Joint Optimization of Task Scheduling and Image Placement in Fog Computing Supported Software-Defined Embedded System

Deze Zeng, *Member, IEEE*, Lin Gu, *Member, IEEE*, Song Guo, *Senior Member, IEEE*, Zixue Cheng, *Member, IEEE*, and Shui Yu, *Senior Member, IEEE*

Abstract—Traditional standalone embedded system is limited in their functionality, flexibility, and scalability. Fog computing platform, characterized by pushing the cloud services to the network edge, is a promising solution to support and strengthen traditional embedded system. Resource management is always a critical issue to the system performance. In this paper, we consider a fog computing supported software-defined embedded system, where task images lay in the storage server while computations can be conducted on either embedded device or a computation server. It is significant to design an efficient task scheduling and resource management strategy with minimized task completion time for promoting the user experience. To this end, three issues are investigated in this paper: 1) how to balance the workload on a client device and computation servers, i.e., task scheduling, 2) how to place task images on storage servers, i.e., resource management, and 3) how to balance the I/O interrupt requests among the storage servers. They are jointly considered and formulated as a mixed-integer nonlinear programming problem. To deal with its high computation complexity, a computation-efficient solution is proposed based on our formulation and validated by extensive simulation based studies.

Index Terms—Fog computing, software-defined embedded system, task scheduling, resource management, optimization

1 Introduction

With the rapid development in embedded and mobile devices during the last decade, the modern computing environment is shifting toward high heterogeneity with various hardwares, softwares and operation systems. At the same time, people also demand more customized services. However, traditional application-specific embedded systems fail to catch up with the growing diversity of user demands. To address this issue, some recent studies advocate the "software-defined" concept [1], [2], [3], [4] by leveraging software to tackle the complexity and inflexibility of traditional embedded system. By such means, it is possible to fully explore the potential of embedded system hardware and make the system more flexible.

Standalone embedded systems are limited in their functionalities and processing capbilities. Cloud computing platform with abundant resources is naturally regarded as a good platform to tackle such limitation by offloading some embedded tasks onto it. However, some time-sensitive embedded tasks require fast response. In such cases, long and unstable latency between the embedded client and cloud is not desirable. To overcome these disadvantages, fog computing [5], also known as "clouds at the edge", emerges as an alternative solution to support future cloud services and applications, especially to the Internet-of-Things (IoT) applications featured by geographical distribution, latency sensitivity, and high resilience.

In this paper, we explore the features of fog computing to support software-defined embedded system and propose a fog computing supported software-defined embedded system (FC-SDES) as shown in Fig. 1. The network edges (e.g., cellular base stations) are equipped with certain storage or computation resources and the embedded clients are general-purpose bare hardware automated by intelligent software. Different from the traditional standalone embedded system, the task image may not be fully loaded into the embedded system initially, but resides on an edge storage server and shall be retrieved from the edge server in an on-demand manner at runtime. As a result, during the execution of a task, I/O interrupts (e.g., page faults) happen unexpectedly. They are handled by local hard disk in traditional embedded system, while by storage server in SD-SDES following the procedures shown in Fig. 2.

Although fog computing has been studied from different aspects [6], [7], [8] in the literature, most existing studies mainly focus on the application and resource management on the cooperation between cloud computing and fog computing. When it comes to FC-SDES, we shall consider the interaction between fog computing and embedded system. For example, we can designate an edge

- *D. Zeng is with the Hubei Key Laboratory of Intelligent Geo-Information Processing, School of Computer Science, China University of Geosciences, Wuhan, China. E-mail: dazzae@gmail.com.*
- *L. Gu is with the Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China. E-mail: linny.gu@gmail.com.*
- *S. Guo and Z. Cheng are with the Department of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu, Japan. E-mail: {sguo, zx-cheng}@u-aizu.ac.jp.*
- *S. Yu is with the School of Information Technology, Deakin University, Melbourne, Australia. E-mail: shui.yu@deakin.edu.au.*

Manuscript received 30 Nov. 2014; revised 22 Feb. 2016; accepted 23 Feb. 2016. Date of publication 28 Feb. 2016; date of current version 14 Nov. 2016. Recommended for acceptance by G. Min.
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/TC.2016.2536019

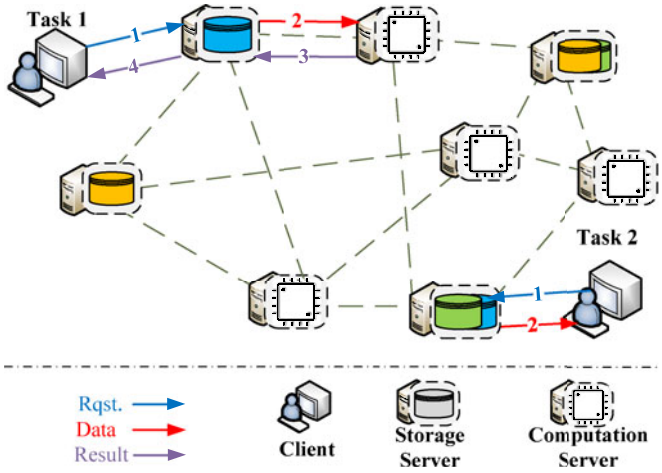


Fig. 1. Task processing in fog computing supported software-defined embedded system.

server to execute tasks from embedded clients (e.g., Task 1 in Fig. 1). However, the resources on the network edges are limited and the latency varies. Some tasks could also be processed on clients for faster response, e.g., Task 2 shown in Fig. 1. How to systematically manage the resources and schedule the tasks on FC-SDES is still an open issue. An effective resource management and task scheduling mechanism is required in order to provide high user experience, e.g., task completion time minimization. Specially, the following questions shall be answered for task completion time minimization.

1) Where to place the task images? The placement could be affected by the popularity of the corresponding requests, e.g., locating the replica near to computation. Further, certain task image is not only essential to reliability but also helpful for load balancing. The task image placement shall have deep influence on the I/O interrupt handling response time.
2) Where computation shall take place, on computation servers or clients? Generally, computation server shall have faster processing speed than clients. However, the computation server is shared by multiple tasks. If it is not scheduled well, the task completion time on the server may be even longer than on the clients. Carefully balancing the workloads on edge and client side is also a critical issue to minimize the task completion time.
3) Which computation server shall be used when a user request is distributed to the edge? The task processing response time is influenced by workloads on a server. The task shall be also well balanced among computation servers. Besides, the connections between edge servers and clients will highly affect the response latency.
4) Which storage server shall be used to handle the I/O interrupt of a task? Similar to the last problem, the I/O interrupt handling response time is affected by both the workloads allocated onto storage server and the transmission latency among storage server, computation server and client.

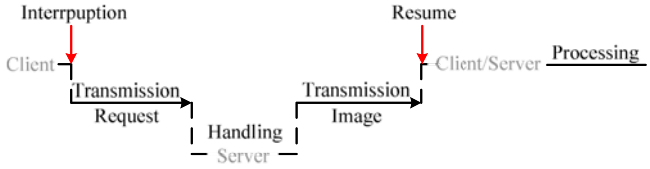


Fig. 2. I/O interrupt handling in FC-SDES.

In this paper, we are motivated to investigate the request completion time minimization problem with joint consideration of task image placement and load balancing for a hybrid computing environment. Our main contributions are summarized as follows:

- To our best knowledge, we are the first to investigate the task scheduling problem in FC-SDES. In particular, we consider a scenario where computation can be processed in either client side or edge side. By balancing the workload on both sides, we can minimize the overall computation and transmission latency of all requests. We also investigate how to place replica of task image on the storage servers.
- We formulate the task completion time minimization problem with joint consideration of task scheduling and image placement as a mixed-integer nonlinear programming (MINLP) problem.
- We propose a low-complexity three-stage algorithm for the task completion time minimization problem. Experimental results validate the high efficiency of our algorithm.

The rest of the paper is organized as follows. Section 2 introduces our system model. The task completion time minimization problem is formulated in Section 3 and a heuristic algorithm is proposed in Section 4. The effectiveness of our proposal is verified by experiments in Section 5. Section 6 summaries the related work. Finally, Section 7 concludes our work.

2 System Model and Problem Statement

The logical view of FC-SDES in a three-tier architecture is shown in Fig. 3, where there are a set J of storage servers, a set K of computation servers and a set I of embedded clients. All the servers and clients are inter-connected. The transmission latency between any two nodes, e.g., $i \in I$ and

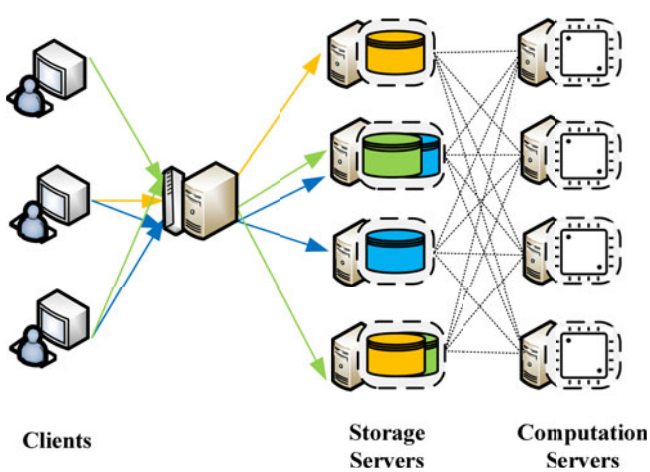


Fig. 3. Logical view of FC-SDES.

TABLE 1
Notations

I	Embedded client set
μ_i	Service rate for FC-SDES on client $i \in I$
J	Storage server set
S_j	Storage resource for FC-SDES of storage server $j \in J$
μ_j	I/O processing rate for FC-SDES on storage server $j \in J$
K	Computation server set
μ_k	Service rate for FC-SDES on computation server $k \in K$
T	Task set
s_t	Image size of task $t \in T$
λ_{ti}	Average request rate of task $t \in T$ from client $i \in I$
n_t	Average number of interrupts required by task $t \in T$
p_{tijk}	The probability that task $t \in T$ from client $i \in I$ is handled by storage server $j \in J$ and computation server $k \in K$
x_{tj}	Whether the program of task $t \in T$ is stored in storage server $j \in J$ or not
Ω	The required replica number for each task image
l_{ij}	Transmission latency between client $i \in I$ and storage server $j \in J$
l_{jk}	Transmission latency between storage server $j \in J$ and computation server $k \in K$
l_{ik}	Transmission latency between edge device $i \in I$ and computation server $k \in K$

$j \in J$, is denoted as l_{ij} and other major symbols are summarized in Table 1.

Task processing requests arise randomly as a Poisson process on each client. For each task t in a set T of tasks, the average task arrival rate on a client $i \in I$ is λ_{it}. Without loss of generality, we consider that storage server $j \in J$ is with storage capacity S_j for FC-SDES and a computation server $k \in K$ is with computation rate μ_k. Let s_t denote the image size of task $t \in T$. To ensure the reliability and to balance the disk reading requests across different servers, Ω copies are maintained in different storage servers in the FC-SDES.

All the I/O interrupts arising unexpectedly during task processing on a client or a computation server must be handled by a storage server. Fig. 2 illustrates the procedures upon an I/O interrupt. The computation device shall first suspend current task and issue a data request to a storage server. After the storage server handles the request and transmits the desired data to the execution device, the computation can then resume. For a task t, let n_t denote the average number of interrupts during one task session. That is to say, for one task, its task image is not fully loaded to the embedded client initially, but shall be loaded by n_t I/O interrupts at runtime. Generally, a larger size implies more I/O interrupts. We follow the findings in [9] that the I/O interrupt handling time follows exponential distribution. The storage server can process an I/O interrupt from a computation server or a client with average service rate $\mu_j, \forall j \in J$.

In the FC-SDES, a task can be processed either by the client itself or a computation server. Although it is possible to obtain some statistics of task processing (such as base image size, average interrupt number, etc.), it is hard to accurately predict the characteristics of a specific task request. Alternatively, we could make the task scheduling decisions in a probabilistic manner. We denote p_{tijk} as the probability that a task request $t \in T$ from client $i \in I$ is distributed to storage server $j \in J$ and computation server $k \in K$, to handle the disk reading and task processing, respectively. Let p_{tij} be the probability that I/O interrupt handling requests of task t are sent to storage server j for disk reading but executed locally on the client.

A task image replica can be stored on any storage server. As a storage server may be shared by multiple clients for the same task, where the task image is stored shall have a deep influence on the quality-of-service (QoS) in terms of task completion time. A task may be processed on a computation server, other than always on the client itself. Usually the computation server may have higher service rate than a client, but it would be shared by multiple clients for many tasks. How to balance the requests to the client and a computation server is also critical to task completion time. Further, the transmission latency is another issue that cannot be ignored. We are interested in finding out the task image placement strategy and the decision probabilities that can minimize the maximum task completion time for a given traffic pattern.

3 Problem Formulation

In this section, we provide a formal description of our problem with joint consideration of task scheduling and task image placement by formulating it into a mixed-integer non-linear programming problem.

3.1 Task Completeness Constraints

To ensure the QoS, it is first required that all the requests arising from client i must be processed, either by the computation server or by the client itself. This leads to

$$\sum_{j \in J} \sum_{k \in K} p_{tijk} + \sum_{j \in J} p_{tij} = 1, \forall t \in T, i \in I. \tag{1}$$

3.2 Storage Constraints

A storage server can handle the I/O interrupt if and only if it holds the corresponding task image. We use a binary variable to denote whether the task image of t is stored in storage server j or not as

$$x_{tj} = \begin{cases} 1, & \text{if the task image } t \in T \text{ is} \\ & \text{stored in server } j \in J, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

For any task t, Ω replica shall be maintained on different servers for reliability and load balancing consideration. Therefore, we have

$$\sum_{j \in J} x_{tj} = \Omega, \forall t \in T. \tag{3}$$

A server may also store multiple images for different tasks, provided that the storage capacity constraint is not violated, i.e.,

$$\sum_{t \in T} x_{tj} s_t \leq S_j, \forall j \in J. \tag{4}$$

The I/O requests of a task can be scheduled to a storage server if it stores the corresponding image. Therefore, no matter where a task is processed, we always have

$$p_{tijk} \leq x_{tj} \leq A p_{tijk}, \forall t \in T, i \in I, j \in J, k \in Ks, \tag{5}$$

and

$$p_{tij} \leq x_{tj} \leq Ap_{tij}, \forall t \in T, i \in I, j \in J, \tag{6}$$

where A is an arbitrarily large number.

3.3 Task Completion Time Analysis

Basically, the task completion time consists of three basic components, i.e., computation time, I/O time and transmission time.

3.3.1 Computation Time

The computation time of a task $t \in T$ depends on where the task processing is scheduled. If t is distributed on computation server $k \in K$ with the service rate μ_k, sever k may be shared by multiple clients for different tasks. As the sum of multiple independent Poisson processes is still a Poisson process, the overall task arrival rate at server k thus can be calculated as

$$\Lambda_k = \sum_{t \in T} \sum_{i \in I} \sum_{j \in J} p_{tijk} \cdot \lambda_{ti}, \forall k \in K. \tag{7}$$

Recall that the task computation time is exponentially distributed on a computation server, which can then regarded as an M/M/1 queue. The average computation time of all tasks at computation server $k \in K$ can be thus calculated as

$$\tau_k^c = \frac{1}{\mu_k - \sum_{t \in T} \sum_{i \in I} \sum_{j \in J} p_{tijk} \cdot \lambda_{ti}}, \forall k \in K, \tag{8}$$

where we must ensure

$$\mu_k > \sum_{t \in T} \sum_{i \in I} \sum_{j \in J} p_{tijk} \cdot \lambda_{ti}, \forall k \in K. \tag{9}$$

Similarly, the computation on a client can be also described as an M/M/1 queue. We can derive the average computation time on a client $i \in I$ as

$$\tau_i^c = \frac{1}{\mu_i - \sum_{t \in T} p_{it} \lambda_{it}}, \forall i \in I. \tag{10}$$

We shall also guarantee

$$\mu_i > \sum_{t \in T} p_{it} \lambda_{it}, \forall i \in I. \tag{11}$$

3.3.2 I/O Time

For task executed on any device, I/O interrupt may arise unexpectedly. Corresponding requests shall be sent to a remote storage server with the task image. Therefore, the requests at a storage server j come from two kinds of sources, i.e., client and computation server. The I/O interrupt handling process can also be described as an M/M/1 queuing model [9]. Thus, we can derive the average I/O time for each interrupt handling as

$$\tau_j^d = \frac{1}{\mu_j - \sum_{t \in T} \sum_{i \in I} \sum_{k \in K} (p_{tijk} + p_{tij}) \lambda_{ts} i_t}, \forall j \in J. \tag{12}$$

3.3.3 Transmission Time

If the I/O requests and computation of a task t from client i are handled by storage server j and computation server k, respectively, the transmission latency between clients i, j and k shall be considered. We use binary variables to denote the storage and computation server selection as

$$x_{tijk} = \begin{cases} 1, & \text{the task } t \text{ from client } i \text{ are handled by} \\ & \text{storage server } j \text{ and computation server } k, \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

Similarly, we define

$$x_{tij} = \begin{cases} 1, & \text{the task } t \text{ from client } i \text{ are handled by} \\ & \text{storage server } j \text{ and edge client itself,} \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

We can derive that whenever $p_{tijk} > 0$, the value of x_{tijk} shall be one, indicating that the storage server j and computation server k are selected. Therefore, we have the following relationships

$$p_{tijk} \leq x_{tijk} \leq Ap_{tijk}, \forall t \in T, i \in I, j \in J, k \in K, \tag{15}$$

and

$$p_{tij} \leq x_{tij} \leq Ap_{tij}, \forall t \in T, i \in I, j \in J, \tag{16}$$

where A is an arbitrarily large number.

For the tasks scheduled onto a remote computation server $k \in K$, all the transmissions for I/O interrupt handling happen between k and a storage server $j \in J$. There are averagely n_t I/O interrupts during one task execution session. Therefore, the expected total transmission time of task t from client i allocated to j and k can be calculated as

$$\tau_{tijk}^t = 2n_t l_{jk} + l_{ij} + l_{ik}, \forall t \in T, i \in I, j \in J, k \in K. \tag{17}$$

Similarly, if task t is handled by storage server j and client itself, the total expected transmission time becomes

$$\tau_{tij}^t = 2n_t l_{ij}, \forall t \in T, i \in I, j \in J. \tag{18}$$

3.4 An MINLP Formulation

In this paper, we intend to minimize the maximum average task completion time. A new variable τ to denote the maximum time is introduced. Hence, we have

$$x_{tijk} \cdot (\tau_j^c + \tau_k^d + \tau_{tijk}^t) \leq \tau, \forall t \in T, i \in I, j \in J, k \in K, \tag{19}$$

and

$$x_{tij} \cdot (\tau_i^c + \tau_k^d + \tau_{tij}^t) \leq \tau, \forall t \in T, i \in I, j \in J. \tag{20}$$

Our target then becomes to minimize the value of τ.

By summarizing all issues discussed above, we can formulate the task maximum completion time minimization program with joint consideration of task scheduling and image placement as an MINLP problem:

$$\begin{aligned} &\text{MINLP}: \\ &\quad \min: \tau, \\ &\quad \text{s.t.}: (1), (3)-(6), (8)-(12), \text{and } (15)-(20). \end{aligned}$$

4 Algorithm Design

It is difficult to solve the MINLP problem directly due to its high computational complexity [10]. To tackle this issue, we propose a three-stage heuristic algorithm based on the MINLP formulation. The main concept of the algorithm is "partition and join". We first try to minimize the I/O and computation time of any task in the first two stages. There is no strict coupling on the computation and storage servers in the first two stages. Then, we re-couple the storage and computation servers to minimize the overall task completion time with the consideration of transmission time in the third stage.

4.1 Minimize the I/O Time

In this stage, our objective is to find the task image placement and I/O request scheduling for each task to minimize the maximum I/O time without considering the computation time and transmission time. We introduce new variables q_{tij} to denote the probability that the I/O requests for a task $t \in T$ from client $i \in I$ are sent to storage server $j \in J$.

All I/O requests for task t shall be sent to storage servers with corresponding task image. Therefore, we have

$$\sum_{j \in J} q_{tij} = 1, \forall t \in T, i \in I. \tag{21}$$

Moreover, if requests for task t retrieves data from storage server j, then program for task t must be stored in j, i.e.,

$$q_{tij} \leq x_{tj} \leq A q_{tij}, \forall t \in T, i \in I, j \in J, \tag{22}$$

where A is an arbitrarily large number.

By modeling a storage server as an M/M/1 queue, the average I/O time on j can be calculated as

$$\tau_t^d = \frac{1}{\mu_j - \sum_{t \in T} \sum_{i \in I} q_{tij} \lambda_{ti}}, \forall j \in J, \tag{23}$$

where

$$\mu_j > \sum_{t \in T} \sum_{i \in I} q_{tij} \lambda_{ti}, \forall j \in J. \tag{24}$$

Similar to the MINLP formulation, we introduce a variable τ_d to denote the maximum I/O time of all tasks. We shall then have

$$\frac{1}{\mu_j - \sum_{t \in T} \sum_{i \in I} q_{tij} \lambda_{ti}} \leq \tau_d, \forall j \in J. \tag{25}$$

Thus, the problem of minimizing the maximum I/O time can then be described as

$$\begin{aligned} &\text{MINLP-IO}: \\ &\quad \min: \tau_d, \\ &\quad \text{s.t.}: (3), (4), (21), (22), (24), \text{ and } (25). \end{aligned}$$

It is still an MINLP as (25) is non-linear. Fortunately, we notice that it can be translated into a linear form as follows by introducing a new variable $\bar{\tau_d} = 1/\tau_d$:

$$\mu_j - \sum_{t \in T} \sum_{i \in I} q_{tij} \lambda_{ts} \geq \bar{\tau_d}, \forall j \in J. \tag{26}$$

The problem of minimizing τ_d is thus equivalent to maximizing the value of $\bar{\tau_d}$. Then, MINLP-IO can be reformulated as

$$\begin{aligned} &\text{MILP-IO}: \\ &\quad \max: \bar{\tau_d}, \\ &\quad \text{s.t.}: (3), (4), (21), (22), (24), \text{ and } (26). \end{aligned}$$

It becomes a mixed integer linear programming (MILP) problem, which can be approximately solved by linear programming relaxation.

4.2 Minimize the Computation Time

Next, we try to find the optimal task scheduling to minimize the maximum computation time, regardless of image placement and transmission. We denote q_{ti} and q_{tik} as the probability that task $t \in T$ from client $i \in I$ is scheduled to client i and to computation server $k \in K$, respectively.

All requests must be processed on either a computation server or the client, i.e.,

$$q_{ti} + \sum_{k \in K} q_{tik} = 1, \forall t \in T, i \in I. \tag{27}$$

By viewing the computation process as an M/M/1 queue, the average computation time on a computation server $k \in K$ and the client can be expressed as

$$\tau_k^c = \frac{1}{\mu_j - \sum_{t \in T} \sum_{i \in I} q_{tik} \lambda_{ti}}, \forall k \in K, \tag{28}$$

and

$$\tau_i^c = \frac{1}{\mu_i - \sum_{t \in T} q_{ti} \lambda_{ti}}, \forall i \in I, \tag{29}$$

respectively. Inspired by the problem formulation on minimizing the maximum I/O time, the problem of minimizing the maximum computation time can be formulated into a linear programming (LP) problem by translating the min-max problem into a max-min problem. To this end, we introduce a variable $\bar{\tau_c} = 1/\tau_c$ such that the problem can be directly formulated into a LP problem as follows:

$$\begin{aligned} &\text{LP-Comp}: \\ &\quad \max: \bar{\tau_c}, \\ &\quad \text{s.t.}: \mu_j - \sum_{t \in T} \sum_{i \in I} q_{tik} \lambda_{ti} \geq \bar{\tau_c}, \forall k \in K, \\ &\qquad \mu_i - \sum_{t \in T} q_{ti} \lambda_{ti} \geq \bar{\tau_c}, \forall i \in I, \\ &\qquad q_{ti} + \sum_{k \in K} q_{tik} = 1, \forall t \in T, i \in I. \end{aligned}$$

4.3 Joint Optimization

After minimizing the maximum I/O time and computation time independently, we further try to optimize the transmission time by re-coupling the I/O handling and computation, i.e., selecting the optimal routine among clients, storage servers and computation servers with known computation time $\tau_k'^c, \forall k \in K$ and I/O time $\tau_j'^d, \forall j \in J$. The initial I/O request handling and task scheduling plan, i.e., $q_{tij'}$, $q_{tik'}$ and $q_{ti}, \forall t \in T, i \in I, j' \in J, k' \in K$, obtained in the first

two stages do not enforce the exact processing servers as they are homogeneous. In other words, the workloads, either I/O request or task processing, on one server can be freely migrate to another server as a whole. Therefore, we shall consider the mapping between the server assignments obtained in the first two stages and the ones in the final solution. A new variable $m_{hh'}$ is defined as

$$m_{hh'} = \begin{cases} 1, & \text{if server } h' \text{ is mapped by server } h \\ 0, & \text{otherwise.} \end{cases} \tag{30}$$

Note that we may have $h' = h$, which indicates that server h' is mapped to itself in the final solution.

In the final solution, a server must be mapped with another server with the same type, i.e., computation or storage. This enforces

$$\sum_{h \in \{J \text{ or } K\}} m_{hh'} = 1, \forall h' \in \{J \text{ or } K\}. \tag{31}$$

On the other hand, we have

$$\sum_{h' \in \{J \text{ or } K\}} m_{hh'} = 1, \forall h \in \{J \text{ or } K\}. \tag{32}$$

Once a server h is mapped with server h', all the initial task assignment to h shall be reassigned to server h' to ensure the correct mapping with workload conservation. For computation server mapping, we shall have

$$\sum_{k \in K} \sum_{j \in J} p_{tijk} m_{kk'} = q_{tik'}, \forall t \in T, i \in I, k' \in K, \tag{33}$$

while the workload conservation constraints after storage server mapping can be described as

$$\sum_{j \in J} \sum_{k \in K} p_{tijk} m_{jj'} + \sum_{j \in J} p_{tij} m_{jj'} = q_{tij'}, \forall t \in T, i \in I, j' \in J. \tag{34}$$

In (33) and (34), p_{tijk} and $p_{ij}, \forall i \in I, j \in J, k \in K$ are with the same definition in MINLP. Note that there are quadratic terms in both (33) and (34). Fortunately, it is possible to linearize them by introducing auxiliary variables $a_{tijkk'}$, $a_{tijj'k}$ and $a_{tijj'}$ as

$$a_{tijkk'} = p_{tijk} m_{kk'}, \forall i \in I, j \in J, k, k' \in K, \tag{35}$$

$$a_{tijj'k} = p_{tijk} m_{jj'}, \forall i \in I, j, j' \in J, k \in K, \tag{36}$$

$$a_{tijj'} = p_{tij} m_{jj'}, \forall i \in I, j, j' \in J, \tag{37}$$

which can be equivalently replaced by the following linear constraints

$$0 \leq a_{tijkk'} \leq p_{tijk}, \forall i \in I, j \in J, k, k' \in K, \tag{38}$$

$$p_{tijk} + m_{kk'} - 1 \leq a_{tijkk'} \leq m_{kk'}, \forall i \in I, j \in J, k, k' \in K, \tag{39}$$

$$0 \leq a_{tijj'k} \leq p_{tijk}, \forall i \in I, j, j' \in J, k \in K, \tag{40}$$

$$p_{tijk} + m_{jj'} - 1 \leq a_{tijj'k} \leq m_{jj'}, \forall i \in I, j, j' \in J, k \in K, \tag{41}$$

$$0 \leq a_{tijj'} \leq p_{tij}, \forall i \in I, j, j' \in J, \tag{42}$$

$$p_{tij} + m_{jj'} - 1 \leq a_{tijj'} \leq m_{jj'}, \forall i \in I, j, j' \in J, k \in K. \tag{43}$$

Then, constraints (33) and (34) can be rewritten in a linear form as

$$\sum_{k \in K} \sum_{j \in J} a_{tijkk'} = q_{tik'}, \forall t \in T, i \in I, k' \in K, \tag{44}$$

$$\sum_{j \in J} \sum_{k \in K} a_{tijj'k} + \sum_{j \in J} a_{tijj'} = q_{tij'}, \forall t \in T, i \in I, j' \in J. \tag{45}$$

Similar to p_{tijk} and p_{tij}, variables x_{tijk} and x_{tij} bounded by (15) and (16) are also introduced. Thus, with known computation time and I/O time, (19) and (20) can be rewritten as

$$x_{tijk} \cdot (\tau_j'^c + \tau_k'^d + \tau_{tijk}^t) \leq \tau, \forall t \in T, i \in I, j \in J, k \in K, \tag{46}$$

and

$$x_{tij} \cdot (\tau_i'^c + \tau_k'^d + \tau_{tij}^t) \leq \tau, \forall t \in T, , i \in I, j \in J, \tag{47}$$

respectively.

The joint optimization towards minimizing the maximum task completion time can be then described as follows:

$$\begin{aligned} &\text{MILP-Joint}: \\ &\quad \min: \tau, \\ &\quad \text{s.t.}: (15), (16), (31), (32), \text{ and } (38) - (47). \end{aligned}$$

By summarizing all the above issues together, we finally have our three-stage algorithm shown in Algorithm 1. In line 2, we first relax all the binary variables x_{tj} referring to the task image placement decisions into real variables. Next, we try to find a feasible solution that does not violate the storage capacity constraints by greedily converting the real values back to binary ones until all the task images and their replica are correctly placed (lines 3-12). With the decisions on task image placement, we can solve LP-IO-2 to obtain the I/O request distribution probabilities $q_{tij}, \forall t \in T$, $i \in I, j \in J$ (line 14). Then, we start the second stage for obtaining the task computation scheduling probabilities q_{tik} and $q_{ti}, \forall t \in T, i \in I, k \in K$ to minimize the maximum computation time. As LP-Comp is already in a linear form, we solve it directly in line 16. The third stage starts from line 18. We first take in the values of the probabilities obtained in the first two stages to calculate the expected I/O time on $j \in J$ and computation time on $k \in K$, respectively. Next, we also apply linear programming relaxation method to solve MILP-Joint. We first relax the binary variables $m_{hh'}$ in MILP-Joint to obtain LP-Joint-1. After solving LP-Joint-1, we then try to convert the values of $m_{hh'}$ back to binary ones provided that both constraints in (31) and (32) are not violated (lines 19-31). Finally, we take in the binary values of $m_{hh'}$ and solve LP-Joint-2 shown in line 32 to obtain the final solutions of p_{tijk} and p_{tij}.

5 Performance Evaluation

To evaluate the efficiency of our proposed three-stage heuristic algorithm ("Joint"), we develop a simulator strictly following the system model defined in Section 2. Specially, in order to show the advantage of our flexible task scheduling scheme, we introduce two competitors, i.e., server-greedy ("Server") and client-greedy ("Client") task scheduling schemes. The former greedily schedules all computation

tasks onto servers until all the servers are fully loaded, while the latter first tries to make them all solely on the client side until the clients are fully loaded.

Algorithm 1. Three-Stage Task Completion Time Minimization Algorithm

Stage 1:
Relax the integer variables in MILP-IO and solve the resulting linear programming LP-IO-1
for $t \in T$ **do**
  Sort $x_{tj}, \forall j \in J$ in descending order into set X_t
  for $x_t \in X_t$ and $\sum_{j \in J} x_{tj} < \Omega$ **do**
    $x_t \leftarrow 1$ and $x'_t \leftarrow 0, \forall x'_t \in X_t \setminus \{x_t\}$.
    $j \leftarrow$ the corresponding storage server of x_t
    if $\sum_{t \in T} x_{tj} s_t > S_j$ **then**
      recover the values of elements in X_t
    end if
  end for
end for
Take in the values of binary variables x_{tj} and solve

$$\text{LP-IO-2}: \quad \min: \bar{\tau}_d, \quad \text{s.t.}: (21), (22), (24), \text{ and } (26)$$

to obtain the values of $q_{tij}, \forall i \in I, j \in J$
Take the values of $q_{tij}, \forall i \in I, j \in J$ and derive the expected I/O time $\tau'^d_j, \forall j \in J$
Stage 2:
Solve LP-Comp and derive the values of q_{tik} and $q_{ti}, \forall t \in T, i \in I, k \in K$.
Stage 3:
Take the values of q_{tik} and $q_{ti}, \forall t \in T, i \in I, k \in K$ into (28) and (29) to derive the expected computation time on computation server $k \in K$ and client $i \in I$, respectively
Relax the integer variables in MILP-Joint and solve the resulting LP-Joint-1 by incorporating the values of q_{tik} and q_{tij}
for $h' \in J \cup K$ **do**
  Sort $m_{hh'}, \forall h \in J$ or K into set $M_{h'}$
  for $m_{h'} \in M_{h'}$ **do**
    $m_{h'} \leftarrow 1$ and $m'_{h'} \leftarrow 0, \forall m'_{h'} \in M_{h'} \setminus \{m_{h'}\}$
    $h \leftarrow$ the corresponding mapping server indicated by $m_{h'}$
    if $\sum_{h' \in J \text{or} K} m_{hh'} > 1$ **then**
      recover the values of elements in $M_{h'}$
    else
      break
    end if
  end for
end for
Take in the values of binary variables $m_{hh'}$ and solve

$$\text{LP-Joint-2}: \quad \min: \tau, \quad \text{s.t.}: (15), (16), (33), \text{ and } (34)$$

to obtain the final solutions of p_{tijk} and $p_{tij}, \forall t \in T, i \in I, j \in J, k \in K$

The default settings in our experiments are as follows: each storage server is with storage capacity of 60 and I/O handling rate of 90 while each computation server is with total service rate of 30. The computation service rates on the clients are all set as 10. Since network edges and clients are shared by many tasks, we randomly set the current resource availability rate of each device in the range of [0.7, 1.0]. In other words, if the resource availability of one storage server is 0.7, its available storage is 42. Each task totally requires 3 replicas to be saved in the storage servers and the size of task image is randomly set within the range of $[15, 25]$. The task arrival rate from a client is set as 9. The number of interrupts is set according to the task size as $n_t = \lceil \frac{s_t}{5} \rceil$. The transmission latencies between storage and computation servers in the same edge node is set as 0, while those between edge nodes and clients are randomly set in the range of $[0.1, 0.4]$. //We investigate how our "joint" algorithm performs in different settings and how various parameters affect the minimum value of the maximum task completion time.

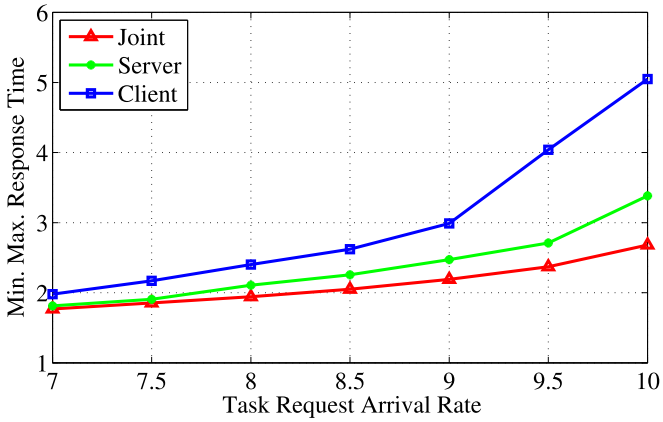


Fig. 4. On the effect of arrival rate.

In our three-stage algorithm, there are several LP formulations involved. All the LPs are solved by commercial solver Gurobi.[1] We investigate how our algorithm performs and how it is affected by various parameters by varying the settings in each experiment group. For each setting, average task completion time is obtained by 100 simulation instances.

5.1 On the Effect of Request Rate

We first compare the min-max task completion time of "Joint", "Server-greedy" and "Client-greedy" algorithms under different task arrival rates λ_{it} varying from 7 to 10 in Fig. 4. It can be noticed that the task completion time shows as an increasing function of the task arrival rate. As the task arrival rate increases, more requests for each task shall be handled by both storage servers and computation server/client. With the same I/O handling rate and computation service rate, the I/O and computation time becomes larger due to longer queuing delay. Nevertheless, the advantage of "Joint" algorithm over "Client-greedy" and "Server-greedy" algorithms can be always observed, especially when the task arrival rates are large. This is attributed to its flexibility in exploring the computation resources in both the computation servers and the clients.

1. http://www.gurobi.com

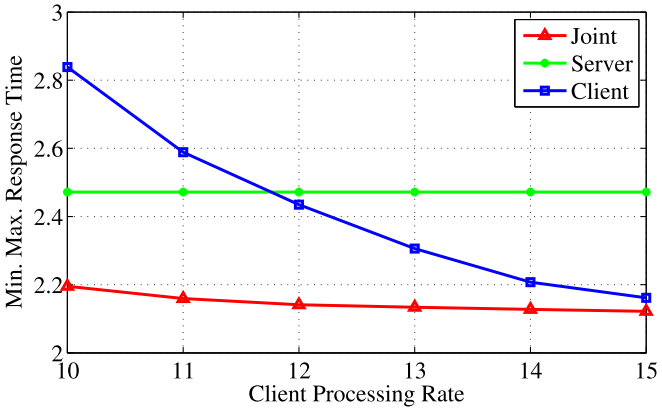


Fig. 5. On the effect of client service rate.

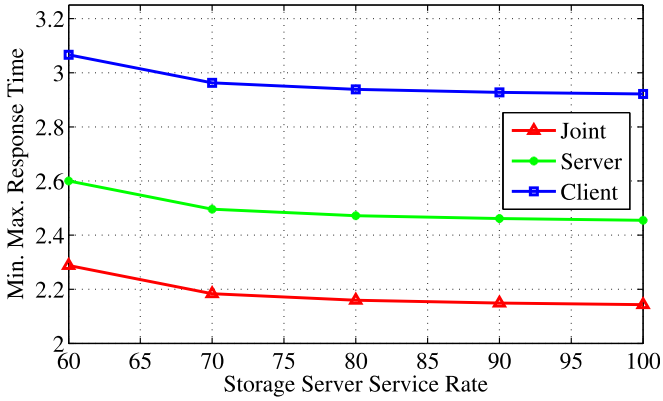


Fig. 7. On the effect of disk reading rate.

5.2 On the Effect of Computation Service Rate

Fig. 5 illustrates the task completion time as a function of the client service rate from 10 to 15 and Fig. 6 plots the task completion time under different computation server service rates increasing from 16 to 36. In Fig. 5, we first notice that the task completion time of our "Joint" and "Client-greedy" algorithms show as an decreasing function of the client service rate while the result of "Server-greedy" algorithm shows as a static value. This is because when the client server rate increases, more requests can be handled on client side with a faster processing speed, resulting in shorter computation time. In this case, increasing the client service rate will exert great positive effects on cutting the client computation time. Hence for "Client-greedy" computation, the response latency will drop significantly as shown in Fig. 5. With our "Joint" algorithm, part of requests shall be processed on client side with a faster speed. Increasing the computation rate also benefits the overall task completion time and hence the results also decrease as the client service rate increases. As for the "Server-greedy" algorithm, all requests are first processed on server side and therefore no much benefit can be obtained from increasing the client service rate.

Another interesting observation is that when increasing the client service rate, the results of our "Joint" and "Client-greedy" algorithms converge. For example, in Fig. 5 the response latency of "Joint" is 2.20 compared with "Client-greedy" with 2.83 when the client service rate is 10. However when client service rate researches 15, such advantage drops to 2.12 versus 2.17. The reason is that with the increasing of client service rate, the advantage of clients over computation servers also increases. Therefore, in our "Joint" algorithm more requests will be also processed on client side. There will be less optimization space left on server side. Therefore, "Joint" and "Client-greedy" algorithms give similar performance.

Similar phenomenon can be observed in Fig. 6. When the computation service rate on server is low, e.g., 20, the response latency of our "Joint" algorithm is 2.71 while that of "Server-greedy" is 3.24. This is because our algorithm reduces the response latency by assigning more tasks onto clients. Although remote I/O interrupts are required, we can still have higher performance thanks to the fast computation service rate on client. But when the service rate of computation servers increases, e.g., 36, more computation resources can be allocated to each task. Bearing the benefit on fast I/O interrupt handling, many tasks experience a low processing latency. In this case, the gap between "Joint" and "Server-greedy" decreases to 0.06. Nevertheless, our algorithm can always find a balancing point for appropriate task assignment to achieve better performance on any occasions.

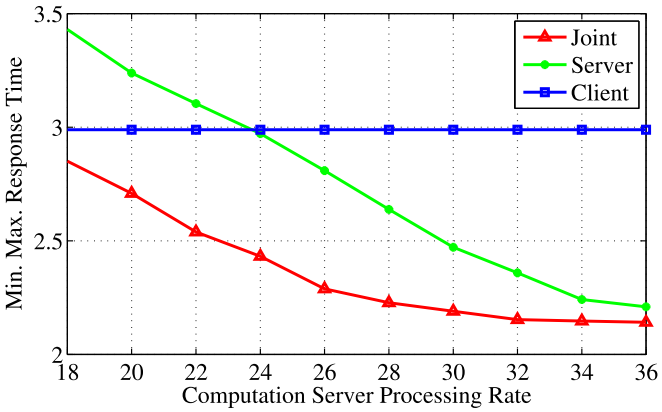


Fig. 6. On the effect of server service rate.

5.3 On the Effect of Storage Service Rate

Then, we investigate how the service rate of storage servers (i.e., I/O handling rate) affects the task completion time via varying its value from 60 to 100. The evaluation results are shown in Fig. 7. Note that no matter where the computation is handled, the I/O interrupt handling is inevitable. Hence, when the service rate increases, more I/O interrupts can be handled on each storage server per unit time. With the same task arrival rate, the I/O handling time of each storage server decreases, and so does the overall task completion time for all three algorithms. In summary, the advantage of our "Joint" algorithm can be observed under any values of service rate due to its flexility on exploring the computation resources.

5.4 On the Effect of Transmission Latency

To evaluate the effect of transmission latency, we introduce another competitor called "Joint w/o Mapping" to show the advantage of the mapping function in our heuristic algorithm "w. Mapping". The first two stages are the same for both algorithms. The only difference is on the third stage where "w/o Mapping" randomly selects the computation server and storage server for each task from a client provided that (33) and (34) are satisfied. In this group of experiments,

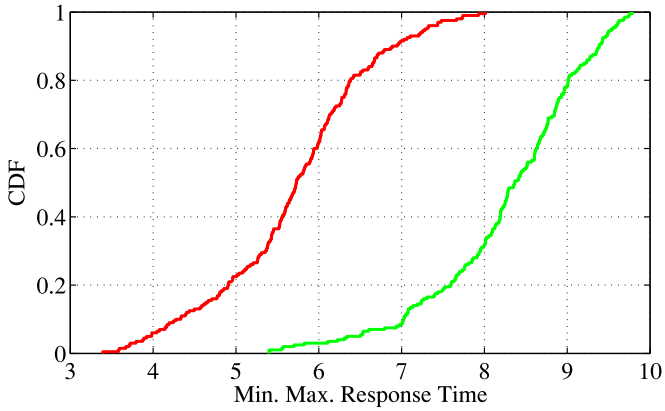


Fig. 8. CDF of the task completion time (w. mapping versus w/o mapping).

we set the transmission latency as a random value uniformly distributed in range $(0, 1)$ and total 500 simulation instances are conducted. We plot the Cumulative Distribution Function (CDF) of the min-max task completion time in Fig. 8. Note that even with the same I/O handling time and computation time, elaborate planning on the server mapping exhibits substantial advantage over the one without such consideration. In particular, the average task completion time of "w. Mapping" is almost 68.97 percent of the one achieved by "w/o Mapping". This further validates the efficiency of our design on server mapping in the third stage.

6 Related Work

6.1 Fog Computing

The advantages of fog computing are increasingly attracting the attentions of researchers and organizations for reliability, security and expenditure benefits. Zhu et al. [6] discover that it is possible to adapt the user conditions (e.g., network status, device computing load, etc.) to improve the web site performance by edge servers in fog computing architecture. Vaquero and Rodero-Merino [7] discuss several key technologies related fog computing and mention the importance of management in fog computing, but no actual solution is proposed. Mukherjee et al. [8] envision that it is possible to utilize the edge devices for computation, thereby extending the computation to the edges of a compute cloud rather than restricting it to the core of servers. Recently, Stantchev et al. [11] exemplify the benefit of cloud computing and fog computing to healthcare and elderly-care by an application scenario developed in OpSIT-Project in Germany.

6.2 Task Scheduling

Existing task scheduling studies mainly target cloud computing platform, where requests can be dynamically dispatched among all available servers. For example, Fan et al. [12] study power provisioning strategies on how much computing equipment can be safely and efficiently hosted within a given power budget and response latency. Rao et al. [13] investigate how to reduce electricity cost by routing user requests to geo-distributed data centers with accordingly updated sizes and guaranteed QoS. Delimitrou and Kozyrakis [14] present Paragon, a heterogeneity- and interference-aware online scheduler, which is derived from robust analytical methods, instead of by profiling each application. Recently, Liu et al. [15] re-examine the same problem by taking network delay as a part of overall data center cost. Gao et al. [16] propose an optimal workload control and balancing method by taking account of response latency, carbon footprint, and electricity costs. Liu et al. [17] reduce electricity cost and environmental impact using a holistic approach of workload balancing that integrates renewable supply, dynamic pricing, and QoS.

6.3 Resource Management

The resource management problem in distributed computing system, e.g., cloud computing, has been widely studied in the literature, especially in the form of VM placement. With virtualization technology, servers in data centers are organized as VMs with specific types, to meet requirements of various cloud service providers. User requests can be only distributed to these types of VMs that are deployed by the corresponding service provider. Liu et al. [18] propose GreenCloud architecture which enables comprehensive online-monitoring, live VM migration, and VM placement optimization. Cohen et al. [19] study the problem of VM placement for the traffic-intense data centers. Meng et al. [20] improve the network scalability by optimizing the traffic-aware VM placement according to the traffic patterns among VMs. Actually, task scheduling and VM placement are usually jointly investigated. For example, Zeng et al. [21] address the problem of VM placement to minimize the aggregated communication cost within a data center under the consideration of both architectural and resource constraints. Jiang et al. [22] study VM placement and routing problem to minimize traffic costs and propose an efficient online algorithm in a dynamic environment under changing traffic loads. Xu and Fortes [23] propose a two-level control system to manage the mappings of workloads to VMs and VMs to physical resources. It is formulated as a multi-objective optimization problem that simultaneously minimizes total resource wastage, power consumption and thermal dissipation costs. Virtualization technique has been adopted to network resources [24] as well.

Previous studies on performance optimization by resource management and task scheduling mainly focus on fog or cloud computing system. They cannot be applied to the software-defined embedded system due to the I/O interrupt handling on clients. To our best knowledge, we are the first to investigate the joint problem of resource management and task scheduling for FC-SDES in the literature.

7 Conclusion and Future Work

In this paper, we investigate the maximum task completion time minimization problem in FC-SDES via joint consideration of task image placement and task scheduling. The optimization problem is first formulated as an MINLP problem. To tackle the high computational complexity, we further propose a three-stage task completion time minimization algorithm. The first two stages individually deal with I/O time and computation time via transforming the min-max MINLP problems into equivalent max-min MILP ones. The third stage re-couples the I/O handling and task computation together with the incorporation of transmission time. Linear programming relaxation is applied to approximate

the solution of MILP problems in the algorithm. Performance evaluation results validate the high efficiency of our algorithm in minimizing the maximum task completion time in a FC-SDES.

With the consideration of memory gap between memory and disk, it is significant to study the memory management for FC-SDES. Different from traditional standalone embedded system where the memory is only used for applications on the embedded device, the memory on the storage server is shared by multiple embedded clients. Since traditional memory management scheme may not be applicable or does not perform well, we will investigate, as part of our future work, the memory management in FC-SDES by considering the I/O interrupt handling in a fine-grained manner.

Acknowledgments

This research was partially supported by Strategic International Collaborative Research Program (SICORP) Japanese (JST)—US (NSF) Joint Research "Big Data and Disaster Research" (BDD), the NSF of China (Grant No. 61272470, 61305087, 61402425, 61502439, 61501412), the Fundamental Research Funds for National University, China University of Geosciences, Wuhan (Grant No. CUG14065, CUGL150829), and the Provincial Natural Science Foundation of Hubei (Grant No. 2015CFA065).

References

[1] L. Gomes and J. J. Rodríguez-Andina, "Embedded and reconfigurable Systems," *IEEE Trans. Ind. Inform.*, vol. 9, no. 3, pp. 1588–1590, Aug. 2013.

[2] C. Li, B. Brech, S. Crowder, D. M. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, et al., "Software defined environments: An introduction," *IBM J. Res. Develop.*, vol. 58, no. 2/3, pp. 1–1, 2014.

[3] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, A. Rindos, et al., "SDIoT: A software defined based internet of things framework," *J. Ambient Intell. Humanized Comput.*, vol. 6, no. 4, pp. 453–461, 2015.

[4] J. C. Lyke, C. G. Christodoulou, G. A. Vera, and A. H. Edwards, "An introduction to reconfigurable systems," *Proc. IEEE*, vol. 103, no. 3, pp. 291–317, Mar. 2015.

[5] Flavio, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. 1st Edition MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[6] J. Zhu, D. S. Chan, M. S. Prabhu, P. Natarajan, H. Hu, and F. Bonomi, "Improving web sites performance using edge servers in fog computing architecture," in *Proc. IEEE 7th Int. Symp. Service Oriented Syst. Eng.*, 2013, pp. 320–323.

[7] L. M. Vaquero, and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *Proc. Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, 2014.

[8] A. Mukherjee, S. Dey, H. Paul, and B. Das, "Utilising condor for data parallel analytics in an iot context: An experience report," in *Proc. IEEE 9th Int. Conf. Wireless Mobile Comput. Netw. Commun.*, Oct. 2013, pp. 325–331.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. New York, NY, USA: Elsevier, 2012.

[10] C. Adjiman, I. Androulakis, and C. Floudas, "Global optimization of MINLP problems in process synthesis and design," *Comput. Chem. Eng.*, vol. 21, pp. S445–S450, 1997.

[11] V. Stantchev, A. Barnawi, S. Ghulam, J. Schubert, and G. Tamm, "Smart items, fog and cloud computing as enablers of servitization in healthcare," *Sensors Transducers*, vol. 185, pp. 121–128, 2015.

[12] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 13–23.

[13] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment," in *Proc. 29th Int. Conf. Comput. Commun.*, 2010, pp. 1–9.

[14] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. 18th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 77–88.

[15] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew, "Greening geographical load balancing," in *Proc. Int. Conf. Meas. Modeling Comput. Syst.*, 2011, pp. 233–244.

[16] P. X. Gao, A. R. Curtis, B. Wong, and S. Keshav, "It's not easy being green," in *Proc. ACM Special Interest Group Data Commun.*, 2012, pp. 211–222.

[17] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser, "Renewable and cooling aware workload management for sustainable data centers," in *Proc. Int. Conf. Meas. Modeling Comput. Syst.*, 2012, pp. 175–186.

[18] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "Greencloud: A new architecture for green data center," in *Proc. 6th Int. Conf. Ind. Session Autonomic Comput. Commun. Ind. Session*, 2009, pp. 29–38.

[19] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Almost optimal virtual machine placement for traffic intense data centers," in *Proc. IEEE INFOCOM*, 2013, pp. 355–359.

[20] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[21] H. H. S. Y. Deze Zeng, Song Guo, and V. C. Leung, "Minimizing inter-VM communication cost in data centers with VM placement," *Int. J. Auton. Adaptive Commun. Syst.*, vol. to appear, pp. 1–10, 2014.

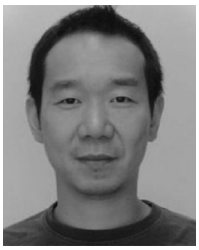
[22] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2876–2880.

[23] J. Xu and J. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *Proc. Green Comput. Commun.*, Dec. 2010, pp. 179–188.

[24] S. Zhang, Z. Qian, S. Guo, and S. Lu, "FELL: A flexible virtual network embedding algorithm with guaranteed load balancing," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2011, pp. 1–5.

Deze Zeng received the BS degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, China, and the MS and PhD degrees in computer science from the University of Aizu, Aizu-Wakamatsu, Japan, in 2007, 2009 and 2013, respectively. He is currently an associate professor in the School of Computer Science, China University of Geosciences, Wuhan, China. His current research interests include: cloud computing, software-defined sensor networks, data center networking, networking protocol design and analysis. He is a member of IEEE.

Lin Gu received the MS and PhD degrees in computer science from the University of Aizu, Fukushima, Japan in 2011 and 2015. She is currently a lecturer in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. Her current research interests include cloud computing, vehicular cloud computing, big data and software-defined networking. She is a member of IEEE.



Song Guo received the PhD degree in computer science from the University of Ottawa, Canada. He is currently a full professor at the University of Aizu, Japan. His research interests include the areas of wireless communication and mobile computing, cloud computing, big data, and cyber-physical systems. He has authored/edited 7 books and more than 300 papers in refereed journals and conferences in these areas. He serves/served in editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Emerging Topics in Computing*, *IEEE Communications Magazine*, *Wireless Networks, Wireless Communications and Mobile Computing*, and many other major journals. He has been the general/program chair or in organizing committees of numerous international conferences. He is a senior member of IEEE, a senior member of ACM, and an IEEE Communications Society Distinguished Lecturer.

Zixue Cheng (M'95) recieved the master's and doctor degrees in engineering from the Tohoku University, Japan, in 1990 and 1993, respectively. He joined the University of Aizu in April 1993 as an assistant professor, became associate professor in 1999, and has been a full professor since 2002. His research interests include design and implementation of protocols, distributed algorithms, distance education, ubiquitous computing, ubiquitous learning, embedded systems, functional safety, and Internet of Things (IoT). He served as the director of University-business innovation center during April 2006–March 2010, the head of the Division of Computer Engineering, during April 2010–March 2014, and has been the vice president of the University of Aizu since April 2014. He is a member of ACM, IEEE, IEICE, and IPSJ.

Shui Yu is currently a senior lecturer of the School of Information Technology, Deakin University, Australia. His research interests include security and privacy, big data, networking theory, and mathematical modelling. He has published two monographs and edited two books, more than 120 technical papers, including top journals and top conferences, such as IEEE TPDS, IEEE TC, IEEE TIFS, IEEE TMC, IEEE TKDE, IEEE TETC, and IEEE INFOCOM. He pioneered the research field of networking for big data in 2013. His h-index is 20. He is currently serving the editorial boards of *IEEE Transactions on Parallel and Distributed Systems, IEEE Communications Surveys and Tutorials, IEEE Access*, and a number of other international journals. He has organized eight security or networking related special issues with prestigious international journals, such as *IEEE Network* and *IEEE Cloud Computing*. He has served more than 70 international conferences as a member of organizing committee. He is a senior member of IEEE, and a member of AAAS.