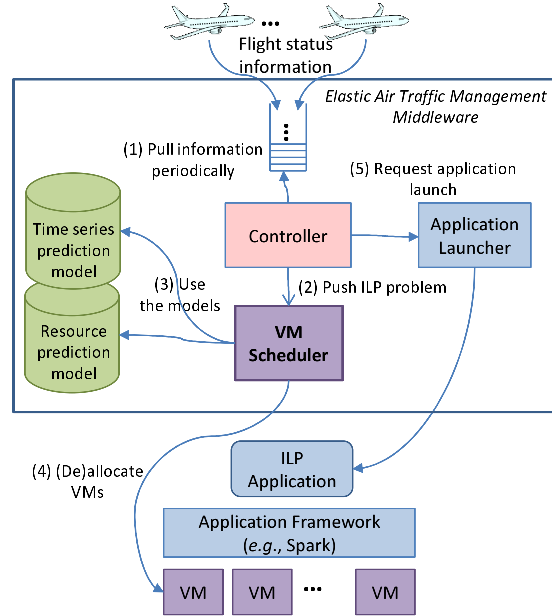


Figure 3. Architecture of the elastic air traffic management middleware framework.

4. Virtual Machine Scheduling

In Section 4.1, we first confirm how the ILP optimization application described in Section 3.2 performs on actual VMs through a preliminary experiment. Next, in Section 4.2, we describe a resource prediction model created using linear regression. Finally, in Section 4.3, based on observations from the preliminary experiment and the resource prediction model, we present two VM scheduling algorithms: $baseScheduler$ and $specScheduler$.

4.1. Performance Characterization of ILP Optimization

To understand how the Spark application works, we conducted a preliminary experiment with the following settings:

- Number of links per route: generated from a Gaussian distribution (average = 19, variance = 9).
- Number of sectors: 1024 made of a 32 by 32 grid just as shown in Figure 2.
- Convergence criterion: the value of master dual objective in Equation (7) does not improve more than 1% for 1000 iterations (*i.e.*, $\delta = 1, I = 1000$ in Algorithm 1).
- Spark setting: 1 executor per core.

We tested 50 application runs with randomly selected VM instances and number of routes from the following options:

- VM instances for Spark worker nodes: {c4.large, c4.xlarge, c4.2xlarge} instance types available from Amazon EC2 (see Table 1). Up to five instances can be created for each instance type.

- Number of routes: $\{128, 256, 512, 1024\}$

TABLE 1. AMAZON EC2 VM INSTANCE TYPES USED IN EXPERIMENTS (INFORMATION AS OF NOVEMBER 2015).

Name	vCPU cores	Cost [USD/hr]	Instance limits
c4.large	2	0.11	5
c4.xlarge	4	0.22	5
c4.2xlarge	8	0.441	5

Figure 4 presents the relationship between total number of cores used by the VMs and the application execution time. First, we observe that the performance variance is relatively small (at most 7%) regardless of VM configurations as long as we use the same number of cores for the same number of routes. This is due to the fact that we assign one thread per core, and therefore, we end up using the same number of threads even for different VM configurations as long as they have the same number of cores. Second, as we can clearly see from the graph, the application execution time does not improve significantly from around 18 to 20 cores for all numbers of routes. This behavior is consistent with a performance analysis of a K-means Spark application reported in [22], in which the performance converges at around 15 threads. They concluded that multi-threaded computation overhead (*i.e.*, work time inflation [23]) and load imbalance cause the scalability bottleneck. Since our application and K-means have a similar synchronization pattern (*i.e.*, both are iterative and synchronize all workers between every iteration), this analysis applies to our case as well. These

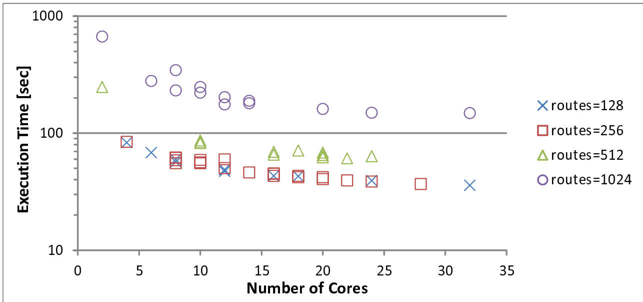


Figure 4. Characteristics of the ILP problem execution time.

observations lead to the following decisions for the resource allocation method design:

- The unit of resource (de)allocation is the number of cores. Since the performance and cost per core is equal among {c4.large, c4.xlarge, c4.2xlarge}, we do not distinguish one VM instance type from another.
- We set the upper limit for the number of cores that we allocate to match the application's inherent scalability limitations.

4.2. Resource Prediction Model

The VM Scheduler introduced in Section 3.3 needs a model to determine how many resources it should allocate to achieve the target processing latency. We use linear regression with a non-linear transform to model the relationship between the two input parameters: processing latency l and number of routes r, and the output: number of cores c. We sampled 50 application runs using the same experimental settings as the preliminary experiment in Section 4.1, but this time with uniformly random numbers of routes between 100 and 1200. Subsequently, we tested five non-linear transforms to determine which one best fits the sampled training data as shown in Table 2. As the transformation becomes more complex, correlation between the model and the training data improves. Based on this result, we have identified $\Phi_{\text{-}2:2}$ to be the most correlated with the data. We thus use this model in our algorithms. The number of cores can be obtained as follows:

$$f(l, r) = \mathbf{w}^\intercal \cdot \Phi_{\text{-}2:2}(l, r), \tag{10}$$

where $\mathbf{w} = [w_1, w_2, ..., w_{11}]$ is a weight vector acquired from linear regression of the number of cores given the latency and the number of routes.

TABLE 2. NON-LINEAR TRANSFORMS USED FOR LINEAR REGRESSION.

Name	Transformation vector	Correlation
$\Phi_{\text{-}1}$	$[1/r, 1/l, 1]^\intercal$	0.615
Φ_2	$[1, r, l, r^2, rl, l^2]^\intercal$	0.765
$\Phi_{\text{-}2}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1]^\intercal$	0.870
$\Phi_{\text{-}2:1}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1, r, l]^\intercal$	0.873
$\Phi_{\text{-}2:2}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1, r, l, r^2, rl, l^2]^\intercal$	0.890

4.3. Elastic Scheduling Algorithms

The VM Scheduler periodically calls one of the scheduling algorithms to keep the processing latency consistent. We describe two scheduling algorithms for the VM Scheduler that use the model presented in Section 4.2. Key notations used in the algorithms are summarized in Table 3.

TABLE 3. KEY NOTATIONS USED IN SCHEDULING ALGORITHMS.

Name	Description
l_{req}	Requested processing latency.
r_t	Number of routes to process at time t.
t_{up}	VM startup time.
$f(l, r)$	Resource prediction model that predicts the number of cores to satisfy latency l when processing r routes.
V_{act}	Set of VMs that are actively used by the application.
V_{idle}	Set of VMs to be removed at the end of next billing cycle.
V_{spec}	Set of speculative VMs to be allocated.
V	Set of currently allocated VMs. $V = V_{\text{act}} \cup V_{\text{idle}}$.
$t_{\text{bc}}(v)$	Next billing cycle time of a VM $v \in V$.
$c(v)$	Number of cores of a VM $v \in V$.

4.3.1. Baseline Scheduling. The baseline scheduling algorithm ($baseScheduler$) is shown in Algorithm 2. This algorithm responds to increasing computational demand by creating new VMs, while at the same time, tries to take

advantage of existing VMs even when they are not needed to achieve required processing latency.

First, we compare the available number of cores c_{all} with required cores c_{req} estimated from the resource prediction model f (Line 1-3). If there are enough cores, we sort V in descending order of next billing cycles (the VM with the latest billing cycle comes first) (Line 5). Then, we let *selectVMs* select at least c_{req} worth of VMs from the sorted V and put them to V_{act} and the rest of VMs to V_{idle} (Line 6 and 7). Further, if there still remains VMs in V_{idle} such that their billing cycles come after the time that we expect the application to finish, we utilize those VMs V_{extra} too (Line 9-11). At this point, VMs in V_{idle} are expected to end their billing cycles before the application finishes, and therefore they will be deallocated.

If there are not enough cores to satisfy the requested latency, we have to allocate new VMs to satisfy the requested latency. Since newly allocated VMs take t_{up} time before they become fully operational, we can only start running the application after t_{up} time has passed. Therefore, we set a tighter deadline $l_{\mathrm{req}} - t_{\mathrm{up}}$ and estimate the number of cores to allocate c_{alloc} again (Line 15). Then, we call the *allocVMs* sub-routine to allocate at least c_{alloc} worth of VMs and update V_{act} and V_{idle} accordingly.

Algorithm 2: *baseScheduler*
(Baseline VM scheduling algorithm)

```
   input  : l_req, r_t, t_up, V
   output: V_act, V_idle
1  c_req ← ⌈f(l_req, r_t)⌉;
2  c_all ← Σ_{v∈V} c(v);
3  if c_req ≤ c_all then
4      // There are enough cores
5      Sort v ∈ V in descending order of t_bc(v);
6      V_act ← selectVMs(c_req, V);
7      V_idle ← V − V_act;
8      V_extra = {v | v ∈ V_idle,  t + l_req < t_bc(v)};
9      if V_extra ≠ ∅ then
10         V_act ← V_act ∪ V_extra;
11         V_idle ← V_idle − V_extra;
12     end
13 else
14     // Not enough cores, allocate VMs
15     c_alloc ← ⌈f(l_req − t_up, r_t)⌉ − c_all;
16     V_act ← V ∪ allocVMs(c_alloc);
17     V_idle ← ∅;
18 end
19 return V_act, V_idle;
```

4.3.2. Speculative Scheduling. The speculative scheduling algorithm (*specScheduler*) is shown in Algorithm 3. This algorithm takes advantage of future computational demand prediction and tries to allocate VMs before they are needed. By doing so, we can avoid waiting a VM startup time before running the application. We predict the number of routes for time $t+1$ using a slope computed from r_t and r_{t-1} as follows.

$$r_{t+1} = \frac{r_t - r_{t-1}}{t-(t-1)} + r_t = 2r_t - r_{t-1}. \tag{11}$$

This prediction model is equivalent to an autoregressive model (*i.e.*, AR(2)) just as used in [24], [25].

The *specScheduler* first obtains a baseline configuration using the *baseScheduler* and computes all of available number of cores in c_{all} (Line 2-3). Next, *specScheduler* predicts the number of routes for the next step in $\hat{r}_{t+1}$ by using the prediction model of Equation (11) (Line 5). Using $\hat{r}_{t+1}$ and the resource prediction model f, we estimate a speculative required cores $\hat{c}_{\mathrm{req}}$. Finally, if we need more cores at next time step than what we currently have ($c_{\mathrm{all}} < \hat{c}_{\mathrm{req}}$), then we schedule to launch VMs that are worth $\hat{c}_{\mathrm{req}} - c_{\mathrm{all}}$ cores just before the next time step (Line 8 and 10).

Algorithm 3: *specScheduler*
(Speculative VM scheduling algorithm)

```
   input  : l_req, r_t, r_{t−1}, t_up, V
   output: V_act, V_idle, V_spec
1  // Obtain a baseline configuration first
2  (V_act, V_idle) ← baseScheduler(l_req, r_t, t_up, V);
3  c_all ← Σ_{v∈V} c(v);
4  // Speculative VM allocation
5  r̂_{t+1} ← predictNumRoutes(r_t, r_{t−1});
6  ĉ_req ← ⌈f(l_req, r̂_{t+1})⌉;
7  V_spec ← ∅;
8  if c_all < ĉ_req then
9      // Schedule to finish launching V_spec
          VMs before the next time step
10     V_spec ← scheduleAllocVMs(ĉ_req − c_all);
11 end
12 return V_act, V_idle, V_spec;
```

4.3.3. VM Allocation Policy. Given the number of cores, we allocate VMs from a limited pool of VMs when executing *allocVMs* (Line 16, Algorithm 2) and *scheduleAllocVMs* (Line 10, Algorithm 3).

When selecting VMs, we try to allocate a VM type with smaller number of cores. If a VM type reaches its instance creation limit, then we try to allocate VM types with bigger number of cores until at least the requested number of cores is allocated. In case of Amazon EC2, we try to allocate c4.large instances first, and then we try c4.xlarge followed by c4.2xlarge. The reason that we give priority to smaller instances is because they have finer core granularity. That is, we would have a higher chance of allocating exact number of cores so that we can avoid over provisioning of VMs.

5. Evaluation

We first introduce simulation based experimental settings in Section 5.1. Next, we evaluate the proposed algorithms' elastic behavior in Section 5.2. Then, we compare the proposed algorithms' performance with static VM scheduling and threshold-based auto scaling in Section 5.3 and 5.4 respectively.

5.1. Experimental Settings

The experiments are simulation-based. We develop a simulator that executes proposed VM scheduling algorithms. Using the generated schedules by the simulator, we manually allocate and deallocate VMs on Amazon EC2 cloud and run the Spark application to evaluate used VM hours, cost, and latency violations based on actual execution time. We use two 3-hour route datasets for testing: the first one is called *Nationwide* that we create from the 24-hour real nationwide flights shown in Figure 1, and the second one is called *Dallas* that we create based on a simulated flights over Dallas/Fort Worth area [26]. Both have almost the same peak number of routes, about 1200, but the patterns of fluctuation are different. While Nationwide has a smooth curve, Dallas has steep spikes, as shown in Figure 7.

We use the following test parameters for evaluation:

- Scheduling interval: 5 minutes (36 scheduling problems over 3 hours).
- Requested processing latency (l_{req}) : 4 minutes.
- VM startup time (t_{up}): 90 seconds.
- VM instances for Spark's worker nodes: {c4.large, c4.xlarge, c4.2xlarge} (see Table 1 for details). Up to five instances can be created for each instance type.
- Billing cycle: 1 hour (Amazon EC2's default).

5.2. Elastic Behavior Confirmation

5.2.1. Nationwide Dataset. Results for $baseScheduler$ and $specScheduler$ for the Nationwide dataset are shown in Figure 5(a)-(d). From Figure 5(a), we see that the baseline scheduler allocates VMs, initially for 8 cores at 1900 seconds and then for 12 cores at 3900 seconds. Looking at Figure 5(b), we notice that there are two "dips" in requested latency at the same time as the scheduler allocates new VMs. This lower latency corresponds to the value of $l_{\mathrm{req}} - t_{\mathrm{up}}$ (= 150 seconds) at Line 15 of Algorithm 2. To account for the VM startup time, we intentionally set a tighter latency. Therefore, the scheduling algorithm has to allocate relatively large number of cores. Since these cores are more than enough to satisfy the regular required latency l_{req} (= 240 seconds), there are periods (2100 to 2700 seconds, 3900 seconds to the end) when execution time stays lower than required, that is, resources are over-provisioned during these periods. This is a limitation of the reactive approach. There are four latency violations occurring at 1800, 3300, 3600, and 3900 seconds respectively. At these times, there are exactly the same number of cores available as the resource prediction model estimated. This means that the prediction model underestimated the number of cores needed to satisfy the requested latency. The average prediction error of execution time for the four violations is 12%. This result suggests that the accuracy of resource prediction is limited and we may need to over-provision VMs intentionally. The total cost for the base scheduler is \$1.72 and latency violations are 4 out of 36.

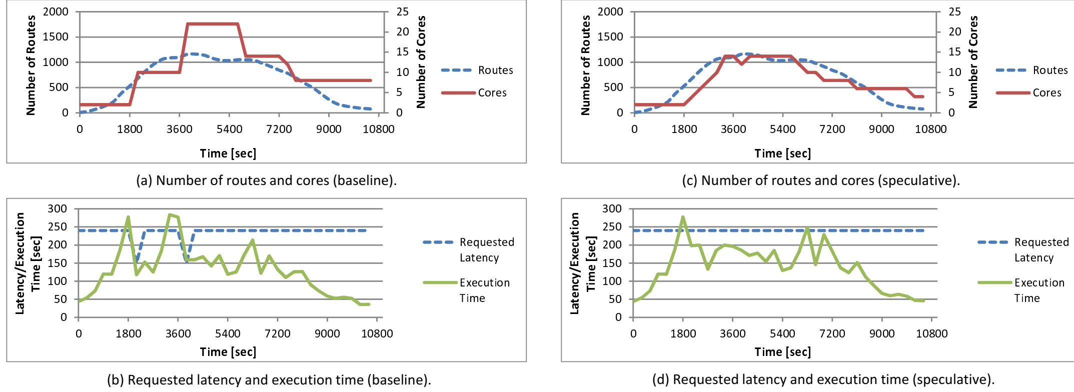
From Figure 5(c), we can visually confirm that the speculative scheduler gradually allocates smaller numbers of cores, unlike the baseline scheduler which abruptly allocates larger numbers of cores. This is a direct effect of the speculative VM allocation. In fact, all the allocated VMs are launched by $scheduleAllocVMs$ at Line 10 in Algorithm 3. Since there are already enough cores by the time $baseScheduler$ at Line 2 tries to schedule, it does not need to create any new VMs. As a result, there are no latency drops in Figure 5(d). The total cost for the speculative scheduler is \$1.01 and latency violations are 2 out 36. The cost is a 41% improvement compared to the baseline scheduler.

Figure 6 shows a VM allocation sequence from the speculative scheduler created from the Nationwide dataset. We can see that five c4.large instances (ID = 0 to 4) are allocated by 2910 seconds, and then a c4.xlarge instance (ID = 5) is allocated at 3210 seconds. Interestingly, at 9900 seconds, the scheduler chooses to keep the c4.xlarge instance instead of the c4.large (ID = 4) instance even though the c4.large can also satisfy the requested processing latency. This is because the c4.xlarge will have the billing cycle later than the c4.large does; however, in a truly continuous optimization scenario, it may be less critical because wasted VM hours will be negligible compared with the application execution time.

5.2.2. Dallas Dataset. Results for $baseScheduler$ and $specScheduler$ for the Dallas dataset are shown in Figure 7(a)-(d). From Figure 7(a), we can confirm that the baseline scheduler allocates VMs for 6 cores at 1200 seconds and then for 12 cores at 7800 seconds. In Figure 7(c), the speculative scheduler follows changes of the routes smoothly for the first stage of the sequence; however, at 7500 seconds, it fails to predict the number of routes correctly and ends up allocating less VMs than actually needed at 7800 seconds. It allocates two more VMs at 7800 seconds and that is the reason why we see a requested latency drop at 7500 seconds of Figure 7(d). The current slope-based time series predictor cannot keep up with sudden route changes. Apart from the time series prediction failure of the speculative scheduler, both schedulers are able to adapt to the spike and allocate/deallocate VMs successfully. For the base scheduler, the total cost is \$0.83 and latency violations are 2 out of 36. For the speculative scheduler, they are \$1.38 and 1 out 36 respectively.

5.3. Comparison with Static Scheduling

Elastic scheduling can adapt to unforeseen fluctuating demand whereas static scheduling cannot. We compare static scheduling against our proposed elastic algorithms to confirm the effectiveness of our approach's adaptivity. Experimental settings are the same as Section 5.2, and we use both Nationwide and Dallas datasets. For the static scheduling, we test VM configurations with cores = $\{2, 4, 8, 10, 12, 14, 16\}$ for Nationwide and $\{2, 6, 8, 10, 12\}$ for Dallas. Comparison of VM hours, cost, and the percentage of latency violations are shown in Tables 4 and 5, respectively, for the Nationwide and Dallas datasets. Since static schedules do not waste any VM hours at all (*i.e.*, they



(a) Number of routes and cores (baseline).

(c) Number of routes and cores (speculative).

(b) Requested latency and execution time (baseline).

(d) Requested latency and execution time (speculative).

Figure 5. Experimental results for the Nationwide dataset.

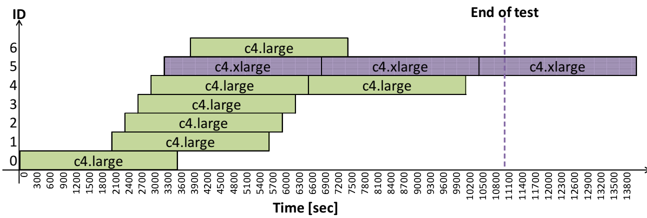


Figure 6. VM allocation sequence for the speculative scheduling algorithm created from the Nationwide dataset.

do not visit a situation as in Figure 6), we compute the cost for our elastic schedulers in proportion to the execution time for fairness. In the tables, VM hours means the net number of cores used over 3 hours of the experiments. The percentage of latency violations is computed out of 36 scheduling problems over 3 hours.

For the Nationwide dataset, the speculative scheduler successfully improves over the baseline scheduler in terms of latency violations by 50% with 41% less VM hours and cost. The performance of the static schedule with 12 cores is comparable to the speculative scheduler (0% vs. 5.56% violations). When comparing the two, the speculative scheduler achieves a similar performance with 49% less VM hours and cost. For the Dallas dataset, the base scheduler improves latency violations over the closest static allocation approach (6 cores) by 66% despite it uses 17% less VM hours. When comparing to the static schedule with 8 cores, the speculative scheduler slightly over-provisions due to inaccuracy of both time series and resource predictions. That is, it spends 5% more VM hours and cost, but equally performs as the static scheduler with 8 cores in terms latency violations.

While our elastic scheduling policy exhibits a small percentage of latency violations, we note that any static scheduler, other than a very highly provisioned one, will not be able to guarantee zero latency violations. For any static VM allocation, there is a possibility that it will not be sufficient for some level of demand. Our elastic schedulers,

TABLE 4. VM HOURS, COST, AND LATENCY VIOLATIONS FOR ELASTIC AND STATIC SCHEDULING ALGORITHMS (NATIONWIDE DATASET).

Policy	Cores	VM hours [core·hour]	Cost [USD]	Violations [%]
Static	2	6	0.33	63.89
	4	12	0.66	44.44
	8	24	1.32	19.44
	10	30	1.65	13.89
	12	36	1.98	0
	14	42	2.31	0
	16	48	2.64	0
Auto Scaling	2 to 8	15.96	0.88	25
Elastic (base.)	2 to 22	31.33	1.72	11.11
Elastic (spec.)	2 to 14	18.33	1.01	5.56

TABLE 5. VM HOURS, COST, AND LATENCY VIOLATIONS FOR ELASTIC AND STATIC SCHEDULING ALGORITHMS (DALLAS DATASET).

Policy	Cores	VM hours [core·hour]	Cost [USD]	Violations [%]
Static	2	6	0.33	61.11
	6	18	0.99	16.67
	8	24	1.32	2.78
	10	30	1.65	2.78
	12	36	1.98	0
Elastic (base.)	2 to 14	15	0.83	5.56
Elastic (spec.)	2 to 18	25.15	1.38	2.78

on the other hand, successfully adapt to unforeseen computational demand changes and scale VMs accordingly with reasonably low cost.

5.4. Comparison with Auto Scaling

Since threshold-based auto scaling is commonly used as an application-agnostic scaling technique, we test it against our application aware approach. We use the same experimental settings as Section 5.2. We implement the following rules that are compatible to Amazon Auto Scaling [10]:

- VM instance type: c4.large.
- VM allocation: minimum 1, maximum 5 instances.

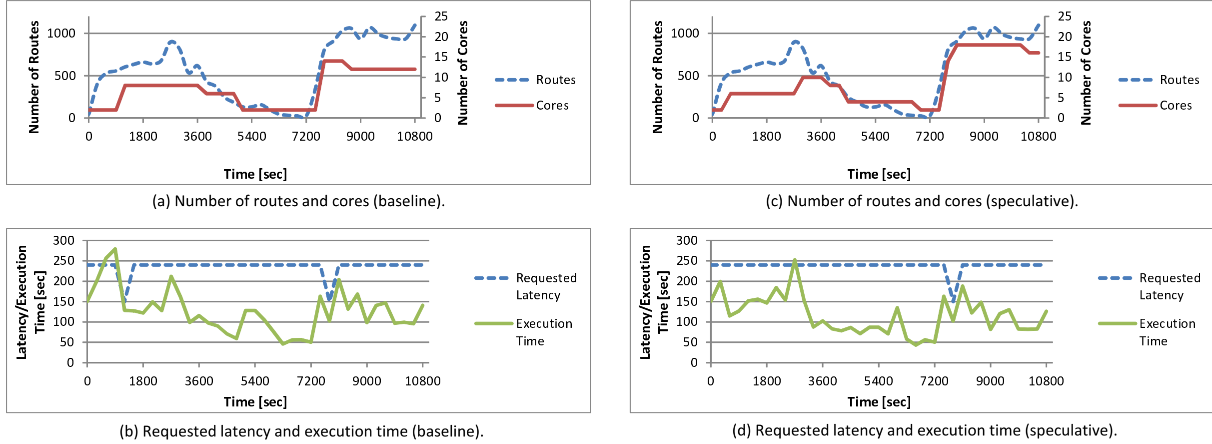


Figure 7. Experimental results for the Dallas dataset.

- Rule to scale up: if the average CPU utilization of allocated VMs is consistently above 70% for 2 minutes, add one VM.
- Rule to scale down: if the average CPU utilization of allocated VMs is consistently below 30% for 2 minutes, reduce one VM.
- Cooldown period: once scaling decision is made, no new scaling activity is performed for 300 seconds.
- VM termination: the instance that is closest to the next billing cycle is chosen to terminate.

Figure 8 shows average CPU utilization and the number of VMs over the 3 hour experiment period. The auto scaler successfully increases VMs up to 4, and then decreases them to 1. The results are summarized in Table 4. Since the threshold-based auto scaler is not aware of the application performance requirement (*i.e.*, 240 seconds latency) at all, it under-provisions the VMs and ends up producing relatively many latency violations compared to our elastic schedulers.

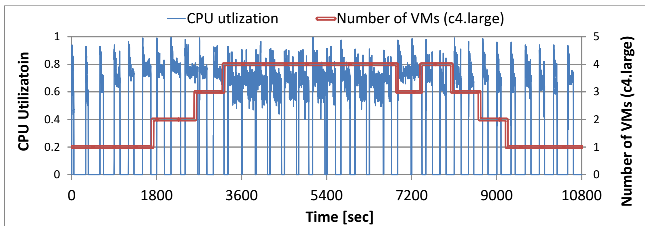


Figure 8. CPU utilization and VM allocation by a threshold-based auto scaling.

6. Related Work

First and foremost, related work is an air traffic flow management model optimizing nationwide air traffic [16]. Cao and Sun formulate the air traffic management problem as an ILP problem and decompose it into multiple subproblems using Lagrangean decomposition. Further, they use MapReduce [15] to solve the Lagrangean relaxation of the large LTM problem in parallel [16]. They only consider fix-sized clusters as a running environment; however, we apply it to an IaaS cloud and support auto scaling of VMs. Also, our approach is aware of billing cycles and tries not to renew VM leases unless necessary.

Many techniques have been developed to automatically scale VM allocations [27]. Autoregression and ARMA are widely used to predict time series and have been successfully applied to auto-scaling cloud systems [24], [25]. If the system foresees a demand spike in advance, it can allocate new VMs and make them ready before the spike actually arrives. As long as prediction accuracy is reasonably high, time series prediction approaches are effective. Another common approach is threshold-based such as offered in Amazon Auto Scaling [10]. For example, users can configure a policy that creates a new VM if the CPU utilization consistently goes above 70%. This approach is simple and easy to use; however, since it only reacts to simple metrics, from our experiment, it cannot optimally allocate VMs to meet the application specific performance target (*e.g.*, processing latency).

Other auto scaling techniques include reinforcement learning and control theory based approaches. Reinforcement learning (RL) techniques try to scale VMs without any prior knowledge about the application [11], [12]. RL agents learn appropriate actions by interacting with environments, but it requires a long time to find an optimal solution. In control theory based approaches, a controller tries to control the system to follow some desired value by using feedback from the system. For example, the controller controls the number of VMs to keep the system's throughput consistent [28], [29]. Unlike the control theory based approaches, our model based approach does not require online model updates. Therefore, it is expected to work as soon as the system starts running. As a trade-off, the models in our approach cannot adapt at run time.

7. Conclusion and Future Work

In this paper, we presented an elastic middleware framework that is specifically designed to solve ILP problems generated from continuous air traffic streams over an IaaS cloud. We proposed a speculative VM scheduling algorithm with time series and resource prediction models. Experiments show that our speculative VM scheduling algorithm can achieve a similar performance to a static schedule while using 49% less VM hours for a smoothly changing air traffic. However, for a sharply changing air traffic, our speculative VM scheduling algorithm costs slightly more VM hours to achieve the same performance. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy. We plan to improve model prediction accuracy especially for time series using a more complex model (*e.g.*, ARMA).

We have several potential directions for future work. First, we would like to apply our middleware to other application areas since the concept of solving large scale ILP problems created from continuous data stream is widely applicable. Candidate application areas include: public transportation routing [4], investment portfolio optimization [5], and marketing budget optimization [6], [7]. Second, we plan to explore other modeling techniques for predicting resource allocation. Finally, we plan to extend our framework to support other optimization policies such as budget constrained and deadline constrained policies.

Acknowledgments

This research is partially supported by the DDDAS program of the Air Force Office of Scientific Research, Grant No. FA9550-15-1-0214 and NSF Awards, Grant No. 1462342, 1553340, and 1527287. The authors would like to thank an Amazon Web Services educational research grant and a Google Cloud Credits Award.

References

[1] International Air Transport Association (IATA), "New IATA Passenger Forecast Reveals Fast-Growing Markets of the Future," http://www.iata.org/pressroom/pr/pages/2014-10-16-01.aspx, October 2014.

[2] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[3] GE, "GE Flight Quest Challenge 2," http://www.gequest.com/c/flight2-final/data.

[4] J.-F. Cordeau, P. Toth, and D. Vigo, "A Survey of Optimization Models for Train Routing and Scheduling," *Transportation Science*, vol. 32, no. 4, pp. 380–404, 1998.

[5] C. Papahristodoulou and E. Dotzauer, "Optimal portfolios using linear programming models," *Journal of the Operational research Society*, vol. 55, no. 11, pp. 1169–1177, 2004.

[6] T. Lu and C. Boutilier, "Dynamic segmentation for large-scale marketing optimization," in *International Conference on Machine Learning 2014 Workshop on Customer Life-Time Value Optimization in Digital Marketing*, June 2014.

[7] Z. Abrams, O. Mendelevitch, and J. Tomlin, "Optimal delivery of sponsored search advertisements subject to budget constraints," *Proceedings of the 8th ACM Conference on Electronic Commerce - EC '07*, p. 272, 2007.

[8] Google, "Google AdWords," https://www.google.com/AdWords/.

[9] E. Even-dar, Y. Mansour, V. Mirrokni, S. Muthukrishnan, and U. Nadav, "Bid Optimization in Broad-Match Ad Auctions," *Proceedings of the 18th International Conference on World Wide Web*, p. 10, 2009. [Online]. Available: http://arxiv.org/abs/0901.3754

[10] Amazon Web Services, "Auto Scaling," https://aws.amazon.com/autoscaling/.

[11] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[12] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[13] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1439–1451, 2006.

[14] Y. Cao and D. Sun, "A Link Transmission Model for Air Traffic Flow Management," *Journal of Guidance, Control, and Dynamics*, vol. 34, no. 5, pp. 1342–1351, 2011.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004.

[16] Y. Cao and D. Sun, "Migrating Large-Scale Air Traffic Modeling to the Cloud," *Journal of Aerospace Information Systems*, vol. 12, no. 2, pp. 257–266, 2015. [Online]. Available: http://arc.aiaa.org/doi/10.2514/1.I010150

[17] FlightAware, "FlightAware," http://flightaware.com/.

[18] Amazon Web Services, "Amazon Elastic Compute Cloud (Amazon EC2)," https://aws.amazon.com/ec2/.

[19] The Apache Software Foundation, "Apache Spark," http://spark.apache.org/.

[20] LGPL open source project, "lp_solve: linear integer programming solver," http://lpsolve.sourceforge.net/.

[21] Amazon Web Services, "Amazon Elastic Compute Cloud (Amazon EC2) API Reference," http://docs.aws.amazon.com/AWSEC2/latest/APIReference/.

[22] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server," *arXiv:1506.07742*, 2015.

[23] S. L. Olivier, B. R. De Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," *Scientific Programming*, vol. 21, pp. 123–136, 2013.

[24] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services." in *NSDI*, vol. 8, 2008, pp. 337–350.

[25] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 500–507.

[26] Yiyuan Zhao and Joachim K. Hochwarth and Adrianne A. Hersrud, "Comprehensive Dynamic Air Traffic System Simulation (ComDATSS," https://www.aem.umn.edu/research/atc/projects/ComDATSS/.

[27] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014. [Online]. Available: http://link.springer.com/10.1007/s10723-014-9314-7

[28] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 410–417.

[29] X. Bu, J. Rao, and C.-Z. Xu, "Coordinated self-configuration of virtual machines and appliances using a model-free learning approach," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 4, pp. 681–690, 2013.