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. Experiments show that our VM scheduling algorithm with time-series prediction can achieve similar performance to a static schedule while using 49% fewer VM hours for a realistic air traffic pattern.**

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 demand for air traffic optimization also changes dynamically as shown in Figure 1. To keep up with the fluctuating computing demand in a cost-efficient way, we can dynamically allocate and deallocate virtual machines (VMs) from Infrastructure-as-a-Service (IaaS) cloud computing providers.

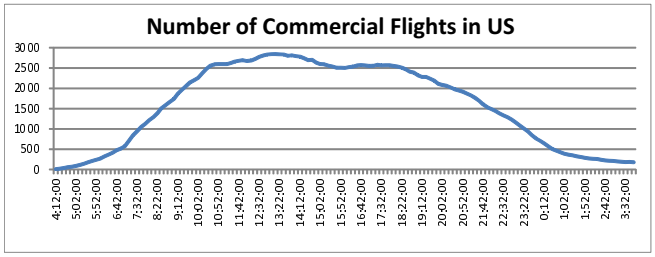


Figure 1. U.S. flights on January 18th, 2014 (created from [3]).

The Link Transmission Model (LTM) is an air traffic flow management model that optimizes nationwide air traffic and is formulated as an ILP problem [4]. Since ILP is NP-hard, it is common to use relaxation methods such as Lagrangean relaxation to find an approximate solution in a reasonable amount of time. Using Lagrangean decomposition, Cao and Sun decomposed LTM into multiple sub-problems, where each sub-problem corresponds to the optimization of one air traffic route [5]. We use a simplified version of the original LTM that is also decomposable to route-wise sub-problems using Lagrangean decomposition [6]. For each route i, a decomposed sub-problem of the simplified LTM problem is given as follows:

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

where $\boldsymbol{\lambda}$ is a vector of Lagrangean multipliers, $A_i, \mathbf{c_i}, d_i$ are given parameters associated with route i, and $\mathbf{x_i}$ is a vector of variables representing the number of traffic on each *link* of route i. For air traffic consists of K routes, we iteratively solve the K sub ILP problems shown in (1) to find optimal $\mathbf{x}_i (i = 1, ..., K)$ for a specific $\boldsymbol{\lambda}$, and then update $\boldsymbol{\lambda}$ for the master dual problem (see [6] for details). Since the number of sub ILP problems is determined by the number of routes, fluctuating number of flights will impact the computational demand.

ILP has a lot of practical applications, and some of them have a strong time-dependency. Examples of such applications include: public transportation routing [7], investment portfolio optimization [8], and marketing budget optimization [9]. Just as air traffic management, public transportation routing has similar characteristics: *e.g.*, the number of buses changes depending on time of the day. 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 passively adjust the number of VMs at run time or proactively predict the number of VMs using a resource prediction model. The former approach includes a threshold-based scaling, as used in Amazon's Auto Scaling [10], reinforcement learning [11], and control theory [12]. We

take the latter approach with the full application domain knowledge of air traffic management. Moreover, we use an autoregressive time series prediction model to decide when to allocate/deallocate VMs in a speculative manner.

In this paper, to provide cost-efficient scalable resource management infrastructure for ILP problems with fluctuating computational demand, we propose an elastic middleware framework specifically designed to solve ILP problems. In particular, we evaluate the simplified LTM problem that we have shown in (1) as the target ILP problem. This paper is a summary of a technical report [6].

2. Elastic Air Traffic Management Middleware

2.1. 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 or airplanes directly send the latest flight status information to the middleware (see Figure 2). 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 3.2 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 [10]). The scheduler only terminates VMs just before their billing cycle so that the application can use the VMs' computing power until right before the next billing cycle.

2.2. Application Implementation

We use Spark 1.5.1 [13], a general cluster computing engine, to implement an iterative ILP solver of the simplified LTM problem presented in Section 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. Sub-problems defined in (1) are executed in parallel by *executors* running on the worker nodes while the rest of the solver application runs 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 [14] 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 our ILP problem solver.

2.3. Middleware Architecture

Figure 2 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 [10].
- **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 [13].

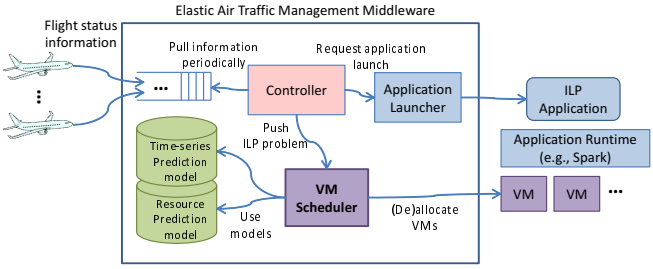


Figure 2. Architecture of the elastic air traffic management middleware framework.

3. Virtual Machine Scheduling

The VM Scheduler introduced in Section 2.3 needs a VM scheduling algorithm to determine how many VMs it should allocate to achieve the target processing latency. We show our resource estimation approach in Section 3.1. Then, in Section 3.2, we present our elastic VM scheduling algorithm with a time-series prediction capability.

3.1. Resource Prediction Model

From a preliminary experiment, we observed non-linear relationship between the number of cores and execution time for different numbers of routes as shown in Figure 3. To determine how many resources the VM scheduler should allocate to achieve the target processing latency, we use linear regression with a polynomial 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 to create a training data set and evaluated five polynomial transforms. We chose the most complex transform among them that showed the best correlation of 0.890 to the training data set as follows:

$$\begin{aligned} f(l,r) &= \mathbf{w}^\intercal \cdot \Phi(l,r) \qquad (2) \\ &= [w_1, w_2, ..., w_{11}] \cdot \\ &\quad [1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1, r, l, r^2, rl, l^2]^\intercal, \end{aligned}$$

where $\mathbf{w}$ is a weight vector acquired from linear regression using the polynomial transform $\Phi(l,r)$.

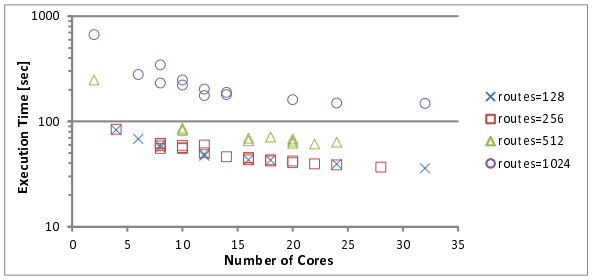


Figure 3. Characteristics of the ILP problem execution time.

3.2. Elastic Scheduling Algorithm

We describe a speculative VM scheduling algorithm, *specScheduler* shown in Algorithm 1, that uses the resource prediction model presented in Section 3.1. Key notations used in the algorithms are summarized in Table 1.

TABLE 1. 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. $V = V_{\text{act}} \cup V_{\text{idle}}$.
$c(v)$	Number of cores of a VM $v \in V$.

The speculative VM scheduling algorithm takes advantage of future computational demand prediction. VM scheduling and time series prediction are performed every Δt time. We predict the number of routes for time step $t+1$ using a slope computed from r_t and r_{t-1}: $\hat{r}_{t+1} = 2r_t - r_{t-1}$. This time-series prediction model is equivalent to an autoregressive model (*i.e.*, AR(2)) just as used in [15].

In Algorithm 1, the *specScheduler* first obtains a baseline configuration using the *baseScheduler* (Line 1). The *baseScheduler* only considers the current number of routes r_t, but is aware of billing cycle and tries to take advantage of existing VMs even when they are not needed to achieve required processing latency [1]. Then, all of available number of cores is computed in c_{all} (Line 2). Next, the *specScheduler* predicts the number of routes for the next step in $\hat{r}_{t+1}$ by using the time series prediction model (Line 3). Using $\hat{r}_{t+1}$ and the resource prediction model f, we estimate a speculative required cores $\hat{c}_{\text{req}}$. Finally, if we need more cores at next time step than what we currently have ($c_{\text{all}} < \hat{c}_{\text{req}}$), then we schedule to launch VMs that are worth $\hat{c}_{\text{req}} - c_{\text{all}}$ cores just before the next time step (Line 8). Since the speculative VM scheduler takes the VM startup time t_{up} into account, by the time the middleware receives a next processing request, newly allocated VMs are expected to be already available.

Algorithm 1: *specScheduler*
(Speculative VM scheduling algorithm)

```
   input : l_req, r_t, r_{t-1}, t_up, V
   output: V_act, V_idle, V_spec
1  (V_act, V_idle) ← baseScheduler(l_req, r_t, t_up, V);
2  c_all ← Σ_{v∈V} c(v);
3  r̂_{t+1} ← predictTimeSeries(r_t, r_{t-1});
4  ĉ_req ← ⌈f(l_req, r̂_{t+1})⌉;
5  V_spec ← ∅;
6  if c_all < ĉ_req then
7  |   // Schedule to finish launching V_spec
   |   |   VMs before the next time step
8  |   V_spec ← scheduleAllocVMs(ĉ_req − c_all);
9  end
```

4. Evaluation

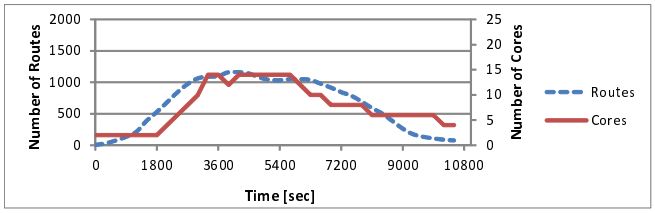
4.1. Elastic Behavior Confirmation

Experiments are simulation-based. We develop a simulator that executes the proposed VM scheduling algorithm. 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 create a test data set from the 24-hour real nationwide flights shown in Figure 1, which has the peak number of routes about 1200. We use the following test parameters for evaluation:

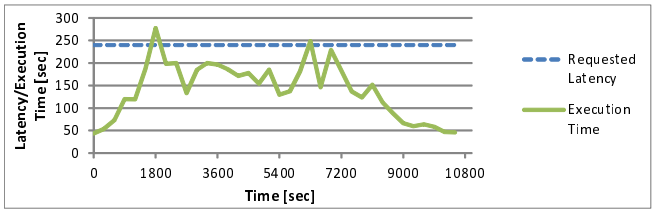
- Requested processing latency (l_{req}) : 4 minutes.
- Scheduling interval (Δt): 5 minutes (36 scheduling problems over 3 hours).
- VM startup time (t_{up}): 90 seconds.
- VM instances for Spark's worker nodes: {c4.large, c4.xlarge, c4.2xlarge}. Up to five instances can be created for each instance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

The result for *specScheduler* is shown in Figure 4(a)-(b). From Figure 4(a), we can visually confirm that the speculative scheduler gradually allocates smaller numbers of cores. Since the *scheduleAllocVMs* at Line 8 in Algorithm 1 creates VMs ahead of time, the *baseScheduler* does not create any new VMs in this experiment. From Figure 4(b), there are two latency violations which occurred at around 1800 and 6300 seconds respectively due to inaccurate time-series predictions. The total cost is $1.01 every 3 hours.

1. We cannot present the algorithm of *baseScheduler* here due to the space limitation. For the complete algorithm, see [6].



(a) Number of routes and cores.



(b) Requested latency and execution time.

Figure 4. Experimental results for the Nationwide dataset.

4.2. Comparison with Other Schedulers

We compare our proposed elastic algorithms to static scheduling and Amazon EC2's threshold-based *Auto Scaling* to confirm the effectiveness of our approach. Experimental settings are the same as Section 4.1. For the static scheduling, we test VM configurations with cores = $\{2, 4, 8, 10, 12, 14, 16\}$. For Auto Scaling, we implemented the compatible scaling rules used in [10]. Comparison of VM hours, cost, and the percentage of latency violations are shown in Tables 2.

TABLE 2. VM HOURS, COST, AND LATENCY VIOLATIONS FOR ELASTIC AND STATIC SCHEDULING ALGORITHMS PER 3 HOURS.

Policy	Cores	VM hours [core·hour]	Cost/3hrs [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
Auto Scaling	2 to 8	15.96	0.88	25
Elastic (spec.)	2 to 14	18.33	1.01	5.56

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. 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, on the other hand, successfully adapt to unforeseen computational demand changes and scale VMs accordingly with reasonably low cost. Since the threshold-based Auto Scaling 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.

5. Conclusion

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 predictions. 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 realistic air traffic pattern.

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

[4] 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.

[5] ——, "Migrating Large-Scale Air Traffic Modeling to the Cloud," *Journal of Aerospace Information Systems*, vol. 12, no. 2, pp. 257–266, 2015.

[6] S. Imai, S. Patterson, and C. A. Varela, "Elastic Virtual Machine Scheduling for Continuous Air Traffic Optimization," Dept. of Computer Science, Rensselaer Polytechnic Institute, Tech. Rep. 16-01, Feb. 2016.

[7] 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.

[8] 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.

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

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

[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*, pp. 67–74.

[12] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *IEEE CLOUD 2010*, pp. 410–417.

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

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

[15] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *IEEE CLOUD 2011*, pp. 500–507.