Predictive Switch-Controller Association and Control Devolution for SDN Systems

Xi Huang[1], Simeng Bian[1], Ziyu Shao[1], Hong Xu[2]
[1]School of Information Science and Technology, ShanghaiTech University
[2] NetX Lab, Department of Computer Science, City University of Hong Kong
Email: {huangxi,biansm,shaozy}@shanghaitech.edu.cn, henry.xu@cityu.edu.hk

Abstract—For software-defined networking (SDN) systems, to enhance the scalability and reliability of control plane, existing solutions mainly adopt either multi-controller architecture with static switch-controller association, or static control devolution by delegating certain processing of some requests back to switches. Such solutions can fall short due to uncertainties in temporal variations of request traffic, leading to excessive communication costs between switches and controllers, and local computation costs on switches. So far, it still remains an open problem to develop a joint online control scheme that conducts dynamic switch-controller association and dynamic control devolution. In addition, the fundamental benefits of predictive scheduling to SDN systems still remains unexplored. In this paper, we address the above challenges. First, we identify the trade-off between the long-term queueing stability and the reduction in total costs of communication and computation in SDN systems. Then we formulate the joint control problem as a stochastic network optimization problem which aims to minimize the time-average total costs of communication and computation with queueing stability guarantee in the long run. By exploiting unique problem structure and taking non-trivial transformations, we propose *GRAND*, a distributed, effective, and online joint control scheme. *GRAND* solves the long-term optimization problem asymptotically optimally through a series of online decision making over time slots, with no priori knowledge of request traffic statistics but only instant and limited information about system dynamics. Furthermore, by leveraging sliding window techniques and integrating predictive scheduling with *GRAND*, we propose *PRAND*, a joint control scheme that proactively exploits the value of predicted future information and induces even better system performance than *GRAND*. Results from our theoretical analysis and extensive simulations show that both *GRAND* and *PRAND* effectively achieves near-optimal total system costs with a tunable trade-off for long-term queueing stability. What's more, with mild-value of future information, *PRAND* outperforms *GRAND* with a significant reduction in request response time, even in the presence of mis-prediction.

Index Terms—SDN, switch-controller association, control devolution, predictive scheduling.

1 Introduction

In the past decade, software-defined networking (SDN) has initiated a profound revolution in networking system design towards more efficient network management and more flexible network programmability. The key idea of SDN is to separate control plane from data plane [18]. In this way, control plane maintains a centralized view over the whole network and processes requests that are constantly generated from SDN-enabled switches in the data plane; meanwhile, data plane only needs to carry out basic network functions such as monitoring and packet forwarding.

As data plane scale expands, control plane can become a potential bottleneck of SDN systems for scalability and reliability concerns. For example, the control plane, if implemented as a singleton controller, can be overloaded by ever-increasing request traffic, leading to excessively long processing latencies and belated response to network events. The singleton controller is a single point of failure which can result in the breakdown of the whole networking system.

To address such issues, existing solutions basically fall into two categories. One is to implement the control plane as a distributed system with multiple controllers [12] [23]; then each switch associates with more than one controller for fault-tolerance and load balancing [14] [7] [13] [25] [26] [9] [17]. The other is to devolve some request processing that does not require global information from controllers to switches, to reduce workloads on control plane [6] [10] [30].

When it comes to switch-controller association, the first category of solutions, the usual design choice is to make a static switch-controller association [12] [23]. However, solutions with static association are often inflexible when dealing with temporal variations of request traffic, thereby inducing workload imbalance across controllers and increased request processing latency. To mitigate such issues, Dixit *et al.* proposed an elastic distributed controller architecture with an efficient protocol design for switch migration among controllers [7]. However, the design for *switch-controller association* still remained unresolved. Later, Krishnamurthy *et al.* took a further step by formulating the controller association problem as an integer linear problem with prohibitively high computational complexity [13]. A local search algorithm was proposed to find the best possible association within a given time limit (*e.g.*, 30 seconds). Wang *et al.* modeled the controller as an *M/M/1* queueing system [25]; they formulated the association problem with a steady-state objective function as a many-to-one stable matching problem with transfers. Then they developed a novel two-phase algorithm that connects stable matching to utility-based game theoretic solutions, *i.e.*, coalition formation game with Nash stable solutions. Later, they extended the problem with an aim to minimize the long-term cost in SDN systems [26]. By decomposing it into a series of per-time-slot controller assignment sub-problems, Wang *et al.* applied receding horizon control techniques to solve the problem. In parallel, Filali *et al.* [9] formulated the problem as an one-to-many matching game, then developed another matching-based algorithm that achieves load-balancing by assigning minimum quota of workload to each controller. Lyu *et al.* [17] presented an adaptive decentralized approach for joint switch-controller association and controller activation with periodic on-off control to save operational costs.

For control devolution, the second category of solutions, most works have focused on static delegation of certain network functions to switches [6], [10], [30]. Some recent works [27] have even proposed more flexible schemes to perform delegation based on real-time distribution of network states or workloads on controllers.

Based on the above investigations, we identify several interesting but still unresolved questions regarding the control plane design, as follows.

- ◇ Instead of conducting deterministic switch-controller association with infrequent re-association [13] [25], can we directly perform dynamic association with respect to request traffic variation? What is the benefit of fine-grained control at the request level?
- ◇ How can dynamic devolution be conducted?
- ◇ Are there any trade-offs in the joint design of dynamic switch-controller association and dynamic control devolution? If so, how do we manage such trade-offs?
- ◇ Since the uncertainties in request traffic is one of the key factors that bring challenges to SDN system design, then if they can be learned, what are the fundamental benefits of such predicted information to SDN systems?

Notably, the last question is motivated by the recent growing interests in applying predictive scheduling [8] [4] based on various machine learning techniques to improve system performance. For instance, to promote quality-of-experience, Netflix predicts user behavior and preferences, then preloads videos onto user devices. Although such prediction-based approaches [28] [29] [5] [21] [19] are widely adopted, the fundamental benefits of predictive scheduling for SDN systems still remains unexplored.

In this paper, we consider a general SDN network with request traffic variations, incurring dynamic requests to handle network events. We assume that each request can be either processed at a switch (with computation costs) or be uploaded to certain controllers (with communication costs). [1] We aim to reduce the computational cost by control devolution at data plane, the communication cost by switch-user association between data plane and control plane, and the response times experienced by switches' requests, which is mainly caused by queueing delays on controllers. Regarding predictive scheduling, switches are assumed able to predict requests to arrive in limited time slots ahead through lightweight prediction modules with recent time-series forecasting techniques [3]. Further, we assume such future requests can be generated and pre-served before their arrival, and, if mis-predicted, they will incur extra system costs of communication and computation. [2] Under such settings, we open up a new perspective to answer those questions. The following summarize our contributions.

- ◇ **Modeling and Formulation:** We formulate the problem stated above as a stochastic network optimization problem. We make a careful choice of modeling by characterizing the system dynamics at the request level and conducting the scheduling on a per-time-slot basis, achieving a balance between model accuracy and the complexity in decision making. The optimization objectives include the minimization of the long-term time-average total costs of switch-to-controller communication and local computation on switches, as well as the long-term queueing stability guarantee that requests would receive timely processing instead of long queueing delay.
- ◇ **Algorithm Design:** By adopting existing techniques [20] and exploiting unique sub-problem structure, we propose *GRAND*, an efficient and greedy scheme that decides the association and devolution opportunistically over time slots, achieving the optimality asymptotically with limited information. *GRAND* is the first to take control decisions at the granularity of request level which, compared to the coarse-grained control schemes, induces more accurate decision making but with low overheads. We also discuss methods to deploy *GRAND*, and their pros and cons in practice.
- ◇ **Performance Analysis:** We conduct theoretical analysis and show that *GRAND* yields a tunable trade-off between $O(1/V)$ deviation from minimum long-term average sum of communication cost and computational cost and $O(V)$ bound for long-term average queue backlog size. We also illustrate the physical image and choice of parameters in the algorithm.
- ◇ **New Degree of Freedom in the Design Space of SDN Systems**: By adopting the idea of sliding lookahead window model [11], we further propose *PRAND*, a scheme that goes beyond *GRAND* by extending it with predictive scheduling. *PRAND* effectively reduces request response time by proactively leveraging surplus system resources to pre-serve requests. To the best of our knowledge, this is the first design that explores and exploits the benefits of predictive scheduling for SDN systems, opening up a new perspective in the design of SDN systems.
- ◇ **Experimental Evaluation and Verification:** We conduct extensive simulations to evaluate the performance of *GRAND* and *PRAND* against baseline schemes. Under various settings such as different request arrival processes and networking topologies [1] [2] [16] [22], our results show that *GRAND* and *PRAND* achieve near-optimal system costs while maintaining a tunable trade-off with queueing stability. Furthermore, given only mild-value of predicted information, *PRAND* induces a significant reduction in request response time, even in face of mis-prediction.

We organize the rest of paper as follows. Section 2 illustrates the basic idea, modeling, and problem formulation of dynamic switch-controller association and control devolution. Then Section 3 shows the algorithm design of *GRAND* and its performance analysis. Section 4 extends the previous formulation with predictive scheduling and presents the algorithm design of *PRAND*. Section 5 evaluates *GRAND* and *PRAND*, while Section 6 concludes this paper.

2 Problem Formulation

In this section, we first present a motivating example that reveals the non-trivial trade-off in the joint design of dynamic switch-controller association and control devolution; then we introduce our system model and problem formulation.

2.1 Motivating Example

Figure 1 shows the evolution of an SDN system during one time slot. Particularly, Figure 1 (a) presents the initial system state at the beginning of the time slot. Figure 1 (b) – (e) show two scheduling decisions and their consequent system

1. The scenario that some requests can only be processed by a controller is a special case of our model.

2. The scenario that some future requests may not be pre-served due to their statefulness is also a special case of our model.

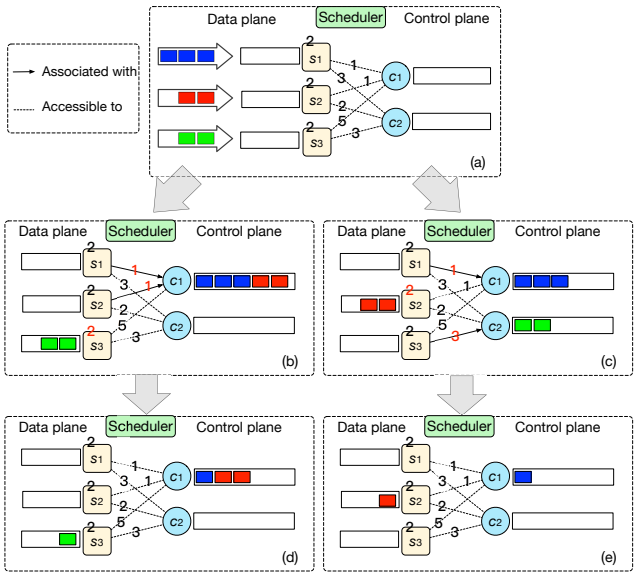


Fig. 1. *The scheduling process at the request level in one time slot.* There are 3 switches(s_1, s_2, s_3), 2 controllers(c_1, c_2), and 1 global scheduler. Switches and controllers maintain queue backlogs to buffer requests. Between controllers and switches, each dotted line denotes a potential connection in between, while each solid line denotes the actual association made in current time slot. Each connection and each switch are associated with a number denoting the unit communication or local computation cost (on switches) for transferring and processing one request, respectively. Each controller can serve up to 2 requests for each time slot, while each switch serves only 1 request. The goal of the scheduler is to minimize the sum of communication cost and computational cost and ensure the total backlog size as small as possible. The system proceeds as follows: At the beginning of time slot t, s_1, s_2, and s_3 generate 3, 2, and 2 requests, respectively. The scheduler then collects system dynamics and decides the switch-controller association and control devolution (could be (b) or (c)). According to the decisions made, each switch either processes its requests locally or sends them to controllers.

evolutions. In Figure 1 (b), both s_1 and s_2 are associated (using solid lines) to c_1, whereas s_3 chooses processing the two requests locally. In Figure 1 (d), s_1 and s_3 are associated to c_1 and c_2, respectively, whereas s_2 chooses processing the two requests locally. We use red color to mark the incurred per-request communication costs and computation costs.

We focus on the behavior of s_3. In Figure 1 (b), s_3 chooses to process its requests locally, and that incurs a computational cost of 2 per request. In Figure 1 (c), s_3 decides to upload requests to c_2 and that incurs a communication cost of 3 per request. Although the computational cost is less than communication cost, the decision of locally processing leaves one request not processed yet at the end of the time slot. Hence, it is not necessarily a smart decision for a switch to perform control devolution when its computational cost is lower than its communication cost. Instead, the scheduler should jointly decide control devolution and switch-controller association at the same time.

Next, we compare the two scheduling decisions in Figure 1 (b) and (c). Figure 1 (b) shows the switch-controller association with (s_1, c_1) and (s_2, c_1) (s_3 processes requests locally), denoted by X_1. In Figure 1, we can see X_1 results in uneven queue backlogs, leaving four requests unfinished at the end of the time slot, although it incurs the total cost of communication and computation by only 9. Figure 1 (c) shows another association with (s_1, c_1) and (s_3, c_2) (s_2 processes requests locally), denoted by X_2. In Figure 1(e), we can see X_2 does better in balancing queue backlogs than X_1, but it incurs higher cost by 13. Thus there is a non-trivial trade-off between minimizing the total cost of communication and computation and maintaining small queue backlogs on each controller.

TABLE 1
Key Notations

Symbol	Description
$\mathcal{C}$	The set of controllers in the control plane
$\mathcal{S}$	The set of switches in the data plane
$Q_i^s(t)$	Switch i's local queue backlog size in time slot t
$Q_j^c(t)$	Controller j's queue backlog in time slot t
$A_i(t)$	The number of request arrivals on switch i in time t
$B_i^s(t)$	The service capacity of switch i in time slot t
$B_j^c(t)$	The service capacity of controller j in time slot t
$M_{i,j}(t)$	The per-request communication cost between switch i and controller j
$P_i(t)$	The per-request computational cost on switch i
$X_{i,j}(t)$	The association decision for switch i and controller j in time slot t
$Y_i(t)$	The admission decision for switch i

2.2 Problem Formulation

We develop a fine-grained queueing model to characterize SDN systems, with key notations summarized in Table 1.

We consider a time slotted network system, indexed by $\{0, 1, 2, \dots\}$. Its control plane comprises a set $\mathcal{C}$ of physically distributed controllers, while its data plane consists of a set of switches $\mathcal{S}$. Each switch $i \in \mathcal{S}$ keeps a queue backlog of size $Q_i^s(t)$ for locally processing requests, while each controller $j \in \mathcal{C}$ maintains a queue backlog $Q_j^c(t)$ that buffers requests from data plane. We denote $[Q_1^c(t), \dots, Q_{|\mathcal{C}|}^c(t)]$ as $\mathbf{Q}^c(t)$ and $[Q_1^s(t), \dots, Q_{|\mathcal{S}|}^s(t)]$ as $\mathbf{Q}^s(t)$. We use $\mathbf{Q}(t)$ to denote $[\mathbf{Q}^s(t), \mathbf{Q}^c(t)]$.

We consider the request arrivals on each switch $i \in \mathcal{S}$ in time slot t. We denote by $A_i(t)$ ($\leq a_{max}$ for some constant a_{max}) the number of new requests that are actually arriving at switch i, and such arrivals $A_i(t)$ are assumed i.i.d. over time slots. Next, we consider processing capacities of switches and controllers. We assume all the requests to be homogeneous. Each of them can be handled by both switches and controllers. In time slot t, each controller $j \in \mathcal{C}$ has a service capacity of $B_j^c(t)$ requests, while each switch $i \in \mathcal{S}$ has a service capacity of $B_i^s(t)$ requests. For simplicity, we denote all service rates $\{B_j^c(t)\}_{j\in\mathcal{C}}$ and $\{B_i^s(t)\}_{i\in\mathcal{S}}$ by $\mathbf{B}(t)$. Due to limited resources on switches and controllers, we assume there exist constants b_{max}^c and b_{max}^s such that, $B_j^c(t) \leq b_{max}^c$, $\forall j \in \mathcal{C}$ and $B_i^s(t) \leq b_{max}^s$, $\forall i \in \mathcal{S}$. We also assume the existence of $E\{(A_i(t))^2\}$, $E\{(B_j^c(t))^2\}$, and $E\{(B_i^s(t))^2\}$.

At the beginning of time slot t, the scheduler collects system dynamics information $(\mathbf{A}(t), \mathbf{B}(t), \mathbf{Q}(t))$ and makes a scheduling decision, denoted by an association matrix $\mathbf{X}(t) \in \{0, 1\}^{|\mathcal{S}|\times|\mathcal{C}|}$. Here $\mathbf{X}(t)_{i,j} = 1$ if switch i will be associated with controller j during current time slot and 0 otherwise. An association is feasible if it guarantees that each switch is associated with at most one controller during each time slot. We denote the set of feasible associations as

$$\mathcal{A} \triangleq \Big\{\mathbf{X} \in \{0,1\}^{|\mathcal{S}|\times|\mathcal{C}|} \mid \sum_{j\in\mathcal{C}} \mathbf{X}_{i,j} \leq 1 \text{ for } i \in \mathcal{S}\Big\}. \quad (1)$$

The scheduler then notifies switches with the decisions being made. The notification mechanism depends on the

scheduler implementation. We'll discuss the implementation in Section 4. Then for switch i, it sends its request to controller j if $\mathbf{X}_{i,j} = 1$. If switch i is associated with no contorllers, *i.e.*, $\sum_{j\in\mathcal{C}} \mathbf{X}_{i,j} = 0$, it appends the requests to its local queue backlog. After the association and devolution are finished, switches and controllers serve as many requests in their queues as they could. As a result, the update equation for the queue backlog $Q_i^s(t)$ at switch i is

$$Q_i^s(t+1) = \left[Q_i^s(t) + \Big(1 - \sum_{j\in\mathcal{C}} \mathbf{X}_{i,j}(t)\Big) A_i(t) - B_i^s(t)\right]^+, \tag{2}$$

and the update equation for $Q_j^c(t)$ at controller j is given by

$$Q_j^c(t+1) = \left[Q_j^c(t) + \sum_{i\in\mathcal{S}} \mathbf{X}_{i,j}(t)\cdot A_i(t) - B_j^c(t)\right]^+, \tag{3}$$

where $[\,x\,]^+ = \max(x, 0)$. Having introduced the necessary notations, we switch to the problem formulation part.

2.2.1 Time-Average Communication Cost

Request transmissions from data plane to control plane would incur some communication cost in terms of, *e.g.*, the number of hops or round-trip times. A lower communication cost often implies better responsiveness to the requests. Thus it is desirable to minimize the total communication cost. In time slot t, we define $M_{i,j}(t)$ as the communication cost incurred by forwarding one request from switch i to controller j. [3] Denoting the set $\{M_{i,j}(t)\}_{i\in\mathcal{S},j\in\mathcal{C}}$ by $\mathbf{M}(t)$, Fixing some association $\mathbf{X} \in \mathcal{A}$, we define the total communication cost in time slot t is

$$f_{\mathbf{X}}(t) = \hat{f}(\mathbf{X}, \mathbf{A}(t)) \triangleq \sum_{j\in\mathcal{C}}\sum_{i\in\mathcal{S}} M_{i,j}(t)\cdot \mathbf{X}_{i,j}\cdot A_i(t), \tag{4}$$

where we can regard $M_{i,j}(t)$ as the price of transmitting one request from switch i to controller j.

2.2.2 Time-average Computational Cost

Considering the scarcity of switches' computational resources, the scheduler should also carefully take the local computation cost into account when performing control devolution. We use $P_i(t)$ to denote the computational cost for keeping a request locally on switch i in time t, with $\mathbf{P}(t)$ as $\{P_i(t)\}_{i\in\mathcal{S}}$. Given some association $\mathbf{X} \in \mathcal{A}$, we define the one-time-slot computational cost as

$$g_{\mathbf{X}}(t) = \hat{g}(\mathbf{X}, \mathbf{A}(t)) \triangleq \sum_{i\in\mathcal{S}} P(i)\cdot \Big(1 - \sum_{j\in\mathcal{C}} \mathbf{X}_{i,j}\Big)\cdot A_i(t). \tag{5}$$

2.2.3 Queueing Stability

To ensure the timely processing of requests, it is necessary to balance queue backlogs on controllers and switches, so that no queue backlogs would suffer from being overloaded. Hence, we also require the stability of all queue backlogs in the system. For the data plane and the control plane, we define the queueing stability [20] as

$$\limsup_{T\to\infty} \frac{1}{T}\sum_{\tau=0}^{T-1}\sum_{i\in\mathcal{S}} \left[E\left\{Q_i^s(\tau)\right\} + E\left\{Q_j^c(\tau)\right\}\right] < \infty. \tag{6}$$

3. The communication cost can be the number of hops, round-trip times (RTT), or transmission powers.

2.2.4 Problem Formulation

Consequently, our formulate the following stochastic network optimization problem that aims to minimize the long-term time-average expectation of the weighted sum of total communication costs and computation costs, while ensuring the long-term queueing stability.

$$\begin{aligned} \underset{\{\mathbf{X}(t)\in\mathcal{A}\}_t}{\text{Minimize}} \quad & \limsup_{T\to\infty} \frac{1}{T}\sum_{\tau=0}^{T-1}\Big[\mathbb{E}\left\{f(\tau)\right\} + \gamma\mathbb{E}\left\{g(\tau)\right\}\Big] \\ \text{subject to} \quad & (2), (3), (6), \end{aligned} \tag{7}$$

where γ is a non-negative constant that weighs the scarcity of computation resources on switches.

3 Algorithm Design and Analysis

In this section, we solve our stochastic optimization problem (7) by first transforming it into a series of one-time-slot problems, and designing optimal algorithm that solves the problem in each time slot. Our algorithm design is then followed by a theoretical analysis on its performance.

3.1 Algorithm Design

To design a scheduling algorithm that solves problem (7), we adopt the Lyapunov optimization technique [20].

First, we define the quadratic Lyapunov function as

$$L(\mathbf{Q}(t)) \triangleq \frac{1}{2}\Big[\sum_{j\in\mathcal{C}}\Big(Q_j^c(t)\Big)^2 + \sum_{i\in\mathcal{S}}\Big(Q_i^s(t)\Big)^2\Big]. \tag{8}$$

Next, we define the conditional Lyapunov drift for two consecutive time slots as

$$\Delta\left(\mathbf{Q}(t)\right) \triangleq E\Big\{L(\mathbf{Q}(t+1)) - L(\mathbf{Q}(t)) \,|\, \mathbf{Q}(t)\Big\}. \tag{9}$$

This conditional difference measures the general change in queues' congestion state. We want to push such difference as low as possible, so as to prevent queues $\mathbf{Q}^s(t)$ and $\mathbf{Q}^c(t)$ from being overloaded. However, to maintain small queue backlogs, the action we take, *e.g.* $\mathbf{X}$, might incur considerable communication cost $f_{\mathbf{X}}(t)$ or computational cost $g_{\mathbf{X}}(t)$, or both. Hence, we should jointly consider both queueing stability and the total cost $f_{\mathbf{X}}(t) + g_{\mathbf{X}}(t)$.

Given any feasible association $\mathbf{X} \in \mathcal{A}$, we define the one-time-slot conditional drift-plus-penalty function as

$$\Delta_V(\mathbf{Q}(t)) \triangleq \Delta(\mathbf{Q}(t)) + V\cdot E\left\{f_{\mathbf{X}}(t) + g_{\mathbf{X}}(t)|\mathbf{Q}(t)\right\}. \tag{10}$$

where V is a positive constant parameter that weighs the penalty brought by communication costs between switches and controllers ($f_{\mathbf{X}}(t)$) and the local computation costs on switches due to control devolution ($g_{\mathbf{X}}(t)$). By minimizing the upper bound of the drift-plus-penalty term (34), the time-average communication cost can be minimized while stabilizing the network of request queues [20]. We then employ the concept of *opportunistically minimizing an expectation* [20]. Then we transform the long-term stochastic optimization problem (7) into the following drift-plus-penalty minimization problem in every time slot t. [4]

$$\begin{aligned} \underset{\mathbf{X}\in\mathcal{A}}{\text{Minimize}} \quad & V\cdot\Big(\hat{f}(\mathbf{X}, \mathbf{A}(t)) + \hat{g}(\mathbf{X}, \mathbf{A}(t))\Big) + \\ & \sum_{j\in\mathcal{C}} Q_j^c(t)\cdot\Big[\sum_{i\in\mathcal{S}}\mathbf{X}_{i,j}\cdot A_i(t)\Big] + \\ & \sum_{i\in\mathcal{S}} Q_i^s(t)\cdot\Big[(1 - \sum_{j\in\mathcal{C}}\mathbf{X}_{i,j})\cdot A_i(t)\Big]. \end{aligned} \tag{11}$$

4. The detail of the transformation can be found in Appendix A.

After rearranging the terms in (11), our optimization problem turns out to be

$$\underset{\mathbf{X}\in\mathcal{A}}{\text{Minimize}} \sum_{i\in\mathcal{S}}\Big[V\cdot P_i(t)+Q_i^s(t)\Big]A_i(t)+\sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{C}}\Big[VM_{i,j}(t) +Q_j^c(t)-VP_i(t)-Q_i^s(t)\Big]\mathbf{X}_{i,j}A_i(t). \tag{12}$$

Since the first summation term $\sum_{i\in\mathcal{S}}[VP_i(t)+Q_i^s(t)]$ in (35) has nothing to do with $\mathbf{X}$, we regard it as constant and focus on minimizing only the second summation term of (35). To simplify notations, we define

$$\begin{aligned}\omega(i,j) &\triangleq -\Big(V\cdot M_{i,j}(t)+Q_j^c(t)-V\cdot P_i(t)-Q_i^s(t)\Big)\\ &= V\cdot(P_i(t)-M_{i,j}(t))+\Big(Q_i^s(t)-Q_j^c(t)\Big).\end{aligned} \tag{13}$$

Equivalently, we can rewrite problem (35) as

$$\underset{\mathbf{X}\in\mathcal{A}}{\text{Maximize}} \quad \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{C}}\omega(i,j)\mathbf{X}_{i,j}A_i(t). \tag{14}$$

We find that the association decisions for different switches can be decoupled in problem (14). Thus for each $i\in\mathcal{S}$, we split $\mathcal{C}$ into two disjoint sets $\mathcal{J}_1^i$ and $\mathcal{J}_2^i$ (*i.e.* $\mathcal{J}_1^i\dot{\bigcup}\mathcal{J}_2^i=\mathcal{C}$) such that

$$\mathcal{J}_1^i\triangleq\{j\in\mathcal{C}\,|\,\omega(i,j)<0\} \text{ and } \mathcal{J}_2^i\triangleq\{j\in\mathcal{C}\,|\,\omega(i,j)\geq 0\}. \tag{15}$$

As a result, for each switch $i\in\mathcal{S}$,

$$\begin{aligned}&\sum_{j\in\mathcal{C}}\omega(i,j)\mathbf{X}_{i,j}A_i(t)\\ &=\Big\{\sum_{j\in\mathcal{J}_1^i}\omega(i,j)\mathbf{X}_{i,j}+\sum_{j\in\mathcal{J}_2^i}\omega(i,j)\mathbf{X}_{i,j}\Big\}A_i(t).\end{aligned} \tag{16}$$

Next, we show how to minimize (16) with $\mathbf{X}\in\mathcal{A}$. We use $\mathbf{X}^*$ to denote the optimal solution to maximize (16). For each switch $i\in\mathcal{S}$, we consider two different cases.

i. If $\mathcal{J}_2^i=\emptyset$, *i.e.*, $\omega(i,j)<0$ for all $j\in\mathcal{C}$, then the only way to maximize (16) is setting $\mathbf{X}_{i,j}^*=0$ for all $j\in\mathcal{C}$.

ii. If $\mathcal{J}_2^i\neq\emptyset$, then we handle with $\mathbf{X}_{i,j}$ for $j\in\mathcal{J}_1^i$ and $j\in\mathcal{J}_2^i$ separately.

– For $j\in\mathcal{J}_1^i$, to maximize (16), it is not hard to see one should set $\mathbf{X}_{i,j}^*=0$ for all $j\in\mathcal{J}_1^i$.

– For $j\in\mathcal{J}_2^i, \omega(i,j)\geq 0$. Then one should set $\mathbf{X}_{i,j^*}^*=1$ for such j^* that

$$j^*\in\underset{j\in\mathcal{J}_2^i}{\arg\max}\quad\omega(i,j), \tag{17}$$

and $\mathbf{X}_{i,j}^*=0$ for $j\in\mathcal{J}_2^i-\{j^*\}$.

Consequently, we obtain the optimal solution $\mathbf{X}^*$ that maximizes (16), and equivalently minimizes (35). We present the pseudocode for our algorithm in Algorithm (1).

3.1.1 Remarks:

Remark 1. For switch i and controller j, $\omega(i,j)$ actually reflects the willingness of switch i to send its new requests to controller j. According to (13), its willingness is determined by two factors.

1) One is the relative size of switch i's local computation cost $P_i(t)$ compared to its communication cost $M_{i,j}(t)$ to controller j. If $P_i(t)$ is relatively greater than $M_{i,j}(t)$, then switch i will tend to uploading its requests instead of locally processing them.
2) Before making the decision, switch i also needs to consider the other factor: the relative size of local queue backlog size $Q_i^s(t)$ in time slot t compared to controller j's backlog $Q_j^c(t)$. If switch i's queue backlog is more loaded than controller j's, *i.e.*, $Q_i^s(t)\geq Q_j^c(t)$, then its willingness to upload requests will be increased, too. Otherwise, the willingness will be decreased.

Next, we consider the willingness in different scenarios.

1) If switch i has a considerable computation cost and more loads on its local queue than controller j, then its willingness to upload requests will be positive.
2) If switch i has a low computation cost and less loads on its local queue than controller j, then it will be more unwilling to send requests since local processing can incur faster response times.
3) However, if switch i has relatively low computation cost but with less loads than controller j, then its willingness is up to which factor is more important: to avoid high cost or to keep its local backlog small. The relative importance between the two factors is determined by parameter V. With a greater value of V, switch i will focus more on avoiding high cost. Its willingness is thus a result of the two factors.
4) If switch i has relatively high computation cost but with less loads than controller j, similar to previous case, the willingness also depends on V.

Remark 2. *GRAND* is online because it makes all the decisions based on only the available system dynamics in the current time slot. Next, *GRAND* is greedy. For switch i, it determines the switch's association and devolution by the following process. Among switch i's potential controllers, if there's no such controller that switch i is willing to upload its requests, *i.e.*, $\omega(i,j)<0$ for all j, then switch i will process requests locally. To the contrary, if there exists at least one controller j that switch i is willing to upload requests, then *GRAND* greedily associates switch i to the controller with the highest willingness.

Algorithm 1 GRAND (GReedy switch-controller Association aNd control Devolution)

Input: During each time slot t, the scheduler collects system dynamics, *e.g.*, *i.e.* $\mathbf{Q}^c(t)$, $\mathbf{Q}^s(t)$, $\mathbf{A}(t)$, and $\mathbf{U}(t)$.
Output: A scheduling association $\mathbf{X}\in\{0,1\}^{|\mathcal{S}|\times|\mathcal{C}|}$.

1: Initially, set $\mathbf{X}_{i,j}\leftarrow 0$ for all $(i,j)\in\mathcal{S}\times\mathcal{C}$
2: **for** each switch $i\in\mathcal{S}$ **do**
3: Divide all controllers $\mathcal{C}$ into two sets $\mathcal{J}_1^i$ and $\mathcal{J}_2^i$, where $\mathcal{J}_1^i\leftarrow\{j\in\mathcal{C}\,|\,\omega(i,j)<0\}$ and $\mathcal{J}_2^i\leftarrow\{j\in\mathcal{C}\,|\,\omega(i,j)\geq 0\}$.
4: If $\mathcal{J}_2^i=\emptyset$, then skip the current iteration.
5: If $\mathcal{J}_2^i\neq\emptyset$, then choose a controller $j^*\in\mathcal{J}_2^i$ such that

$$j^*\in\underset{j\in\mathcal{C}}{\arg\max}\,\omega(i,j),$$

6: and set $\mathbf{X}_{i,j^*}\leftarrow 1$.
7: **end for**
8: **return** $\mathbf{X}$

According to the association decision $\mathbf{X}$, switches upload requests to controllers or append requests to their local queues. Then controllers and switches update their queue backlogs as in (2) and (3) after serving requests.

3.2 Performance Analysis

We assume that the expectation of total system processing capacity is greater than that of the total request arrival rate. Then *GRAND* achieves the classic $[O(V), O(1/V)]$ trade-off between the time-average expectation of total cost and total queue backlog size in the system. The proof is relegated to Appendix B.

As for the time complexity of *GRAND*, note from Algorithm 3.1 that GRAND can be run in a distributed manner. In particular, after acquiring the instant information system dynamics at the beginning of each time slot, each switch conducts decision making independent of each other. For each switch, the computation and searching for the best candidate to process requests require only time complexity in $O(|\mathcal{C}|)$. Such a low complexity allows system designers to trade off only little overheads for notable improvement in system performance. We will further discuss its implementation in Section 5.

4 Extension with Predictive Scheduling

In this section, we take a further step by integrating the joint control of switch-controller association and control devolution with predictive scheduling. We present an example that shows the potential benefits of predictive scheduling in Appendix C.

4.1 Model with Predictive Scheduling

4.1.1 Pre-service Model

In addition to the processing of actually arriving requests, as defined in Section 2.2, we take a further step by considering the case when the system can predict future request arrivals in a finite number of time slots ahead. Meanwhile, pre-serving future requests is assumed applicable. In particular, each switch is embedded with a learning module that predicts the request arrival, and maintains extra queue backlog to buffer requests that are yet to arrive, but predicted and generated. Such requests are recorded by the switch and marked as predicted by using only one bit in their headers. Both arriving and predicted requests, if scheduled, are distributed to corresponding processing queues and served by some queueing policy, *e.g.*, in a first-in-first-out (FIFO) manner. When predicted requests arrive, if pre-served, they will be considered finished instantly.

Formally, in time slot t, we consider each switch i having access to its future request arrivals in a prediction window of size D_i ($< D$ for some constant D), denoted by $\{A_i(t+1), \ldots, A_i(t+D_i)\}$. There are two things to note in practice. First, due to the resource scarcity on switches and the uncertainty in request arrivals, one should leverage time series forecasting techniques with low computational complexity [3], and the size of the prediction window would not be very large, only few time slots. Second, the prediction may be inaccurate often times, which can lead to extra resource consumption. We delegate the evaluation of misprediction's impact to Section V.

With predictive scheduling, some future requests might have been pre-admitted into or pre-served before time t, thus we use $Q_i^{(d)}(t)$ ($0 \le d \le D_i$) to denote the number of untreated requests in the d-th slot ahead of time t, such that

$$0 \le Q_i^{(d)}(t) \le A_i(t+d). \tag{18}$$

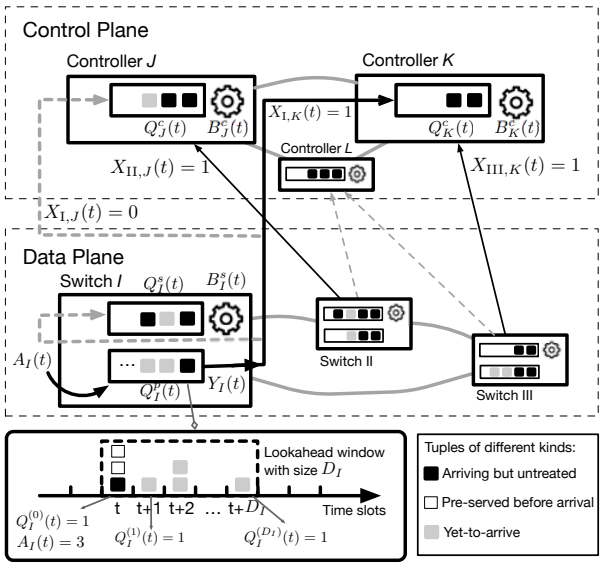


Fig. 2. An example of queueing model with predictive scheduling: At the beginning of each time slot, requests arrive at switches over time slots. Along with predicted request arrivals, they are buffered in queue $Q_i^p(t)$. Meanwhile, each switch i makes the admission decision $X_i(t)$ and association decision $Y_{i,*}(t)$, then either forwards requests to controllers or process them locally. Such requests are queued up, respectively, and await to be processed in a first-in-first-out order.

Note that $Q_i^{(0)}(t)$ denotes the number of untreated requests that actually arrive in time slot t. We denote the total number of untreated requests on each switch i by $Q_i^p(t) = \sum_{d=0}^{D_i} Q_i^{(d)}(t)$. We regard $Q_i^p(t)$ as a virtual prediction queue backlog that buffers untreated future requests for switch i.

To simplify the notations, we define $\mathbf{Q}(t)$ as the vector of all queue backlogs $\{Q_i^p(t)\}_{i\in\mathcal{S}}$, $\{Q_i^s(t)\}_{i\in\mathcal{S}}$, and $\{Q_j^c(t)\}_{j\in\mathcal{C}}$.

4.1.2 Scheduling Decisions

Unlike *GRAND*, upon new requests' arrivals, each switch should make two decisions with predictive scheduling.

The first is **admission decision**, *i.e.*, deciding the total number of untreated requests to be admitted, including requests that actually arrive in the current time slot and untreated future requests in its own prediction window. We use $Y_i(t)$ to denote the number of requests to be admitted from switch i in time slot t. These admitted requests should at least include all untreated requests that actually arrive, while not exceeding $Q_i^p(t)$, *i.e.*,

$$Q_i^{(0)}(t) \le Y_i(t) \le Q_i^p(t), \quad \forall i \in \mathcal{S}, t \ge 0. \tag{19}$$

The second is **association decisions**, *i.e.*, deciding to process these admitted requests locally (control devolution), or associate with and forward requests to one of its potentially connected controllers. We adopt $X_{i,j}(t)$ to denote such decisions, as defined in Section 2.2.

We denote sets $\{Y_i(t)\}_{i\in\mathcal{S}}$ and $\{X_{i,j}(t)\}_{i\in\mathcal{S},j\in\mathcal{C}}$ by $\mathbf{Y}(t)$ and $\mathbf{X}(t)$, respectively.

4.1.3 System Workflow and Queueing Model

In each time slot t, the system workflow is as follows. At the beginning of time slot t, new requests arrive at switches. The scheduler collects all available system dynamics such as $\mathbf{Q}(t)$ and $\mathbf{B}(t)$. Next, the scheduler makes scheduling decisions $[\mathbf{X}(t), \mathbf{Y}(t)]$ and notifies all the switches. Each

switch $i \in \mathcal{S}$ then admits $Y_i(t)$ requests, possibly including untreated future requests. All the admitted future requests are marked treated and will not be served again thereafter. Meanwhile, based upon $\{X_{i,j}(t)\}_{j\in\mathcal{C}}$, each switch i sets up its association and forward the admitted requests accordingly. Next, all switches and controllers process requests from their processing queues. At the end of time t, the learning module on each switch makes its new prediction about the request arrival in the furthest slot. Based on the new prediction, all prediction windows move one slot ahead and the new future requests are appended to the corresponding prediction queues.

With the above system workflow, we have the following update equations for different queue backlogs.

1) For the prediction queue with respect to switch i,

$$Q_i^p(t+1) = [Q_i^p(t) + A_i(t+D_i+1) - Y_i(t)]^+ \tag{20}$$

2) For the processing queue on switch $i \in \mathcal{S}$,

$$Q_i^s(t+1) = \Big[Q_i^s(t) + (1 - \sum_{j\in\mathcal{C}} X_{i,j}(t))Y_i(t) - B_i^s(t)\Big]^+. \tag{21}$$

3) For the processing queue on controller $j \in \mathcal{C}$,

$$Q_j^c(t+1) = \Big[Q_j^c(t) + \textstyle\sum_{i\in\mathcal{S}} X_{i,j}(t)Y_i(t) - B_j^c(t)\Big]^+, \tag{22}$$

where $[x]^+ \triangleq \max\{x, 0\}$.

Note that the queueing model in (2) and (3) is a special case of the above mode, where in each time slot, the switch consistently admits requests that arrive in current time slot only. Figure 2 shows an example of the queueing model with predictive scheduling.

4.1.4 Optimization Objectives

Communication Cost: Recall that we denote by $M_{i,j}(t)$ the communication cost of transferring a request from switch i to controller j. With predictive scheduling, we can write the total communication cost in time t as

$$f_p(t) \triangleq \hat{f}_p\left(\mathbf{X}(t), \mathbf{Y}(t)\right) = \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{C}} M_{i,j}(t)X_{i,j}(t)Y_i(t). \tag{23}$$

Computational Cost: Recall that $P_i(t)$ to denote the computational cost for keeping a request locally on switch i in time t. Thus we can write the total computational cost in the extended model as

$$g_p(t) \triangleq \hat{g}_p\left[\mathbf{X}(t), \mathbf{Y}(t)\right] = \sum_{i\in\mathcal{S}} P_i(t)\Big[1 - \sum_{j\in\mathcal{C}} X_{i,j}(t)\Big]Y_i(t). \tag{24}$$

Queueing Stability: We first denote the weighted total queue backlog size in time slot t as

$$\begin{aligned} h_p(t) &\triangleq \hat{h}(\mathbf{Q}(t)) \\ &= \sum_{j\in\mathcal{C}} Q_j^c(t) + \beta_1 \sum_{i\in\mathcal{S}} Q_i^s(t) + \beta_2 \sum_{i\in\mathcal{S}} Q_i^p(t). \end{aligned} \tag{25}$$

where β_1 and β_2 are constants that weight the importance of balancing switches' queues and prediction queues compared to controllers' queues. We define the long-term queueing stability [20] constraint as

$$\limsup_{T\to\infty} \frac{1}{T} \sum_{\tau=0}^{T-1} \mathbb{E}\left\{h_p(\tau)\right\} < \infty. \tag{26}$$

4.2 Problem Formulation

We extend the problem formulation in (7) and write

$$\begin{aligned} \underset{\{(\mathbf{X}(t),\mathbf{Y}(t))\}_{t=0}^{T-1}}{\text{Minimize}} \quad & \limsup_{T\to\infty} \frac{1}{T} \sum_{\tau=0}^{T-1} \left[\mathbb{E}\left\{f_p(\tau)\right\} + \gamma \mathbb{E}\left\{g_p(\tau)\right\}\right] \\ \text{Subject to} \quad & X_{i,j}(t) \in \{0,1\},\ \forall i \in \mathcal{S}, j \in \mathcal{C}, \\ & Y_i(t) \in \mathbb{Z}_+,\ \forall i \in \mathcal{S}, \text{ and } (19)-(22), (26), \end{aligned} \tag{27}$$

where γ is a non-negative constant that weighs the scarcity of computation resources on switches.

4.3 Algorithm Design

Following similar steps in Section 3.1, we transform the long-term stochastic optimization problem (27) into a series of subproblems over time slots. The detail of the transformation can be found in Appendix A. The subproblem in time slot t can be written as

$$\begin{aligned} \underset{\mathbf{X}(t),\mathbf{Y}(t)}{\text{Maximize}} \quad & \sum_{i\in\mathcal{S}} l_i(t)Y_i(t) + \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{C}} u_{i,j}(t)X_{i,j}(t)Y_i(t) \\ \text{Subject to } & X_{i,j}(t) \in \{0,1\},\ \forall i \in \mathcal{S}, j \in \mathcal{C}, \\ & Y_i(t) \in \mathbb{Z}_+,\ \forall i \in \mathcal{S}, \text{ and } (19)-(22), \end{aligned} \tag{28}$$

where $l_i(t)$, $u_{i,j}(t)$ are defined as

$$l_i(t) \triangleq \beta Q_i^p(t) - \alpha Q_i^s(t) - V\gamma P_i(t), \tag{29}$$

$$u_{i,j}(t) \triangleq \left[\alpha Q_i^s(t) - Q_j^c(t)\right] + V\left[\gamma P_i(t) - M_{i,j}(t)\right]. \tag{30}$$

As in *GRAND*, V is a positive parameter to adjust the trade-off between queueing stability and reduction in the total system cost. The objective function of (28) can be decoupled for different switches. In time slot t, for each switch $i \in \mathcal{S}$,

$$\begin{aligned} \underset{\mathbf{X}(t),\mathbf{Y}(t)}{\text{Maximize}} \quad & l_i(t)\cdot Y_i(t) + \Big[\sum_{j\in\mathcal{C}} u_{i,j}(t)X_{i,j}(t)\Big]\cdot Y_i(t) \\ \text{Subject to } & X_{i,j}(t) \in \{0,1\},\ Y_i(t) \in \mathbb{Z}_+,\ \forall\, j \in \mathcal{C}, \\ & \sum_{j\in\mathcal{C}} X_{i,j}(t) \le 1,\ Q_i^{(0)}(t) \le Y_i(t) \le Q_i^p(t). \end{aligned} \tag{31}$$

Notice that $X_{i,*}(t)$ and $Y_i(t)$ are coupled in the objective function, making (31) a combinatorial problem. However, by exploiting the subproblem's structure, it can still be solved optimally. We unfold the algorithm design as follows.

Given $[\mathbf{Q}(t), \mathbf{P}(t), \mathbf{M}(t)]$, $l_i(t)$ and $u_{i,j}(t)$ are constants in time t. For switch i, problem (31) is equivalent to finding the maximum product of two terms: 1) $Y_i(t)$ (positive and bounded), 2) $l_i(t) + \sum_{j\in\mathcal{C}} X_{i,j}(t)u_{i,j}(t)$, dependent on $X_{i,*}(t)$ that at most one of them equals one.

To achieve the maximum, we consider three cases.

1) If $l_i(t) < 0$ and $l_i(t) + u_{i,j}(t) < 0$ for all j, then one should consider two sub-cases. In either case, $Y_i(t)$ should be set as its minimum $Q_i^{(0)}$, and we aim to decide $X_{i,*}(t)$ that maximizes $l_i(t) + \sum_{j\in\mathcal{C}} X_{i,j}(t)u_{i,j}(t)$.
   a) If $l_i(t) > \max_j(l_i(t) + u_{i,j}(t))$, *i.e.*, if $u_{i,j}(t) < 0$ for all j, then choose no controllers, *i.e.*, $X_{i,j}(t) = 0$ for all j.
   b) Else, if $\hat{\mathcal{C}} \triangleq \{j' \in \mathcal{C} \mid u_{i,j'}(t) \ge 0\} \neq \emptyset$, then one should choose the controller $j^* \in \arg\max_{j'\in\hat{\mathcal{C}}} u_{i,j'}(t)$ by setting $X_{i,j^*}(t) = 1$ and others to be zero.

2) If $l_i(t) \geq 0$ and $l_i(t) + u_{i,j}(t) < 0$ for all j, then the best choice is to choose no controllers (so that the second term equals $l_i(t)$) and set $Y_i(t)$ to its maximum $Q_i^p(t)$.
3) If $\tilde{\mathcal{C}} \triangleq \{j' \in \mathcal{C} \mid l_i(t) + u_{i,j}(t) \geq 0\} \neq \emptyset$, then the optimal solution is attained by setting $X_{i,j^*}(t) = 1$ for $j^* \in \arg\max_{j' \in \tilde{\mathcal{C}}} u_{i,j'}(t)$ and others to be zero. Still, $Y_i(t)$ should be set as $Q_i^p(t)$.

In such a way, subproblem (31) can be solved optimally. Accordingly, we propose *PRAND*, an online and distributed algorithm that solves (31) in every time slot. The pseudocode of *PRAND* is shown in Algorithm 2.

Algorithm 2 POSCAD (Predictive Online Switch-Controller Association and Control Devolution) in each time slot t

Input: Current queue backlog sizes $\mathbf{Q}(t)$, service rates $\mathbf{B}(t)$, computation costs $\mathbf{P}(t)$ on switches, and communication costs $\mathbf{M}(t)$ between switches and controllers.
Output: Admission and association decisions.

for each switch $i \in \mathcal{S}$
  Calculate $l_i(t)$ and $w_{i,j}(t)$ for every controller $j \in \mathcal{C}$ according to (29) and (30), respectively.
  if $w_{i,j}(t) < 0$ for every controller $j \in \mathcal{C}$ **then**
    if $l_i(t) > 0$ **then**
      admit all $Q_i^p(t)$ requests from i's prediction queue.
    else admit $Q_i^{(0)}(t)$ untreated requests in time slot t.
  endif
    Append all admitted requests to the local processing queue $Q_i^s(t)$ on switch i.
  else associate switch i with the controller j^* such that

$$j^* \in \arg\max_{j \in \mathcal{C}} w_{i,j}(t)$$

    if $l_i(t) + w_{i,j^*}(t) > 0$ **then**
      admit all $Q_i^p(t)$ requests from i's prediction queue.
    else admit $Q_i^{(0)}(t)$ untreated requests in time slot t.
  endif
    Forward all admitted requests to controller j^* and append them to $Q_{j^*}^c(t)$ therein.
  endif
endfor
All switches and controllers then consume requests from their respective processing queues.

4.4 Discussion:

First, we note that, without predictive scheduling, *PRAND* degrades to *GRAND* by setting $Y_i(t)$ as $A_i(t)$ for all t; *i.e.*, *PRAND* only admits and treats requests that actually arrive and in the current time slot. However, integrated with predictive scheduling, *PRAND* also considers when to preserve future requests, inducing more complex decision making. In such a way, *PRAND* goes beyond *GRAND* by pre-admitting future requests opportunistically and exploiting the surplus system processing capacity in every time slot to pre-serve them, leading to even lower request response time. We verify such effectiveness of *PRAND* in Section 5.

Next, we discuss the roles of $l_i(t)$ and $u_{i,j}(t)$ in *PRAND*, as defined in (29) and (30). On one hand, similar to *GRAND*, $u_{i,j}(t)$ reflects switch i's willingness of associating with controller j in time t. On the other hand, $l_i(t)$ quantifies the weighted balance between the number of untreated requests in the prediction queue and switch i's local queue backlog size and computation cost. The larger the value of $l_i(t)$, the more untreated future requests in switch i's prediction queue, and the more requests switch i tends to admit. Note that both $l_i(t)$ and $u_{i,j}(t)$ utilize queue backlog sizes and estimate communication cost as the indicator of congestion, which are directly attainable from switches and controllers at the beginning of each time slot. However, for information such as service capacity, they remain unknown until the end of time slot, after requests being processed.

The above illustration leads to a better understanding of how *PRAND* works. If switch i decides to process new requests locally, *i.e.*, $u_{i,j}(t) < 0$ for all $j \in \mathcal{C}$, the number of admitted requests depends on $l_i(t)$. If $l_i(t) > 0$, switch i would admit future requests to its local queue; otherwise, only untreated requests in current time slot will be admitted. If $u_{i,j}(t) > 0$ for some controllers, then switch i will associate with the controller j^* with the maximum $u_{i,j^*}(t)$. In this case, PRAND decides request admission based on $l_i(t) + u_{i,j^*}(t)$, which according to (29) and (30), turns out to be

$$l_i(t) + u_{i,j^*}(t) = Q_i^p(t) - Q_{j^*}^c(t) - V \cdot M_{i,j^*}(t). \quad (32)$$

Intuitively, $l_i(t) + u_{i,j^*}(t)$ reflects the weighted balance among the number of all untreated requests in the prediction queue, controller j^*'s queue backlog size, and their communication cost in between. Switch i tends to admit more future requests when $l_i(t) + u_{i,j^*}(t) > 0$, *i.e.*, controller j^* possesses a queue backlog with a small size and a low communication cost.

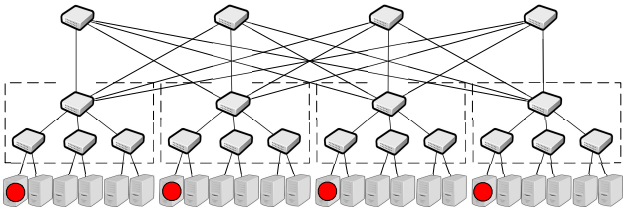
Similar to *GRAND*, the value of *PRAND*'s parameter V determines the relative importance of minimizing communication cost to maintaining queueing stability for switches and controllers. With an extremely large value of V, the terms related to queue backlogs in (29) and (30) become negligible. In this case, *PRAND* inclines to omit the balance between queue backlogs, and admit only arriving requests by sending them to the processing queue with least cost. In contrast, if the value of parameter V approaches zero, similar to *GRAND*, *PRAND* would ignore the impact of such costs and greedily forwards requests to the queue with the smallest backlog size. The choice of V reflects the main objective of system design. Last but not least, *PRAND* can also be run in a distributed fashion with a time complexity of $O(|\mathcal{C}|)$. The detailed analysis is similar to that of *GRAND* and hence omitted here.

5 Simulation Results

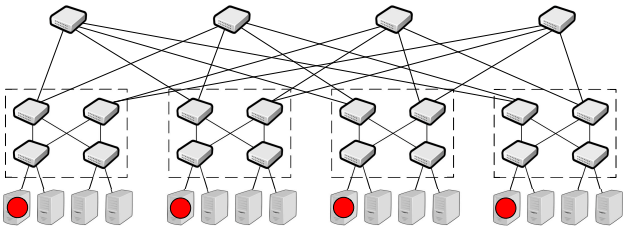
We conduct trace-driven simulations to evaluate the performance of *GRAND* and *PRAND* under different settings against various baseline schemes. Particularly, we first introduce the settings adopted in the simulation. Then we compare *GRAND* with baseline schemes in terms of the reduction in the total cost of communication and computation, and the total queue backlog size in the system. Next, we evaluate *PRAND* against *GRAND* to investigate the benefits of predictive scheduling, by conducting analysis on the results from the simulation.

5.1 Basic Settings

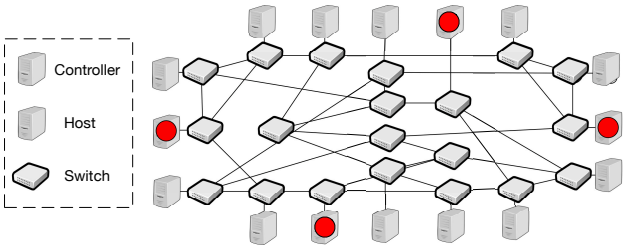
Topology: We conduct extensive simulations under four well-known data center topologies: Canonical 3-Tiered topology [2], Fat-tree [1], Jellyfish [22], and F10 [16]. We show one instance for each of them, respectively, in Figure 4.4. To make our performance analysis comparable among the four topologies, we construct instances of these



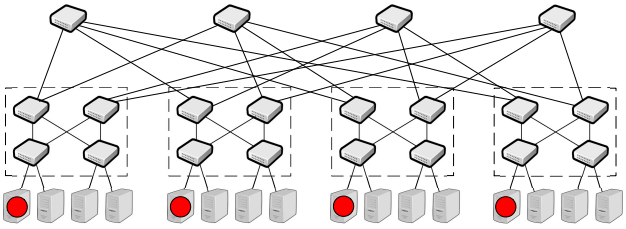
(a) Canonical 3-Tiered topology with $k = 4$, where k denotes the switch port number. In this paper, the number of aggregate switches is also set to k, and each connects to $k-1$ edge switches. The total number of switches is k^2+k. Each edge switch is directly connected to $k/2$ hosts. There are $k^3-k^2/2$ hosts in total.



(b) Fat-tree topology with $k = 4$, where k denotes the number of switch ports. The number of core, aggregate, edge switches are $k^2/4$, $k^2/2$, and $k^2/2$, respectively. The total number of switches and hosts are $5/4k^2$ and $k^3/4$, respectively. Each edge switch is directly connected to $k/2$ hosts.



(c) Jellyfish topology with $k = 4$, where k denotes the number of switch ports. Jellyfish topology is created by constructing a random graph at the top-of-rack (ToR) switch layer, and comprises $5/4k^2$ switches and $k^3/4$ hosts.



(d) F10 topology with $k = 4$. It has identical numbers of switches and hosts. They only differ lies in the interconnections among switches. F10 breaks the symmetry of Fat-tree's structure by employing AB-tree to enhance its fault tolerance [16].

Fig. 3. Instances of Canonical 3-Tiered topology [2], Fat-tree topology [1] , Jellyfish topology [22], and F10 topology [16], respectively.

topologies at almost the same scale, by assuming that all switches have identical port number. These topologies are comparable to the size of commercial data centers [2].

In these topologies, we deploy controllers on the hosts, which are denoted by hosts with red circles. In deterministic topologies (Fat-tree, Canonical 3-Tiered, and F10), we deploy one controller in every pod[5]. In random topology (Jellyfish), we deploy the same number of controllers on hosts with non-neighboring ToRs.

Request Arrival Settings: We conduct trace-driven simulations, where the request arrival process on each switch follows the distribution of request inter-arrival time that are drawn from measurements within real-world data centers. Accordingly, the average request inter-arrival time is about $1700\mu s$. We then set the length of each time slot as $10ms$. Accordingly, the average request arrival rate on each switch is about 5.88 requests per time slot. Considering that the existence of hot spots in real-world data center networks, where some switches have intensive request arrivals, in our simulation, we pick the first pod as the hot spot and each switch therein has a much higher request arrival rate (200 requests per time slot). As for controllers, each of them has a service capacity of 600 requests per time slot, which is consistent with the capacity of a typical NOX controller [24].

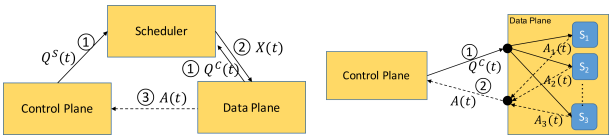
System Cost Settings: Given any network topology, we define the communication cost $M_{i,j}(t)$ between switch i and controller j as the length (number of hops) of shortest path from i to j. We set a common computation cost α for all switches, which equals to the average hop number between switches and controllers of its underlying topology. In both Fat-tree and F10 topologies, $\alpha = 4.13$; in 3-Tiered and Jellyfish topologies, α is 4.81 and 3.56, respectively. Besides, by setting weighting factor $\gamma = 1$, we assume the equal importance of reducing communication cost and local computational cost on switches.

Queueing: Regarding *PRAND*, we treat all queue backlogs equally by setting weighting factors β_1 and β_2 as 1 for (25). In practice, the value of parameter β_1 and β_2 can be tuned proportional to the ratio between the capacity of prediction queues, switches' queues, and controllers' queues. Besides, recall that the value of parameter V determines the importance of communication cost reduction to queueing stability. When applying *PRAND*, one can tune the value of parameter V proportional to the ratio between the magnitude of queue backlog size (in number of requests) and communication cost (in milliseconds). In our simulation, the ratio is about 100 and thereby we adjust the value of parameter V from 1 to 100. The queueing policy is set as *first-in-first-out* (FIFO) for all queues.

Scheduler Implementations: Figure 4 (a) and (b) illustrate how *GRAND* and *PRAND* can be implemented in a centralized and decentralized manner, respectively.

In the centralized way, the scheduler is independent of both control plane and data plane. The scheduler collects system dynamics including queue backlogs on both switches and controllers to make a centralized scheduling decision. Next, it spreads the scheduling decision onto switches; then switches upload or locally process their requests according to the decision. The abstract process is presented in Figure 4 (a). The advantage of centralized architecture is that it doesn't require modification on data plane, *i.e.*, all the system dynamics such as the communication cost and queue backlogs can be obtained via standard OpenFlow APIs. This is well-suited for the situation where the data plane is at a large scale and switches' compute resource is scarce. In fact, the scheduler could also be deployed on control plane. There are disadvantages, too. Centralized scheduler is a potential single point of failure, or even a bottleneck with considerable computation. Besides, it requires

5. In Canonical 3-Tiered topology, we regard the group of switches that affiliate the same aggregation switch as one pod (including the aggregation switch itself).



(a) Centralized (b) Decentralized

Fig. 4. Two scheduler implementations for *GRAND* and *PRAND*

back-and-forth message exchange between the SDN system and the scheduler, which leads to longer response time.

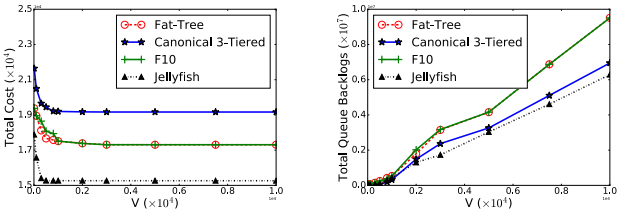
In the decentralized way, as Figure 4 (b) shows, switches periodically update their information about queue backlogs from control plane. Each of them makes scheduling decisions independently. Though requiring modification on switches, the decentralized way still has the following advantages. It requires less amounts of message exchange than that in the centralized way, thus switches would response even faster to handling network events. Meanwhile, the computation of both schemes decision-making is distributed onto switches, leading to better scalability and fault tolerance. In our simulation, we choose the decentralized implementation for both *GRAND* and *PRAND*.

Prediction Settings: We vary switches' prediction window sizes by sampling them uniformly at random in $[0, 2\times D]$, with mean D. The value of D ranges from 0 to 20. Besides, considering that prediction errors are inevitable in practice, we evaluate *PRAND*'s performance with prediction errors at different levels. Particularly, we use e_t to denote the prediction deviation for time slot t, *i.e.*, the difference between the number of predicted and actual arrivals. $e_t = 0$ indicates that the prediction is perfect. When the prediction is imperfect, $e_t > 0$ implies that the future request arrivals are over-estimated; otherwise, they are under-estimated. We also assume e_t to be i.i.d. over all time slots, and generate e_t by 1) sampling a value x_t from some probability distribution with zero mean and then 2) obtaining e_t by rounding x_t, *i.e.*, $e_t \triangleq \text{Round}(x_t)$. The second step is due to that the probability distribution may be continuous but the number of requests must be an integer value. With e_t and x_t, we define r as prediction error rate, *i.e.*, the probability of misprediction. Thus $r \triangleq \Pr\{e_t \neq 0\} = \Pr\{x_t \notin (-0.5, 0.5)\}$. In our simulation, we sample x_t from a *normal distribution* with zero mean and variance σ^2. Accordingly, we have

$$r = 2\left[1 - \Phi\left(0.5/\sigma\right)\right], \tag{33}$$

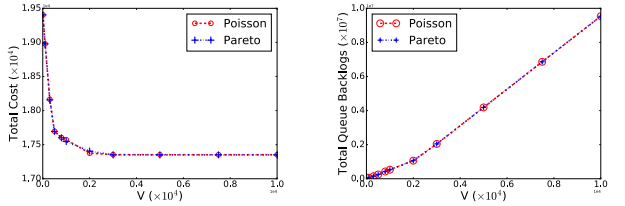
where $\Phi(\cdot)$ is the CDF of *standard normal distribution*. Based on (33), we can adjust the prediction error rate r by choosing an appropriate σ. We vary r from 0% to 50%.

Response Time Metric: As for evaluation metrics, note that the total queue backlog size in the non-prediction case ($D = 0$) and the predictive case ($D > 0$) are incomparable. The reason is that with predictive scheduling, the total queue backlog size, as defined in (25), also includes the future untreated requests in the prediction queues. All such requests are still counted, although they haven't actually arrived or even not been pre-served. The comparison is even more inappropriate with predictive scheduling in the presence of errors, where some future requests may not even exist due to over-estimation. Meanwhile, *PRAND* is promising to reduce request response times, since it exploits future information to pre-serve future requests with the spare system processing capacities. Hence, instead of comparing total queue backlog size, we evaluate *PRAND* in terms of the average request response time. In the simulation, we define a request's response time as the number of time slots from its actual arrival to its eventual completion. If pre-served before its actual arrival, a request will be responded instantly and experience a near-zero response time. The average request response time is obtained over completed requests.



(a) Total cost vs. V (b) Total queue backlog vs. V

Fig. 5. Performance of *GRAND* under different topologies.



(a) Total cost vs. V (b) Total queue backlog vs. V

Fig. 6. Performance of *GRAND* under Fat-tree topology with other two arrival processes.

5.2 Evaluation of *GRAND*

Figure 5 (a) presents how the summation of long-term average communication cost and computational cost change with different values of V in the four topologies. We make the following observations.

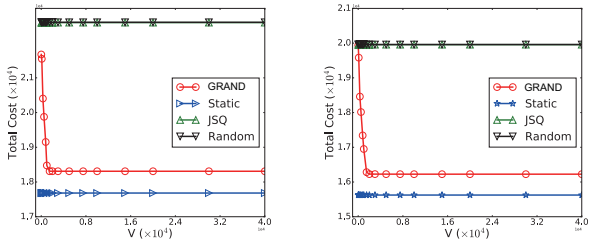
First, as the value of V varies from 0 to 1.0×10^4, it shows that the total cost goes down gradually. This is consistent with our previous theoretic analysis. The intuition behind such decline is as follows. Recall that V controls the switches' willingness of uploading requests. For switches that are close to controllers (their communication cost is less than the average), large V makes them unwilling to process requests locally unless the controllers get too heavy load. As V increases, those switches will choose to upload requests to further reduce the costs since for those switches, communication costs are less than the computation costs.

Second, the total cost in 3-Tiered topology is more than the other schemes'. The reasons are two-fold. One is 3-Tiered has a higher computational cost ($\alpha = 4.81$ compared to 4.13 and 3.56) and it costs even more when switches process requests locally. The other is that switches in 3-Tiered topology usually take longer path to controllers and incur more communication cost compared to other topologies.

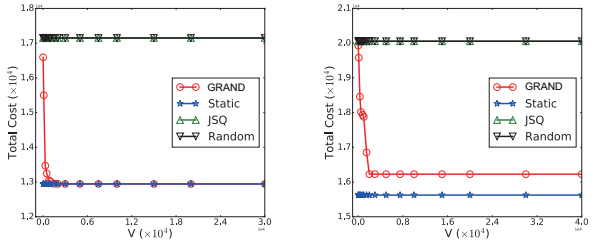
Third, the total cost in Jellyfish topology is significantly lower than the others. As illustrated in [22], compared to deterministic topologies, Jellyfish takes the advantages that all its paths are on average shorter[6] than in other topologies of the same scale.

Figure 5 (b) shows the varying of total queue backlog size with different values of V. We notice a linearly rising trend in the total queue backlog size for all four topologies. This is consistent with the $O(V)$ queue backlog size bound. Recall our previous analysis: larger V invokes most switches

6. Recall that α is set to be the average path length in our settings.

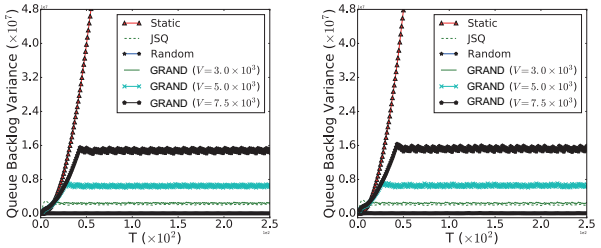


(a) Canonical 3-Tiered topology (b) Fat-tree topology

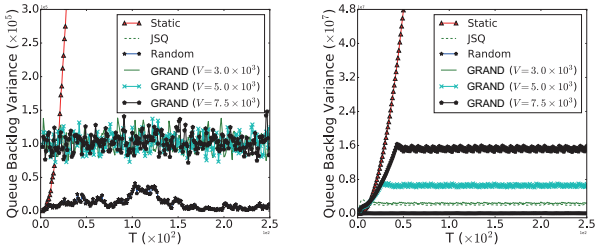


(c) Jellyfish topology (d) F10 topology

Fig. 7. Comparison of communication cost among four scheduling schemes under different topologies.



(a) Canonical 3-Tiered topology (b) Fat-tree topology



(c) Jellyfish topology (d) F10 topology

Fig. 8. Variance of queue backlog size comparison among four scheduling schemes under different topologies.

to spend more time uploading requests to control plane. However, requests on control plane will keep accumulating with fixed control plane capacity. Thus when V becomes sufficiently large, control plane will eventually hold most of requests in the system. This explains the increasing queue backlog size in Figure 5 (b).

Figure 6 (a) shows the total cost of *GRAND* in Fat-tree topology with other two request arrival processes. The curves of total cost with Poisson and Pareto almost overlap, with a gradual declined reduction to the minimum. Similarly, in Figure 6 (b), we can see the total queue backlog size in Fat-tree topology when we apply *GRAND* with request arrivals that follow Poisson and Pareto processes. The queue backlogs under both arrival processes remain overlapping all the time. Note that we do not show the curves here for the other three topologies, because curves are also overlapping as those in Figure 6(a) and Figure 6(b).

Insight: By tuning parameter V, under different request arrival processes and different topologies, *GRAND* consistently achieves a trade-off between $O(1/V)$ deviation from optimal communication cost and $O(V)$ increase in total queue backlog size.

5.3 Comparison with Other Association Schemes

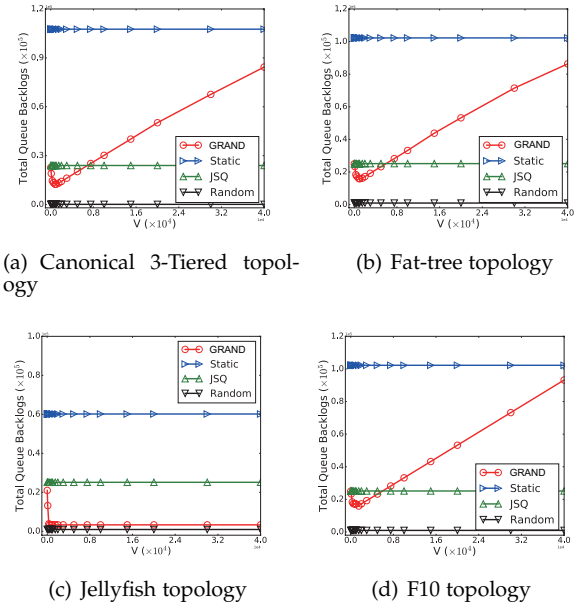
In this subsection, we consider the extreme case by setting common computational cost $\alpha = 2.0 \times 10^{28}$ for all switches. This means the cost of local processing requests are prohibitively high and in each time slot switches would only choose to upload requests to controllers. Such a setting emulates the scenarios where switches' computing resources are extremely scarce or local processing is not supported. As a result, *GRAND* degenerates into a dynamic switch-controller association algorithm.

We compare *GRAND*'s performance along with three other schemes: *Static*, *Random* and *JSQ (Join-the-Shorest-Queue)*. In *Static* scheme, each switch i chooses the controller j with minimum communication cost $M_{i,j}(t) = \min_{k\in\mathcal{C}} W_{i,k}$ and then fixes such an association in all time slots. In *Random* scheme, each switch is scheduled to pick up a controller uniformly randomly during each time slot. In *JSQ* scheme, each switch i is scheduled to pick the controller with smallest queue backlogs, among its available candidates. After choosing the target controller, each switch pushes all its available requests (those haven't been put into local processing queue yet) to the controller's queue.

Figure 7 presents a comparison among *Static*, *Random*, *JSQ*, and *GRAND* in terms of communication cost in different topologies. We make the following observations.

First, the communication cost under *Static* is the minimum among all schemes, which is consistent with its only goal of minimizing the overall communication cost. *GRAND* cuts down the communication cost with increasing V. Eventually, when V is sufficiently large (around 1.0×10^4 to 2.0×10^4), communication cost stops decreasing and remains unchanged. Both *Random* and *JSQ* exhibit much higher communication costs, compared to *GRAND* and *Static*. This is due to the blindness of *Random* and *JSQ* to the communication cost in decision making.

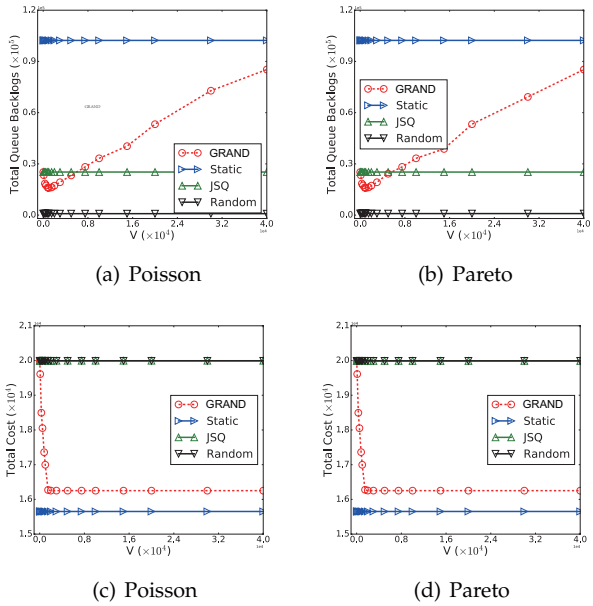
Besides, we also observe that: there is still a gap between the communication cost of *Static* and the minimum cost that *GRAND* can reach. Here is the reason behind. With the growth of V's value, *GRAND*'s scheduling behavior becomes increasingly similar to *Static*'s, which will lead to the reduction in cost and rise in queue backlogs. However, when the controllers' queue backlog size exceeds some threshold (about $2 \times V$ in our simulation), especially for those close to hot spots, the scheduling decisions by *GRAND* and *Static* would be different again. For *Static*, its decision would continue pursuing minimum communication cost. This would accumulate even more requests onto heavily loaded controllers. For *GRAND*, however, some switches would rather turn to controllers with higher cost, so as to avoid the long queueing delay on those with lower cost. The difference in scheduling decisions would continue until the queue backlog size falls below the threshold again. Thus we can regard the gap as the price that *GRAND* takes to stabilize the controllers' queue backlogs. The gap is much less significant in Jellyfish topology, because there are more



(a) Canonical 3-Tiered topology (b) Fat-tree topology

(c) Jellyfish topology (d) F10 topology

Fig. 9. Total queue backlog size comparison among four scheduling schemes under different topologies.



(a) Poisson (b) Pareto

(c) Poisson (d) Pareto

Fig. 10. Performance comparison among different schemes under Fat-tree topology, with Poisson and Pareto request arrivals, respectively.

switches (around 75% in Jellyfish, higher than others) with multiple choices of minimum-cost controllers in Jellyfish than other topologies. Consequently, the range of request arrival fluctuation around the threshold (which results in different scheduling decisions of *GRAND* and *Static*) would be smaller, leading to a smaller gap.

Insight: As V increases, *GRAND* achieves a comparably low communication cost among the four schemes. The gap between *GRAND* and *Static* is the price which has to be taken in order to guarantee the queueing stability.

Figure 8 presents a comparison among the four schemes in terms of the variance of queue backlog size under those four topologies, respectively. In fact, smaller queue backlog size variance indicates better capability of load balancing. The variance of *Static* grows exponentially with time, showing that *Static* is incompetent in load balancing. The reason is that *Static* greedily associates switches with their nearest controllers, ignoring different controllers' loads, especially those controllers close to hot spots. When it comes to *Random* and *JSQ*, the variance is significantly lower, which shows the two schemes' advantage in load balancing. As for *GRAND*, in deterministic topologies (Fat-tree, F10, and Canonical 3-Tiered), its variance is in between the other three: the variance increases at the beginning and then remains stable soon after only about hundreds of time slots. With larger V, *GRAND* exhibits higher variance of queue backlog size, *i.e.*, the load of controllers is more imbalanced. In contrast, in the random topology, *i.e.* Jellyfish, increased V in *GRAND* seems to have insignificant impact on the variance of queue backlogs. The reason is that Jellyfish has both smaller variance and average of shortest path lengths between switches and controllers than other topologies. Even though hot spots exist, the arriving requests would be spread more evenly to controllers in Jellyfish topology.

Insight: By tuning the value of parameter V, *GRAND* can guarantee the queueing stability at different levels and thus outweighs *Static* which incurs a highly imbalanced load. Although *JSQ* and *Random* takes the advantage of load-balancing the request loads among different queues, *GRAND* can also achieve comparably small queue backlog size with much lower communication.

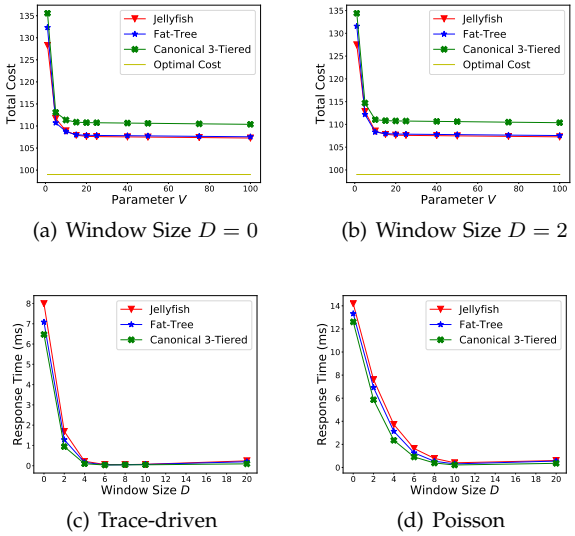
Figure 9 shows a comparison among the four schemes in terms of the total queue backlog size under those four topologies, respectively. The curves of *Static*, *JSQ*, and *Random* in Figure 9 are very consistent with our observation from Figure 8. Intuitively, fixed the service rate on each controller, the more balanced the loads on control plane are, the more controllers' service are utilized, and hence the smaller of the total queue backlog size. In Figure 8, the variance of *Static* is high while that of *Random* and *JSQ* are much lower, so the total queue backlog size of *Static* is large while that of *Random* and *JSQ* is small in Figure 9. When it comes to *GRAND*, for deterministic topologies, we observe a declining trend at the very beginning, then the curve of total queue backlog size rises linearly after reaching a valley at around 1.0×10^3. The explanation is as follows.

When the value of V is small, switches prefer controllers with shorter queues. [7] A switch's scheduling decision is independent of the others'. This will lead to arriving requests being intensively uploaded to just few controllers. In this way, controllers close to hot spots are more likely to get heavier loads, though *GRAND* would adjust the load spread in the next time slot.

As the value of V becomes larger, some switches would reach a tipping point and choose other controllers instead. As a result, this would mitigate the skewness of controllers' loads; *i.e.*, the loads at control plane would become more and more balanced. This explains the declination of the curve. With the continual increasing in V, switches' interest in minimizing communication cost becomes dominant. Hence, the skewness of controllers' loads is aggravated and turns to linear rise.

When *GRAND* is applied in Jellyfish, its curve is very different from other three topologies. The curve decreases at the beginning and then stays at a low level constantly. To explain the difference, we notice that the variation of queue backlog size is highly related to the variance of request arrivals among controllers. To measure the variance, given a controller j, we define j's *minimum-cost request arrival rate* as the summation of the arrival rates from those switches that j is one of those controllers with minimum cost. Thereby

7. Note that *JSQ* is just a special case of *GRAND* with $V = 0$.



(a) Window Size $D = 0$ (b) Window Size $D = 2$

(c) Trace-driven (d) Poisson

Fig. 11. Performance under different values of parameter V.

greater variance of controllers' minimum-cost request arrival rates will result in greater skewness of controllers' loads, when switches put less concern on queue backlogs and more on minimizing communication cost. In our setting, the variances of request arrivals among controllers are 1.62×10^5, 7.43×10^5, 7.43×10^5, and 5.25×10^5, for Jellyfish, Fat-tree, F10, and Canonical 3-Tiered, respectively. We find that for Fat-tree, F10, and Canonical 3-Tiered, they have more imbalanced numbers of hops between switches and controllers; as a result, with the existence of hot spots and aim to minimize communication costs, more requests accumulate on some particular controllers, inducing larger queue backlog sizes therein and hence the linearly rising curve. For Jellyfish, however, because incoming requests are spread more evenly among controllers, increased V has insignificant impact on the skewness of controllers' queue backlogs; hence its curve stays at a low level and is quite different from the others.

Insight: When V is small, *GRAND*'s behavior is similar to *JSQ*. When V is large, the queue backlog size it incurs increases but never exceeds that of *Static*, because *GRAND* always guarantee queueing stability while *Static* doesn't. The actual variation of *GRAND*'s queue backlog size depends on the characteristics of underlying topologies. The more balanced switches and controllers are connected, the less significant queue backlog skewness there will be.

In addition, we also conduct simulations under two kinds of request arrival processes, *i.e.*, *Poisson* and *Pareto* processes, which are widely adopted in traffic analysis. For Poisson process, we set its arrival rate as 5.88; while for Pareto process, we set its shape parameter as 2 and its scale parameter as 2.94. We only show the simulation results under Fat-tree topology, because the simulation results in other three topologies are qualitatively similar. From Figure 10, we find that the scheduling policies perform qualitatively consistent under different arrival processes.

Summary: Among the four schemes, **Static** is on the one end of performance spectrum: it minimizes communication cost while incurring extremely large queue backlogs; both **Random** and **JSQ** are on the other end of performance spectrum: they maintain the total queue backlog at a low level while incurring much larger communication costs. In contrast, our *GRAND* scheme achieves a trade-off between

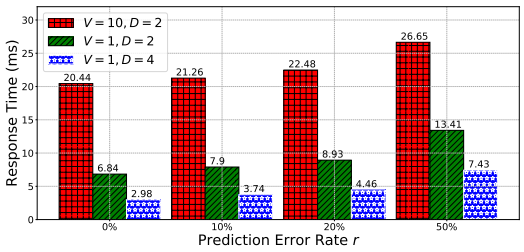


Fig. 12. Average response time vs. prediction error rate under Fat Tree topology with different parameter V and prediction window size D.

minimization of communication costs and minimization of queue backlogs. Through a tunable parameter V, we can achieve different degrees of balance between cost minimization and latency (queue backlog) minimization.

5.4 Evaluation of *PRAND*

5.4.1 Evaluation with Perfect Prediction

To explore the benefits of predictive scheduling, we first consider the case where switches have perfect information about future request arrivals, *i.e.*, with $r = 0$. Considering the similarities between curves under Fat-Tree and F10 topologies in previous figures. We omit the results regarding F10 in the following.

Time-average total costs vs. V: We compare the total costs incurred by POSCAD with various parameter V among different topologies in Fig. 11. In particular, Fig. 11 (a) and (b) present the results with a prediction window size $D = 0$ (*non-prediction*) and $D = 2$, respectively. From both plots, we find that as V increases from 1 to 20, all the curves fall rapidly under different topologies. When V continues increasing after 25, the total costs stops decreasing and remains stable thereafter, with a merely 10% gap to the optimal total costs (denoted by the solid horizontal line). Comparing (a) and (b), we observe that the incurred total costs are almost the same even when the window size D is increased. Based on the observations above, we have the following insight.

Insight: As the parameter V increases, POSCAD reduces the time-average total costs and achieves the optimal system cost asymptotically under different topologies. Note that the gap between the eventual total costs and the optimal cost is the price paid for balancing and stabilizing queue backlogs in the system. In the mean time, compared to the non-prediction case, POSCAD performs predictive scheduling without increasing the total costs in the system.

Average response time vs. prediction window size: Next, we switch to the impact of different prediction window sizes on the average request response time. Figure 11 (c) and (d) present the results drawn from simulations with trace-driven and Poisson arrival process, respectively. Figure 11 (c) shows the curves under the trace-driven setting. We observe a sharp fall of the average request response time from around $14ms$ to less than $1ms$, with the window size rising from 0 (non-prediction) to 6 under different topologies. As the window size continues growing, the reduction stops and the average response time remains stable ($0.45ms$). The results are similar in Fig. 11 (d).

Insight: Figure 11 shows the benefits of predictive scheduling. By exploiting predictive information and preserving future requests, POSCAD effectively reduces the average response time without inducing extra system cost.

Moreover, with mild-value of future information, POSCAD achieves a near-zero average response time.

5.4.2 Evaluation with Imperfect Prediction

Prediction errors are inevitable in practice. Hence, we are also interested in the robustness of POSCAD in face of such errors. In the following, we vary error rate from 0% to 50%.

Average response time vs. prediction error rate: Figure 12 shows the average response times with prediction error rate ranging from 0% (perfect prediction) to 50%, under various settings of parameter V and window size D.

First, we focus on the red mesh bars, where parameter $V = 10$ and prediction window size $D = 2$. We observe a growth ($6.21ms$) of the average response time as the prediction error rate rises from 0% to 50%. This suggests that prediction errors lead to an increased request response time. Next, with a fixed prediction error rate $r = 10\%$ and a prediction window size $D = 2$, we compare the response times with different Vs. Figure 12 shows that the response time ($21.26ms$) of the red mesh bar ($V = 10$) is higher than that ($7.9ms$) of the green stripped bar ($V = 1$). The result suggests that we can reduce the average response time by reducing parameter V. The reason is that a smaller V leads to a smaller total queue backlog size which, by *Little's theorem* [15], implies a lower average response time. Nonetheless, reducing V will also increase total costs. Last but not least, by fixing the prediction error rate ($r = 10\%$ as an example), we see the blue bar ($D = 4$) is lower than the green bar ($D = 2$), implying that more future information is helpful to reduce the response time by up to 50% even under inaccurate prediction.

Insight: POSCAD is robust against prediction errors. By choosing a smaller V and enlarging the prediction window size (with more future information), POSCAD effectively cancels the negative effect of prediction errors and shortens average request response time.

6 Conclusion

In this paper, we studied the problem of joint dynamic switch-controller association and dynamic control devolution, and investigate the benefits of predictive scheduling for SDN systems. We proposed *GRAND*, an efficient joint control scheme that solves the problem through a series of online decision making in a distributed manner. Our theoretical analysis showed that *GRAND* achieves a near-optimal total cost with a tunable trade-off for queueing stability guarantee. By integrating *GRAND* with predictive scheduling, we proposed *PRAND*, an effective scheme that goes beyond *GRAND* by proactively pre-serving future requests with surplus system capacities. Extensive simulation results verified the theoretical results and the effectiveness of both proposed schemes.

Appendix A
Problem Transformation by Opportunistically Minimizing an Expectation

By minimizing the upper bound of the drift-plus-penalty expression, [8]

$$\Delta_V(\mathbf{Q}(t)) \triangleq \Delta(\mathbf{Q}(t)) + V \cdot E\left\{f_{\mathbf{X}}(t) + g_{\mathbf{X}}(t)|\mathbf{Q}(t)\right\}, \tag{34}$$

8. Recall that V is a positive constant parameter that weighs the penalty brought by communication costs between switches and controllers ($f_{\mathbf{X}}(t)$) and the local computation costs on switches due to control devolution ($g_{\mathbf{X}}(t)$).

the time average of communication cost can be minimized while stabilizing the network of request queues. We denote the objective function to be solved in time slot t by $J_t(\mathbf{X})$, *i.e.*,

$$\begin{aligned} J_t(\mathbf{X}) \triangleq \sum_{i\in\mathcal{S}} \Big[V \cdot P_i(t) + Q_i^s(t)\Big] A_i(t) + \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{C}} \Big[VM_{i,j}(t) \\ + Q_j^c(t) - VP_i(t) - Q_i^s(t)\Big]\mathbf{X}_{i,j}A_i(t) \end{aligned} \tag{35}$$

and its optimal solution by $\mathbf{X}^* \in \mathcal{A}$.

Therefore, for any other scheduling decision $\mathbf{X} \in \mathcal{A}$ made during time slot t, we have

$$J_t(\mathbf{X}) \geq J_t(\mathbf{X}^*). \tag{36}$$

Taking the conditional expectation on both sides conditional on $\mathbf{Q}^c(t)$, we have

$$E\left[J_t(\mathbf{X}) \,|\, \mathbf{Q}^c(t)\right] \geq E\left[J_t(\mathbf{X}^*) \,|\, \mathbf{Q}^c(t)\right], \tag{37}$$

for any $\mathbf{X} \in \mathcal{A}$. In such a way, instead of directly solving the long-term stochastic optimization problem, we can opportunistically choose a feasible association that minimizes (35) during each time slot.

Appendix B
Proof of the $[O(V), O(1/V)]$ Trade-off

We assume there is an S-only algorithm achieves optimal time-average total costs (infimum) $\bar{f}^* + \gamma\bar{g}^*$ with action $\mathbf{X}^*(t)$ and $\mathbf{Y}^*(t)$, $t = \{0, 1, 2, \cdots\}$ [20]. With the existence of randomized scheduling scheme which ensures that the expectation of total system processing capacity to be strictly greater than that of the total request arrival rate, *i.e.*, there exists some $\epsilon \geq 0$ such that

$$\mathbb{E}\left\{\sum_{i\in\mathcal{S}} X_i^*(t)\big|\mathbf{Q}(t)\right\} + \epsilon \leq \mathbb{E}\left\{\sum_{j\in\mathcal{C}} B_j^c(t) + \sum_{i\in\mathcal{S}} B_i^s(t)\big|\mathbf{Q}(t)\right\}. \tag{38}$$

Next, we denote $\mathbf{X}'(\mathbf{t})$, $\mathbf{Y}'(\mathbf{t})$ as the decisions over time, and $f'(t)$, $g'(t)$ as the corresponding costs given by POSCAD algorithm. Accordingly, the one-slot drift-plus-penalty is

$$\begin{aligned} \Delta_V(\mathbf{Q}(t)) \leq\, & B + C(\mathbf{Q}(t)) + \\ & \mathbb{E}\left\{\sum_{j\in\mathcal{C}} Q_j^c(t)\left(\sum_{i\in\mathcal{S}} X_i^*(t)Y_{i,j}^*(t)\right)\Big|\mathbf{Q}(t)\right\} \\ & + \alpha\mathbb{E}\left\{\sum_{i\in\mathcal{S}} Q_i^s(t)\left(1 - \sum_{j\in\mathcal{C}} Y_{i,j}^*(t)\right)X_i^*(t)\Big|\mathbf{Q}(t)\right\} \\ & - \beta\mathbb{E}\left\{\sum_{i\in\mathcal{S}} Q_i^p(t)X_i^*(t)\Big|\mathbf{Q}(t)\right\} + \\ & V\mathbb{E}\left\{f^*(t) + \gamma g^*(t)\big|\mathbf{Q}(t)\right\}. \end{aligned} \tag{39}$$

Using (38) and the definition in (25), we obtain

$$\begin{aligned} & \mathbb{E}\left\{L(t+1) - L(t)\big|\mathbf{Q}(t)\right\} + V\mathbb{E}\left\{f'(t) + \gamma g'(t)\big|\mathbf{Q}(t)\right\} \\ & \leq B + V\mathbb{E}\left\{f^*(t) + \gamma g^*(t)\big|\mathbf{Q}(t)\right\} - \epsilon\mathbb{E}\left\{h(t)\big|\mathbf{Q}(t)\right\}, \end{aligned} \tag{40}$$

where $L(t) \triangleq L(\mathbf{Q}(t))$. Then taking expectation over both sides and summing over $t \in \{0, 1, \cdots, T-1\}$, we have

$$\mathbb{E}\left\{L(T-1)\right\} - \mathbb{E}\left\{L(0)\right\} + V\mathbb{E}\left\{\sum_{t=0}^{T-1}\left(f'(t) + \gamma g'(t)\right)\right\}$$

$$\leq BT + V\mathbb{E}\left\{\sum_{t=0}^{T-1}\left(f^*(t)+\gamma g^*(t)\right)\right\} - \epsilon\cdot\mathbb{E}\left\{\sum_{t=0}^{T-1}h(t)\right\}. \tag{41}$$

We are ready to prove the $[O(V), O(1/V)]$ trade-off.

1) First, dividing both sides of (41) by VT, rearranging items, and then canceling non-positive quantities, we obtain

$$\begin{aligned}&\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{\left(f'(t)+\gamma g'(t)\right)\right\}\\ &\leq \frac{B}{V}+\frac{\mathbb{E}\{L(0)\}}{VT}+\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{\left(f^*(t)+\gamma g^*(t)\right)\right\}.\end{aligned} \tag{42}$$

Taking lim-sup as $T\to\infty$ in (42) yields

$$\begin{aligned}&\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{\left(f'(t)+\gamma g'(t)\right)\right\}\\ &\qquad\leq \limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{\left(f^*(t)+\gamma g^*(t)\right)\right\}+\frac{B}{V}.\end{aligned} \tag{43}$$

2) Similarly, by dividing both sides of (41) by ϵT, we have

$$\begin{aligned}&\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{h(t)\right\}\leq\\ &\frac{B}{\epsilon}+\frac{\mathbb{E}\{L(0)\}}{\epsilon T}+\frac{V\sum_{t=0}^{T-1}\mathbb{E}\left\{\left(f^*(t)+\gamma g^*(t)\right)\right\}}{\epsilon T}.\end{aligned} \tag{44}$$

Since $L(0)$ is constant, by taking lim-sup as $T\to\infty$, we obtain

$$\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{h(t)\right\}\leq\frac{V}{\epsilon}\left(\bar{f}^*+\gamma\bar{g}^*\right)+\frac{B}{\epsilon}<\infty. \tag{45}$$

□

APPENDIX C
MOTIVATING EXAMPLE

Figure 13 presents an example that compares the cases with and without predictive scheduling, where Figure 13(a) shows the system state at the beginning of time slot t. Wherein there are two switches potentially connected to one controller through dashed arrows. All requests are assumed homogeneous and can be handled by both switches and the controller. Upon new requests' arrival, each switch either associates with the controller through a solid arrow and uploads requests, or stores them in its own queue for local processing. In each time slot, a switch processes at most 1 request and the controller processes at most 2 requests, both in a FIFO manner. The objective is to minimize the average request response time. (a) shows the system state at the beginning of time slot t, where one new request arrives at switch 1, one is already stored in switch 1's local queue, and no requests arrive at switch 2. The future request arrivals (marked with stripped colors) in time slot $(t+1)$ are also visible to switches. (b) and (c) present the scheduling process that handles current requests only, whereas (d) and (e) exhibit the case with predictive scheduling.

First, we consider the case without predictive scheduling and handle only the arriving requests, as shown in Figure 13 (b) and (c). For switch 1, it chooses to make an association

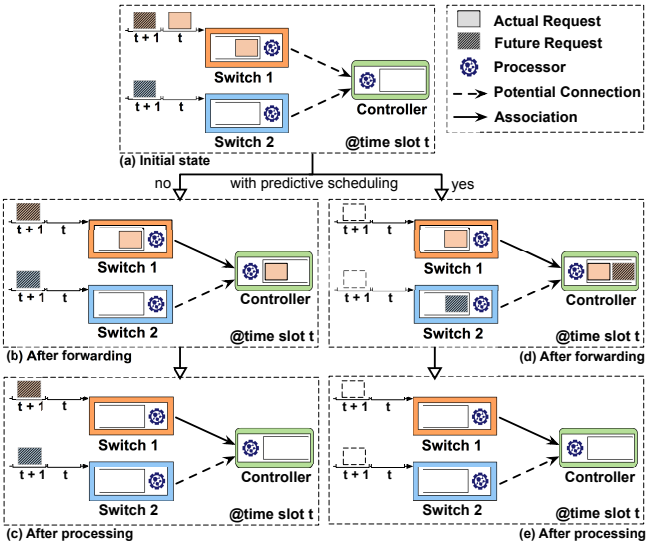


Fig. 13. An example that shows the benefits of predictive scheduling.

with the controller and uploads the new request, considering that its local queue has already buffered one request. Meanwhile, switch 2 takes no action since no requests arrive at present. After the only new request is forwarded, as shown in Figure 13 (b), the controller and switches process the requests in their respective queues. Figure 13 (c) shows the system state at the end of time slot t, where the two requests in time slot $(t+1)$ are still left unprocessed.

Next, we focus on the case with predictive scheduling, as shown in Figure 13 (d) and (e). Switch 1 associates with the controller and uploads the new request that arrives at present. Considering that the controller has a service capacity of two requests per time slot, it pre-admits the request which will arrive in time $(t+1)$, then uploads the request to the controller. Similarly, switch 2 pre-admits the future request in time $(t+1)$ and stores the request in its local queue. Figure 13 (d) shows the system state after the (pre-)admission of requests. Then the controller and switches consume the requests from their queues. Figure 13 (e) shows that with predictive scheduling, all the requests in time t and $(t+1)$ are completed by the end of time t. Consequently, both future requests will receive instant response upon their arrivals in time $(t+1)$.

The above example shows that predictive scheduling can effectively reduce the request response time by taking advantages of utilizing the present surplus processor capacities. In the next subsection, we extend our previous model and formulation in Section 2 with predictive scheduling.

REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of ACM SIGCOMM*, 2008.

[2] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of ACM IMC*, 2010.

[3] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[4] J. Broughton, "Netflix adds download functionality," https://technology.ihs.com/586280/netflix-adds-download-support, 2016.

[5] N. Chen, J. Comden, Z. Liu, A. Gandhi, and A. Wierman, "Using predictions in online optimization: Looking forward with an eye on the past," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1.

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proceedings of ACM SIGCOMM*, 2011.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *Proceedings of ACM HotSDN*, 2013.

[8] J. Dunn, "Introducing fblearner flow: Facebook's ai backbone," https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/, 2016.

[9] A. Filali, A. Kobbane, M. Elmachkour, and S. Cherkaoui, "Sdn controller assignment and load balancing with minimum quota of processing capacity," in *Proceedings of IEEE ICC*, 2018.

[10] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of ACM HotSDN*, 2012.

[11] L. Huang, S. Zhang, M. Chen, X. Liu, L. Huang, S. Zhang, M. Chen, and X. Liu, "When backpressure meets predictive scheduling," *IEEE/ACM Transactions on Networking (TON)*, vol. 24, no. 4, pp. 2237–2250, 2016.

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *Proceedings of OSDI*, 2010.

[13] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson, "Pratyaastha: An efficient elastic distributed sdn control plane," in *Proceedings of ACM HotSDN*, 2014.

[14] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proceedings of ACM HotSDN*, 2012.

[15] J. D. Little, "A proof for the queuing formula," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[16] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson, "F10: A fault-tolerant engineered network." in *USENIX NSDI*, 2013.

[17] X. Lyu, C. Ren, W. Ni, H. Tian, R. P. Liu, and Y. J. Guo, "Multi-timescale decentralized online orchestration of software-defined networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2716–2730, 2018.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[19] S. Nanda, F. Zafari, C. DeCusatis, E. Wedaa, and B. Yang, "Predicting network attack patterns in sdn using machine learning approach," in *Proceedings of IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016.

[20] M. J. Neely, "Stochastic network optimization with application to communication and queueing systems," *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1–211, 2010.

[21] I. S. Petrov, "Mathematical model for predicting forwarding rule counter values in sdn," in *Proceedings of IEEE Conference of Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2018.

[22] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers, randomly." in *USENIX NSDI*, 2012.

[23] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, 2010.

[24] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proceedings of ACM Hot-ICE*, 2012.

[25] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *Proceedings of IEEE INFOCOM*, 2016.

[26] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic sdn controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017.

[27] J. Yang, X. Yang, Z. Zhou, X. Wu, T. Benson, and C. Hu, "Focus: Function offloading from a controller to utilize switch power," in *Proceedings of IEEE NFV-SDN*.

[28] H. Yu, M. H. Cheung, L. Huang, and J. Huang, "Power-delay tradeoff with predictive scheduling in integrated cellular and wi-fi networks," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 4, pp. 735–742, 2016.

[29] S. Zhang, L. Huang, M. Chen, and X. Liu, "Proactive serving decreases user delay exponentially," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 2, pp. 39–41, 2015.

[30] K. Zheng, L. Wang, B. Yang, Y. Sun, Y. Zhang, and S. Uhlig, "Lazyctrl: Scalable network control for cloud data centers," in *Proceedings of IEEE 35th International Conference on Distributed Computing Systems (ICDCS)*, 2015.