Efficient Event Scheduling of Network Update

Ting Qu, Deke Guo, Jie Wu, Fellow, IEEE, Xiaolei Zhou, Xin Lu, Zhong Liu

Abstract—Changes in network state are a common source of instability in networks. An update event typically involves multiple flows that compete for network resources at the cost of rescheduling and migrating some existing flows. Previous network updating schemes tackle such flows independently, rather than as the entity of an update event. They only optimize the flow-level metrics for the flows involved in an update event. In this paper, we present an event-level abstraction of network update that groups flows of an update event and schedules them together to minimize the event completion time (ECT). We then study the scheduling problem of multiple update events for achieving high scheduling efficiency and preserving fairness. The designed least migration traffic first (LMTF) method schedules all update events in the FIFO order, but it avoids head-of-line blocking by randomly fine-tuning the queue order of some events. It can considerably reduce the update cost, the average, and tail ECTs of update events. In addition, we design a general parallel-LMTF (P-LMTF) method to guarantee fairness and further improve scheduling efficiency among update events. This improves the LMTF method by opportunistically updating multiple events simultaneously. The comprehensive evaluation results indicate that the average ECT of our approach is up to $10\times$ faster than the flow-level scheduling method for network update events, and its tail ECT is up to $6\times$ faster. Our P-LMTF method incurs a 75% reduction in the average ECT compared with FIFO when the network utilization exceeds 70%, and it achieves a 42% reduction in tail ECT.

Index Terms—Network update, fairness, efficiency.

1 Introduction

Due to updating issues, such as upgrades of switches and VM migrations [1], triggered by operators, applications, and network devices, network condition consistently undergoes changes. When upgrading a switch, all flows initially passing through it should be rerouted along other parts of the network to ensure the normal execution of network applications. There are two general consequences of such update issues: change of the network topology and change of the traffic matrix. These updating issues are common sources of instability in networks. Therefore, each network update should be planned well in advance and should be tackled by designing effective and efficient update schemes.

For a network update event under the initial network configuration, the update plan usually includes the network topology and traffic distribution to the desired final network state. The update process usually undergoes many intermediate network states, and it may result in serious traffic congestion and other issues. For this reason, previous updating schemes focus on realizing the correct transition from an initial network state to a final network state. Previous approaches can be roughly divided into two categories. One is the consistent update [2], in which a packet/flow traverses the network obeying either the old or new network configurations. The two-phase update method and its variants [3–5] fall into such a category. The other is the congestion-free update, which creates an update plan for any update event in advance. This plan contains a series of intermediate states, but the transition across adjacent states is lossless. SWAN [6], zUpdate [1] and their variants [7, 8] fall into this category.

An update event typically involves a collection of new flows, such as a set of flows caused by VM migration. The update event is not finished until all flows in it have been scheduled. However, existing updating schemes treat each flow in such a collection in isolation rather than organizing involved flows of an update event as an entity. Updating schemes that attempt to optimize flow-level metrics will fail to optimize event-level metrics like the event completion time (ECT) of an update event or the average and tail ECTs of multiple update events. In fact, the abstraction of a per-flow update cannot capture event-level or inter-event level requirements in a collection of update events. Previous updating schemes fail to provide a frame to represent event-level update semantics. For example, some flows of an update event may be blocked because the network resources they required are occupied by the heavy flows of other update events. This would lead to high average and tail ECTs.

In this paper, we define an event-level abstraction of network update and schedule flows together contained in an update event. If the network cannot serve any flow in such an update event, a few existing flows are locally migrated to other appropriate paths (if they exist) to satisfy the bandwidth requirement of the new flows in the update event. The event-level abstraction can decrease the ECT by cooperatively allocating network resource for an update event. This considerably speeds the network update process and proves very important for network management

- *T. Qu, D. Guo, X. Lu and Z. Liu are with College of Systems Engineering, National University of Defense Technology, Changsha, Hunan, 410073, P. R. China. D. Guo is also with School of Computer Science and Technology, College of Intelligence and Computing, Tianjin University, Tianjin, 300350, China. E-mail: {quting14, dekeguo, luxin, liuzhong}@nudt.edu.cn.*
- *X. Zhou is with the Sixty-third Research Institute, National University of Defense Technology, Nanjing 210007, China. E-mail: xlzhou@126.com.*
- *J. Wu is with the Department of Computer and Information Sciences, Temple University, Philadelpha, USA,19122. E-mail: wujie@163.com.*
- *D. Guo is corresponding author.*

[9]. We define the update cost of an update event as the total amount of occupied bandwidth of migrated existing flows caused by an update event. However, it is a NP-complete problem to identify which existing flows should be migrated, guarantee a minimal update cost and satisfy the requirements of an update event. Accordingly, we propose an efficient method to calculate the set of migrated flows to approximate the optimal solution.

Operators, applications, and network devices create multiple update events in a shared network. For example, a switch upgrade can be treated as an update event which contains all flows passed through it. VM migration also can be an update event where incoming flows will be rerouted to the new location of VMs. Within one time slot, there may exist multiple such update events, which exhibit wide variations in flow number, the size of individual flows, and in total size. They may lead to different bandwidth requirements for the migrated traffic. Simple scheduling mechanisms like FIFO [10] remain inapplicable to this type of inter-event scheduling problem. Using a scheduling method like FIFO usually incurs the serious head-of-line blocking problem. That is, the head-event may be heavy, and it requires more network resources which are available after a longer period of time because of the occupation by other applications. Therefore, many light-weight update events, which arrive later, would wait a long time in the update queue. This would increase the average ECT and tail ECT of a set of update events.

In this paper, we investigate the scheduling problem of multiple update events, and we focus on two different objectives: 1) speeding up the network-update process by decreasing the average and tail ECTs and 2) preserving update fairness. We propose the least migration traffic first (LMTF) method, which schedules update events based on their arrival order, but dynamically fine-tune the execution order while facing the heavy-weight update events. A direct method to arrange execution order is to dynamically compute the update costs for all update events in the queue and execute the update event with the lowest cost first. However, this would cause non-trivial computation and time overhead. In contrast, LMTF compares the head-event with a few update events randomly selected from the queue, and the executes the one with the least cost first. This policy ensures that not all small update events are blocked behind the heavy-weight update events as the smaller events at least have a chance to be executed earlier in each round. Additionally, LMTF considerably simplifies and accelerates the decision-making process.

Although LMTF can effectively decrease the average and tail ECTs, it delays some heavy events and affects the FIFO fairness of queued events. To further improve update efficiency and fairness, we design a more general method called P-LMTF, which introduces opportunistic updating. After getting the new head-event like LMTF method, P-LMTF checks whether the other chosen update events can be executed alongside the head-event. A heavy-weight update event that arrives earlier but is delayed by LMTF still has a chance to be executed in time. It will be scheduled in a timely manner during the process of opportunistic updating. This opportunistic updating policy further increases scheduling efficiency and fairness. The major contributions of this paper are summarized as follows:

- We give an event-level abstraction of network update and minimize the migration cost of an update event to guarantee fairness and efficiency in network update.
- We formulate the problem of updating multiple events as an optimization problem, and we devise the LMTF and P-LMTF methods to reduce the average and tail ECTs effectively.
- We evaluate the scheduling performance via trace-driven simulations. The results indicate that the average and tail ECTs of our approaches is up to $10\times$ faster than the flow-level scheduling method for network update events.

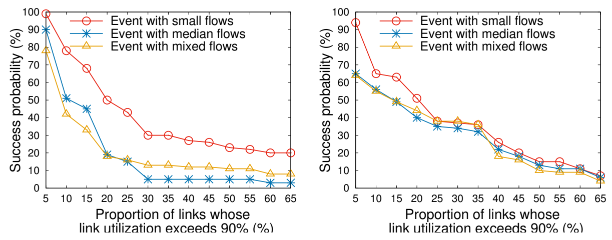
The remainder of this paper is organized as follows. In §2, we discuss related work and motivation. In §3, we present an event-level abstraction of network update. In §4, we discuss multiple events scheduling model. In §5, we design multiple events scheduling methods for achieving both fairness and efficiency. We report the evaluation methodology and results in §6. Finally, we discuss future work in §7 and conclude this paper in §8.

2 Overview

2.1 Related Work

Consistent update. Reitblatt et al. present the concept of per-packet/per-flow consistent network update [2]. This means that a packet/flow traverses the network according to either the old network configuration or the new configuration. In addition, they propose a two-phase method to guarantee a consistent update. Katta et al. try to reduce the overhead of keeping both the new and old configurations at related switches at the cost of increasing the overall update duration [3]. Dionysus finds a consistent migration sequence by searching through a dependency graph of possible migrated steps [9]. Moreover, timed-based update methods [4, 5] aim to achieve consistent update using an accurate time to trigger a network update at each phase. Such methods effectively reduce the duration of flow rules on the switches and the update duration. In addition, the authors in [8] propose an effective method to decide if a consistent update is possible. Foerster et al. study the power of random choices in consistent network update for updating forwarding rules in a loop-free manner and migrating flows without congestion [11]. They propose an effective algorithm to migrate two-splittable flows and an alternative when no consistent migration exists.

Congestion-free update. zUpdate [1], SWAN [6], and Caesar [12] try to make a congestion-free update plan in advance for any update event. This plan contains a sequence of intermediate states – from an initial network state to the final network state – so that the transition across any pair of adjacent states is lossless. However, this strategy has several drawbacks. First, achieving such an update plan means solving a series of LPs. The time complexity is very high for large-scale network update events. Second, to guarantee a series of congestion-free transitions, a portion ($10\% - 15\%$) of the link capacity has to be obligated in advance, which decreases the network utilization [6]. Cupid



(a) Yahoo! Trace (b) Random Trace

Fig. 1. The success probability of accommodating an update event without migrating other flows.

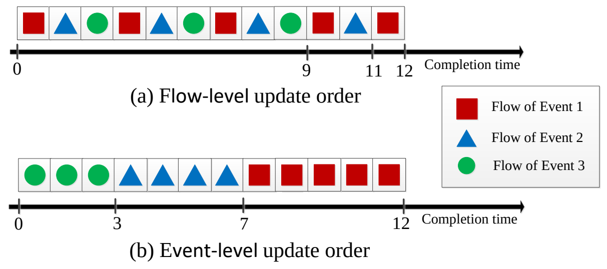


Fig. 2. The update orders of flows under flow-level and event-level methods.

locally constructs a dependency graph among key-nodes so that congested links to avoid high overhead among updates and congestion-free data plane update can be guaranteed. This work also proposes a heuristic algorithm to update flow tables consistently and effectively [13]. Zheng et al. apply timed-based update thoughts [4, 5, 14] to achieve congestion-free update [15, 16].

Accelerate the update process. B4 speeds up the update process using custom hardware [17, 18]. The work in [7] speeds the update process at the cost of incurring a given level of congestion. The basic idea is to minimize the transient congestion during the network update and achieve a better tradeoff between the update speed and the transient congestion.

For any update event, the aforementioned updating schemes focus on optimizing flow-level metrics; they do not perform well in optimizing event-level metrics. Moreover, they cannot capture the inter-event requirements in a queue of update events. In summary, previous updating schemes do not provide an abstraction to represent event-level update semantics. Event-level abstract, expressing the semantics of a collection of flows caused by an update event, is different from the network abstraction Coflow [19], which is a collection of flows between two groups of machines with associated semantics and a collective objective.

2.2 Motivation

An update event is completed only when all flows of the update event are scheduled. The event completion time (ECT) starts at the time an update event is pushed in the scheduling queue and ends at the time the update event is performed successfully. That is, the ECT also means the time duration for the new network configuration be in effect for that update event which includes the completion time of the migrated existing flows. Therefore, given an update event, ECT partly depends on the last scheduled flow in an update event; hence, the group-based scheduling can help to get a better performance when tacking all involved flows of multiple update events. We assume that these incoming flows are divided into several groups based on update events.

All flows in the same group have to compete for network resources along desired routing paths. However, the desired paths for such an update event may not offer sufficient residual bandwidth resources. In this scenario, the desired paths will exhibit congestion once the flows of an update event are inserted into the network, especially when the link utilization is high. As shown in Fig. 1, the success probabilities of updating events with small flows, middle flows and big flows decrease along with the increase of link utilization under the trace from the Yahoo! data center [20] and random trace with a distribution of traffic [21] in an 8-pods Fat-Tree data center network.

For this reason, it is necessary to first check if all links along the desired path could offer sufficient link bandwidth when tackling each flow of an update event. If they do not, then the update event needs to be carefully addressed. A straightforward method is to assign priorities to all flows in the network [22]. Existing flows with lower priorities will be removed if they block new flows with higher priorities. This policy incurs a large volume of burst traffic due to the retransmission of all removed flows. What is worse, determining which flows should be removed in an update event is an NP-hard problem, as proved in [8].

Another method is to re-route all the existing flows to supply the flows of the update event with sufficient network resources. This policy aims to achieve a better network performance, in terms of load balance and link utilization, when the network topology or traffic changes. Solving a series of linear programming (LP) problems is time-consuming. Moreover, globally re-routing all existing flows will lead to serious network-scale traffic migration.

Despite such considerations, the network update problem still lacks efficient solutions. In this paper, we present a novel strategy to locally adjust a few existing flows on the congested links of the desired path to accommodate each new flow in an update event. To update a new flow, we first check if there exists a feasible path with sufficient residual network resource, which can meet the new flow's requirement. If not, we locally migrate a few existing flows on the bottleneck links to accommodate the new flow. Because of the aggregate communication pattern, the number of existing flows in the bottleneck links can be very large [21]. Therefore, such migration of existing flows may incur considerable extra overhead and significantly disrupt other network applications as explained in our previous work [23]. Furthermore, reconfiguration of those existing flows also hurts the quality of service (QoS) and the system stability [24]. Paris et al. optimize the gap of the flow reconfiguration and the price paid for per unit of a routed flow [25]. In this paper, we do not consider price but prefer to find a local re-routing solution to minimize the amount of the migrated existing traffic, so as to release sufficient network resource to accommodate new flows of a given update event.

TABLE 1
Symbols and notations.

V	The set of all switches
E	The links between all switches
G	The direct network graph $G = (V, E)$
$e_{i,j}$	The link connecting node i and node j, where $i, j \in V$
$c_{i,j}$	The residual bandwidth of the link $e_{i,j}$
C	The matrix of available bandwidth of each link
C_{f_i}	The matrix of bandwidth requirements of transforming the flow f_i and migrating several existing flows on the congested links.
C_{U_i}	The matrix of bandwidth requirements of updating an event and migrating several existing flows on the congested links.
D	The network diameter
F	All existing flows in the network
f	A flow in the network
d^f	The bandwidth requirement of a flow f
$d^{f_{i,j}}$	The occupied bandwidth by a flow f on the link $e_{i,j}$
p	A path taken by a flow f. $p \in P(f)$
$E^c_{f_a}$	The set of congestion links caused by the flow f_a
F_A	The set of existing flows which pass any congested links on the path p allocated for the flow f_a
F_a	The set of existing flows which will be migrated caused by the new flow f_a, where $F_a \subset F_A$
$Sum(F_a)$	The sum of bandwidth requirements of flows in F_a.
U	An update event is defined as $U = \{f_1, f_2, ..., f_w\}$
$Cost(U)$	The update cost of the update event U

Consider a series of flows caused by three update events. We may schedule such flows independently, as shown in Fig. 2(a). Alternatively, we may regard the flows of an update event as a collection and schedule them in a certain order, as shown in Fig. 2(b). The average ECT of the three events is $(3+7+12)/3=22/3$ under the event-level scheduling manner, which is lower than $(9+11+12)/3=32/3$ under the flow-level scheduling manner. Since we assume that each flow of an update event can get enough bandwidth and the time for migration of existing background traffic to insert each flow of an update event is the same, the tail ECTs of these three update events under these two scheduling manners are the same. In real networks, the migration of background traffic for each flow of an update event usually varies from each other. Therefore, the tail ECTs of these three update events will change as well.

3 Event-level Abstraction of Network Update

In this section, we start with the event-level abstraction of a network update and discuss the cost optimization of an update event. Accordingly, we characterize the inter-event scheduling problem of multiple update events.

3.1 Abstraction of event-level network update

Before characterizing our model, we first report all used notations and symbols in Table 1. The network is defined as a graph $G=(V, E)$, where V and E denote a set of switches and a set of links connecting those switches. Let $c_{i,j}$ be the residual bandwidth of link $e_{i,j}\in E$, while D denotes the network diameter. In addition, F refers to all flows in the network. For any flow $f\in F$, its bandwidth requirement is defined as d^f. Flow f is routed along a selected path p from set $P(f)$, which denotes feasible shortest paths for that flow. For each link $e_{i,j}$ in the selected path p, $d^{f_{i,j}}$ denotes the consumed bandwidth by a flow f on link $e_{i,j}$. The network is congestion-free if the following constraints are satisfied:

- $\forall e_{i,j}\in E, \;\; c_{i,j}\geq 0.$

The aforementioned constraints ensure that each flow $f\in F$ is unsplit and is forwarded along a certain path p. The last condition indicates that each link in the network is congestion-free after the network accommodates the entire flow set F.

Definition 1 (Migration of existing flows for a new flow). When a new flow f_a from an update event is inserted into the network, we first try to find a feasible path p with sufficient residual bandwidth on each link to accommodate it. If the flow f_a would cause congestions on some links of the path p. Then, we define the set of congested links as $E^c_{f_a}$, i.e.:

- $\forall e_{i,j}\in p, \text{ if } d^{f_{i,j}}>c_{i,j}, \text{ then } e_{i,j}\in E^c_{f_a},$

where $c_{i,j}$ denotes the residual bandwidth of the link $e_{i,j}$. A set F_A contains existing flows, each of which passes through at least one congested link $e_{i,j}$ ($e_{i,j} \in E^c_{f_a}$) on the path p. We can denote F_A as:

- $\forall f_{i.j}\in F, \text{if } \exists e_{i,j}\in E^c_{f_a}, \text{ then } f_{i,j}\in F_A.$

3.2 Cost optimization problem

Since a remote controller knows the load of each link, it can migrate existing flows on the congestion links to satisfy the bandwidth requirement of the new flows. We consider the negative impact of traffic migration on the network application. In this paper, we attempt to minimize the possible existing migrated flows because of the potential congested links while inserting new flows to the network, which we call the cost optimization problem. The goal of the cost optimization problem is to find an optimal subset F_a of F_A ($F_a \subseteq F_A$) for each new flow f_a. Consequently, the flow f_a could be accommodated by the network, if only these existing flows in the subset F_a are migrated to other feasible paths of the network. That is

- $F_a\subseteq F_A, \forall e_{i,j}\in E^c_{f_a}, \sum\limits_{f\in F_a} d^{f_{i,j}} + c_{i,j}\geq d^{f_a}$

We can get the minimum subset F_a of the set F_A for each flow $f_a\in U$ if and only if equality holds in the above formula. However, it is NP-complete to calculate such an optimal subset F_a for any flow $f_a\in U$ (see appendix A for the proof of theorem 1).

Theorem 1. The problem of identifying an optimal subset F_a from set F_A is NP-complete.

In this paper, we design a novel strategy to reduce the amount of migrated existing flows and get a set F_a. Any flow $f_a\in U$ will get enough bandwidth after migrating these existing flows.

Definition 2 (Event-level abstraction of network update). We abstract an update event U as a set of new flows which need to be inserted in the network. We denote them as $U=\{f_1, f_2, \cdots, f_a, \cdots, f_w\}$. A flow f_a is delivered from the source s_a to the destination d_a. A new flow f_a ($f_a \in U$) leads to the migration of existing flows of the set F_a. To complete an update event quickly, these existing flows must be migrated during network update.

Then, the cost of an update event U is the total amount of occupied bandwidth of those migrated existing traffic. That is,

$$Cost(U) = \sum_{a=1}^{w} \sum_{F_a \subseteq F_A, \mathfrak{f} \in F_a} d^{\mathfrak{f}}.$$

4 Scheduling models among multiple update events

A shared network usually needs to schedule a series update events to reduce the average and tail ECTs. Simple scheduling mechanisms like FIFO [10] do not perform well in this environment. If the head-event is heavy and has a long execution time, it would block many smaller events and increase the average and tail ECTs. Therefore, we study inter-event scheduling among multiple update events and give scheduling models for network update.

Fine-tuning the order of update events. As shown in Fig. 1, an update event with small flows is easier to be inserted into a Fat-Tree data center than the other two types of flows. Thus, the event with less migrated traffic have more possibilities to be updated faster. For example, if the bandwidth of each link is sufficient for executing an update event, the amount of migrated traffic is zero ($Cost(U) = 0$). This event will be executed more quickly than other update events, which incurs a less mount of traffic migration of existing flows. Therefore, preferentially scheduling these events with less amount of migrated traffic helps to reduce the average ECT of all update events where the principle is similar to the shortest job first [26] when scheduling a set of jobs over a shared cluster. However, rerouting of these existing flows means that we need to reconfigure flows which will hurt quality of service (QoS) and system stability [24]. Therefore, we utilize the cost of an update event as defied in Definition 2 as the metric to schedule all update events in a queue.

A simple way to do this is to reorder all queued events based on their update costs and choose the smallest event to execute first. Note that the network is in flux due to the changing of traffic. Consequently, we have to reorder all queued events frequently. This causes non-trivial computation and time overhead, especially for large-scale networks and update events. Moreover, the entire reordering fully breaks the order of the queue and destroys fairness among update events.

Therefore, in this paper, we prefer to schedule update events based on their arrival order, but dynamically fine-tunes the execution sequence to tackle the head-of-line blocking problem. Moreover, locally rerouting a set of existing flows also hurts quality of service (QoS) and system stability [15]. Thus, we must minimize the migrated traffic. Considering a set of n update events $\{U_1, U_2, \cdots, U_n\}$, we randomly choose two update events U_i and U_j and then compare the head-event with these two selected update events. Finally, we execute the one with the smallest update cost first.

Parallel updating. Let $Q = \{U_1, \cdots, U_n\}$ stand for an update queue, the aforementioned scheduling model fine-tunings the order of update events to reduce the average ECT. However, if multiple events can be updated simultaneously, it will provide an enormous speed boost for network update. Note that we assume that multiple events will not migrate the same existing flow. For an update event U_k, let the matrix C_{U_k} stand for the traffic distribution of the flows in update event U_k and several migrated existing flows on the congested links. $(C_{U_k})_{i,j}$ is the element of the i^{th} row, j^{th} column of C_{U_k}, which stands the bandwidth requirement on link $e_{i,j}$ for accommodating the update event U_k. $c_{i,j}$ is the residual bandwidth on link $e_{i,j}$. Each boolean variable x_k denotes whether the event U_k is chosen to be updated. Then, the optimization problem (PE) of getting the maximum number of parallel events can be formed as an integer linear programming (ILP).

$$\max \quad x_1 + x_2 + \cdots x_k + \cdots + x_n \tag{1}$$

$$s.t. \begin{cases} x_1(C_{U_1})_{0,0} + \cdots + x_k(C_{U_k})_{0,0} + \cdots + x_n(C_{U_n})_{0,0} \\ \leq c_{0,0} \\ x_1(C_{U_1})_{0,1} + \cdots + x_k(C_{U_k})_{0,1} + \cdots + x_n(C_{U_n})_{0,1} \\ \leq c_{0,1} \\ \cdots \\ x_1(C_{U_1})_{|v|,|v|} + \cdots + x_k(C_{U_k})_{|v|,|v|} + \cdots \\ + x_n(C_{U_n})_{|v|,|v|} \leq c_{|v|,|v|} \end{cases} \tag{2}$$

$$\forall e_{i,j} \in E, (C_{U_k})_{i,j} \leq C_{i,j}, 0 \leq k \leq n \tag{3}$$

$$x_k \in \{0, 1\} \tag{4}$$

Formula 2 guarantees that the bandwidth requirement on any link $e_{i,j}$ is less than the residual link capacity $c_{i,j}$ of this link. Formula 3 guarantees that we can at least get one update event that can be executed.

For a queue of n events, if each event owns w flows, each of which has v feasible paths. Then, the optimization space consists of v^{wn} combinations for updating those n update events. Even if we randomly select a path from multiple feasible shortest paths for a flow, there still exists w^n combinations. Thus, getting an exact scheduling sequence, which takes into account of the impact of the reconfiguration on future scheduled events, would cost much time and become infeasible. However, PE is NP-hard (see appendix B for the proof of theorem 1).

Theorem 2. The problem PE under the constraints of Formulas (2), (3) and (4) is NP-hard.

Opportunistic updating. The parallel updating model can decrease the average and tail ECTs. However, to get the maximum number of parallel events in each round, we must very first get the matrix of bandwidth requirements of each queued event. If the number of queued events is too big, the time overhead of getting these matrixes is unsustainable, especially when the network utilization is high and lots of update events compete for common scarce bandwidth resources.

Therefore, we propose another opportunistic updating model based on the parallel updating strategy. The basic idea is to find the first event following the same policy as the Fine-tuning scheduling model. In addition, we check other random selection of update events to decide whether they can be updated alongside the first event based on parallel updating model. A heavy-weight update event, which arrives earlier but is delayed by the fine-tuning model, will have a chance to be quickly executed in the process of opportunistic updating. This model increases scheduling efficiency and improves fairness.

5 Efficient methods for mutilple events scheduling

In this section, we propose an approximation method to reduce the update cost of each flow from a single update event and then reduce the ECT. Finally, we design two event scheduling methods, LMTF and P-LMTF to guarantee the fairness and efficiency for scheduling multiple network update events.

5.1 Cost optimization method for any update event

For any update event, the following two questions dominate the event completion time. First, is it necessary to reroute some existing flows if the desired paths lack sufficient bandwidth to transmit the flows of an update event? If some existing flows need to be migrated, which paths should be reallocated to ensure sufficient bandwidth for the migrated flows? Second, if needed, which existing flows should be migrated so that the network resource becomes just enough to accommodate the flows of an update event? This problem is NP-complete. Therefore, we aim to design an approximation algorithm to determine the minimum set of the migrated flows that can provide sufficient bandwidth both for the flows in the update event and for the migrated existing flows.

Any migrated flow for an update event competes for network resources with other traffic during the rerouting process. This behavior further affects the network's ability to handle more update events. Migrating existing flows would take non-trivial time and decrease the completion process of the update event. Thus, reducing the updating cost, i.e., the bandwidth requirements of migrated existing flows, is crucial for reducing the ECT for each update event. However, for an update event in the queue, the update cost of this event usually changes due to the dynamics of network traffic. This brings more challenges to reduce the update cost.

Given an update event $U = \{f_1, f_2, \cdots, f_w\}$ with w flows, the flow sequence and their paths are fixed. If the allocated path does not have enough link bandwidth, we need to migrate a mount of existing traffic to accommodate the new flow, especially when the network utilization is high. Therefore, different migration scheme of existing flows exhibit different update costs. The path selection for each flow of an update event and migrated traffic caused by the flows of an update event both will effect the update cost of each event. However, it appears reasonable to choose the path with very few congested links. This method, however, does not mean to migrate less amount of traffic from these congested links. Furthermore, we can choose a path, which incurs the minimum migration cost of existing flow. That is, we have to calculate the migration cost of each feasible path for a flow. This would cost much time and considerably increase the decision-making time in each scheduling round. Therefore, to get the update cost of each event, we allocate a fixed path to each flow of an update event when we optimize the update cost of a flow. Algorithm 1 illustrates our cost optimization method, which minimizes the amount of existing flows to be migrated for flows in an update event and its approximation ratio is at most 2 (see proof for theorem 3 in appendix C).

Algorithm 1 Cost optimizing method(U,δ)

```
Require: An update event $U$ and a variable $\delta=\varepsilon/2n$, where
    $0\leq\varepsilon\leq 1$ and $n=|F_A|$.
 1: for flow $f_a \in U$ do
 2:     Let an array $F_A$ record all sizes of existing flows passing
        through any congested link in $E^c_{f_a}$.
 3:     Append $-sum(F_A)$ to an array $L_0$.
 4:     for $i$=1 to $|F_A|$ do
 5:         for $j$=0 to $|L_{i-1}|$ do
 6:             Append $L_{i-1}[j]$ and $L_{i-1}[j] + d^{F_A[i]}$ to the array $L_i$
 7:         Sort $L_i$ in the ascending order
 8:         for $j$=1 to $|L_i|$ do
 9:             if $L_i[j] > L_i[j-1]/(1+\delta)$ then
10:                 Append $L_i[j]$ to an array $L_i^{'}$
11:         $L_i \leftarrow L_i^{'}$
12:     Select the biggest value $-k$ in $L_i$ but smaller than $-d^{f_a}$
13:     $Cost(f_a) \leftarrow k$
14:     $Cost(U) \leftarrow Cost(U) + Cost(f_a)$
15: return $Cost(U)$
```

Theorem 3. The approximation ratio of Algorithm 1 is at most 2.

It takes an update event U and a variable δ as inputs. For each flow f_a in the update event, the candidate set F_A contains all existing flows, each of which passes through at least one of potential congested links of the path which the flow f_a will pass through. We first sum the amount of traffic in set F_A and record the opposite number of it as $-sum(F_A)$ and then put it in an array L_0 (line 3). For each flow in F_A, we will take it out from the sum $-sum(F_A)$ to see if any number in L_{i-1} can approximate it (lines 4 - 9). If no, the new sum will be added to the array L_i (lines 10 - 11). Finally, we can get the biggest value -k from the array and get the cost (lines 12-15).

5.2 Fine-tuning the order of update events

The aforementioned optimization method can effectively decrease the completion time of a single update event. How to schedule multiple update events that form a queue according to their arrival order is still unknown. The scheduling order determines not only the average and tail ECTs (two metrics related to efficiency), but also the fairness of such update events. In this environment, FIFO is attractive because it is easy to implement and guarantees strict fairness. If the durations of such update events are similar, FIFO

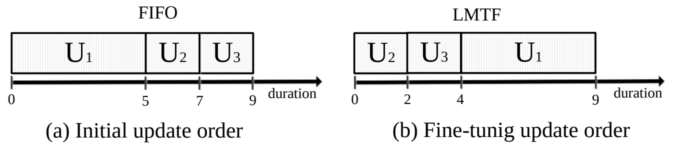


Fig. 3. Our LMTF can reduce the average ECT against the FIFO. U_1, U_2 and U_3 stand for three update events with various event durations.

Algorithm 2 LMTF(U_{all},α)

```
Require: A set $U_{all}$ contains all queued update events
1: Invoke Algorithm 1 to get the update cost $Cost(U)$ for the head-event.
2: if $U_{all}$ contains at least $\alpha + 1$ update events then
3:     Randomly sample other $\alpha$ events except the head-event and calculate their update costs
4: else
5:     Randomly sample other $\alpha \geq 1$ events except the head-event and calculate their update costs
6: Let $U$ denote the new head-event with the least cost among $\alpha + 1$ candidates, which will be scheduled first
7: return $U$
```

Algorithm 3 P-LMTF(U_{all})

```
Require: A set $U_{all}$ contains all queued update events and a parameter $\alpha$.
1: Invoke Algorithm 2 to get the update cost of each queued event.
2: if $\sum_{k=1}^{n} (C_{U_k})_{i,j} \leq c_{i,j}$, $(0 \leq i, j \leq n)$ then
3:     return $U_{all}$
4: while $U \neq U_{ALL}$ do
5:     Select $\alpha$ update events from the set $U_{all} - U_{head}$ randomly. Then, the event with the least update cost chosen from these $\alpha$+1 events becomes the new head-event.
6:     Find the max number $m$ $(1 \leq m \leq \alpha)$ of parallel events $U_{r_1}, \cdots, U_{r_m}$ from the rest of $\alpha$ update events that satisfy $\sum_{k=1}^{m} (C_{U_{r_k}})_{i,j} \leq c_{i,j} \leq \sum_{k=1}^{m+1} (C_{U_{r_k}})_{i,j}$, $(0 \leq i, j \leq n)$.
7:     $U \leftarrow \{U_{r_1}, \cdots, U_{r_m}\}$
8:     Execute events $U_{head}$ and $U_{r_1}, \cdots, U_{r_m}$ simultaneously.
9:     Check the queue to delete some events from $U_{all}$, which do not have to be updated at this moment.
10: return $U$
```

is proven to be optimal for minimizing the tail ECT and achieving tight fairness [27]. If the duration exhibits heavy-tailed distribution, FIFO usually leads to the head-of-line blocking, caused by the heavy update events which arrived earlier. This problem will increase the average and tail ECTs. In this scenario, FIFO guarantees strict fairness, but fails to supply efficiency to update events. Therefore, for a queue of update events, we first focus on decreasing the average and tail ECTs at the cost of slightly relaxing the fairness requirement.

A straightforward method is to reorder all queued events based on their update costs and choose the lowest-cost event to execute first. Fig. 3 shows an example of the scheduling of three update events. The execution time of each update event is 1 second. The update cost is 4 seconds for event U_1 and 1 second for events U_2 and U_3. The average ECT of such update events is $(5+7+9)/3=7$ seconds, and the tail ECT is 9 seconds under the FIFO method, as shown in Fig. 3(a). If we order those update events according to their update costs, the ideal sequence is shown in Fig. 3(b). The average ECT is reduced to $(2+4+9)/3=5$ seconds and the tail ECT is the same. Theoretically, such a reordering of all update events could tackle the head-of-line blocking problem, and consequently, reduce the wait time of the lower-cost events that arrive later. As discussed in Section 5, this method suffers from huge computation and time overheads, the loss of fairness.

In this paper, we propose the least migration traffic first method (LMTF), a lightweight but effective scheduling method. Algorithm 2 reports the details of our LMTF method. It prefers to schedule update events based on their arrival order, but it can dynamically fine-tune the execution sequence of a few selected events to tackle the head-of-line blocking problem. The basic idea is to randomly sample α update events except for the head-event from the queue. At the same time, we compare these chosen events with the head-event and the update event with the lowest cost among the α+1 candidates is chosen to be executed first (lines 2-3). The head-of-line blocking problem, however, is well-tackled by selecting the update event with the lowest cost among α+1 events. Note that LMTF does not persist in sampling α update events when the queue contains less than $\alpha + 1$ update events (line 5). The evaluation results indicate that our LMTF method effectively decreases the average and tail ECTs for any queue of update events, even when the sampling number α is set as 2. This is a regular pattern inspired by the load-balance theory of the power of two random choices [28].

5.3 Opportunistic updating

In this section, we will present the opportunistic updating method, a general inter-event scheduling policy to improve fairness based on fine-tuning policy. The basic idea is first to select α update events randomly and compare them with the head-event to get the new head-event with the least update cost. At the same time, it tries to search opportunities for updating other α events, including the heavy events, which are always scheduled later even if they arrive earlier by LMTF. The opportunistic updating process, however, would check whether those α events can be updated with the selected new head-event simultaneously. If yes, they will have the chance to be executed quickly.

A heuristic schedule method, parallel-LMTF (P-LMTF), is proposed to realize this design. As shown in Algorithm 3, we form a candidate set consisting of the initial head-event and the other α update events sampled randomly from the queue. The event with the lowest update cost among the α+1 candidates is selected to be the new head-event (line 5) as in LMTF. In the second step, only the other α candidates will be checked, to determine whether performing them simultaneously with the new head-event (line 6). That is, the second step offers priority for update events that are heavy but arrive earlier; hence, it effectively improves fairness compared with fine-tuning updating strategy and guarantees update efficiency using the parallel updating. Finally, it updates the traffic distribution of the network and checks the queued events to delete the events which do not have to be updated after waiting a while in the queue (lines $8-9$).

Note that P-LMTF does not check the entire queue to search for update events that can be executed alongside the new head-event because it would cause huge computation and time overheads, especially for large-scale networks and in a long queue. Instead, P-LMTF reduces the variables space to the chosen $\alpha{+}1$ events which reduces the difficulty greatly and increases the operation-rate to get the answer when we search for parallel update events. The evaluation results indicate that the α random update events still effectively exploit the benefits of opportunistic updating, even in the case of $\alpha{=}2$.

Since OpenFlow controller can handle 4175 packet-in requests per second [29] and the method in Time4 [30] can trigger network configurations on different network devices simultaneously, the generated configuration schemes of multiple parallel events can go into effect simultaneously and these parallel events can be executed in parallel.

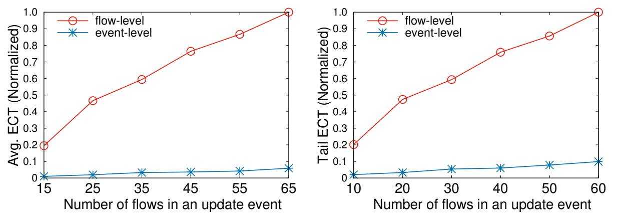
6 Experimental Evaluation

We start with the settings of our trace-driven evaluations in an 8-pod Fat-Tree datacenter network. We then compare the performance of our event-level and flow-level abstractions of network update. Finally, we evaluate our LMTF and P-LMTF scheduling methods against the FIFO method over a real data-set from Yahoo!'s data center [20]. Note that we use Gurobi Optimizer (version 7) to solve the ILP problems [31].

6.1 Evaluation settings

Topology. We consider an 8-pod Fat-Tree [32] datacenter network where the bandwidth of each link is fixed as 1 Gbps. In a Fat-Tree data center, the number of servers and switches is determined by the setting of parameter k. A Fat-Tree topology accommodates $5k^2/4(80)$ switches and $k^3/4(128)$ servers, which form k pods. The parameter k is set to 8 in our experiments.

Workloads. To enable the trace-driven evaluation of our methods and related work, we inject a large amount of traffic into the Fat-tree datacenter as the background traffic and observe the expected link utilization. All injected flows are generated from a real traffic data-set from Yahoo!'s data center [20]. This trace records the basic information of each flow, including the IP addresses of both the source and destination servers, the size and duration of each flow, etc. Note that the real IP addresses in the trace are anonymous. We use a hash function to map the IP addresses of the source and destination of each flow into our datacenter network. We further generate a set of heterogeneous network update events which differ in the number of flows, flow sizes, and flow durations. We set the average number of flows caused by each update event as a random integer ranging from 10 to 100. We then generate new flows with demand ranging from 20 Mbps to 40 Mbps for randomly chosen source-destination top-of-rack switch pair for each events.



(a) Average ECT (b) Tail ECT

Fig. 4. The changing trends of two metrics under 10 update events when network utilization approaches 50%. The average number of flows in each event increases from 15 to 60.

Metrics. For a queue of network update events, we evaluate the benefits of event-level abstraction by comparing six metrics of the flow-level scheduling method, our LMTF method, and our P-LMTF method. The metrics are as following: the total update cost of all update events, the average event completion time (ECT), the tail ECT, the total plan time, the event queuing delay and fairness. The update cost of an update event means the amount of occupied bandwidth of migrated existing flows. The average and tail ECTs indicate the average and tail completion times of all update events in the queue. The total plan time indicates the time it takes to make the update plan for all queued events. The event queuing delay is the time from the moment an update event is pushed in an update queue until its execution starts. Finally, we evaluate the updating fairness ζ. Let η be the total number of queued events, and δ denote the number of the update events that are scheduled no latter than the corresponding time under the FIFO principle. Then, the updating fairness can be defined as:

- $\zeta{=}\delta/\eta \ \ (0{\le}\zeta{\le}1)$.

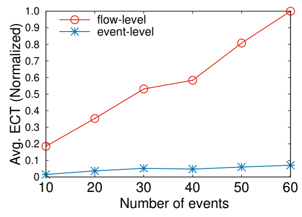
As the increase of ζ towards to 1, the updating method is close to be an absolute fair updating method, vice versa. Specially, $\zeta_{FIFO}{=}1$, hence, FIFO method is an absolute fairness updating.

We first compare the flow-level and event-level scheduling methods in a queue of update events. We then regulate several essential factors to evaluate their impacts on the performance metrics. They are the number of queued events, network utilization, the type of queued events, the event queueing delay and the fairness.

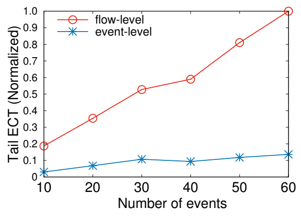
6.2 Flow-level vs. Event-level scheduling methods

To evaluate the effectiveness of our event-level abstraction, we compare it with the flow-level method which schedules the flow with the smallest bandwidth demand first. We construct a set of update events, each of which has 10 to 100 flows. We report only the normalized results of each metric, which are achieved by dividing the maximum value of the flow-level method.

Fig. 4 plots the average and tail ECTs of 10 update events where the average number of flows in each event varies from 15 to 60. The average and tail ECTs of our event-level method are up to 10× faster than the flow-level method. Fig. 5 shows that more events in the queue would lead to larger average and tail ECTs for both methods. Scheduling the flow with the smallest bandwidth demand has a higher probability to have a lower update cost and thus a lower average



(a) Average ECT



(b) Tail ECT

Fig. 5. The changing trends of two metrics under different numbers of update events when network utilization is 50%. Note that the number of flows in each event ranges from 10 to 100 randomly.

ECT for all flows of all events. However, such flow-level scheduling methods do not mention which events these flows belong to and thus cannot help to reduce average and tail event ECTs. Our event-level scheduling method groups the flows belonged to the same event and schedules them together. Therefore, it gives a chance to the event with lower update cost to be executed first and significantly reduces average and tail ECTs.

6.3 Impact of the number of update events

We evaluate our LMTF and P-LMTF scheduling methods against the fairness scheduling method FIFO as we vary the number of update events in the queue. The parameter α is set as 4, which means that we choose 4 events randomly from the update queue and compare the update costs of these 4 update events with the cost of the initial head-event. The network utilization varies from 50% to 70%; while each update event has 10 to 100 flows. Fig. 6 plots the evaluation results.

Fig. 6(a) reports reduction in the total update cost of our method against the FIFO method. P-LMTF achieves a stable reduction by $34\% - 45\%$ as the number of queued events varies from 10 to 50, which means the ratio of the difference between FIFO and our methods to FIFO is $34\% - 45\%$. In this setting, LMTF also reduces the total update cost, but its achievement is always smaller than that of P-LMTF. Note that the total update costs of P-LMTF are less then LMTF method. Because the update sequence leads to the different residual bandwidth of each link which will effect the traffic migration for following update events. On the other hand, P-LMTF tries to find more update events which can be performed with the head event in parallel using remaining network capacity. That is, if we can find more events to be performed with the head event, the cost of multiple events can be reduced. As expected, the changing trends of average and tail ECTs are similar to the total update cost. For example, in the setting of 20 update events, the reduction in the update cost by LMTF is dramatic. At the same time, the average and tail ECTs also decrease, especially the tail ECT. That is to say, there must exist multiple heavy update events in the update queue that perform before light-weight events. Such heavy events arrive earlier, and therefore, block later events. As a result, they increase the ECT of some later events. This scenario, however, will not affect the performance of P-LMTF. As the analysis in Section 5.3 shows, P-LMTF improves LMTF by appending an opportunistic updating process and has a better performance.

Fig. 6(b) indicates that P-LMTF achieves a $69\% - 80\%$ reduction in the average ECT compared with FIFO, and LMTF achieves a $22\% - 36\%$ reduction. This significant improvement comes from the introduction of opportunistic updating. It permits multiple events to be executed simultaneously if possible, i.e., heavy events that arrive earlier have a chance to be executed at the same time as the head-event. Thus, P-LMTF further decreases the average ECT. On the other hand, P-LMTF reduces the tail ECT by $35\% - 48\%$ and LMTF reduces by $5\% - 26\%$ compared with FIFO, as shown in Fig. 6(c).

Finally, we measure the total plan time for all queued update events, as shown in Fig. 6(d). As expected, FIFO takes the least amount of time because it does not conduct other actions during the decision process. Our LMTF and P-LMTF methods cause a longer plan time because LMTF calculates the update costs for $\alpha + 1$ update events to find the new head-event with lowest update cost. LMTF and P-LMTF take about 4.5 times and 2 times more plan time than FIFO, respectively, regardless of the number of update events. P-LMTF takes less plan time than LMTF since it has the chance to make an update plan for multiple events simultaneously in one round. This is an acceptable tradeoff to achieve significant reductions in the other three metrics.

6.4 Impact of the network utilization

Network utilization reflects the network's ability to provide services and it influences the execution of update events. In this section, we evaluate the performance of involved methods over a longer period. The network utilization fluctuates from 10% to 90% when the number of update events is 10. The number of flows in each update event varies from 10 to 100.

Fig. 7(a) reports reduction in the total update cost of our method against the FIFO method. P-LMTF achieves a stable reduction by $29\% - 52\%$. The benefit of P-LMTF smoothly grows with the increase of the network utilization, while the benefit of LMTF decreases when the network utilization reaches 70%. P-LMTF tries to find more update events which can be performed in parallel with the head event using the remaining network capacity. That is, if we can find more events to be performed with the head event, we can avoid migrating background traffic multiple times and reduce the the cost of multiple update events. Therefore, P-LMTF has a lower update cost than LMTF.

Fig. 7(b) indicates that P-LMTF achieves a $63\% - 82\%$ reduction in average ECT, and LMTF reduces by $21\% - 38\%$ compared with FIFO. P-LMTF and LMTF reduce the tail ECT by $28\% - 53\%$ and $5\% - 26\%$, respectively, compared with FIFO, as shown in Fig. 7(c). That is, both of our methods effectively reduce the average and tail ECTs compared with FIFO, regardless of the network utilization. The benefit of P-LMTF is more remarkable for its higher network utilization.

Fig. 7(d) plots the ratio of the total plan time of our two methods in relation to the FIFO method. We can see that the metric of our methods, except for the LMTF at 90%, grows as the network utilization increases. In this setting, FIFO also takes more time to make the update plan. In summary, LMTF and P-LMTF reduce 4 times and 2 times on

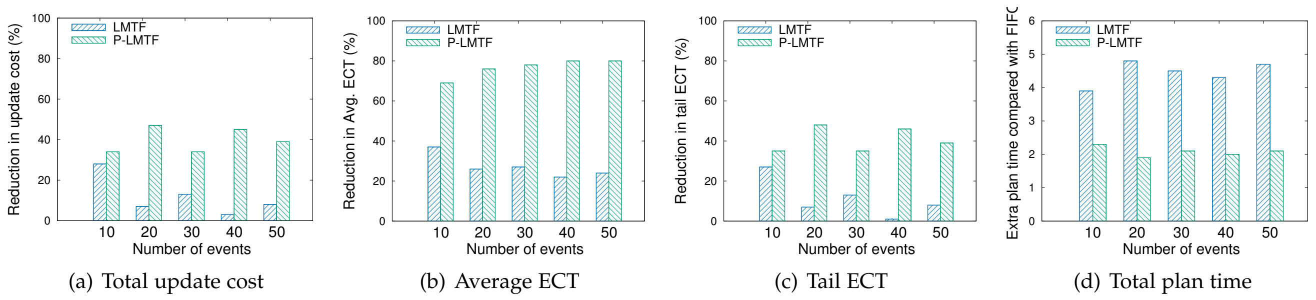


Fig. 6. Reductions in three metrics against FIFO and the changing trend of the plan time of our scheduling methods under different numbers of events where the network utilization fluctuates between 50% and 70% and α=4. Note that the number of flows in each update event is a random integer from 10 to 100.

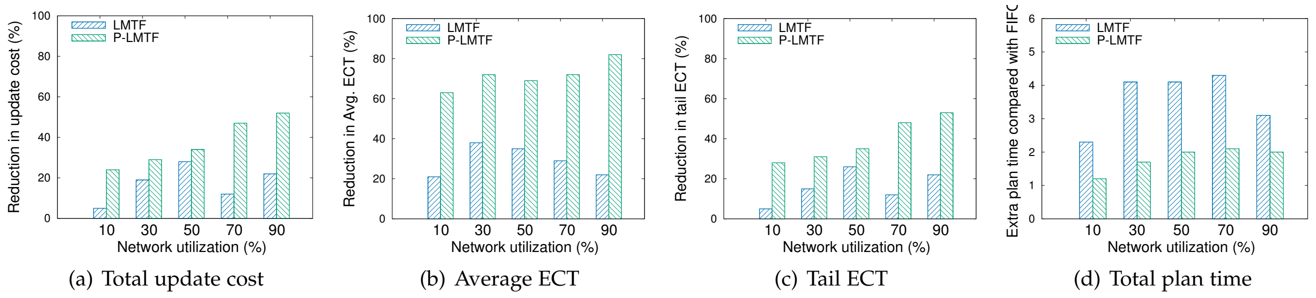


Fig. 7. Reductions in three metrics against FIFO and the changing trend of the plan time under different scheduling methods where the network utilization varies from 10% to 90% and α=4. Note that the number of update events is 10 and the number of flows in each update event ranges from 10 to 100.

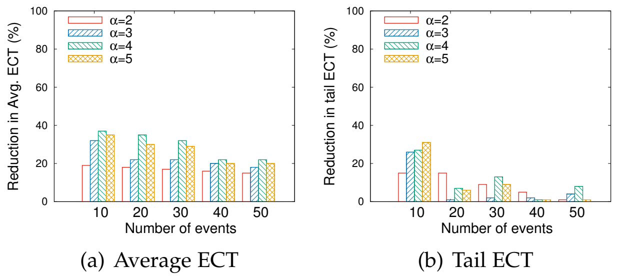


Fig. 8. Reduction in two metrics with our LMTF method against FIFO under different settings of α and different numbers of events where the network utilization fluctuates from 50% to 70%.

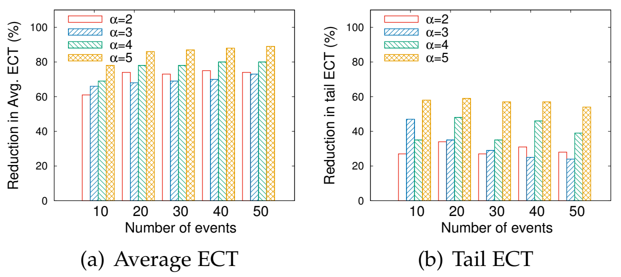


Fig. 9. Reduction in two metrics with our P-LMTF method against FIFO under different settings of α and different numbers of events where the network utilization fluctuates from 50% to 70%.

plan time than FIFO, respectively, irrespective of the value of network utilization. We can infer that such extra plan time is reasonable when achieving significant reductions in the other three metrics.

6.5 Impact of the number of sampled events from the queue

Note that LMTF and P-LMTF are dominated by the parameter α since they focus on $\alpha+1$ sampled events in each round.

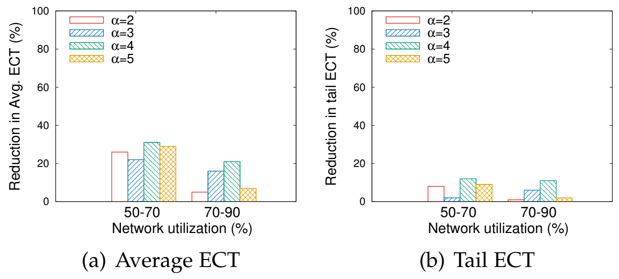
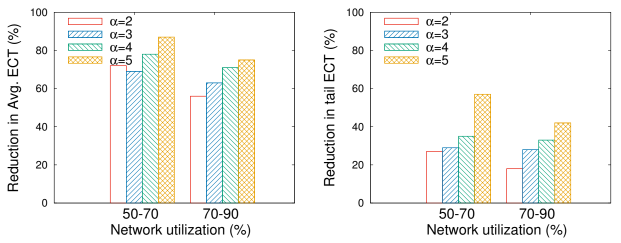


Fig. 10. Reduction in two metrics with our LMTF method against FIFO under different settings of α and different network utilizations where the number of update events is 30.

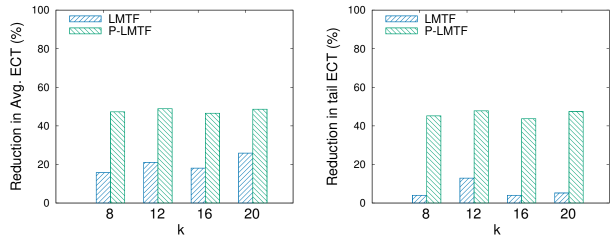
For the P-LMTF method, α still means the largest number of update events to be executed alongside the head-event. A larger α brings more sampled events to be processed by the two methods in each round but more decision time, and therefore, we can make a tradeoff and optimize the average and tail ECTs.

Varying the number of events. We first measure the average and tail ECTs of our two methods under varied settings of α and different numbers of update events when the network utilization fluctuates from 50% to 70%. Fig. 8 reports the results of our LMTF method against the FIFO method. The gains of LMTF actually reduce when the α value becomes 5 from 4. Note that we have to compute the update costs of $\alpha+1$ events only for selecting the new head-event in each scheduling round. It will cost much time to get such update costs while facing the heavy-events when the network utilization is high. Sampling more events means that we have to get the update cost of each event and compare them, which increases the decision time. Sometimes, if we can get more reduction in average ECT by less sampling, we do not have to sample more events.



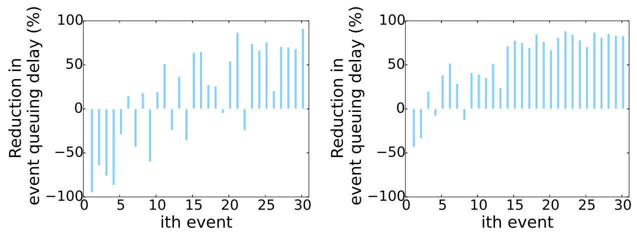
(a) Average ECT (b) Tail ECT

Fig. 11. Reduction in two metrics with our P-LMTF method against FIFO under different settings of α and different network utilizations where the number of update events is 30.



(a) Average ECT (b) Tail ECT

Fig. 12. Reduction in two metrics with our P-LMTF and LMTF methods against FIFO under different settings of k where the network utilization approaches 50% and α=4. Note that the number of update events is 10 and that the average number of flows in each update event is 20.



(a) LMTF (b) P-LMTF

Fig. 13. Reduction in queuing delay of each event with our methods against FIFO, where the network utilization fluctuates from 50% to 70% and α=4. Note that the number of update events is 30 and that the number of flows in each update event ranges from 10 to 100.

As expected, the reduction in average and tail ECTs of LMTF against FIFO decreases for any fixed setting of α, when the number of queued events changes from 10 to 50. This is because the sampled $\alpha+1$ events become sparse during this process compared with the total number of queued events. Thus, the probability of finding a desired update event with the lowest update cost from $\alpha+1$ sampled events decreases with the increase of the event number. That is, it is easier to find a desired update event with lowest-cost from α update events chosen randomly from $n(n \geq \alpha)$ candidates than from $2n$ candidates. Obviously, there exists no explicit, optimal solution for any fixed queue length. The reduction in average and tail ECTs of LMTF against FIFO changes irregularly while the setting of α is increasing, because of the cost comparison among multiple sampled events.

Fig. 9 depicts the results of our P-LMTF method against the FIFO method. The performance of the two metrics remains relatively stable as the number of events increases form 10 to 50. It incurs $61\%-89\%$ and $27\%-59\%$ reductions in average and tail ECTs, respectively, when compared with FIFO. Additionally, it is clear that higher values of α achieve better performances in terms of the reduction of both average and tail ECTs. Consequently, it is impossible to compute an optimal solution for the fixed length of a queue. The results indicate that P-LMTF always outperforms LMTF under the same evaluation scenario, regardless of the setting of α, due to the gain of opportunistic update.

Varying network utilization. To understand the influence of parameter α under different network load, we measure two metrics under various parameters, when the network utilization changes from 50% to 90%. Fig. 10 shows the reduction in average and tail ECTs of LMTF against FIFO. It is obvious that the achievements decrease along with the increase of the network utilization, irrespective of the setting of parameter α. When the network utilization increases, it costs much time to migrate existing flows for each update event, which leads to more time overhead to get the head-event in each scheduling round. In addition, the gains of LMTF actually reduce when α becomes 5 from 4 for the same reason as in Fig. 8. However, our P-LMTF method has solved this problem by introducing parallel updating mechanism.

Fig. 11 shows a reduction in average and tail ECTs with P-LMTF against FIFO. More precisely, it offers $56\% - 87\%$ and $18\% - 57\%$ reductions in average and tail ECTs, respectively. Additionally, it is clear that our P-LMTF achieves a better performance under higher α, regardless of the network utilization.

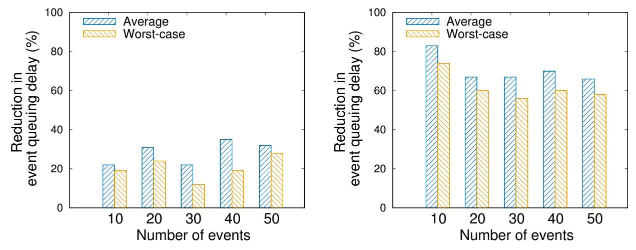
6.6 Impact of the network scale

Fig. 12 shows the reduction in average and tail ECTs with P-LMTF and LMTF against FIFO when k equals to 8, 12, 16 and 20. Since the benefits of P-LMTF and LMTF increase with the number of flows in each update event as shown in Fig. 4, we use 20 as the average flow number of each event in our experiments. The benefits of P-LMTF do not decrease as the increase of network scale when the network utilization approaches to 50%. The reason is that the number of paths between two servers is the same as the parameter k of network scale for Fat-Tree topology. Therefore, even though network scale increases, P-LMTF still has a higher probability to find more parallel events in each round.

6.7 Event queuing delay and Fairness

We first observe the queuing delay of 30 events in the update queue. Then, we study the average event queuing delay during a network update with multiple events. Finally, we quantify the updating fairness of a queue with multiple update events. Our results are depicted in Fig. 13 and Fig. 14.

Fig. 13 and Fig. 14 plot the reductions in queuing delay of each event and the average queuing delay of all events with LMTF and P-LMTF compared with FIFO. As shown in Fig. 13(a), because of the fine-tuning of the execution sequence, LMTF leads to the delay of several update events. Even so, we can see from the Fig. 14(a), LMTF provides $20\% - 40\%$ reduction in average event queuing delay and $10\% - 30\%$ reduction in the worst-case delay. Moreover, P-LMTF, based on the LMTF and parallel updating, provides more opportunities for update events with high update costs to



(a) LMTF (b) P-LMTF

Fig. 14. Reduction in average event queuing delay with our methods against FIFO under different number of events where the network utilization fluctuates from 50% to 70% and α=4.

be executed earlier. Therefore, it both reduces the queuing delay of an update event and guarantees the fairness, as shown in Fig. 13(b). With P-LMTF, the event queuing delay of the worst-case reduces by $60\% - 74\%$ and the average by $67\% - 83\%$, as shown in Fig. 14(b). As expected, with the increase of the number of event, the benefits have a stable fluctuation.

We can get the metric of updating fairness from Fig. 13. For each update event, if the reduction in event queueing delay is greater than or equal to 0, this update event is scheduled earlier than its scheduling time under the FIFO principle. The evaluation results show that our scheduling methods achieve $\zeta_{LMTF} = 0.63$ and $\zeta_{P-LMTF} = 0.87$, respectively. Note that the FIFO scheduling method achieves $\zeta_{FIFO} = 1$. Our LMTF and P-LMTF significantly improve the update efficiency at the cost of relaxing the fairness slightly. That is, there exists a tradeoff between the efficiency and fairness when dealing with multiple update events. Fortunately, P-LMTF achieves a higher fairness than LMTF when performing a queue of update events, due to exploit the opportunities of parallel updating.

7 Discussion and Future work

Multiple update events may compete for the same scarce bandwidth resources. The update event that queues in the back has to wait for resource releasing of other events. We may expect to migrate the same existing flows. However, there may exist some cases that we cannot find the paths with enough link bandwidth for these existing flows using the solutions provided by Brandt et al. and Foerster et al. about how to check the network capacity for splittable flows [8] or unsplittable flows [33]. As a result, these existing flows will be migrated to other paths and have to share the limited link bandwidth with other existing flows. If the other path cannot provide enough bandwidth for migrated flows, they may experience the rate limitation. Note that we assume that the migrated flows have the lowest priority, rate limitation will not affect the application performance which they belong to. Since we just migrate the part of existing flows on the possible congested links even if the existing flows pass through multiple paths, this imposes no constraint on the routing algorithm for the existing flows. Therefore, our event scheduling algorithms are compatible to employing multi-path routing algorithms.

In this paper, we treat flows from any update events as the new flows and existing flows as the background traffic which can be migrated to make space for new flows. It is true that some update events caused by a latency-sensitive application may have a strict deadline and the same existing flows may be expected to be migrated by different update events. Currently, we only consider the cases that no migration collision during the update process and the update events are independent to each other with no deadline and priority. It remains as the future work to investigate more complex scenarios [34] to deal with their correlative dependence and avoid migrating the same existing flow multiple times. The priority queue can be used to solve these problems by allocating queue for update events with different priorities.

With the development of the fast programming data planes on switches, many applications have been moved to the network, to the dataplane to get more details and fast processing within line-rate transmission. Since the event-level scheduling algorithms need the global information, such as background traffic and their paths to compute the cost of each event and then decide the update sequence, they cannot be achieved only in the data plane. However, scheduling packets in the dataplane can help us to guarantee that the flows are processed in the order as we expect. For example, if queueing time of a flow exceeds the limitation, we can enqueue/dequeue packets of this flow in/from any place of a queue and schedule them as we want on the programmable data plane [35, 36].

8 Conclusion

Due to updating issues, the network condition is constantly in flux. Prior updating schemes tackle all the flows affected by such updating issues individually while ignoring the event-level update requirements. In this paper, we use the event-level abstraction of network update to optimize the update cost and event completion time (ECT) of each update event. We further propose two efficient approaches, LMTF and P-LMTF, to schedule multiple update events while simultaneously improving the average and tail ECTs of queued update events and preserving fairness. Trace-driven evaluations indicate that our event-level LMTF method achieves a performance $10\times$ better in average ECT and $6\times$ better in tail ECT compared to flow-level method. When network utilization exceeds 70%, P-LMTF method reduces average ECT by 75% and tail ECT by 42% compared with FIFO.

Acknowledgements

This work is partially supported by National Program for Support of Top-Notch Young Professionals of National Program for Special Support of Eminent Professionals, Tianjin Science and Technology Foundation under Grant No. 18ZXJMTG00290, and National Natural Science Foundation of China under Grant No.61772544.

References

[1] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zUpdate: updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, Hong Kong, 2013.
[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, Helsinki, Finland, 2012.

[3] N. P. Katta, J. Rexford, and D. Walker, “Incremental consistent updates,” in *Proc. ACM SIGCOMM*, Hong Kong, 2013.

[4] T. Mizrahi, E. Saat, and Y. Moses, “Timed consistent network updates,” in *Proc. SOSR*, Santa Clara, CA, 2015.

[5] T. Mizrahi, O. Rottenstreich, and Y. Moses, “Timeflip: Scheduling network updates with timestamp-based TCAM ranges,” in *Proc. INFOCOM*, Hong Kong, 2015.

[6] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven WAN,” in *Proc. ACM SIGCOMM*, Hong Kong, 2013.

[7] J. Zheng, H. Xu, G. Chen, and H. Dai, “Minimizing transient congestion during network update in data centers,” in *Proc. IEEE ICNP*, San Francisco, CA, 2015.

[8] S. Brandt, K.-T. Foerster, and R. Wattenhofer, “On consistent migration of flows in SDNs,” in *Proc. INFOCOM*, San Francisco, CA, 2016.

[9] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *Proc. ACM SIGCOMM*, Chicago, Illinois, CA, 2014.

[10] A. Endres and H. Stork, “Fifo-Optimal placement on pages of independently referenced sectors,” *Inf. Process. Lett.*, 1977.

[11] K. T. Foerster and R. Wattenhofer, “The power of two in consistent network updates: Hard loop freedom, easy flow migration,” in *Proc. ICCCN*, Waikoloa, Hawaii, 2016.

[12] S. Ghorbani and M. Caesar, “Walk the line: consistent network updates with bandwidth guarantees,” in *Proc. ACM HotSDN*, Helsinki, 2012.

[13] W. Wang, W. He, J. Su, and Y. Chen, “Cupid: Congestion-free consistent data plane update in software defined networks,” in *Proc. 35th IEEE INFOCOM*, San Francisco, CA, 2016.

[14] J. Zheng, B. Li, C. Tian, K. Foerster, S. Schmid, G. Chen, J. Wu, and R. Li, “Congestion-free rerouting of multiple flows in timed sdns,” *IEEE Journal on Selected Areas in Communications*, 2019.

[15] J. Zheng, G. Chen, S. Schmid, H. Dai, J. Wu, and Q. Ni, “Scheduling congestion-and loop-free network update in timed sdns,” *IEEE Journal on Selected Areas in Communications*, 2017.

[16] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, and J. Wux, “Scheduling congestion-free updates of multiple flows with chronicle in timed sdns,” in *IEEE ICDCS*, 2018.

[17] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, “B4: experience with a globally-deployed software defined wan,” in *Proc. ACM SIGCOMM*, Hong Kong, 2013.

[18] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev *et al.*, “B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[19] M. Chowdhury and I. Stoica, “Coflow: A networking abstraction for cluster applications,” in *Proc. ACM Workshop on Hot Topics in Networks*, Redmond, WA, USA, 2012.

[20] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu, “A first look at inter-data center traffic characteristics via yahoo! datasets,” in *Proc. INFOCOM*, Orlando, FL, 2011.

[21] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proc. ACM SIGCOMM*, New Delhi, 2010.

[22] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, “RSVP-TE: extensions to RSVP for LSP tunnels,” *RFC Editor*, 2001.

[23] T. Qu, D. Guo, Y. Shen, X. Zhu, L. Luo, and Z. Liu, “Minimizing Traffic Migration During Network Update in IaaS Datacenters,” *Accepted to appear at IEEE Transaction on Service Computing*, 2016.

[24] S. Paris, A. Destounis, L. Maggi, G. S. Paschos, and J. Leguay, “Controlling flow reconfigurations in sdn,” in *Proc.35th IEEE INFOCOM*, San Francisco, CA, 2016.

[25] S. Paris, G. S. Paschos, and J. Leguay, “Dynamic control for failure recovery and flow reconfiguration in sdn,” in *Design of Reliable Communication Networks (DRCN), 2016 12th International Conference on the*. IEEE, 2016.

[26] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, 2015.

[27] A. Wierman and B. Zwart, “Is Tail-Optimal scheduling possible?” *Operations Research*, 2012.

[28] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *IEEE Transaction Parallel Distribute System*, vol. 12, pp. 1094–1104, 2001.

[29] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, “Modeling and performance evaluation of an openflow architecture,” in *Proc. ITC*, 2011.

[30] T. Mizrahi and Y. Moses, “Software defined networks: It’s about time,” in *Proc. IEEE INFOCOM*, 2016.

[31] G. OPTIMIZATION, “Inc. gurobi optimizer reference manual, 2015,” *URL: http://www. gurobi. com*, 2014.

[32] C. E. Leiserson, “Fat-trees: universal networks for hardware-efficient supercomputing,” *IEEE transactions on Computers*, vol. 100, pp. 892–901, 1985.

[33] K.-T. Foerster, “On the consistent migration of unsplittable flows: Upper and lower complexity bounds,” in *NCA*, 2017.

[34] K.-T. Foerster, S. Schmid, and S. Vissicchio, “Survey of consistent software-defined network updates,” *IEEE Communications Surveys & Tutorials*, 2018.

[35] V. Shrivastav, “Fast, scalable, and programmable packet scheduler in hardware,” 2019.

[36] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, “Programmable packet scheduling at line rate,” in *proc. SIGCOMM*, 2016.

[37] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of computer computations*. Springer, 1972.

[38] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proc. ACM symposium on Theory of computing*, 1971.

Appendix A
Proof for Theorem 1

The problem of identifying an optimal subset F_a from set F_A is NP-complete.

Proof: According to Formula (1), we try to find a subset of migrated existing flows, i.e., $f_1, \cdots, f_n$, from the candidate set F_A. The amount of bandwidth occupied by such migrated flows justly satisfies the bandwidth requirement d^{f_a} of flow f_a. This is the certification to verify that there is a solution for this problem. We can find a set of migrated flows in polynomial time and judge whether the remaining bandwidth is sufficient to satisfy the requirement d^{f_a} of flow f_a. Thus, this problem is NP.

Consider a case of 3-Dimensional Matching [37], which is a well-known NP-complete problem. It consists of three sets–X, Y and Z–of size n, and a set of n triples W, each of which is expressed as $\beta = (x_i, y_j, z_k)$. We construct a vector of $3n$ bits where the i^{th}, $(n+j)^{th}$, $(2n+k)^{th}$ bits are all 1 and all other bits are 0. In other words, to a certain radix $d > 1$, $f_\beta = d_{i-1} + d_{n+j-1} + d_{2n+k-1}$. In the case of update cost, according to the rule mentioned above, let m stand for the number of tuples in W; we construct f_β as base $m+1$, and for each triple $\beta = (x_i, y_j, z_k) \in W$. Let d^{f_a} be a vector with $3n$ bits, and let $m+1$ be the basis for it. Each bit of this vector is 1. Thus, we have $d^{f_a} = \sum_{i=0}^{3n-1} (m+1)^i$.

We state that a triple set W meets a perfect 3-Dimensional Matching if and only if there is a subset whose cardinality equals d^{f_a} in the set $\{f_\beta | \beta \in W\}$. Consider a perfect 3-Dimensional Matching consisting of triples $\beta_1, \cdots, \beta_n$. Then, the $3n$ bits of $f_{\beta_1} + \cdots + f_{\beta_n}$ are all 1, equaling d^{f_a}. On the contrary, assuming that there exists a set $\{f_{\beta_1}, \cdots, f_{\beta_n}\}$, whose sum equals d^{f_a}, each f_{β_i} contains three bits of 1 and no carry. Naturally, $k = n$. For each bit among the $3n$ bits, there exists one bit f_{β_i}, which is 1. Therefore, $\beta_1, \cdots, \beta_n$ form a perfect 3-Dimensional Matching. Then, identifying an optimal subset F_a from Formula (1) NP-complete. □

Appendix B
Proof for Theorem 2

The problem PE under the constraints of Formulas (2), (3) and (4) is NP-hard.

Proof: To show that problem PE is NP-hard, we can show that the associated recognition problem PE-r is NP-complete for instead. The problem PE-r is to decide whether the bandwidths occupied by the chosen parallel update events or by migrated traffic (caused by update events), satisfy the capacity limitation of each link $e_{i,j} \in E$. Note that we will migrate several existing flows to satisfy the bandwidth requirement of each update event. This is the certification to verify that there is a solution for this problem, i.e., at least an update event can be executed. Given a binary vector $x_k \in \{0,1\}^n$, we can verify in polynomial time that the residual bandwidths satisfy the bandwidth requirements of parallel update events. Thus, PE-r is NP.

We consider a case of SAT [38], which is a well-known NP-complete problem. It consists of n boolean variables $y_1, \cdots, y_n$ and $|v|$ clauses $z_1, \cdots, z_{|v|}$ (disjunctions over the boolean variables $y_1, \cdots, y_n$ and their complements $\bar{y_1}, \cdots, \bar{y_n}$). We can construct a particular instance of PE-r with n binary variables $x_1, \cdots, x_n$ and $|v|$ linear inequalities such that the instance of SAT has answer yes if and only if the corresponding instance of PE-r has answer yes. For each clause z_j , with $1 \leq j \leq |v|$, we construct a linear inequality as follows. If the k-th variable in clause z_j is y_i, the k-th term of the j-th constraint is x_i. If the k-th variable is $\bar{y_i}$, the k-th term is $(1 - x_i)$. Moreover, the disjunction operator in z_j is replaced with the addition operator.

Clearly, this is a one-to-one correspondence between the truth assignments of the boolean variables $y_1, \cdots, y_n$ which make all the clauses $(z_1, \cdots, z_{|v|})$ true and the binary vectors $\mathbf{x} = (x_1, \cdots, x_n)$ which satisfy all the linear inequalities of the corresponding ILP-r instance. Indeed, $y_i = true$ if and only if $x_i = 1$. Then, PE problem is NP-hard. □

Appendix C
Proof for Theorem 3

The approximation ratio of Algorithm 1 is at most 2.

Proof: In Algorithm 1, the value obtained in line 5 takes the negative number of the sum of a subset of F_A. Supposing P_i denotes a set, each of its different elements is the sum of a subset got from F_A. Let $y^* \in P_n$ denote one of the optimal solutions. We can infer from line 13 that $m^* \leq y^*$.

When we trim L_i, the relative error between the number before the trimming and the remaining, representative number is no more than ε/n. By inducting i, we can prove that there exists $m \in L_i$ for any element $y \in L_i$ that is no more than d^{f_a}. Thus, we can establish the following formula:

$$y/(1+\epsilon/n)^i \leq y/(1+\epsilon/2n)^i \leq m \leq y^*. \tag{5}$$

Inequality (6) is established for $y^* \in P_n$, and then $m \in L_n$ satisfies the following formula:

$$y^*/m \leq (1+\epsilon/2n)^n. \tag{6}$$

Consider that the value of m^* is the biggest element in L_i and that the involved migrated flows can be removed. Therefore, Inequality (7) is established for m^*. That is,

$$y^*/m^* \leq (1+\epsilon/2n)^n. \tag{7}$$

Obviously, $\lim\limits_{n\to\infty} (1+\epsilon/2n)^n = e^{\epsilon/2}$. In addition, we can prove that $\frac{d}{dn}(1+\epsilon/2n)^n > 0$. So the formula $(1+\epsilon/2n)^n$ is monotonically increasing. Thus,

$$(1+\epsilon/2n)^n \leq e^{\epsilon/2} \leq 1+\epsilon/2+(\epsilon/2)^2 \leq 1+\epsilon. \tag{8}$$

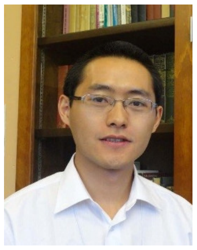
Therefore, from Inequality (7) and Inequality (8) we can conclude that $m^*/y^* \leq 1+\epsilon$. The approximation ratio of Algorithm 1 is at most 2. □

Ting Qu received her B.Eng. degree in computer science from Xidian University, Xian, China, in 2014. She is currently working toward her Ph.D. at the College of System Enginneering, National University of Defense Technology, Changsha, China. Her research interests include datacenter networks, software defined networks and programmable data plane.

Deke Guo received the B.S. degree in industry engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008. He is currently a Professor with the College of System Engineering, National University of Defense Technology, and is also with the College of Intelligence and Computing, Tianjin University. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a senior member of the IEEE since 2005 and a member of the ACM.

Jie Wu is the chair of the Department of Computer and Information Sciences at Temple University. Prior to joining Temple University, he was a program director at the US National Science Foundation. His research interests include wireless networks and mobile computing, routing protocols, fault-tolerant computing, and interconnection networks. He has published more than 450 papers in various journals and conference proceedings. He has served as an IEEE Computer Society distinguished visitor, and he is the chairman of the IEEE Technical Committee on Distributed Processing (TCDP). He is an IEEE Fellow since 2005.

Xiaolei Zhou is an assistant professor of the Sixty-third Research Institute at the National University of Defense Technology, Nanjing, China. He received his BA degree in information management and information systems from Nanjing University in 2009, and his ME and PhD degrees in operational research from National University of Defense Technology in 2011 and 2016, respectively. His research interests include mobile and pervasive computing, mobile crowd sensing.



Xin Lu received the B.S. degree in Management Science from Sichuan University, Sichuan, China, in 2006, and the Ph.D degree in Medical Science from Karolinska Institutet, Stockholm, Sweden, in 2013. He is currently working as an associate professor at the College of System Engineering at National University of Defense Technology, Changsha, China. His research interests include big data analytics, collective human behavior, complex networks and emergency response.

Zhong Liu received his B.S. degree in physics from Central China Normal University, Wuhan, Hubei, China, in 1990 and his Ph.D. in management science and engineering from the National University of Defense Technology, Changsha, China, in 2000. He is a professor with the College of System Engineering, National University of Defense Technology, Changsha, China. His research interests include information systems, cloud computing, and big data.