Minimize the Make-span of Batched Requests for FPGA Pooling in Cloud Computing

Yangming Zhao, Chen Tian, Zhuangdi Zhu, Jie Cheng, Chunming Qiao, *Fellow, IEEE*, and Alex X. Liu

Abstract—Using FPGA as accelerators is gaining popularity in Cloud computing. Usually, FPGA accelerators in a datacenter are managed as a single resource pool. By issuing a request to this pool, a tenant can transparently access FPGA resources. FPGA requests usually arrive in batches. The objective of scheduling is to minimize the make-span of a given batch of requests, which is the completion time of the entire batch of jobs. As a result, either the responsiveness is improved, or the system throughput is maximized. The key technical challenge is the existence of multiple resource bottlenecks. An FPGA job can be bottlenecked by either computation (i.e., computation-intensive) or network (i.e., network-intensive), and sometimes by both. To the best of our knowledge, this is the first work that minimizes the make-span of batched requests for an FPGA accelerator pool in Cloud computing that considers multiple resource bottlenecks. In this paper, we design several scheduling algorithms to address the challenge. We implement our scheduling algorithms in an IBM Cloud system. We conduct extensive evaluations on both a small scale testbed and a large-scale simulator. Compared with the Shortest-Job-First scheduling, our algorithms can reduce the make-span by 36.25 percent, and improve the system throughput by 36.05 percent.

Index Terms—Cloud computing, FPGA accelerators, job scheduling

1 Introduction

Motivation. FPGA accelerators have become crucial for Cloud computing. In current Cloud datacenters, CPU resources are no longer adequate for many applications, especially for large-scale machine learning tasks. Leading providers/researchers start to integrate various FPGA/GPU accelerators in their platforms [1], [2], [3], [4], [5], [6], [7], [8]. Compared with CPU, these accelerators can significantly boost the performance of many computation-intensive tasks, such as matrix computation, encryption, and signal processing [9], [10], [11], [12]. For many application scenarios, FPGA is more promising than GPU due to its low cost (i.e., hundreds instead of thousands of dollars per piece), low power footprint (i.e., tens instead of hundreds of Watts per piece) and high power efficiency (i.e., 2-3x more Gflops than GPU per Watt) [2].

FPGA accelerators in a datacenter are usually managed as a single resource pool [8]. In such a datacenter, the operator installs FPGA devices in a portion of the server farm due to cost and deployment constraints. Following the Software-as-a-Service (SaaS) model, tenant programs interact with the Cloud by calling the API functions provided by an FPGA service layer. By issuing a request to this layer, a tenant can transparently access FPGA resources. A centralized scheduler maintains status information of each FPGA node. It assigns job requests to accelerators in the resource pool in an online fashion. Tenants are agnostic to the control and status of FPGA accelerators [13].

An application operation usually triggers a large number of computation requests simultaneously for (FPGA) accelerators. For example, the processing of Online Data Intensive applications (OLDI) and real-time analytics involve a multi-tier split-aggregate workflow, and a single process call triggers a large number of computation tasks [14]. Some streaming data processing systems are even batch-based in nature. For example, Spark Streaming aggregates a batch of requests and submits them together for processing [15]. In this paper, we study how to schedule FPGA accelerators when job requests come in batches.

The objective of FPGA accelerator scheduling is to minimize the make-span of a given batch of FPGA requests. Make-span is the time to complete all job requests in a single batch. A new batch of requests is considered as completed only after the completion of the last task of this batch. For continuous systems (e.g., Spark with FPGA [16]), minimizing the make-span leads to maximized system throughput. For periodical batch scheduling mode (e.g., Spark Streaming), minimizing the make-span leads to minimized missing ratio of application deadlines.

Challenges. The key technical challenge is to address multiple resource bottlenecks. An FPGA job can still be bottlenecked by computation. For example, 12 MB of photos

- *Y. Zhao is with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210008, China, and also with the Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY 14260. E-mail: zhaoyangming.uestc@gmail.com.*
- *C. Tian is with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210008, China. E-mail: tianchen@nju.edu.cn.*
- *Z. Zhu and A.X. Liu are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824. E-mail: zhuzhuan@msu.edu, alexliu@cse.msu.edu.*
- *J. Cheng is with Huawei Technologies Co., Ltd., Shenzhen 518129, China. E-mail: jiecheng2009@gmail.com.*
- *C. Qiao is with Department of Computer Science and Engineering, the State University of New York at Buffalo, Buffalo, NY 14260. E-mail: qiao@computer.org.*

Manuscript received 1 Dec. 2017; revised 6 Mar. 2018; accepted 11 Apr. 2018. Date of publication 24 Apr. 2018; date of current version 10 Oct. 2018.
(Corresponding author: Chen Tian.)
Recommended for acceptance by B. He.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2829860

(2 KB each) need about 1 second to be processed by an FPGA accelerator of deep neural network (DNN) [17]. If equipped with a 10/40 Gbps fast network, the communication cost of this job can be negligible. We call such FPGA requests *computation-intensive*. On the other hand, some jobs can be processed by FPGA at line rate. The I/O bottleneck is usually the network, since PCIE bandwidth is much higher. We call such FPGA requests *network-intensive*. A typical network-intensive example is AES encryption [18]. Sometimes, both network and computation can be bottlenecks. Stick to the DNN example, if the network provides only 1 Gbps per host (as in some legacy datacenters), the network transfer time cannot be ignored any more.

To the best of our knowledge, this is the first work that minimizes the make-span of batched requests for an FPGA accelerator pool in Cloud computing and considers different kinds of resource bottlenecks. The work closest to ours is done by Julio et al. [13]. However, their work performs admission control for an FPGA resource pool, whereas we provide services to all requests and minimize the make-span.

Contributions. In this paper, we design several scheduling algorithms to address the challenge. First, we formulate the scheduling of computation-intensive FPGA requests as a parallel machine scheduling problem, which is a well-known NP-hard problem. A polynomial time approximation algorithm is given, with a proven approximation ratio of 2. Second, we formulate the scheduling of network-intensive FPGA requests as a mixed integer programming problem, which is also NP-hard. The network-intensive case is similar to multiprocessor scheduling [19]. To the best of our knowledge, however, this work is the first one which considers the existence of network bottlenecks at both the sender and the receiver sides and yields a 2-approximation algorithm. At last, we extend the algorithms to support cases that both data transmission phase and the computation phase are bottlenecked.

We implement our algorithms in an IBM Cloud system. This system achieves full FPGA virtualization and pooling on an OpenStack-based cloud. We conducted extensive evaluations on both a small scale testbed and a large-scale simulator. Compared with the Shortest Job First (SJF) scheduling, our algorithms reduce the make-span by 36.25 percent, while improve the system throughput by 36.05 percent.

2 System Overview

Our targeted Cloud system is a FIFO system. In this system, *Responder* hosts carry heterogeneous FPGA accelerators, and *Requester* hosts issue FPGA requests. A host can play the roles of both a responder and a requester. All physical machines are connected by a communication network. We assume this communication network has a non-blocking topology such as Fattree [20] and VL2 [21], and network bottleneck only exists at the NIC of physical machines. As a result, we can control the data transmission rate at the NIC port of each physical machine for job scheduling purpose.

Each FPGA request contains three pieces of information: the job type, the input data size and the requester's location. Though we are considering a Cloud system with heterogeneous accelerators, a linear regression method can be used to predict the required processing time of each request on a particular FPGA hardware [13]. It should be noted that in an FPGA accelerator system, a job may contain many individual tasks, since it is not necessary to send a small task to the FPGA accelerator. For example, a DNN request contains hundreds of photos. Though we cannot accurately predict the processing time of one photo, the processing time of hundreds of photos is stable. In addition, we will demonstrate that our scheduling schemes are robust to the prediction error through simulations.

When a batch of requests arrive, a centralized scheduler aggregates all job information, and assigns job requests to accelerators in the resources pool following the algorithms proposed later. In our current system, we only consider jobs that have small outputs, such as word counting, DNN and numerical average. Accordingly, we do not consider the traffic sent back to the requesters. How to schedule the traffic replying to the requesters will be left for our future work.

3 Scheduling Algorithms

For clarity of presentation, in this part, we first use simplified settings: 1) for each case there is only one type of jobs to be scheduled; 2) there is only one accelerator carried by each responder host; 3) requester hosts and responder hosts are not overlapped; 4) the logic reconfiguration time of each FPGA accelerator can be ignored; 5) only one batch of requests are processed in the system and a batch of requests should be started simultaneously. As such, we propose scheduling algorithms for computation-intensive and network-intensive jobs in Section 3.1 and 3.2, respectively. Section 3.3 proposes an algorithm considering both computation and transmission. In Section 3.4, we discuss how to apply the scheduling algorithms to an online system where requests arrive sequentially. At last, we remove these simplified assumptions and extend our scheduling algorithms to cover the most general cases in Section 3.5.

3.1 Computation-Intensive Case

For computation-intensive jobs, the communication cost is negligible and hence we do not focus on by which requester host each job is issued. In this case, we assume there are M responder hosts and J jobs generated by all requester hosts. For each job i to be accelerated, the computation time on responder host j can be estimated as e_{ij}. Due to heterogeneity among FPGA hardware, there could be $e_{ij_1} \neq e_{ij_2}$ when $j_1 \neq j_2$. Based on above assumptions, minimize the make-span for computation-intensive jobs can be modeled as the following Computation Intensive Model (CIM):

$$\text{minimize} \quad T \tag{1}$$

Subject to:

$$\sum_i e_{ij} x_{ij} \leq T, \quad \forall j \tag{1a}$$

$$\sum_j x_{ij} = 1, \quad \forall i \tag{1b}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j. \tag{1c}$$

The objective is to minimize the make-span of all the jobs, denoted by T. Constraint (1a) says that any responder host j should complete all the jobs assigned to it before the make-span T. Constraints (1b) and (1c) indicate that any job i must and can only be placed on one responder host. In fact, this is a parallel machine scheduling problem, which is a well-known NP-hard problem [22] and is intractable in large scale systems due to the binary variable constraint (1c). Therefore, we need an efficient approximation algorithm to solve it.

A common method to solve this binary variable problem is relaxation and rounding. By treating x_{ij} as real variables on $[0,1]$ and solving the resulting Linear Programming (LP) model, we can derive a feasible scheduling scheme based on the solution of the relaxed CIM model. For example, we can assign job i to responder host j if $j = \arg_j \max\{x_{ij}\}$. However, such simple relaxation and rounding method may not derive a good solution and the result can be extremely bad in large scale systems, since a job can be split arbitrarily and assigned to any responder host. Especially, when the optimal make-span is T^* in the relaxed CIM model, the solution can still assign a fraction of a job i to responder host j, even if $e_{ij} > T^*$.

Based on above discussions, we find that we can at least prevent assigning a job i to a responder host j such that $e_{ij} > T$, where T is the make-span, when we derive a relaxation of the CIM. Suppose $E_j(T)$ is the set of jobs that can be completed on responder host j in time T, and $H_i(T)$ is the set of responder hosts that can complete job i in time T, we propose the following relaxation model named CIM-LP:

$$\text{minimize} \qquad T \tag{1L}$$

Subject to:

$$\sum_{i \in E_j(T)} e_{ij} x_{ij} \le T, \quad \forall j \tag{1aL}$$

$$\sum_{j \in H_i(T)} x_{ij} = 1, \quad \forall i \tag{1bL}$$

$$x_{ij} \ge 0, \quad \forall i, j \in H_i(T). \tag{1cL}$$

Compared with relaxing constraint (1c) directly, CIM-LP prevents a job i from being assigned to a responder host that cannot complete it in T even if only job i is assigned to this responder host. Accordingly, the objective value of CIM-LP should be closer to that of CIM than the objective derived by directly relaxing constraint (1c). In addition, the solution $\{x_{ij}\}$ of CIM-LP provides a better indication to round a job scheduling scheme as it never assigns job i to a responder host j that cannot complete job i before the make-span even if job i is the only job assigned to responder host j. This is an important characteristic to prove the approximation ratio of our algorithm.

When T is the objective to minimize (i.e., T is a variable), CIM-LP is not a convex optimization problem, and hence it is difficult to solve directly. However, by fixing T, all the constraints become linear and there are efficient algorithms to check the problem feasibility. Accordingly, we can use the binary search to find the minimum T that can make CIM-LP feasible, which is the objective value of CIM-LP. This algorithm is shown in Algorithm 1. In this algorithm, ϵ is the required accuracy, $t_{\max}$ and $t_{\min}$ are the upper and lower bound of make-span, respectively, which are obtained by Lines 1–2. This algorithm can get the minimum make-span in $\lceil \log \frac{t_{\max} - t_{\min}}{\epsilon} \rceil$ iterations. It should be noted that Algorithm 1 returns $t_{\max}$ which must be a feasible solution, rather than $t_{\min}$ though it is very close to $t_{\max}$.

Algorithm 1. Minimize Make-Span for Splittable Jobs

```
Input: The running time of each job on different machines $\{e_{ij}\}$
Output: Minimal job make-span
 1: Initialize $t_{\max}$ to be the job make-span based on a random
    job assignment
 2: $t_{\min} \leftarrow \max_i \min_j \{e_{ij}\}$
 3: while $t_{\max} - t_{\min} \ge \epsilon$ do
 4:   $t \leftarrow (t_{\max} + t_{\min})/2$
 5:   if CIM-LP is feasible when $T = t$ then
 6:     $t_{\max} \leftarrow t$
 7:   else
 8:     $t_{\min} \leftarrow t$
 9:   end if
10: end while
11: return $t_{\max}$
```

To achieve the make-span derived by solving CIM-LP, a job may be split onto multiple responder hosts. Accordingly, we need to round the solution derived by solving CIM-LP to allocate every job to a unique responder host. To this end, we first propose the following lemma that can guide our algorithm design.

Lemma 1. *Suppose there are J jobs and M responder hosts in the system, then at most (J+M) variables will be non-zero in the optimal solution of CIM-LP.*

Proof. Assume v is the number of variables in CIM-LP when T is fixed, which is also the number of constraints in (1*cL*). When T is the minimum value that makes CIM-LP feasible, the feasible region is a single point determined by v linearly independent constraints, such that each of these constraints is satisfied with the equality.

Consider that there are $v + M + J$ constraints in CIM-LP, but only M constrains in (1*aL*), J constraints in (1*bL*). Accordingly, there are at least $v - J - M$ constraints in (1*cL*) that hold the equality. Hereby, at most $J + M$ constraints in (1*cL*) do not hold equality. It means that at most $J + M$ variables take non-zero value in the optimal solution. □

From Lemma 1, we have the following corollary.

Corollary 1. *We construct a bigraph $BG = \{U, V, E\}$ according to the solution of CIM-LP, x. $U = \{u_1, u_2, \ldots, u_M\}$ is the set of nodes denoting responder hosts, called responder nodes, while $V = \{v_1, v_2, \ldots, v_J\}$ is the set of nodes denoting jobs, called job nodes. There is an edge between v_i and u_j, if and only if $x_{ij} > 0$. In this case, any connected component, P, in BG can be modified to a pseudo tree (a tree or a tree plus one edge) without increasing make-span.*

Without ambiguity, we say job (responder host) v instead of the job (responder host) associating with node v hereafter for brevity.

Proof. Say if there are a connected component P in BG, such that P is not a pseudo tree, then we solve CIM-LP by only using the jobs and responder hosts associated with P, say the solution is x'. It is obvious that the make-span of the jobs in P under the job scheduling associated with x' should be smaller than or equal to that derived by using all the jobs and responder hosts. According to Lemma 1, the non-zero variable number in the solution is at most the number of nodes in P. Repeat this procedure, we can ensure that the number of edges in every connected component P in BM is at most the number of nodes in P. Therefore, any connected component P in BG can be modified to be a pseudo tree (a tree or a tree plus one edge) without increasing make-span. □

Algorithm 2. Job Assignment

```
Input: The solution of CIM-LP $\{x_{ij}\}$
Output: Job assignment
 1: Construct a bigraph $BG$ according to $\{x_{ij}\}$ as in Corollary 1,
    modify every connected component to be a pseudo tree,
    and update CIM-LP solution based on this modification
 2: Remove all the job nodes with only one node degree and
    place these jobs to the connecting responder host
 3: for all connected components $P \in BG$ do
 4:   if $|N(P)| = |L(P)|$ then
 5:     Find the unique cycle in $P$ with depth first search
 6:     Arbitrarily orient the cycle in one direction and assign
        each job to the responder host succeeding it on the
        cycle
 7:     Remove this cycle from $P$, and what remains overall is
        a forest of trees, each of which contains at most one job
        leaf node
 8:     for all the remaining trees do
 9:       Rooting at the unique job leaf node (if there is), or
          arbitrary job node
10:       Assign each job to its child responder host that
          services most fraction of this job
11:     end for
12:   else
13:     Treat arbitrary job as the root to form a tree and assign
        each job to its child responder host that services most
        fraction of this job
14:   end if
15: end for
```

Based on Corollary 1, Algorithm 2 places each job to a unique responder host. The key idea of this algorithm is to ensure that *each responder host serves at most one job that is split to multiple responder hosts according to the solution of CIM-LP*.

In this algorithm, Line 2 deals with the jobs that have only one responder host to place. After that, every remaining job is split to at least two responder hosts. For each connected component $P \in BG$, if $|N(P)| = |L(P)|$, where $N(P)$ is the set of nodes in P and $L(P)$ is the set of edges in P, there must be a cycle in P. In this case, we first find out this cycle by Depth-First Search (DFS), and determine the job assignment on this cycle (Line 6). This is to ensure every responder host on the cycle only serves one split job.

By removing this cycle from P, there must be a forest of trees left, each of which contains at most one job leaf node. (Job leaf nodes might be created upon the deletion of the cycles, but there will be at most one such leaf per resulting tree.) If there is a job leaf node on the resulting tree, we can root the tree at this job leaf node, and assign the job to its child responder host that serves most fraction of this job. Otherwise, we root the tree at arbitrary job node and assign each job to one of its child responsder hosts (Lines 8-11). In this way, *each responder host can get at most one job split to multiple responder hosts according to the solution of CIM-LP*. When P is a tree, the job assignment method for the resulting tree without job leaf node can be adopted. We treat arbitrary job node as the root to form a tree and assign each job to its child responder host that serves most fraction of this job. With above job assignment, any responder host gets at most one job that is split to multiple responder hosts according to the solution of CIM-LP.

The performance of Algorithm 2 is guaranteed by the following theorem:

Theorem 1. *The make-span derived by Algorithm 2 is at most two times of the solution of CIM-LP. Since CIM-LP yields a lower bound of CIM, the approximation ratio of Algorithm 2 is 2.*

Proof. Suppose T is the objective value of CIM-LP, if all the jobs are splittable, the solution of CIM-LP $\{x_{ij}\}$ ensures that all the jobs can be completed in time T. In Line 2, we deal with all the jobs that are not split according to the solution of CIM-LP, and the make-span formed by these jobs must be smaller than or equal to T. From Lines 3–15, we ensure that every responder host accommodates at most one more job. Since each job will only be assigned to one responder host that can finish it in time T, the make-span is at most $2T$. □

3.2 Network-Intensive Case

Some jobs (e.g., word counting) are bottlenecked by the network transfer. In this case, we should determine not only the allocation of each job, but also the job data transmission rate to minimize the make-span. This problem can be formulated as following Network-Intensive Model (NIM):

$$\text{minimize} \qquad T \tag{2}$$

Subject to:

$$\sum_i f_{ij}(t) \le B_j^{in}, \quad \forall j, t \tag{2a}$$

$$\sum_j \sum_{i:g(i)=u} f_{ij}(t) \le B_u^{out}, \quad \forall u, t \tag{2b}$$

$$\sum_j \int_0^T f_{ij}(t)dt = s_i, \quad \forall i \tag{2c}$$

$$f_{ij}(t) \le x_{ij} B_j^{in}, \quad \forall i, j, t \tag{2d}$$

$$\sum_j x_{ij} = 1, \quad \forall i, \tag{2e}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j. \tag{2f}$$

In this model, the first constraint is used to limit the ingress rate of each responder host, where B_j^{in} is the ingress rate of responder host j, and $f_{ij}(t)$ is the flow rate of job i to responder host j at time t. (2b) is used to limit the egress rate of each requester host u, where $g(i)$ is the requester host that issues job i and B_u^{out} is the egress rate limitation of requester host u. With s_i denoting the volume of job i, (2c) means that all the data of job i should be sent out before the make-span. Constraints (2d)–(2f) are used to indicate that every job should be assigned to one and only one responder host.

NIM is NP-hard even if there is only one requester host [23]. It is difficult to solve because: 1) the objective, which is also a variable, is the upper bound of the integration in constraint (2c); and 2) $\{x_{ij}\}$ are binary variables. To address the first issue, we propose the following theorem:

Theorem 2. *Suppose f_{ij}^* and x_{ij}^* are the solutions, and f^* is the objective value of the following optimization problem named Network-Intensive Model with Rate Control (NIM-RC):*

$$\text{maximize} \qquad f \tag{3}$$

Subject to:

$$\sum_i f_{ij} \le B_j^{in}, \quad \forall j \tag{3a}$$

$$\sum_j \sum_{i:g(i)=u} f_{ij} \le B_u^{out}, \quad \forall u \tag{3b}$$

$$\sum_j f_{ij} = s_i f, \quad \forall i \tag{3c}$$

$$f_{ij} \le x_{ij} B_j^{in}, \quad \forall i, j \tag{3d}$$

$$\sum_j x_{ij} = 1, \quad \forall i, \tag{3e}$$

$$x_{ij} \in \{0,1\}, \quad \forall i, j. \tag{3f}$$

then, $T = \frac{1}{f^*}$ is the optimal objective value of NIM. $f_{ij}(t) = \begin{cases} f_{ij}^* & \text{for } t \in (0, \frac{1}{f^*}) \\ 0 & \text{for } t \in (\frac{1}{f^*}, \infty) \end{cases}$, and $x_{ij} = x_{ij}^*$ are the solutions to achieve the optimal objective value.

Proof. Suppose T_{opt} is the optimal objective of (2), and $f_{ij}^{opt}(t)$ is the corresponding solution, then by setting

$$f_{ij} = \frac{\int_0^{T_{opt}} f_{ij}^{opt}(t)dt}{T_{opt}}$$

we have,

$$\sum_i \int_0^{T_{opt}} f_{ij}^{opt}(t)dt = \sum_i f_{ij} T_{opt}.$$

Since

$$\sum_i \int_0^{T_{opt}} f_{ij}^{opt}(t)dt = \int_0^{T_{opt}} \sum_i f_{ij}^{opt}(t)dt \le \int_0^{T_{opt}} B_j^{in} dt = T_{opt} B_j^{in}$$

we know $\sum_i f_{ij} \le B_j^{in}$. In the same way, we can verify that f_{ij} also satisfies constraint (3b) and (3d).

For constraint (3c), we can see that

$$\sum_j \int_0^{T_{opt}} f_{ij}^{opt}(t)dt = \sum_j f_{ij} T_{opt} = s_i.$$

Let $f = \frac{1}{T_{opt}}$, we get

$$\sum_j f_{ij} = \frac{s_i}{T_{opt}} = s_i f.$$

Above discussion shows that $f_{ij} = \frac{\int_0^{T_{opt}} f_{ij}^{opt}(t)dt}{T_{opt}}$ and $f = \frac{1}{T_{opt}}$ is a feasible solution of NIM-RC. Accordingly,

$$f^* \ge \frac{1}{T_{opt}}.$$

In addition, we can easily verify that the variable settings claimed in Theorem 2 is a feasible solution of NIM. Therefore, we have

$$T_{opt} \le \frac{1}{f^*}.$$

Accordingly, we have $T_{opt} = \frac{1}{f^*}$ □

Theorem 2 provides a way to eliminate the variable in the integration upper bound of constraint (2*c*). Below, we discuss how to solve the binary variables in the NIM-RC. First, we combine all the responder hosts as a big one, the optimization problem can be formulated as the following Network-Intensive Model with One Responder (NIM-OR):

$$\text{maximize} \qquad f \tag{4}$$

Subject to:

$$\sum_i f_i \le \sum_j B_j^{in} \tag{4a}$$

$$\sum_{i:g(i)=u} f_i \le B_u^{out}, \quad \forall u \tag{4b}$$

$$f_i = s_i f, \quad \forall i, \tag{4c}$$

where f_i is the transmission rate of job i. NIM-OR is a linear programming problem that is easy to solve. After solving NIM-OR, we derive the maximum transmission rate of each job. When we place all these jobs to different responder hosts, we should scale down the transmission rate. If the scale-down ratio is r, i.e., we can transmit any job with rate $\frac{f_i}{r}$, to minimize the make-span, we should minimize the scale-down ratio, i.e.,

$$\text{minimize} \qquad r \tag{5}$$

Subject to:

$$\sum_i f_i x_{ij} \le r B_j^{in}, \quad \forall j \tag{5a}$$

$$\sum_j x_{ij} = 1, \quad \forall i \tag{5b}$$

$$x_{ij} \in \{0,1\}, \quad \forall i, j. \tag{5c}$$

By defining $e_{ij} = \frac{f_i}{B_j^{in}}$, the model (5) is modified to be CIM, which can be solved by Algorithm 2 with approximation ratio 2. Based on the above discussions, we design Algorithm 3 to schedule the network-intensive jobs in the system. We first determine the flow transmission rate to minimize the make-span by treating all the responder hosts as one big responder host (Line 1). And then assign the jobs to different responder hosts by pursuing the minimum scale-down ratio (Line 2). In Lines 3–8, we scale down the job transmission rate and assign them to responder hosts. However, we do not scale down all the job transmission rates with the same ratio. Instead, we scale down the transmission rate of jobs to different responder hosts by different ratios to fully utilize the responder hosts' ingress bandwidth. Though it cannot further reduce the make-span, it speeds up the completion of some jobs without impacting the others. It should be noted that the scale-down ratio is smaller than 1 means we should scale up the transmission rate of some jobs to particular responder hosts to fully utilize the ingress bandwidth. In this case, the egress bandwidth on the requester side should be the bottleneck, and hence we maintain the data transmission rate. Accordingly, we only scale down the transmission rate of flows that should be sent to the responder hosts whose bandwidth is the bottleneck of the network (Lines 5 – 7).

Following theorem shows the performance of Algorithm 3.

Algorithm 3. Minimize Network-Intensive Job Make-Span

```
Input: The size of each job $s_i$, egress rate of each requester host
    $\{B_u^{out}\}$, ingress rate of each responder host $\{B_j^{in}\}$
Output: Job transmission rate $\{f_{ij}\}$
1: Formulate and solve NIM-OR (the solution is $\{f_i\}$)
2: Formulate model (5) and solve it with Algorithm 2, say the
   solution is $x_{ij}$
3: for all responder host $j$ do
4:     $f_{ij} \leftarrow f_i x_{ij}$, $r_j \leftarrow \frac{\sum_i f_{ij}}{B_j^{in}}$
5:     if $r_j > 1$ then
6:         $f_{ij} \leftarrow \frac{f_{ij}}{r_j}$
7:     end if
8: end for
9: return $\{f_{ij}\}$
```

Theorem 3. *The approximation ratio of Algorithm 3 is 2.*

Proof. There are two cascaded bottlenecks in the network-intensive case: the egress bandwidth on the requester host side and the ingress bandwidth on the responder host side. Say f and r are the optimal objective value of NIM-OR and (5), respectively, the the optimal make-span should be $\frac{1}{f}\max\{1, r\}$. When $r > 1$, the bottleneck is on the responder hosts side, while the bottleneck is the egress bandwidth of requester hosts if $r \leq 1$. As the optimal value of f can be obtained by NIM-OR which is a linear programming model, and Algorithm 1 and 2 can solve (5) with approximation ratio 2, the make-span derived by Algorithm 3 is at most $\frac{1}{f}\max\{1, 2r\}$. Accordingly, the make-span derived by Algorithm 3 is at most 2 times of the optimal make-span. □

It is worth noting that Algorithm 3 may leave some residual bandwidth, especially when the transmission rate of some jobs is scaled down. Though we cannot further reduce the make-span by fully utilizing such residual bandwidth, it can be used to speed up of some the jobs, which may in turn improve another metric: average job completion time. To this end, we assign the residual bandwidth to the jobs following the shortest job first principle.

3.3 Consider Both Transmission and Computation

So far, we only consider the case that either computation or the network transmission is the bottleneck of a job. However, there are some jobs of which neither the transmission nor the computation can be ignored. In this case, the problem is much more complicated. A key observation is that: we can overlap a job's data transmission with the other jobs' computation phase. Accordingly, we develop a heuristic that first determines the placement of each job and then tries to overlap the transmission and the computation phase of different jobs to minimize the make-span.

To determine the job assignment, we leverage the Algorithm 2 to optimize the make-span, without considering the transmission phase. After that, we leverage Shortest Job First scheme to determine job execution order on each responder host. Then, we minimize the time gap between the computation of any two consecutive jobs by controlling the job transmission rate.

Let b_{ik} denote the time to start the computation phase of $k\text{th}$ job on responder host i, i_k denote the $k\text{th}$ job on responder host i and $N(i)$ denote the number of jobs that are assigned to responder host i, the problem to minimize the make-span can be formulated as the following Network-Computation Joint Model (NCJM) if neither the job computation nor the data transmission can be ignored:

$$\text{minimize} \qquad T \tag{6}$$

$$\sum_k f_{i_k i}(t) \leq B_i^{in}, \quad \forall i \tag{6a}$$

$$\sum_i \sum_{k:g(i_k)=u} f_{i_k i}(t) \leq B_u^{out}, \quad \forall u \tag{6b}$$

$$\int_0^{b_{ik}} f_{i_k i}(t)dt = s_{i_k}, \quad \forall i, k \tag{6c}$$

$$b_{ik} + e_{i_k i} \leq b_{i_{k+1} i}, \quad \forall i, k \tag{6d}$$

$$b_{iN(i)} + e_{i_{N(i)} i} \leq T, \quad \forall i. \tag{6e}$$

The first three constraints are the same as that of the first three constraints of NIM. Constraint (*6d*) says the computation phase of $(k+1)\text{th}$ job cannot start before the completion of the $k\text{th}$ job's computation phase; the last constraint means the make-span is larger than the completion time of the last job on any responder host. The ingredient making this problem difficult to solve is that the integration upper bound of constraint (*6c*) is a variable but it cannot be solved with the same technology in Algorithm 3; we design a heuristic Algorithm 4 to solve NCJM.

In Algorithm 4, we first determine the scheduling (both placement and order) of job computation in Lines 1 and 2 with Algorithm 2 and SJF scheme. After that, we minimize the gap between two consecutive jobs on each responder host by optimizing the job transmission. Starting at Line 3, $d_{i_k i}$ is the earliest time for the $k\text{th}$ job on responder host i to start its computation phase. Therefore, we send the data for the job that can start computation earliest first and try to catch up with the earliest starting time (Line 5). To this end, we first check if there is enough bandwidth to send the entire job i_k to the responder host before $d_{i_k i}$. If so, we complete the job transmission exactly before $d_{i_k i}$, in order to minimize the bandwidth consumption (Line 7). Otherwise, we complete the transmission as soon as possible (Lines 9 and 10). This is to minimize the gap between the computation of two consecutive jobs on a responder host. The minimum time that is required to complete the job transmission can be derived by well-known water filling algorithm [24]. Since the computation phases of some jobs are delayed, we update the earliest starting time of the computation phases of later jobs (Line 4). At last, we update the available ingress and egress bandwidth of each host, since it has been assigned to transmit more jobs.

Algorithm 4. Minimize Make-Span Considering Both Transmission and Computation

```
Input: The size of each job $s_i$, egress rate of each requester host
    $\{B_u^{out}\}$, ingress rate of each responder host $\{B_j^{in}\}$, the
    computation time of each job on different responder
    host $\{e_{ij}\}$
Output: Job assignment $\{x_{ij}\}$ and job transmission scheduling
    $\{f_{ij}(t)\}$
 1: Using Algorithm 2 to determine the job assignment by only
    considering the computation, and get $\{x_{ij}\}$
 2: Schedule all the jobs on each host based on SJF to determine
    the execution order
 3: Initialize $d_{i_k i} \leftarrow \sum_{p<k} e_{i_p i}$, $R_i^{in}(t) \leftarrow B_i^{in}$, $R_i^{out}(t) \leftarrow B_i^{out}$, and
    $S$ as all the jobs in the system
 4: while $S \neq \Phi$ do
 5:     $\{i,k\} \leftarrow \arg\min_{i,k} d_{i_k i}$,
        $s \leftarrow \int_0^{d_{i_k i}} \min\{R_{g(i_k)}^{out}(t), R_i^{out}(t)\}$
 6:     if $s \geq s_{i_k}$ then
 7:         $f_{i_k i}(t) = \min\{R_{g(i_k)}^{out}(t), R_i^{out}(t)\} \times \frac{s_{i_k}}{s}$ for
            $t \in [0, d_{i_k i}]$, and $f_{i_k i}(t) = 0$ for $t > d_{i_k i}$
 8:     else
 9:         With water filling algorithm [24], find the $T$, such that
            $\int_0^T \min\{R_{g(i_k)}^{out}(t), R_i^{out}(t)\} = s_{i_k}$
10:         $f_{i_k i}(t) = \min\{R_{g(i_k)}^{out}(t), R_i^{out}(t)\}$ for $t \in [0,T]$, and
            $f_{i_k i}(t) = 0$ for $t > T$
11:         $d_{i_p i} \leftarrow d_{i_p i} + (T - d_{i_k i})$ for $p > k$
12:     end if
13:     $R_i^{in}(t) \leftarrow R_i^{in}(t) - f_{i_k i}(t)$,
        $R_{g(i_k)}^{out}(t) \leftarrow R_{g(i_k)}^{out}(t) - f_{i_k i}(t)$
14:     $S \leftarrow S - i_k$
15: end while
16: return $\{x_{ij}\}$ and $\{f_{ij}(t)\}$
```

3.4 Online Scheduling of Batched Requests

In previous subsections, we assume a batch of requests come into an empty system. However, in practical systems, there are many batches of requests arrive sequentially. In this section, we present how to schedule a batch of requests when there are some uncompleted jobs in the system.

For computation-intensive jobs, the make-span of jobs coming later will be impacted by the previous batches as not all the jobs can be started at the time "0". To solve this problem, we first estimate the completion time of previous jobs on each responder host based on previous scheduling. Assume c_j is the completion time of the last job on responder host j, the (1*a*) in CIM should be changed to be

$$c_j + \sum_i e_{ij} x_{ij} \leq T, \quad \forall j. \qquad (1a')$$

Correspondingly, in CIM-LP, the $E_j(T)$ should be defined as the set of jobs i that can be completed on host j in time $c_j + e_{ij}$, while $H_i(T)$ is the set of hosts j that can complete job i in time $c_j + e_{ij}$. Then Algorithm 2 still works to derive a 2-approximation solution.

When a batch of network-intensive jobs come and there are some uncompleted jobs in the system, they can try to utilize the bandwidth left by the previous batches of jobs to improve its make-span. For simplicity, in our system, we try two schemes: 1) waiting for the completion of previous jobs and calculating the scheduling with Algorithm 3; 2) calculate the scheduling with Algorithm 3 with the remaining bandwidth when the new batch of jobs arrive. The scheduler enforces one of above two schemes with the smaller make-span into the system.

At last, the Algorithm 4 can be directly applied to the case that there are already some jobs in the system.

3.5 Discussions

Multiple Accelerators per Responder Host. Usually, an FPGA chip can be divided into multiple regions and partial reconfigured to support multiple accelerators simultaneously. In this case, a responder host may carry multiple accelerators. Though our algorithm discussion is based on the assumption that one host carries one accelerator, it can be extended to the case that each host carries multiple accelerators. For the computation-intensive case, we treat each accelerator as a responder host, and then Algorithm 2 still works. Algorithm 3 is applicable to the network-intensive case without any modification. For Algorithm 4, we should calculate the earliest starting time of each job on the accelerator level, while scheduling the bandwidth on the host level.

Mixture of Job Types. When there are mixed types of jobs in the system, we first divide the jobs into multiple waves according to their types. We schedule computation-intensive jobs first and schedule network-intensive jobs last.

Mixture of Job Sources. Some jobs may be generated by responder hosts. For a computation-intensive job, it should be scheduled with Algorithm 2 directly, since its data transmission phase can be ignored. If it is a network-intensive job, it should be locally executed since it has a very light computation workload. When neither computation nor data transmission phase can be ignored, we execute these jobs locally for reducing the communication overhead.

Logic Reconfiguration. According to [25], the accelerator reconfiguration overhead is in the micro-second level, and we believe this is a small overhead. Even if this cannot be ignored, we can add the logic reconfiguration time into the job execution time before we run our scheduling algorithms.

Fairness Among Jobs. Algorithm 3 may greatly degrade the completion time of some small jobs. Fig. 1 shows such an

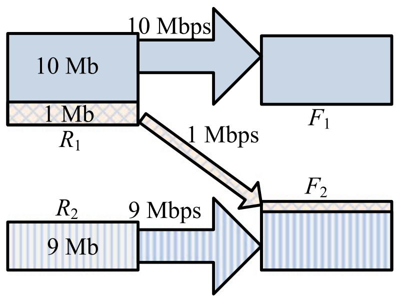

Fig. 1. Performance degradation for the small size job.

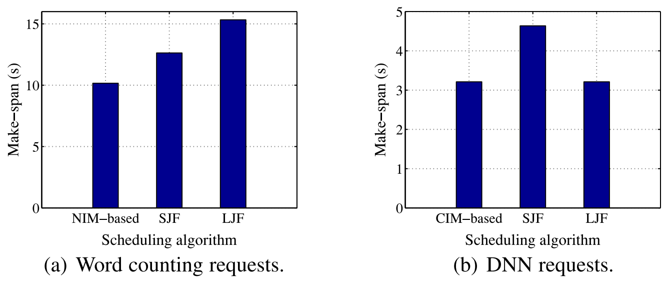

Fig. 2. Experiment results on testbed.

example. There are three jobs on two requester hosts, R_1 and R_2. One of them (R_1), whose egress rate is 11 Mbps, has two jobs with size 10 Mb and 1 Mb. And the other requester host (R_2) issues only one job with size 9 Mb, but its egress rate is 9 Mbps. There are two responder hosts (F_1 and F_2), whose ingress rate are both 10 Mbps. The optimal solution to minimize make-span is shown in Fig. 1, with which all the jobs are completed in 1 second. However, it is a too large latency for the job with only 1 Mb size to last for 1 second, since it requires only 0.1 second to finish if SJF scheme is adopted.

To solve this problem, we can divide all the jobs into multiple groups according to their size and schedule the group containing those smallest jobs first.

4 Testbed and Implementation

We implement the accelerator pooling system on IBM's OpenPower Cloud system testbed with 7 hosts. Two of them are implemented as responder hosts, each of which carries one FPGA accelerator. Four of them work as requester hosts to generate jobs, and the remaining one is used as the scheduler. All of these hosts are connected by a switch, and the NIC rate on each host is 1 Gbps.

Our scheduling algorithms are implemented on the scheduler with CPLEX 12.3 as the linear programming solver. The scheduler should maintain some prepared job information, such as the job type (network-intensive, computation-intensive, or both), and the relationship between the estimated computation time and job size. Whenever a batch of jobs arrive, the requester hosts send the job information to the scheduler; the scheduler adopts suitable algorithms to calculate the scheduling, and sends results back to each requester host.

As we need to control the traffic rate of each job, rate limiting process should be triggered on every requester host for each job. We use two methods. If the scheduler has root privileges, a user-space process is used to control the tc tool in Linux; otherwise, this control is realized by controlling the rate to write the data to the socket buffer.

5 Performance Evaluation

In this section, we evaluate the performance of our scheduling algorithms through both small scale testbed experiments and large scale simulations. With experiments on the testbed, we verify the effectiveness of our algorithms in real systems. With simulations, we study how different workload patterns impact the performance, by varying the number of responder/requester hosts and the number of jobs in a batch. In addition, we study: (1) the relationship between minimizing make-span and maximizing system throughput; (2) the tradeoff between make-span and Average Job Completion Time (AJCT); (3) how mixture of job types impacts our scheduling performance; and (4) how job execution time estimation error impacts our scheduling performance, respectively. At last, we present the running time of the algorithms proposed in this paper to demonstrate their scalability in real systems.

We choose SJF and Largest Job First (LJF) schemes as the baselines for comparison. With SJF, whenever a responder host is available for the next job, we choose the job that can be completed quickest on it, and schedule as many resources as possible to this job; a job can be transmitted as soon as there is remaining bandwidth, even though it cannot start execution before the transmission is completed.

Largest Job First is a good scheme to minimize make-span in the identical machine scenario. However, it cannot be directly applied to our case where the accelerators are heterogeneous. For example, there are two jobs and two accelerators. Job 1 can be completed in 1 second on accelerator A, while it needs 5 seconds to be completed on accelerator B. For Job 2, we need 3 seconds to complete it regardless of which accelerator is selected. In this case, it is difficult to define which job is the larger one. To solve this problem, we first calculate the average completion time of each job if it is assigned to different accelerators. When there is an idle accelerator, we assign the job with largest average completion time to it.

We define a performance improvement of scheme A over scheme B as:

$$\textit{Improvement} = \frac{|\text{Performance of } B - \text{Performance of } A|}{\text{Performance of } B}. \tag{7}$$

5.1 Testbed Experiment

In our experiment, we first generate one word counting job for each requester host with size 300 MB, 300 MB, 300 MB and 900 MB respectively; they should be sent to the responder hosts for fast processing. To emulate the heterogeneous environment, we scale down the ingress bandwidth of one responder to be 500 Mbps. As word counting is a -typical network-intensive job, the scheduler leverages Algorithm 3 to control the job transmission rate and allocation. The make-spans derived by different schemes are shown in Fig. 2a. From this figure, we can see that even in such a small scale experiment, the make-span improvement achieved by our scheduling scheme is 19.56 percent compared with SJF scheme, which is 33.72 percent compared with the LJF scheme.

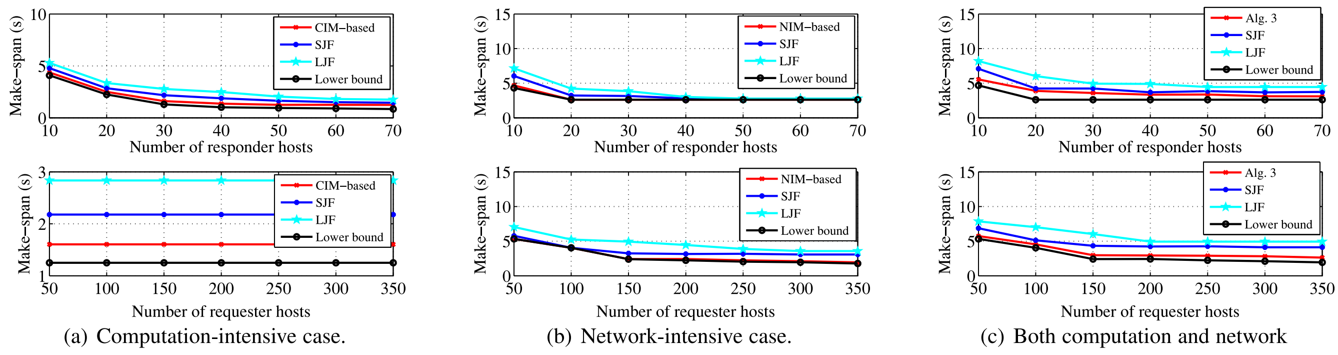

Fig. 3. Algorithm performance changes with number of responder/requester hosts

DNN jobs are used to represent computation-intensive cases on our testbed. We generate one request for each requester host, deal with 6,000/6,000/6,000/18,000 photos respectively. Each photo has 32×32 pixel with RGB channels, and each pixel is quantified with 16 bits. The scheduler calls Algorithm 2 to schedule the jobs. The comparison with SJF and LJF schemes is shown in 2b. We can see that there is 25.80 percent make-span improvement compared with SJF by adopting our scheduling algorithm. In this experiment, the LJF scheme achieves the same performance as our scheduling scheme. This is because currently we only have one type of accelerator for DNN jobs, which makes our testbed to be homogeneous for DNN jobs, where LJF scheme can achieve a good performance. Actually, by enumerating all the possible scheduling schemes, we found that both our algorithms and LJF achieve the optimial solution. However, we will see in the large scale simulation with the heterogeneous environment, LJF performs worst among all these scheduling schemes.

5.2 The Impact of Workload Pattern

From this point, we use a simulator for evaluation; the accuracy of the simulator has been verified by our testbed. To study how workload patterns impact the algorithms' performance, we change the number of responder hosts and the number of requester hosts, respectively. When changing the number of responder hosts, we fix the number of requester hosts to be 200; or, we fix the number of responder hosts to be 30 when we change the number of requester hosts. In these simulations, we inject 1000 jobs into the system. The job size is generated following the exponential distribution with the parameter 0.005, i.e., the average job size is 200 Mb. We assume each responder host carries 5 accelerators. Correspondingly, We set the average ingress bandwidth of responder hosts to be 5,000 Mbps, while the average egress bandwidth of request hosts as 1,000 Mbps. The execution time of each job is set to be approximately proportional to its size with the ratio of 0.001. In this way, the data transmission and computation of each job spend almost the same time. To emulate the heterogeneous environment, we add -20$\sim$20 percent deviation to the execution time of each job when it executes on different accelerators. Correspondingly, we also add -20$\sim$20 percent deviation to the bandwidth of each responder host and request host. For the computation-intensive case, we directly set the ingress/egress bandwidth to be infinite; we set computation time to be 0 for the network-intensive case. To highlight the performance of our scheduling system, we also test the make-span lower bounds. For computation-intensive cases, the lower bound is the objective value of $(1L)$; while for the network-intensive cases, the lower bound is the objective value of (4). When neither computation nor data transmission can be ignored, we set the lower bound to be the larger one of the above two bound values.

Fig. 4 demonstrates the simulation results. From these results, we can make the following observations. First, for the computation-intensive case (Fig. 3a), LJF performs worst among all the comparison schemes. This is because the LJF scheme may assign a large job to the responder that is slowest for this job. It greatly degrades the performance of the scheduling scheme. The same problem may exist in the SLF scheme, however, it happens with much lower probability since an idle accelerator always gets the job it can complete the most quickly. The gap between CIM and SJF first increases, and then shrinks with the number of responder hosts. The largest make-span improvement is achieved when there are 40 responder hosts (i.e., 200 accelerators), and the make-span improvement value is 27.81 percent compared with SJF. When there are only a small number of accelerators in the system, each accelerator has to accommodate a large number of jobs. In this case, the scheduling algorithms' room for make-span improvement is small. When there are a large number of accelerators in the system, each accelerator

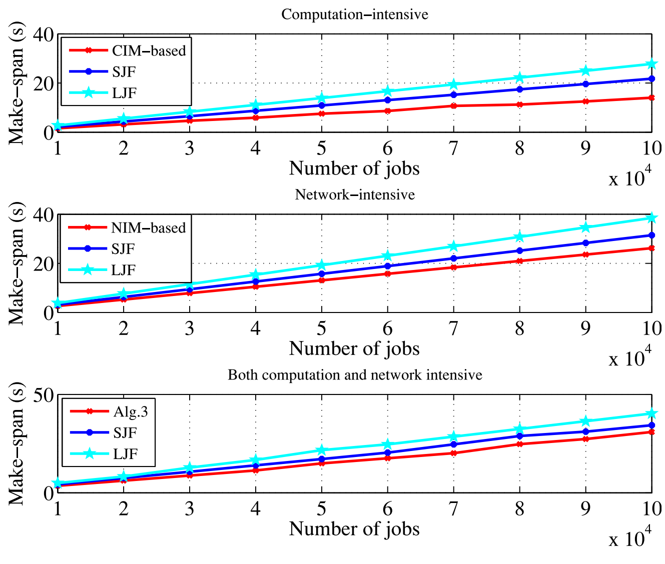

Fig. 4. Impact of workload.

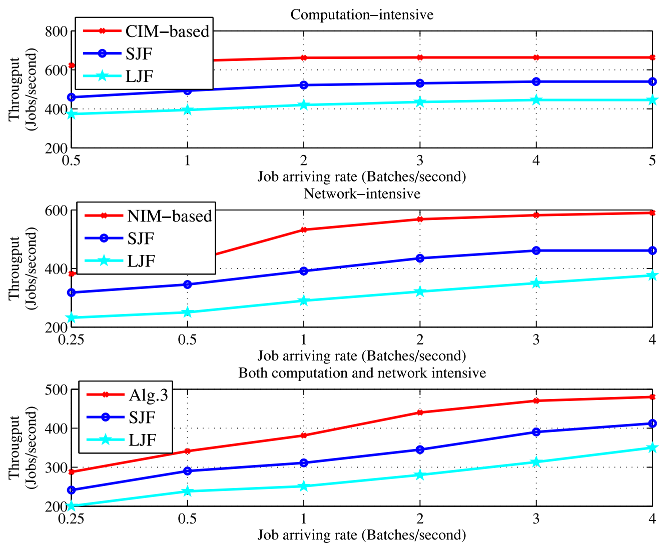

Fig. 5. Job throughput.

executes only a few jobs. The make-span is very small and there is also only a small room for optimization. When we change the number of request hosts, make-span does not change, regardless of which scheduling scheme is adopted. The reason is that make-span is only determined by the job allocation. If the number of accelerators (or, job allocation) does not change, the make-span maintains constant. The performance gap between our scheduling algorithms and the make-span lower bound is relatively stable, as the performance gap only comes from the only split job rounded to each accelerator. In addition, the performance of our algorithm is very close to the make-span lower bound though the theoretic approximation ratio is 2.

Fig. 3b shows the results of the network-intensive case. We can see that initially the make-span improves with the increase of responder hosts or request hosts regardless of which scheduling scheme is adopted; it cannot be significantly reduced any more when there are more than 20 responder hosts (when the number of request host is 200) or 150 request hosts (when the number of responder hosts is 30). This is because that when there are enough requester/responder hosts, the bandwidth bottleneck will be on the responder/requester host side. The increase of only one type of hosts cannot deliver more jobs when this type of host is not the network bottleneck. When there are only a few requester hosts or there are lots of responder hosts, the performance of NIM-based scheduling achieves the make-span lower bound, since the ingress bandwidth of responder hosts is not the network bottleneck. An interesting observation is that make-span improvement of NIM-based scheme will increase with the number of requester hosts. When the the bottleneck is at the egress ports of requester hosts, we should only optimize the bandwidth sharing among the jobs on each requester host; but when the ingress bandwidth of responder hosts is the bottleneck, we should optimize the bandwidth sharing among all the jobs from all the requester hosts. Accordingly, NIM-based scheme derives larger benefit when there are more requester hosts and the ingress ports of the responder hosts are the network bottleneck, since it optimizes the make-span from a global view. Again, LJF performs the worst. It is not surprising that LJF cannot perform as well as NIM-based scheduling, since LJF is a myopic scheme, while NIM-based scheme optimizes the make-span in a global view. Compared with SJF, LJF may assign a large job issued by a requester with large outgoing bandwidth to a responder host with smaller ingress bandwidth, while SJF has much lower probability to do this. Accordingly, LJF yields a larger make-span.

When taking both data transmission and computation into account, the simulation results are shown in Fig. 3c. Obviously, the make-span in this case should be smaller than the sum of the make-span in the computation-intensive case and that in the network-intensive case, as the transmission phase of later jobs can be overlapped with the computation phase of previous jobs. When we change the number of responder hosts, most of the gain comes from the bandwidth scheduling if there are very few responder hosts; the gain comes from the job allocation if there are more responder hosts. When we change the number of request hosts, the make-span improvement trend is the same as that of the network-intensive case. This is because that more requester hosts cannot bring benefit to the job computation phase, and the gain only comes from the bandwidth management. However, when both computation and data transmission cannot be ignored, our algorithm focuses on how to maximize the overlap of the computation phase of each job with the transmission phase of the later jobs. It brings more room for the optimization, and hence our algorithm can outperform SJF more than that in network-intensive case. When there are 350 requester hosts, the performance improvement of our algorithm is 36.25 percent compared with SJF, while it is 46.81 percent compared with LJF. Furthermore, we can see that our scheduling performance is close to the lower bound which is obtained by ignoring the computation phase or data transmission phase. This shows that our algorithm derives a near optimal make-span.

5.3 Impact of Workload

In this section, we investiage how the performance of our scheduling schemes change with the workload. To this end, we assume there are 2000 requester hosts and 300 responder hosts (i.e., 1500 accelerators). Then, we generate different number of jobs in a single batch and measure the make-span derived by different scheduling schemes. The simulation results are shown in Fig. 4 From this figure, we can see that the make-span is increasing almost linearly with the number of jobs injected into the system. This is because that when the system is fully utilized, the queueing delay is proportional to the workload regardless of what scheduling scheme is adopted.

5.4 Job Throughput

Minimizing the make-span of each batch of jobs can also increase the system throughput. In this section, we investigate how much throughput can be improved by our algorithms. To this end, we assume there are 200 requester hosts and 30 responder hosts (i.e., 150 accelerators) in the system. We generate batches of different type of jobs and inject them into the system following different arriving rate, and test the system throughput. The simulation results are shown in Fig. 5. From this figure, we can make following observations.

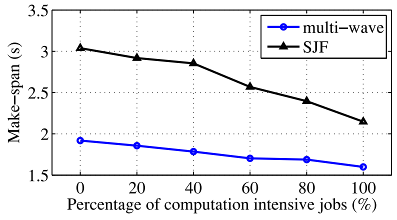

Fig. 6. Performance with job type mixture.

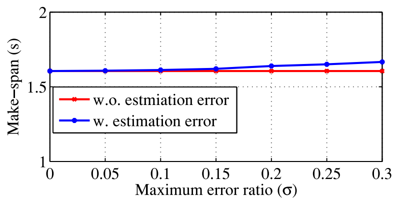

Fig. 8. Impact of execution time estimation error.

First, with the algorithms we proposed to minimize the make-span of each batch of jobs, the system can deal with 22.88–35.64 percent, 20.17–36.05 percent and 16.47–27.57 percent more different type of jobs in each unit of time, respectively, compared with scheduling by SJF. Compared with LJF, algorithms proposed in this paper can complete 32.88–40.08 percent, 36.23–45.47 percent and 27.06–36.38 percent more different type of jobs, respectively. With the algorithms we proposed to minimize the make-span of each batch of jobs, the accelerator and network resources can be utilized more properly. Hereby, the throughput is increased.

Second, the system throughput improvement derived by our scheduling scheme is generally larger than the improvement of reducing the make-span of single batch of jobs. This is because that when the job batches come online, our scheduling scheme does not only minimize the make-span of single batch of jobs, but also optimize how to use the remaining resources left by previous batches of jobs. It brings more optimization opportunity to improve the system throughput. This observation also demonstrates that minimzing make-span of single batch of jobs is a good way to improve the system throughput.

Third, the system throughput is increasing with the job arriving rate. The larger the job arriving rate is, the more jobs are in the system and hence the resources can be used more efficiently. Therefore, more jobs can be dealt with, which increases the system throughput.

Last but not least, when the job arriving rate reaches a threshold, e.g., 4 batches of computation-intensive jobs per second in our simulation, the system throughput will not increase with the job arriving rate. This is because the system is fully utilized and the throughput cannot be increased by injecting more jobs into the system any more.

5.5 Mixture of Job Types

To evaluate the performance of different solutions, we inject 1000 jobs of different mixture patterns into the system and adopt different solutions to schedule the jobs. Fig. 6 shows the simulation results.

We divide all jobs into multiple groups according to their types. We schedule them with different algorithms (labeled as "multi-wave"), as it enforces customized schemes for specific type of jobs. The performance gap between SJF and multi-wave scheme improvements when there are more computation-intensive jobs in the system. This is because that SJF scheme mixes computation and network-intensive jobs and does not try to overlap the computation and the transmission phase of different types of jobs; less transmission results in smaller performance gap.

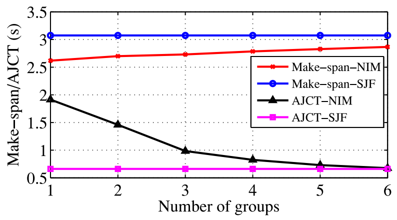

Fig. 7. Tradeoff between AJCT and make-span.

5.6 Tradeoff between AJCT and Make-Span

Pursuing minimum make-span may incur larger AJCT, especially for network-intensive jobs. However, we can divide all the jobs into multiple groups according to their size, and schedule the jobs with smaller size first. Fig. 7 shows the effect of this scheme. We can see that when more groups the jobs are divided into, there will be smaller AJCT with the cost of slightly larger make-span. When we divide all the jobs into 6 groups, we get almost the same AJCT with the SJF scheme but still have 7.2 percent make-span improvement.

5.7 Impact of Job Execution Time Estimation Error

To study how the estimation error of job execution time impacts the performance of our algorithms, we randomly add some error to the estimated execution time in the input of computation-intensive scenarios, since computation-intensive jobs are most sensitive to the job execution time. For each job, we assume the estimation error is proportional to its input data size, and the ratio is evenly distributed in the range of $[-\sigma, \sigma]$. When there are 200 requester hosts, 40 responder hosts, and 1000 jobs, the simulation results are shown in Fig. 8. From this figure, we can see that even when the maximum estimation error is up to 30 percent of the real execution time, our algorithm can still derive a reasonable make-span close to the optimal no-error scenarios. After digging into the results, we found that this is because that estimation error of different jobs assigned to the same host can compensate each other's error impact.

5.8 Algorithm Running Time

As an online system, an important issue is the algorithm running time. By fixing the number of responder hosts and changing the number of requester hosts, we measure the algorithm running time, which is shown in Fig. 9. All the points in this figure are collected on a desktop carrying Intel i7-2600 CPU with 8 GB memory. The algorithm for computation-intensive jobs and the algorithm for both computation and network intensive jobs have nearly the same computation complexity, which is almost independent from the number of request hosts. This is because the computation bottleneck is to solve the same LP model, and this LP model is not related to the number of request hosts. The algorithm for the network-intensive jobs has higher

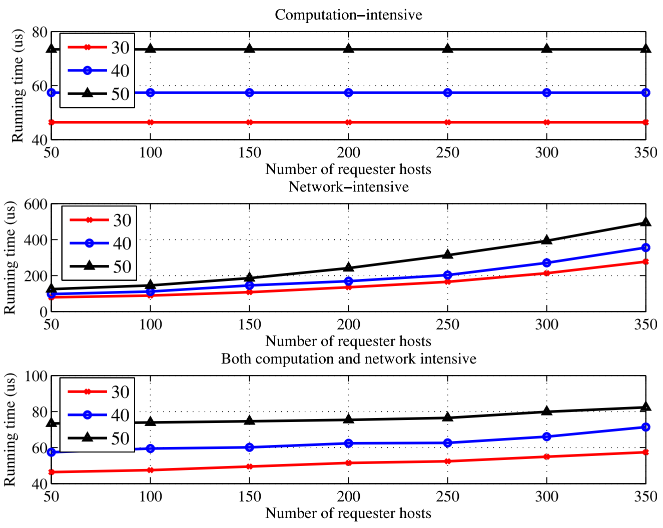

Fig. 9. Algorithm running time.

computation complexity as it should solve two LP models and one of them is scaling with the number of requester hosts. However, the running time of all the algorithms is in the micro-second level even when there are a batch of 1000 jobs in a system with hundreds of accelerators and requester hosts. This means that our scheduling scheme incurs very little overhead to the system. Even if there are millions of jobs and hundreds of responder/requester hosts in the system and we need long time to calculate the scheduling scheme, we can divide the entire system into multiple subsystems to reduce the computation overhead.

6 Related Work

There are several works that propose to virtualize FPGA processing in Cloud [25], [26], [27], [28], [29]. None of them manage FPGA accelerators as a single universal resource pool. *pvFPGA* [26] builds a para-virtualized environment using Xen VMM; it treats FPGA chips as monolithic resources. [27] and [25] exploit the partial-reconfigurable ability in FPGA virtualization, where an FPGA chip is divided into multiple regions and each can act as an independent virtual chip. [27] focuses on how fast a VM can set up and tear down a virtual accelerator in the hosting machine, while [25] works on abstracting FPGA as a consumable resource while avoiding hardware dependencies of specific FPGA techniques.

Both Microsoft and Baidu use FPGA accelerators to improve the performance of their online services [1], [2], [5], [6]. Mostly they are purpose-build clusters for dedicated applications. Instead, our targeted systems follow the SaaS model: tenant programs can interact with the Cloud by calling API functions provided by an FPGA service layer, even without a physical FPGA accelerator installed on their hosts.

There are also some works on job scheduling with accelerators. [30] presents a job scheduler for GPU. However, it is a driver-level work rather than a job level work. Therefore, [30] and our work are compensating for each other. [30] can be used to provide a better accelerator while our work can schedule the job to further improve the system performance. [31] focuses on the job scheduling in heterogeneous Cloud with accelerators. However, it only considers the fairness among jobs, and does not consider the job transmission phase. [32] is the work to present how to program and implement the FPGA accelerators. It develops the API for job scheduling, and provides an easy way to implement our scheduling algorithms in the Cloud systems. Our previous work [33] also considers the scheduling problem in the job level, however, it assumes the egress bandwidth of the requester hosts is infinite, which is not always the case in practical.

7 Conclusions

To minimize the make-span of a given batch of FPGA accelerator requests, in this paper, we proposed 2-approximation algorithms for the computation-intensive case and network-intensive case, respectively. In addition, we also designed an efficient heuristic for the case where both computation and network are bottlenecks. We implemented our scheduling algorithms in a real Cloud system, and conducted extensive evaluations to show that our scheduling algorithms can reduce the make-span compared with the Shortest-Job-First scheme. Different from previous works, this work considers multiple resource bottlenecks for FPGA accelerator pools in Cloud computing.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This work was supported in part by the National Science and Technology Major Project of China under Grant Number 2017ZX03001013-003, the Fundamental Research Funds for the Central Universities under Grant Number 0202-14380037, the National Natural Science Foundation of China under Grant Numbers 61772265, 61602194, 61671130, 61671124, 61502229, 61672276, and 61321491, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al., “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Proc. 41st Annu. Int. Symp. Comput. Architecuture*, 2014, pp. 13–24.
[2] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, “SDA: Software-defined accelerator for largescale dnn systems,” in *Proc. IEEE Hot Chips 26 Symp.*, 2014, pp. 1–23.
[3] D. Ciregan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2012, pp. 3642–3649.
[4] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, “Asynchronous stochastic gradient descent for dnn training,” in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 6660–6663.
[5] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, “Clicknp: Highly flexible and high performance network processing with reconfigurable hardware,” in *Proc. ACM SIGCOMM*, 2016, pp. 1–14.
[6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, “A cloud-scale acceleration architecture,” in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.
[7] Z. AMI, “Getting started with zebra on aws,” 2017. [Online]. Available: http://www.mipsology.com/aws/getting_started.html
[8] A. Putnam, “Large-scale reconfigurable computing in a microsoft datacenter,” in *Proc. IEE Symp. Hot Chips*, 2014, pp. 1–38.
[9] S. Kestur, J. D. Davis, and O. Williams, “BLAS comparison on FPGA, CPU and GPU,” in *Proc. IEEE Annu. Symp. VLSI*, 2010, pp. 288–293.

[10] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Proc. Int. Conf. Field Programmable Log. Appl.*, 2009, pp. 126–131.
[11] S. Zhou, Y. Zhu, C. Wang, X. Gu, J. Yin, J. Jiang, and G. Rong, "An FPGA-assisted cloud framework for massive ECG signal processing," in *Proc. IEEE 12th Int. Conf. Dependable, Autonomic Secure Comput.*, 2014, pp. 208–213.
[12] Q. Liu, Z. Xu, and Y. Yuan, "A 66.1 Gbps single-pipeline AES on FPGA," in *Proc. Int. Conf. Field-Programmable Technol.*, pp. 378–381, 2013.
[13] J. P. Orellana, M. B. Caminero, and C. Carrión, "On the provision of SaaS-level quality of service within heterogeneous private clouds," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, 2014, pp. 146–155.
[14] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013, pp. 219–230.
[15] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proc. 4th USENIX Conf. Hot Top. Cloud Comput.* 2012, pp. 1–6.
[16] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When spark meets FPGAs: A case study for next-generation dna sequencing acceleration," in *Proc. 8th USENIX Conf. Hot Top. Cloud Comput.*, 2016, pp. 64–70.
[17] J. Gu, M. Zhu, Z. Zhou, F. Zhang, Z. Lin, Q. Zhang, and M. Breternitz, "Implementation and evaluation of deep neural networks (DNN) on mainstream heterogeneous systems," in *Proc. 5th Asia-Pacific Workshop Syst.*, 2014, Art. no. 12, pp. 1–7.
[18] N.-F. Standard, "Announcing the advanced encryption standard (AES)," *Federal Inf. Process. Standards Publication*, vol. 197, pp. 1–51, 2001.
[19] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann, *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*, 1st ed. Secaucus, NJ, USA: Springer, 1999.
[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008.
[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.
[22] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," in *Proc. IEEE 28th Annu. Symp. Found. Comput. Sci.*, Oct. 1987, pp. 217–224.
[23] C. Banino-Rokkones, O. Beaumont, and H. Rejeb, "Scheduling techniques for effective system reconfiguration in distributed storage systems," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. Conf.*, 2008, pp. 80–87.
[24] D. P. Palomar and J. R. Fonollosa, "Practical algorithms for a family of waterfilling solutions," *IEEE Trans. Signal Process.*, vol. 53, no. 2, pp. 686–695, Feb. 2005.
[25] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proc. 11th ACM Conf. Comput. Frontiers*, 2014, Art. no. 3, pp. 1–10.
[26] W. Wang, M. Bolic, and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *Proc. 9th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codes. Syst. Synthesis*, 2013, pp. Art. no. 10, pp. 1–9.
[27] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *Proc. IEEE 22nd Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 109–116.
[28] O. Knodel and R. G. Spallek, "Computing framework for dynamic integration of reconfigurable resources in a cloud," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2015, pp. 337–344.
[29] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Proc. 7th Int. Conf. Cloud Comput. Technol. Sci.*, 2015, pp. 430–435.
[30] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 1–12.
[31] G. Lee, B.-G. Chun, and H. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud," in *Proceedings 3rd USENIX Conf. Hot Top. Cloud Comput.*, 2011, pp. 1–5.
[32] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 456–469.
[33] Y. Zhao, X. Liu, and C. Qiao, "Job scheduling for acceleration systems in cloud computing," in *Proc. IEEE ICC*, 2018, pp. 1–6.

Yangming Zhao received the BS degree in communication engineering and the PhD degree in communication and information system from the University of Electronic Science and Technology of China, in 2008 and 2015, respectively. He is a research scientist with SUNY buffalo. He is a visiting scholar with Nanjing University. His research interests include network optimization and data center networks.

Chen Tian received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is an associate professor with State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming, and urban computing.

Zhuangdi Zhu received the bachelor's degree from the College of Elite Education, Nanjing University of Science and Technology, in 2015. She is working toward the PhD degree in the Computer Science Department, Michigan State University. She joined IBM Research China as a research assistant, in 2014. Her research interests include systems, networking, and Internet of Things.

Jie Cheng received the PhD degree from the Huazhong University of Science and Technology, in 2011. He worked as a postdoctoral researcher with Prince Edward Island University from 2011 to 2015. He is currently a senior Sstaff engineer with the Data Center Technology Laboratory, Huawei Technologies Company, Ltd., China. His research interests include cloud computing, artificial intelligence, machine learning, and big data.

Chunming Qiao is a SUNY distinguished professor and also the current chair with the Computer Science and Engineering Department, University at Buffalo. He was elected to IEEE fellow for his contributions to optical and wireless network architectures and protocols. His current focus is on connected and autonomous vehicles. He has published extensively with an h-index of over 69 (according to Google Scholar). Two of his papers have received the best paper awards from IEEE and Joint ACM/IEEE venues. He also has seven US patents and served as a consultant for several IT and Telecommunications companies since 2000. His research has been funded by a dozen major IT and telecommunications companies including Cisco and Google, and more than a dozen NSF grants. He is a fellow of the IEEE.

Alex X. Liu received the PhD degree in computer science from the University of Texas at Austin, in 2006. He received the IEEE & IFIP William C. Carter Award, in 2004, an National Science Foundation CAREER award, in 2009, and the Michigan State University Withrow Distinguished Scholar Award, in 2011. He is an associate editor of the *IEEE/ACM Transactions on Networking*, an editor of the *IEEE Transactions on Dependable and Secure Computing*, and an area editor of the *Computer Communications*. He received Best Paper Awards from ICNP-2012, SRDS-2012, and LISA-2010. His research interest includes networking and security.