BOAT: A Block-Streaming App Execution Scheme for Lightweight IoT Devices

Xuhong Peng, *Student Member, IEEE*, Ju Ren, *Member, IEEE*, Liang She, *Student Member, IEEE*, Deyu Zhang, *Member, IEEE*, Jie Li, and Yaoxue Zhang

***Abstract*—The contradiction between the limited capability of lightweight Internet-of-Things (IoT) devices and ever-increasing user demands is a fundamental and challenging problem in the era of IoT. One of the most important challenges is that lightweight IoT devices are generally embedded with fixed applications on resource-constrained hardware, and cannot enable on-demand service provisioning. In this paper, we propose BOAT, a block-streaming application execution scheme based on transparent computing (TC), which can remotely retrieve the necessary parts of traditional applications from edge servers on demand and run them on IoT devices locally. Specifically, we first exploit TC to build a scalable IoT system, where the applications of lightweight IoT devices are stored in edge servers or cloud but can be dynamically loaded on IoT devices in a block-streaming way. Moreover, we present a code partition approach to split the codes of a whole service into numerous functional blocks at the edge side. Such that, IoT devices only need to load necessary blocks of an application to obtain the requested services without loading the whole application codes. A lightweight I/O virtualization mechanism and a fine-grained relocation technology are then developed to support the block-streaming service loading. Our experimental results on a lightweight wearable device demonstrate that the proposed scheme can efficiently achieve flexible service provisioning with improved scalability and reduced service loading delay and energy consumption.**

***Index Terms*—Application partitioning, dynamic service provisioning, edge computing, Internet-of-Things (IoT), transparent computing (TC).**

Manuscript received December 19, 2017; revised February 24, 2018; accepted February 28, 2018. Date of publication March 6, 2018; date of current version June 8, 2018. This work was supported in part by the Natural Science Foundation of China under Grant 61702562 and Grant 61702561, in part by the Innovation-Driven Project of Central South University under Grant 2016CXS013, in part by the Fundamental Research Funds for the Central South University under Grant 2017zzts148, in part by the Aid Program for Science and Technology Innovative Research Team in Higher Educational Institutions of Hunan Province, in part by the International Science and Technology Cooperation Program of China under Grant 2013DFB10070, and in part by the China Hunan Provincial Science and Technology Program under Grant 2012GK4106. *(Corresponding author: Ju Ren.)*

X. Peng is with the School of Information Science and Engineering, Central South University, Changsha 410083, China, and also with the College of Information Engineering, Shaoyang University, Shaoyang 422000, China (e-mail: xvhongpeng@csu.edu.cn).

J. Ren, D. Zhang, J. Li, and Y. Zhang are with the School of Information Science and Engineering, Central South University, Changsha 410083, China (e-mail: renju@csu.edu.cn; zdy876@csu.edu.cn; jie.li@csu.edu.cn; zyx@csu.edu.cn).

L. She is with the School of Information Science and Engineering, Central South University, Changsha 410083, China, and also with the Mobile E-Business Collaborative Innovation Center of Hunan Province, Hunan University of Commerce, Changsha 410205, China (e-mail: sheliang@csu.edu.cn).

Digital Object Identifier 10.1109/JIOT.2018.2812803

I. Introduction

In the era of Internet-of-Things (IoT), lightweight IoT devices have become the dominating terminals to gather information and provide services for Internet users. Meanwhile, with the rapid development of information technology, the users' requirements on quality-of-service and variety-of-service have increased significantly. However, due to limited power-supply and poor storage/computing capability, lightweight IoT devices are facing great challenges in meeting the ever-increasing user demands. One of the critical problems is that most IoT devices are embedded with fixed applications for specific services, and cannot support dynamic service provisioning. It consequently motivates researchers to focus on improving the service scalability of lightweight IoT devices.

As a type of Internet-based computing, cloud computing was expected as a solution to provide scalable services for IoT devices. It enables the shared processing resources and data to be accessed by IoT devices on demand with minimal management effort [1]. However, due to the explosive growth of IoT devices, cloud computing has significant limitations on supporting context-aware and delay-sensitive IoT applications [2]–[4]. To cope with this problem, handling the data computing and storage requests of IoT devices by nearby devices/infrastructures becomes a new trend and promotes the emergence of edge computing paradigms, e.g., edge computing [5], fog computing [6], and cloudlet [7]. Such kind of computing paradigms have shown a great potential to support real-time and context-aware IoT applications, but few of them can provide on-demand service provisioning solutions for lightweight IoT devices in an efficient way. This is because dynamic services provided by them are generally associated with virtualization technologies or Web services, which are still immature or not suitable for lightweight IoT devices. Moreover, as current Apps are integrating increasing number of functions, it consequently increases the storage cost and energy consumption of running a service through dynamically installing an App on lightweight IoT devices. In fact, most of Apps usually spend 80% of the CPU time in executing only 10% of the codes (sometimes even less 5% or even 1%), and the residual 90% of codes may not be necessary for users. If lightweight IoT devices can only load the necessary codes for providing a service rather than loading the whole application, both the quality-of-service and energy efficiency would be significantly improved.

As a special kind of edge computing, transparent computing (TC) [8] provides such a promising solution to provision scalable services for lightweight IoT devices at the edge of network. It designs an on-demand service execution architecture which can fully utilize the local computing capabilities of client devices to remotely fetch on-demand services from the server side and locally execute them via a block-streaming way. A large number of existing works proposed over the past decade have shown its effectiveness and performance in creating scalable desktops and tablets (e.g., [8] and [9]). However, as we are stepping into the IoT era, the new devices bring new obstacles in applying TC to build scalable IoT platforms. For example, how to devise an efficient solution to provide streaming services for resource-limited IoT devices via unstable wireless networks. As a core technology of TC, streaming service execution is to enable lightweight IoT devices to load the necessary parts of an application from edge servers and locally execute it, without downloading and installing the whole codes of the application. Such that, lightweight IoT devices can run applications larger than their physical memory, and efficiently execute a wide range of applications with their limited computation and storage capability. By such means, it is possible to fully explore the potential of lightweight IoT devices and make them more flexible. However, since most of lightweight IoT devices are embedded with real-time operating systems (e.g., FreeRTOS and TinyOS) and their applications are generally programmed in C or C++ language (not in Java) [10], existing streaming services based on Java Virtual Machine (JVM) are not feasible any more. Thus, how to design an efficient streaming execution scheme becomes critical but also faces some new challenges. First, since different components of an application are dependent on each other and difficult to be executed separately, it consequently makes effectively and efficiently partitioning an application to enable it to be partially fetched and executed on a lightweight OS platform be a challenging problem. Second, in TC-based IoT system, IoT devices should dynamically load the partitioned components of an application from remote edge servers, which means the local input/output (I/O) interrupts should be redirected to the server side via wireless networks. Thus, how to design an efficient I/O virtualization technology for TC-based IoT system needs to be carefully investigated. Third, after a new executable component of an application is retrieved from the edge side, how to prepare for its execution and how to link it with existing components on the IoT devices need to be addressed.

In this paper, we propose BOAT, a TC-based block-streaming application execution scheme, to support efficient dynamic service provisioning on lightweight IoT devices. BOAT works in a TC-based IoT architecture consisting of three layers, including IoT device layer, edge server layer, and cloud server layer. When IoT devices need to load a specific service that is missed in their storage, they send a request to edge servers to download the necessary parts of the application. If the request application is not stored in edge servers, it will be downloaded from cloud servers to edge servers for responding the requests of IoT devices. By leveraging BOAT, lightweight IoT devices can dynamically load and locally execute the necessary blocks of the requested application without installing the whole application. In such a way, dynamical service provisioning on lightweight IoT devices can be achieved efficiently. Specifically, the contributions of this paper can be summarized as follows.

1) We develop an automatic code partition approach to split the machine codes of a whole application into numerous functional blocks based on cross-platform executable and linking format (ELF) files. Such that, IoT devices only need to load necessary blocks to obtain the requested functionalities without loading the whole application codes. Based on that, a block-streaming application execution scheme, named BOAT, is then proposed.
2) We design a lightweight I/O virtualization mechanism to support block-streaming application execution in a TC-based IoT system, and design a fine-grained relocation approach for a new executable block to relocate and link it with existing blocks.
3) We implement the proposed BOAT scheme on TC-based wearable devices named TCWatch. The experimental results on TCWatch demonstrate that BOAT can effectively improve the scalability and energy efficiency of lightweight IoT devices, as well as reduce the service loading delay.

The remainder of this paper is organized as follows. Section II reviews the related works. In Section III, we present the TC-based IoT system architecture and some background information about the file format of IoT service. Section IV presents the details of BOAT, and Section V evaluates its performance based on a prototype system. Finally, we conclude this paper and outline some future research directions in Section VI.

II. Related Work

There have been a number of related works focusing on enhancing the service scalability of lightweight devices. In this section, we briefly review the existing solutions, in terms of dynamic service provisioning and application partitioning.

A. Dynamic Service Provisioning

Software-as-a-Service provided by cloud computing, such as Salesforce [11] and Google Docs [12], is an on-demand software delivery paradigm where business softwares are centrally hosted by vendors and typically accessed through Web browsers [13]. This solution offers some advantages, such as scalability, lower maintenance and management costs, better resource utilization etc. However, applications hosted in the cloud are specialized and dedicated, and it cannot support traditional ones. Another on-demand service provisioning model involving cloud computing are thin-client approaches, such as [14], XenApp [15], and VMware Horizon [16]. In the thin-client/server architecture, servers take the whole processing, management, and deployment burdens for applications, while clients are only designed to provide basic I/O functions. Nevertheless, since computations are almost offloaded to servers in this solution, it is extremely demanding on cloud resources. Recent works on mobile cloud computing seek to jointly leverage local and server computation for application collaborative execution [17]–[19]. However, cloud computing is facing increasing limitations to meet the delay-sensitive, and context-aware service demands of IoT devices. Moving service provisioning close to IoT devices becomes a new option and promotes the emergence of edge computing paradigms [5], [6]. Hassan *et al.* [20] integrated all the personal storage space of a user (her laptops, desktops, etc.) together to expand storage capacity of mobile devices. Satyanarayanan *et al.* [7] enabled users to rapidly instantiate customized application on a proximate infrastructure and then use that software over a one-hop wireless network based on virtual machine technologies. Zeng *et al.* [21] considered a fog computing-based software-defined embedded architecture, where task images are placed in storage servers. This paper focuses on issues such as task image placement among storage servers and systematical resource management. Ren *et al.* [22] proposed a TC-based IoT architecture to improve the scalability of lightweight IoT devices, and clearly identify its benefits and key challenges [23]. Moreover, Ren *et al.* [24], [25] aimed at improving the energy-efficiency of narrow-band and unstable wireless communications to guarantee the service performance of TC-based IoT platforms. In summary, the basic idea behind these works is to move storage/computation to the edge or local network close to IoT devices, hence, to address the challenges, such as low bandwidth, high latency, and enable location-dependent IoT applications.

B. Application Partitioning

Java applets are popular for application streaming [26]. An applet can be run without obtaining entire classes used by it. When a class is not available on the local JVM at runtime, the local JVM can dynamically retrieve the missing class file from the server. Existing works leverage Java reflection mechanism to load classes, interfaces, fields and methods at runtime. For example, in [27], a face recognition software is divided into nine components based on Java classes. Similarly, [28] and [29] enable splitting mobile applications at the method level via special static annotations which are against modifying Java codes. Li and Gao [30] performed automated method-level application partitioning over practical Android OS. Differently, COMET [31] partitions Java applications at the thread level. Moreover, other works, like [32]–[35], attempt to consider the dependency of partitioned components of an application. Deng *et al.* [32] organized components of an application as a workflow. Zhang *et al.* [33] modeled a mobile application with a linear topology graph. Mahmoodi *et al.* [34] illustrated dependencies of a video navigation application's 14 components by a component dependency graph. Wang *et al.* [35] abstracted an application as a graph that has tree topologies. In summary, these mentioned works mostly partition applications based on JVM, while JVMs require a nontrivial memory space and executive efficiency overhead. For example, memory requirements of JVMs range from kilobytes to megabytes, and the execution time of completing a native method by using JVM bytecode is 25.53 times of that by using machine code [36].

III. System Architecture and Background

In this section, we first present a TC-based IoT architecture to support block-streaming service loading on lightweight IoT devices, then introduce some background information about the file format of IoT services and our design objectives.

A. Transparent Computing-Based IoT Architecture

TC-based IoT architecture aims to provide a service-oriented computing, and makes the details of service provisioning transparent to users [22]. It enables users to enjoy their desired services on demand, without concerning the installation, management, and upgrade of services on their devices. With TC, the traditional Von Neumann architecture of a single IoT device is spatio-temporally extended for connected computers/devices in a wireless networking environment. Service codes are stored in edge servers, and can be dynamically streamed to and executed with local resources (CPU and memory) on IoT devices. Fig. 1 shows the system architecture of supporting block-streaming service loading based on TC. It mainly consists of three layers: 1) an IoT device layer; 2) an edge server layer; and 3) a cloud server layer.

1) IoT Device Layer: This layer consists of various lightweight IoT devices, such as sensor nodes, wearable devices, vehicles, smart hardwares, etc. These devices are widely heterogeneous and highly constrained in computation and storage resources, and act as TC clients. Due to the limited resources and energy supply, they are usually embedded with fixed software/functions for specific services running on a lightweight real-time OS, e.g., TinyOS, FreeRTOS, and Contiki. The client-agent for block-streaming service denoted by block-streaming execution platform(i) [BSEP(i)] is located upon the underlying OS to enable dynamic and streaming service loading.

2) Edge Server Layer: This layer is responsible for distributed computing, control, and storage for IoT devices at the edge of the network. Devices in this layer act as TC servers, which can be various mobile smart devices or infrastructures in different IoT scenarios. For example, it can be smart phones for wearable devices, programmable high performance routers for smart devices, or base stations for communication devices. Edge servers maintain App stores that reserve some frequently used services for IoT devices. When an IoT device requests a new service, the edge server can send the codes and related data of the requested service to it through the server-agent, denoted by BSEP(e), in a block-streaming way. The communication between the IoT device and the edge server is based on some specific wireless communication protocols, such as WiFi, ZigBee, and Bluetooth.

Specially, BSEP(e) and BSEP(i) within these two layers are responsible for remotely loading the required codes and data of applications on demand from the server side, and executing the computation with the local CPU and memory resources at the client side, in a timely manner. In such a way, the storage is extended from the local IoT device to the edge server via the network, and the I/O and system interrupts can be redirected from the local device to the edge side.

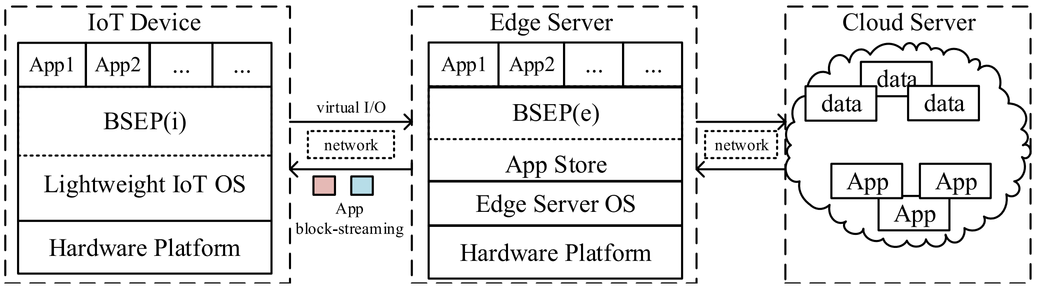


Fig. 1. System architecture.

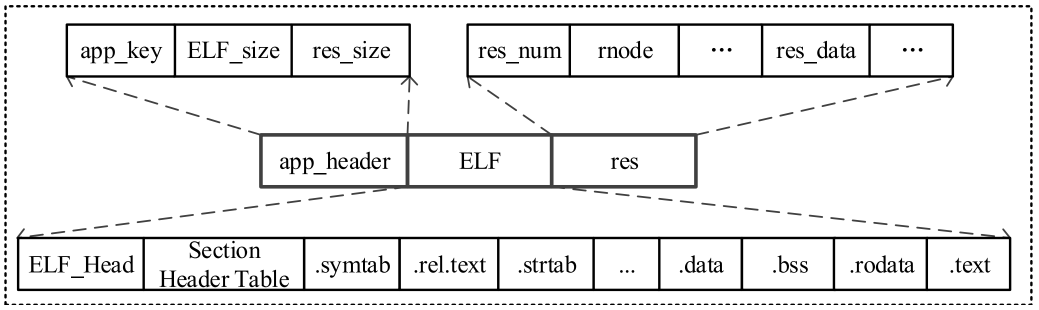


Fig. 2. Structure of APK.

3) Cloud Server Layer: The cloud layer is responsible for large-scale computing and huge data storage. The devices in this layer are composed of a cluster of powerful servers. This layer can also be a commercial cloud platform, e.g., Amazon EC 2 or Google Cloud. The central cloud servers store and maintain full services and application data that can be transmitted to edge servers to respond the service requests of IoT devices and keep all the services up to date.

B. App Installation Package of IoT Services

Since we aim to partition Apps and remotely load them in a block-streaming way, we first briefly introduce an App installation package (APK) format of IoT services in this section.

As shown in Fig. 2, each APK file consists of three parts: *app_header*, *ELF*, and *res*. We illustrate each part in detail as follows.

1) App_header: It resides at the beginning of an APK file and holds some members describing the APK's organization. The *app_key*, *ELF_size*, and *res_size* refer to the identification of an App, the offset within the file of the *ELF* and the *res* separately.

2) ELF: ELF is a common standard file format for executable files, object codes, shared libraries etc. [37]. It is flexible, extensible, cross-platform, and has been widely adopted by many different OSes such as UNIX-based systems on different hardware platforms. It contains multiple auxiliary components for setting App runtime environment, such as *ELF_Head*, *Section Header Table*, *.symtab*, *.rel.text*, *.strtab* etc.

1) *ELF_Head:* It resides at the beginning of an ELF file and holds a road map describing the organization of the ELF.
2) *Section Header Table:* It contains a section entry array that is used to locate all sections of this part.
3) *.symtab:* It holds information needed to relocate a program's symbolic definitions and symbolic references.
4) *.rel.text:* It specifies where relocations are needed within *.text* in order. OSes have to resolve the needed symbol by its name, and then write the symbol address to the place specified in the relocation entry.
5) *.strtab:* It holds null-terminated character sequences, commonly called strings.

These strings are used to represent symbol and section names. A string can be referenced as an index into this section. In addition, as shown in Fig. 3, there are four sections involving App functions' execution.

1) *.data:* It contains initialized static variables, such as global variables and static local variables. Since the values of variables may be altered at runtime, this section can be read and written.
2) *.bss:* It contains entire global variables and static variables that are initialized with zero or do not make explicit initialization in source codes. Typically only the length of this section, while no actual data, is stored in the ELF. The App loader allocates and initializes memory for this section when it loads the program.
3) *.rodata:* It contains static constants rather than variables, such as constants or strings.
4) *.text:* It is usually the major component of an ELF, and it contains executable instructions.

Machine instructions are organized in function modules based on their edited sequence before they are compiled. It is typically read-only and has a fixed size, so on IoT devices it can usually be placed into read-only memory, without the need for loading into RAM.

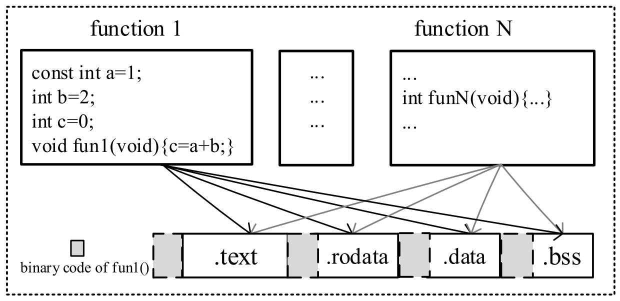


Fig. 3. Functions and binary image.

3) Res: This part contains various resources that codes use, such as images, custom fonts, user interface strings, and more. These resources are commonly used to draw portions of graphical user interfaces for controls and views. They are loaded and used by Apps executable codes in running. When being needed at runtime, Apps obtain the appropriate resource data in *res* through the index information in *res_head*. The *res_head* contains two fields: 1) *res_num* and 2) *rnode*. The *res_num* holds the number of resources, and the *rnode* holds the metadata of its *res_data*. There are four members in *rnode*: 1) the *res_key* and *res_type* represent the identification and type of the *res_data*, respectively; 2) the *offset* keeps the offset within *res* of the *res_data*; and 3) the *size* is the size of the *res_data* in bytes. The *res_data* includes the raw data of each resource. In terms of size, this part is always the major component of an APK.

C. Design Objectives

We aim to design a block-streaming App execution scheme, named BOAT, under the TC-based IoT architecture. The main objectives of BOAT are as follows.

1) *Dynamic Service Provisioning for Lightweight IoT Devices:* In BOAT, Apps of lightweight IoT devices can be stored in edge servers but can be dynamically loaded on IoT devices for execution.
2) *Block-Streaming Service Loading:* Each IoT App should be partitioned to a number of "functional blocks" according to its APK file. When IoT devices request some specific functionalities of a service, the requested "functional blocks" can be dynamically retrieved from edge servers and executed on IoT devices in a streaming way.
3) *Automatic Application Partitioning and Block Loading:* By providing a block-streaming execution platform for users, BOAT should be able to partition traditional Apps and execute the blocks on IoT devices automatically, without any modifications or additional efforts on these Apps.

IV. Block-Streaming App Execution Scheme With I/O Virtualization

In this section, we propose a block-streaming App execution scheme with I/O virtualization, named BOAT, and present its design details.

A. Overview

We first present an overview of our proposed block-streaming application execution scheme, i.e., BOAT.

BOAT works in the TC-based three layer architecture. When an IoT device needs to start a specific App that is not found in its storage, it sends a request to the edge server to download the necessary components of the APK. If the requested App is not stored in the edge server, it will be downloaded from the cloud server to the edge server for responding the request of the IoT device. During the runtime, since there is no whole APK in local, the execution process may be interrupted. IoT devices can capture the interrupts and dynamically load the requested components of the APK from the edge server, then locally execute them.

The designed BSEP in BOAT is shown in Fig. 4. The *App managers* are in charge of Apps' life cycle management, such as installation/uninstallation. In particular, the *block manager* determines which blocks should be locally maintained or destroyed according to network condition, memory availability, and user preference, to make a tradeoff between quality-of-service and resource utilization efficiency. The *block loader* places blocks into memory and prepares them for execution. The *block detector* is used to check which blocks are locally saved. The *virtual I/O protocol* provides reliable communication protocols to fetch required blocks from edge servers for various virtual I/O operations. The *block generator* located at the edge side is used to generate blocks on demand. Specially, the *App partitioner* dynamically splits the APK into separate blocks. Since individual components of an App are mutually dependent on each other, arbitrary partitioning is not feasible. Suppose that a block is partitioned from the binary image as shown in Fig. 3. As the components of *fun*1() are not all included in the current block, there will be multiple interrupts to complete its execution due to the missed components. Note that, in terms of size, *.text* and *res* are the main components of APK. Meanwhile, instructions are organized with function modules within *.text*, and the required functions or resources may be varied and associated with different application functionalities or users actions. Thus, the *App partitioner* only splits *.text* and *res* into blocks on demand, while other parts are delivered to IoT devices as a startup block. The partitioned blocks of *.text* and *res* are named as *.textlet*, *reslet*, respectively.

Specifically, the execution process of an App includes two phases, initialization phase and block-streaming execution phase. The detailed processes are introduced as follows.

1) Initialization Phase: As shown in Fig. 5(1), when an App is launched, a startup block will be partitioned from the APK and delivered to the IoT device. The startup block contains the auxiliary data such as *ELF_Head* and *Section Header Table* etc. (excluding *.rel.text*), the whole data of *.data*, *.bss*, *.rodata*, *res_head*, and a startup *.textlet* (see Section IV-B2). Since *.text* and *.rodata* are read only after relocation, the *block loader* loads them into ROM. Since *.data* and *.bss* are frequently read and written at runtime, they are loaded into RAM. This strategy can help improve the utility of RAM. Afterward, the *relocation module* carries out relocation operations for the

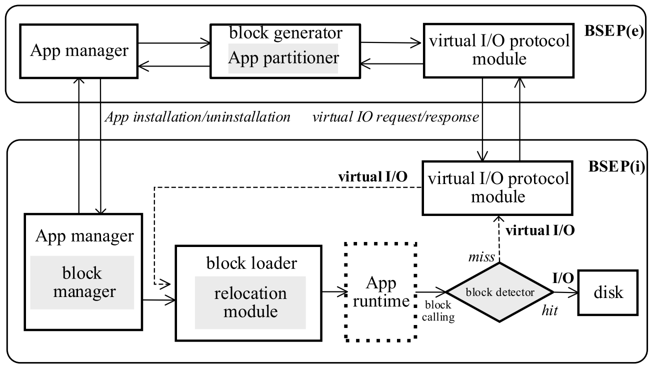


Fig. 4. Module diagram of BOAT.

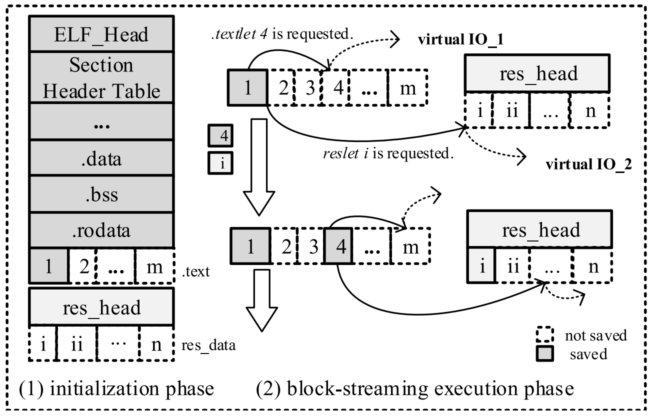


Fig. 5. App execution process.

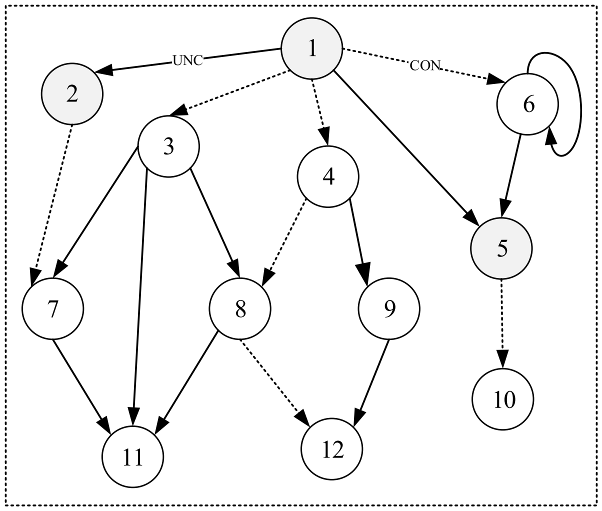


Fig. 6. Example of CRG model.

startup *.textlet*, and then the App is scheduled as a new task by OS kernels for its execution.

2) Block-Streaming Execution Phase: When the App is running, the required *.textlet* or *reslet* will be loaded on demand. The *.symtab* and a Boolean array named *bitmap* are designed to detect which *.textlets* and *reslets* are stored in local, respectively. If an undetected block is requested at runtime, it means a *miss* happens; otherwise a *hit* occurs. As shown in Fig. 5(2), during the execution process of *.textlet* 1, *.textlet* 4 and *reslet i* are requested. Since these blocks are not locally saved, the execution process will be interrupted, and two kinds of virtual I/O requests are generated, denoted as *virtual IO*_1 and *virtual IO*_2, respectively. These requests are captured, and redirected to the edge server according to the virtual I/O protocol via networks. The *App partitioner* located at the edge side splits the target data from the APK, and then generates a new *.textlet* or *reslet* block. When these blocks are received, the *block loader* on the IoT side will load them into memory. If it is a *.textlet* block, the *relocation module* will carry out relocation operations to prepare for its running and link it with existing blocks. After that, the App execution process will be resumed. During the following runtime, if *misses* happen continuously, these procedures will be repeated, thus the block-streaming is formed.

The remainder of this section is organized as follows. The methods of *.textlet* generation, fine-grained relocation, and *virtual IO*_1 generation are introduced in Section IV-B. The approach for loading *reslet* is illustrated in Section IV-C. The virtual I/O protocol is presented in Section IV-D.

B. .textlet-Streaming

In this section, we present an approach to load *.textlet* on demand. First, we explore how to model and partition *.text*. Then, we study how to generate *.textlet* blocks on demand. After that, when a new *.textlet* is loaded, we investigate how to prepare for its running and how to link it with existing blocks. Finally, we describe how a *virtualIO*_1 occurs at runtime.

1) .text Modeling and Partitioning: Instructions within *.text* are organized with function modules. The dependences of functions can fall into three categories: 1) unconditional invocation; 2) conditional invocation (executing a function only if some conditions are met); and 3) no invocation. Thus, *.text* can be abstracted as a call relationship graph (CRG), in which vertices represent functions of *.text*, and arcs represent invocation relationships among vertices.

We define the *.text* as a digraph $G = (V, E)$ which consists of a set of vertices, V, and a set of arcs, E. Each $v \in V$ denotes a function, whereas each $\varepsilon = (v_i, v_j) \in E$ denotes the arc ε between vertices v_i and v_j. The graph G must satisfy these general properties: 1) each vertex should have at least in-degree of one (except the first vertex that has in-degree of zero); 2) each vertex has out-degree of K which can be zero; 3) all vertices should have at least one direct or indirect path from the first vertex so that they are invoked in the program; 4) a vertex may have an arc leading to itself, because a function can invoke itself; and 5) ε has three values, UNC for unconditional invocation which means a path, CON for conditional invocation which means it will be a path when some conditions are met, and NONE means no path. Fig. 6 shows an example of CRG model.

After *.text* is modeled as a digraph G, we can turn its partitioning problem into a process to find a subset of v in G. The execution processes of an App can be viewed as different paths in G, where required functions are associated with the application functionalities or users' actions. When a function

Algorithm 1 Partition()

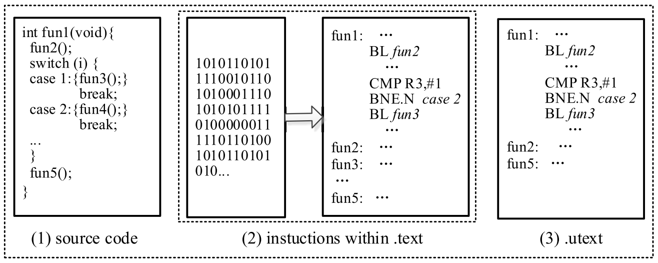
```
Input: Graph $G$, vertex $v$.
Output: $SubGraph$.
 1: $visited[v] \leftarrow TRUE$ ;
 2: while ($w \leftarrow NextAdjVex(G, v)$) do
 3:   if ($visited[w]$ is $FALSE$) then
 4:     if ($e(v, w)$ is $UNC$) then
 5:       if ($status[w]$ is $WHITE$) then
 6:         put $w$ into $SubGraph$;
 7:         $status[w] \leftarrow BLACK$;
 8:       end if
 9:     else
10:       $visited[w] \leftarrow TRUE$ ;
11:       continue;
12:     end if
13:     $Partition(G, w)$;
14:   end if
15: end while
16: return ;
```

v is invoked, all vertices that have at least one direct or indirect path from the vertex v are necessary for its running. We define v, these vertices, and arcs among them as a *SubGraph*. For example, in Fig. 6, v_1, v_2, and v_5 construct a *SubGraph*. How to partition a *SubGraph* from G is shown in Algorithm 1.

In Algorithm 1, we repeatedly perform depth-first partitioning, starting at the input vertex v until all connected vertices have been visited. The output of this algorithm is the *SubGraph*. $NextAdjVex(G, v)$ is designed to find the next connected vertex of v in G. A Boolean array *visited*[] is initialized to *FALSE*, and when visiting a vertex v, we mark it with *TRUE* (line 1). For each unvisited w adjacent to v, we recursively perform depth-first partitioning on all its adjacent vertices. If $e(v, w)$ is UNC (line 4), and w is not partitioned before (line 5), we put it into the *SubGraph* (line 6). A Boolean array *status*[] is initialized to *WHITE*, and when a vertex w is cut down, we mark it with *BLACK* (line 7). After that, we recursively call this procedure for all w's adjacent vertices (line 13). If $e(v, w)$ is CON, we recursively call this procedure for v's next adjacent vertices (line 11).

2) .textlet Generation: In BOAT, since not all functions within *.text* are locally saved, a program's execution process will be interrupted due to the invocation of a missed function. If it happens, the symbol of the function will be captured and delivered to the edge side. The *App partitioner* calls $Partition(G, v)$ to split a *SubGraph* from the graph G using the input v which represents the missed function.

As shown in Fig. 7, we define the instructions of functions contained in the *SubGraph* (e.g., v_1, v_2, and v_{11} in Fig. 6) within *.text* as a *.utext*. Fig. 7(1) shows the C language source codes, where control flow statements can be *switch* and *if*/*else* etc. Fig. 7(2) shows the binary image within *.text* and its assembly instructions. At the assembly language level, control flow instructions are conditional or unconditional branch instructions, such as B, $B <$ *condition* $>$ (supported by



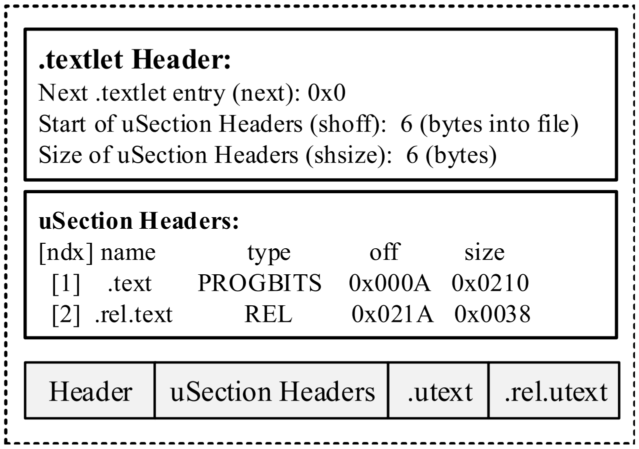
```
int fun1(void){
  fun2();
  switch (i) {
  case 1:{fun3();}
          break;
  case 2:{fun4();}
          break;
  ...
  }
  fun5();
}
```

```
1010110101
1110010110
1010001110
1010101111
0100000011
1110110100
1010110101
010...
```

```
fun1:  ···
       BL fun2
        ···
       CMP R3,#1
       BNE.N  case 2
       BL fun3
        ···
fun2:  ···
fun3:  ···
 ···
fun5:  ···
```

```
fun1:  ···
       BL fun2
        ···
       CMP R3,#1
       BNE.N  case 2
       BL fun3
        ···
fun2:  ···
fun5:  ···
```

Fig. 7. .utext generation.



.textlet Header:
Next .textlet entry (next): 0x0
Start of uSection Headers (shoff): 6 (bytes into file)
Size of uSection Headers (shsize): 6 (bytes)

uSection Headers:

[ndx]	name	type	off	size
[1]	.text	PROGBITS	0x000A	0x0210
[2]	.rel.text	REL	0x021A	0x0038

Header	uSection Headers	.utext	.rel.utext

Fig. 8. Structure of *.textlet*.

ARM cores) etc. Since the function *fun*1() invokes the functions *fun*2() and *fun*5() unconditionally, their instructions are extracted from *.text* and put together to generate an instruction block named as *.utext*, as shown in Fig. 7(3).

The *.utext* is appended with *Header*, *uSection Headers*, and *.rel.utext* to generate a *.textlet*. The structure of *.textlet* is shown in Fig. 8. The *Header* and *uSection Headers* are introduced as follows, and *.rel.utext* related to relocation process will be introduced in Section IV-B3. We define the *.textlet* including an App's entry function [such as "*main*()"] as a startup *.textlet*.

Header resides at the beginning of *.textlet* and holds a road map describing its organization. The *next* indicates the entry address of the next *.textlet*. The *shoff* indicates the byte offset from the beginning of *.textlet* to *uSection Headers*. The *shsize* denotes the *uSection Headers*'s size in bytes. All the section headers are the same size.

uSectionHeaders is an entry array designed to locate all of sections of *.textlet*. A section header is an entry to a section. The *ndx* is a subscript into this array. The *name* denotes the name of the section. The *type* categorizes the section's contents and semantics (e.g., *PROGBITS* indicates program content, including code, data, etc., and *REL* indicates relocation entries). The *addr* gives the address at which the section's first byte should reside. The *off* indicates the byte offset from the beginning of *.textlet* to the first byte in the section. The *size* indicates the section's size in bytes.

3) .textlet Relocation and Virtual IO_1 Generation: There are three steps before running a new *.textlet*: 1) copy the

.symtab:

num	value	size	type	bind	ndx	*name*
...						
15:	0x40355	0	OBJECT	GLOBAL	3	*B*
1c:	0x823	60	FUNC	GLOBAL	1	*fun1*
1d:	VirA	28	FUNC	GLOBAL	1	*fun2*
23:	0x143	40	NOTYPE	GLOBAL	UND	*api1*
...						

.rel.utext:

off	info	type	localoff	*name*
...				
0x038	0x151a	R_ARM_GOT_BREL	0x00	*B*
0x14a	0x230a	R_ARM_THM_CALL	0x00	*api1*
0x1ec	0x1c1a	R_ARM_GOT_BREL	0x00	*fun1*
0x1f0	0x1d1a	R_ARM_GOT_BREL	0x00	*fun2*
...				

Fig. 9. *.symtab* and *.rel.utext*.

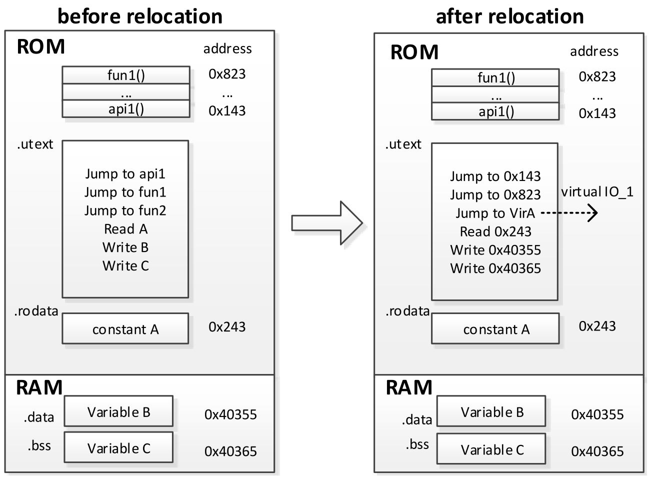


Fig. 10. Relocation and *virtual IO*_1 generation.

.textlet from the load region into the execution region; 2) add the *.textlet* to a linked list connecting the *.textlets*; 3) perform relocation for the *.utext*; and 4) jump to the entry address of the *.utext*. In step 1), *.utext* is assigned to an execution space with an entry address *text_enrty*. As the position-dependent instructions of *.utext* are assigned with physical addresses after step 1), the symbols of function and data within these instructions should be modified to reflect the assigned addresses. In addition, the new loaded block should be linked with the existing blocks. Thus, relocation in step 3) is a critical process to connect symbolic references with symbolic definitions.

As shown in Fig. 9, there are two auxiliary sections related to relocation: 1) *.symtab* and 2) *.rel.text*, which provide auxiliary information on how to perform relocation.

The *.symtab* holds global information needed to relocate a program's symbolic definitions and symbolic references. The *num* represents a string table index that gives the symbol name. The *value* holds the value of the associated symbol. Its value can be a physical address or a virtual address (*VirA*) depending on the context. The *size* holds the size of the associated symbols. The *info* indicates the symbol's type *type* (e.g., *OBJECT* represents data; *FUNC* represents a function; and *NOTYPE* represents an extern symbol), and the binding attribute *bind*. The *ndx* holds the relevant section header table index (e.g., 1 represents an index for the *.text*; 3 represents an index for the *.data*; and *UND* represents an extern symbol).

The *.rel.utext* describes how to make relocation for *.utext*. It holds one or multiple items with the same size denoted as *item_size*. Each item has three members, *offset*, *info*, and *local*. The *off* gives the location where to apply the relocation action. For *.utext*, its value indicates a section offset. The *info* gives both the symbol table index, with respect to which the relocation should be made, and the type of relocation to apply. For example, high byte of *info* holds the symbol table index of the symbol needed to be relocated (i.e., *num* of *.symtab*), and low byte of *info* holds the type of relocation (e.g., *R_ARM_THM_CALL* indicates extern function invocation relocations within Thumb *BL* instructions, and *R_ARM_GOT_BREL* indicates normal function or data call relocation mechanisms). The *localoff* indicates an offset within current *.utext*. If its value is *zero*, it represents that the symbol is not defined within current *.utext*.

Relocation principle is shown in Fig. 10. Specifically, the procedures of *.textlet* relocation are shown in Algorithm 2. The input parameter *.textlet_ptr* is the entry address of *.textlet*. First, the entry addresses of *uSectionHeaders*, *.utext*, and *rel.text* denoted as *sh_tab*, *.utext_entry*, and *.reltab* are calculated separately (lines 2–4). Second, *.utext* is copied to the allocated RAM space (lines 5 and 6). Third, we compute the reference address denoted as *addr* for each symbol (a symbol item is indexed via *idx*) within *rel.text* (lines 7–37). Specifically, through *reltab*[*idx*] within *.rel.text* (line 8), we obtain which symbol (*num*) within *.symtab* is involved in the relocation (line 9). Then, we determine whether current symbol is defined within current *.utext*. If it is, we compute the *addr*, and update *symtab*[*num*].*value* and *is_computed_flag* (*is_computed_flag* is used to mark whether the *addr* of current symbol is calculated or not, and the *DONE* means the *addr* is calculated) (lines 10–13). Otherwise, we compute the *addr* through searching *.symtab* (lines 14–35). In this situation, if *is_computed_flag* indicates *DONE*, we copy the *symtab*[*num*].*value* to *addr* (lines 15 and 16). Otherwise, we compute *addr* according to *symtab*[*num*].*info*. If it indicates *OBJECT*, we compute *addr* according to the entry address of each section and *symtab*[*num*].*value* (*data_entry*, *ro_entry*, and *bss_entry* denote the entry addresses of *.data*, *.rodata*, and *.bss*) (lines 18–27). If it indicates *NOTYPE*, we look up the symbol table of OS and obtain the address of current symbol (the *strtab* represents the entry of *.strtab*) (lines 28–30). If it indicates *FUNC*, we allocate a virtual address *VirA* for *addr* (lines 31 and 32). As symbol values within instructions are processor-bias [38], the computed *addr* needs to be masked (line 38). After that, the masked *addr* is written into the relocation location (line 39). Finally, if all the relocation operations are made, we copy the *.utext* from RAM to ROM. It is note that when we carry out relocation for a new *.textlet*, we also need to iterate over all

Algorithm 2 .textlet Relocation

```
Input: $.textlet\_ptr$.
Output: $void$.
Initialize $idx \leftarrow 0$;
$sh\_tab \leftarrow .textlet\_ptr + shoff$;
$utext\_enrty \leftarrow .textlet\_ptr + sh\_tab[0].off$ ;
$reltab \leftarrow .textlet\_ptr + sh\_tab[1].off$;
allocate $sh\_tab[0].size$ bytes RAM space at $temp$;
copy $.utext$ from $utext\_enrty$ into $temp$;
while ($idx < sh\_tab[1].size$ / $item\_size$) do
    obtain the relocation symbol $reltab[idx]$;
    $num \leftarrow$ high byte of $reltab[idx].rel\_info$;
    if ($reltab[idx].localoff$ is nonzero) then
        $addr \leftarrow utext\_entry + reltab[idx].localoff$ ;
        $symtab[num].value \leftarrow addr$;
        $is\_computed\_flag[num] \leftarrow DONE$;
    else
        if ($is\_computed\_flag[num]$ indicates $DONE$) then
            $addr \leftarrow symtab[num].value$;
        else
            if ($symtab[num].info$ indicates $OBJECT$) then
                if ($symtab[num].ndx$ indicates $.rodata$) then
                    $addr \leftarrow ro\_entry + symtab[num].value$;
                end if
                if ($symtab[num].ndx$ indicates $.data$) then
                    $addr \leftarrow data\_entry + symtab[num].value$;
                end if
                if ($symtab[num].ndx$ indicates $.bss$) then
                    $addr \leftarrow bss\_entry + symtab[num].value$;
                end if
            else if ($symtab[num].info$ is $NOTYPE$) then
                search OS symbol table via $strtab + num$;
                $addr \leftarrow$ the address of the found symbol;
            else if ($symtab[num].info$ is $FUNC$) then
                $addr \leftarrow VirA$ ;
            end if
            $is\_computed\_flag[num] \leftarrow DONE$;
        end if
        update $symtab[num].value \leftarrow addr$;
    end if
    mask $addr$ via processor-specific addends;
    write the masked $addr$ at $temp + reltab[idx].off$;
    $idx \leftarrow idx + 1$;
end while
copy $.utext$ from RAM to ROM;
```

the members of the *.textlet* linked list, and perform relocations for the prior *.utexts* where symbols are defined within current *.utext*.

After relocation, the *.utext* will be executed. A branch instruction within *.utexts* may cause the CPU to execute program in another block. As shown in Fig. 10(2), a *virtual IO*_1 request is generated when the program jumps to the *VirA*, and the App execution process will be interrupted. After entering the interrupt service routine, the *virtual IO*_1 request is packaged according to a virtual I/O protocol described in Section IV-D, and then transmitted to the edge server to retrieve the required block.

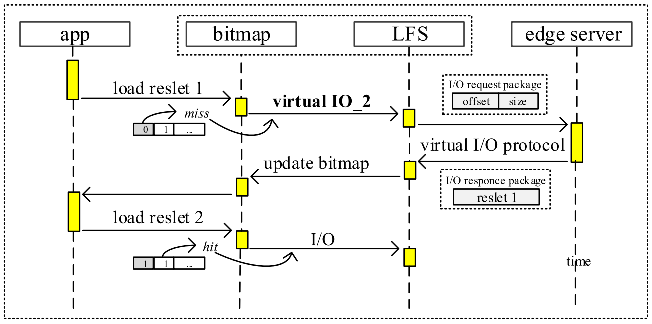


Fig. 11. Loading sequence diagram.

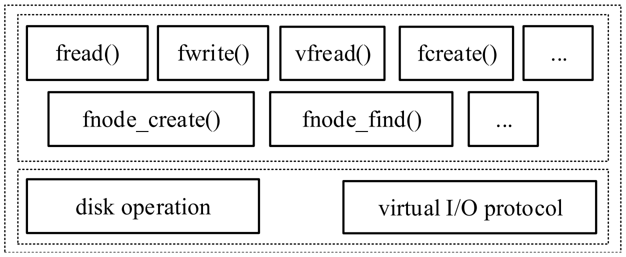


Fig. 12. Structure of LFS.

C. Reslet-Streaming

In this section, we present an approach for loading *reslet* dynamically. First, we give a definition of *reslet* and make an overview about its loading. Then, we design a lightweight file system to support *reslet*-streaming. Finally, we describe *reslet* loading algorithms based on the file system.

1) Overview of Reslet Loading: Apps use resource files to store data that are independent of the code that uses them. The resource files are usually managed by a file system on IoT devices. We design a lightweight file system named LFS to support *reslet*-streaming.

The *reslet* loading sequence diagram is shown in Fig. 11. Since executable codes load resources in the unit of *res_data*, we define a *res_data* within *res* as a *reslet*. When an App needs to load a *reslet* during its runtime, it will first detect whether the block is cached in local or not via a 1-D array named *bitmap*[]. If a *hit* happens, LFS will read the target data from a local disk. Otherwise, a *virtual IO*_2 request will be generated, and the target data will be retrieved from the edge server. Specifically, LFS packs the virtual I/O request into a package according to the virtual I/O protocol, then this package is transmitted to the edge side. The edge server partitions the target *reslet* from the APK, and delivers it to the IoT device with a virtual I/O response package. Finally, the App updates the *bitmap* and loads this block into memory.

2) File System: LFS manages how resources are stored and accessed. As shown in Fig. 12, it consists of two layers logically, a file operation layer and a disk operation layer. The file operation layer provides interfaces to handle I/O operations

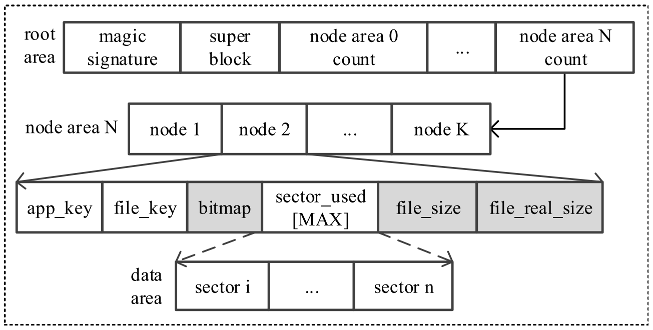


Fig. 13. On-disk layout.

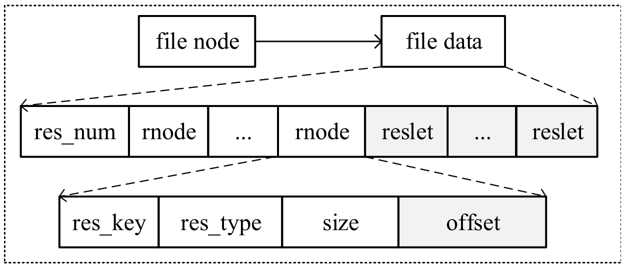


Fig. 14. Structure of resource file.

on files, such as *create*, *read*, and *write* etc. For example, the *fread*() reads data with a given size from the local file, while the *vfread*() reads data from the edge server. When a *virtual IO_2* is generated, the *vfread*() will be invoked (see Section IV-C3). The device operation layer is responsible for interacting with storage device drivers. Note that when the *vfread*() is called, this layer will be replaced by a virtual I/O protocol.

The on-disk layout of LFS is shown in Fig. 13. LFS splits the entire volume into three areas.

1) *Root Area:* The *magic signature* holds default parameters of LFS. The *super block* has sector occupation information. The *node area x count* contains the number of file nodes stored in the *node area x*.
2) *Node Area:* It stores all the file nodes created in the file system. Each node contains the metadata of a file. The *app_key* represents the unique identification of an App. The *bitmap* indicates whether a *reslet* is locally saved or not. The *sector_used*[*MAX*] points to the physical sectors of a file, and the maximum size of a file is $MAX*$ *sector size*. The *file_real_size* indicates the actual size of *res*, while the *file_size* represents its size cached in local.
3) *Data Area:* It stores the actual resource data.

The structure of a resource file is shown in Fig. 14. Since *reslets* are loaded on demand, their organization orders within the file may be different from the *res_data* within *res* of the APK. If a block is locally saved, the *offset* within the *rnode* represents the actual offset of the *reslet* into the file. Otherwise, it indicates the offset of *res_data* within *res* of APK.

Based on the on-disk layout above, the interface *fnode_find*() looks up a target file node as the following steps.

1) Compute the target node area *node area x* by (*file_key*$\&0x0000000F)\%N$.

Algorithm 3 app_res_load()

```
Input: app_key, file_key, res_key.
Output: void.
1:  invoke fnode_find() to obtain fnode;
2:  rnode_off ← (res_key−5000)∗rnode_size ;
3:  offset ← rnode_off;
    size ← rnode_size;
4:  invoke fread() to obtain rnode;
5:  update offset with rnode.offset;
    update size with rnode.size;
6:  if (fnode.bitmap indicates a hit happens) then
7:      invoke fread() to obtain reslet;
8:  else
9:      invoke vfread() to obtain reslet;
10: end if
11: update offset with fnode.file_size;
12: invoke fwrite() to write the reslet;
13: rnode.offset ← file_size;
14: update offset with rnode_off;
    update size with rnode_size;
15: invoke fwrite() to write the rnode back;
16: fnode.file_size ← fnode.file_size + rnode.size;
17: update the res_key−th bit of fnode.bitmap with 1;
18: return ;
```

2) Obtain the numbers of nodes within the target node area (*node area x count*).
3) Look up the target file node that is specified in its input parameters *app_key* and *file_key* among the *node area x count* nodes in *node area x*. Through the metadata provided by the found node, we can obtain an on-disk location stored the file data.

3) Reslet Loading Algorithms: The application program interface (API) *app_res_load*() is designed to handle resource I/O requests. It locates and loads resource data into memory at runtime. The procedures of the *app_res_load*() are shown in Algorithm 3. Line 1 invokes the *fnode_find*() with input parameters *file_key* and *app_key* to find the target file node *fnode*. Line 2 computes the offset of the *rnode* into the file that is specified by the *res_key* (the number 5000 is the *res_key* of the first *res_node*; the *rnode_size* is the size of *rnode*). Line 4 reads out the *rnode* from the resource file. The *fread*() and *vfread*() read data with a given size from a file specified in its input parameters *app_key* and *file_key*. It begins reading at the location that is pointed to by its input parameter *offset*, and continues reading up to the number of bytes that is specified in its input parameter *size*. It returns the read data through the buffer that is pointed to in its out parameter *buffer*. Line 6 checks whether the target *reslet* is saved locally or not. If a *hit* happens, the *reslet* is read from the local disk (line 7). Otherwise, a *virtual IO_2* request is generated, and the *reslet* is read from the edge server (line 9). Line 12 writes the obtained *reslet* into the file. The *fwrite*() writes data from a buffer to a file specified in its input parameters *app_key* and *file_key*. It copies a number of bytes that are specified in its input parameter *size* from the buffer that is pointed to by its

Algorithm 4 vfread()

```
Input: app_key, file_key, offset, size.
Output: The address of the data buffer $*data$.
 1: while ($vrq.rcvsize < size$) do
 2:   Pack vrq into a virtual read request packet;
 3:   Transmit the package via the network driver;
 4:   Wait for a virtual read response from the edge server;
 5:   if a virtual read response package is received then
 6:     Update $address \leftarrow *data + vrq.rcvsize$
        Copy the received data to the address;
 7:     $vrq.rcvsize \leftarrow vrq.rcvsize + vrp.datasize$;
 8:     if ($vrq.rcvsize == size$) then
 9:       break;
10:     else
11:       $vrq.rqsize \leftarrow (size - vrq.rcvsize) > B^* ? B^* : size$;
12:     end if
13:   end if
14: end while
15: return $*data$;
```

input parameter *buffer* into the specified file. It begins placing the buffered data at the location within the file that is pointed to by its input parameter *offset*. Line 15 writes the updated *rnode* back. Lines 16 and 17 update *fnode*.

The procedures of *vfread*() are shown in Algorithm 4. The virtual read request denoted as *vrq* includes five fields: 1) the *vrq.app_key* indicates the key of the requested App; 2) the *vrq.offset* keeps the offset of the requested *reslet* within *res*; 3) the *vrq.total_size* holds the total size of the requested data; 4) the *vrq.rcvsize* indicates the size of data already received; and 5) the *vrq.rqsize* keeps the data size of current request. The *vrq* is initialized via the input parameters. Line 2 packs *vrq* into a virtual read request packet according to the virtual I/O protocol. This package is transmitted to the edge server via the network driver (line 3). When a virtual read response package is received, the target data will be copied to the output buffer (lines 5 and 6). Line 7 updates the *vrq.rcvsize* (the size of data read from the edge side is denoted as *vrp.datasize*). After that, if there are still data left to be read, the *vrq.rqsize* will be updated (lines 10 and 11) (the B^* is the maximum size of requested data in each package), and the procedure will go to line 1. Otherwise, this procedure will be finished (lines 8 and 9).

D. Virtual I/O Protocol

In BOAT, assuring data reliability is crucial as it may cause IoT devices crash. Hence, we use reliable communication protocols such as TCP between the edge and the IoT device.

The communication data of a virtual I/O package is shown in Fig. 15. The *magic_num* field defines the version of the virtual I/O protocol. The *length* holds the size of the package in bytes. The *IO_ID* keeps I/O commands. For example, 01 represents a virtual read request of *virtual IO*_1, while 51 represents its virtual read response. The *magic_check* is used to check the integrity of *data*. The *data* field holds the actual payload data.

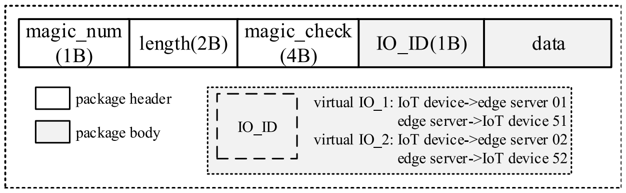

magic_num (1B)
length(2B)
magic_check (4B)
IO_ID(1B)
data
package header
package body
IO_ID
virtual IO_1: IoT device->edge server 01
edge server->IoT device 51
virtual IO_2: IoT device->edge server 02
edge server->IoT device 52

Fig. 15. Structure of virtual I/O package.

TABLE I
CONFIGURATIONS OF TCWATCH AND EDGE SERVER

Parameter	TCWatch	Edge server
CPU	NRF52832 (Cortex-M4 core)	MSM8996 (Quad-Core)
CPU Frequency	64M HZ	1.8G HZ
Network	BLE (S132 SoftDevice)	BLE&LTE&WiFi
Screen Resolution	176*176	1920x1080
RAM	64KB	3GB
ROM	512KB	32GB
Battery Capacity	350mAh	3000mAh
OS	TCOS 1.0	Android 7.0

Each I/O packet is transferred as the communication protocol payload.

V. PERFORMANCE EVALUATION

In this section, we implement a BOAT-based scalable wearable computing platform and evaluate its performance.

A. System Prototype

As shown in Fig. 16, the designed system consists of three devices: 1) a smart watch named TCWatch as the IoT device; 2) a smart phone (MI 5) as the edge server; and 3) a high-performance PC as the cloud server.

TCWatch can keep lightweight hardwares and softwares like FitBit but load a variety of third-party applications dynamically from the edge server. It communicates with MI 5 via the Bluetooth Low Energy 4.0, while MI 5 communicates with the PC via WiFi. The primary hardware and software configurations of TCWatch and MI 5 are listed in Table I. We develop the BSEP(i) based on a lightweight OS (named as TCOS) on TCWatch to remotely load different Apps from the MI 5. TCWatch can select an on-demand App from an application list on MI 5. Compared with TCWatch, the smart phone is equipped with a more powerful processor and a larger storage space. We develop the BSEP(e) based on a software named TCstore on the MI 5. TCstore can provide frequently used Apps to TCWatch when they are requested. If the requested Apps are not stored on the MI 5, they will be downloaded from the cloud server. Moreover, MI 5 can provide real-time data analyzing and storing services for the sensed data of TCWatch. The cloud server stores and maintains a variety of Apps, and also can process complex data analyzing tasks. Developers can publish their developed Apps on the cloud server for TCWatch to use.

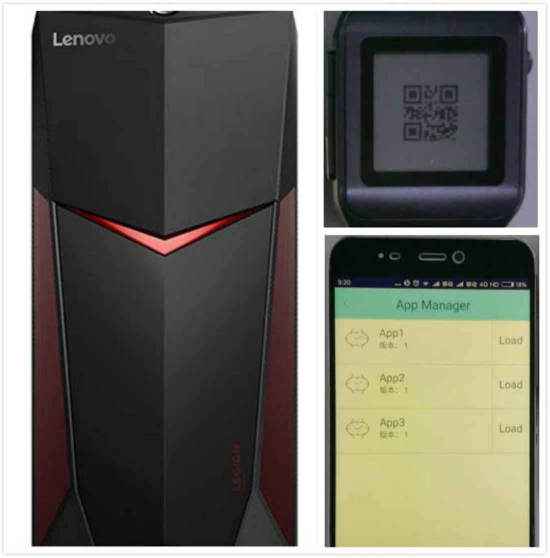


Fig. 16. System prototype.

B. Experimental Results

In order to evaluate the performance of BOAT, we compare BOAT-based TCWatch and a traditional smart watch under varied App sizes, in terms of the startup latency, downloading ratio, energy consumption, and overhead. The compared traditional smart watch adopts the same hardware and software specifications as TCWatch, but follows a traditional App execution procedures where an App is first completely downloaded into its ROM from the cloud and then installed and executed locally on the watch.

1) App Startup Latency: App startup latency is the interval between when it is chosen to download and when the CPU executes its first instruction. As shown in Fig. 17, with the increment of data transmission, the App startup delay of TCWatch can be improved significantly when compared with the traditional smart watch, because the App can start running as soon as the first executable unit is loaded into TCWatch's memory. For example, when the App size is 73.8 KB, TCWatch reduces up to 81% startup delay. In addition, the App startup time is nearly directly proportional to its size on the traditional smart watch, while the larger size of the App is, the greater proportion of startup delay will be reduced on the TCWatch.

2) Downloading Ratio: Downloading ratio is the size of blocks downloaded locally, divided by the total size of an APK. Taking the App with 14.3 KB in Fig. 18 as an example, TCWatch needs only 14.0% of whole program for the App's startup. The download ratios of Apps depend on their internal structure. Since the startup block is maintained during an App's life circle while other blocks can be removed after they are used, TCWatch is more effective in saving storage space where Apps have small startup blocks but with many user interactions. Fig. 19 shows different download ratios versus user actions for an App named pedometer. Pedometer records the number of steps users have walked, how many calories burned, the distance, walking time etc. It displays these data for one day within a watch face (live data display), while draws statistics charts for a week within another watch face (statistics display). These watch faces include image resources for better views. Pedometer also can share its motion data with the smart phone (sharing). As shown in Fig. 19, the download ratio increases sharply when image resources are required, because their sizes are usually large [e.g., the size of a 176×176 image is 11 616 bytes (176×176×3/8)].

3) Energy Consumption: Fig. 20 shows the energy consumption comparison between TCWatch and traditional smart watch under running different Apps for 5 h. It can be seen from the figure that TCWatch can reduce 0.48%–10.78% energy consumption in executing the Apps with different sizes. This is because that only a small portion of App codes are loaded and running in TCwatch, which can effectively improve the energy efficiency.

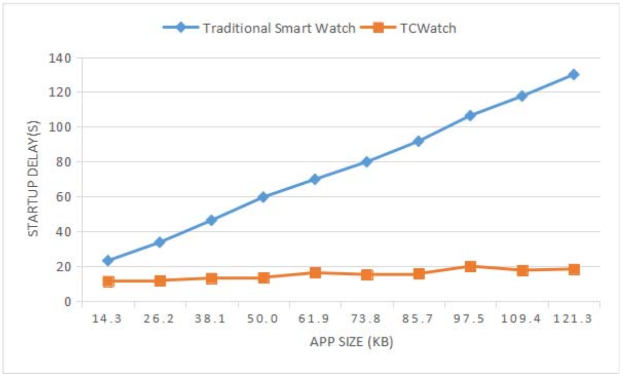


Fig. 17. Startup delay under different App size.

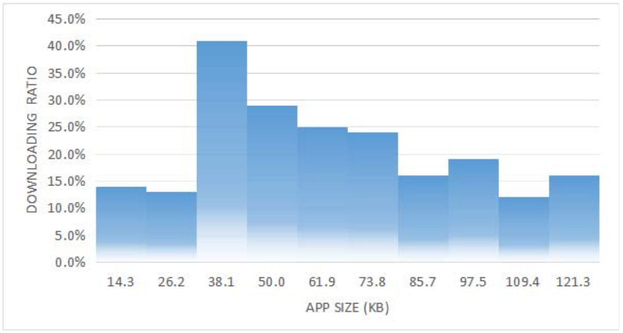


Fig. 18. Downloading ratio when startup.

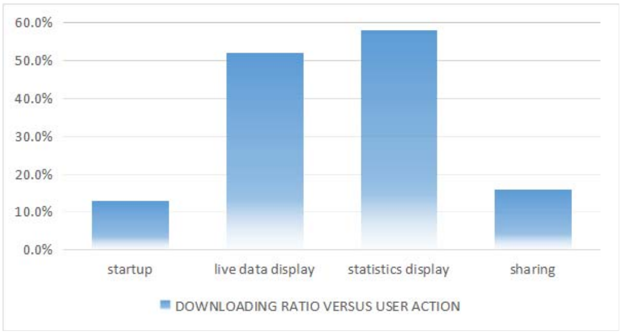


Fig. 19. Downloading ratio versus user action.

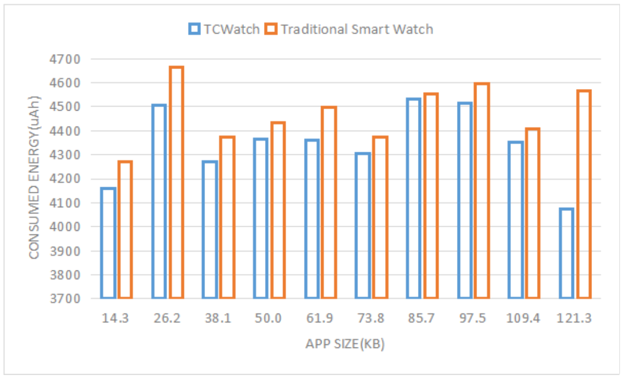


Fig. 20. Energy consumption of different Apps.

C. Discussion on the Overhead of BOAT and TCWatch

One of the overheads in BOAT is the extra bytes added for enabling data streaming. In BOAT, each *.textlet* adds total 20 bytes of extra data to the original data, which consists of: 1) six bytes of *.textlet Header*; 2) six bytes of *uSection Headers*; and 3) eight bytes of communication package header. Each *.reslet* only adds total eight bytes of transmission overhead. Runtime overheads occur when blocks are not in memory, which are involving block partitioning at the edge side, block transmission, and block relocation. The time complexity for a new *.textlet* partitioning is $O(n_\varepsilon)$, where n_ε is the arc number in the *SubGraph* at the edge side. Furthermore, the time complexity for relocation is $O(n)$, where n is the number of function symbols needing to be relocated on IoT devices. The block transmission time is associated with network throughput.

VI. Conclusion

Due to physical limitations, most IoT devices are embedded with fixed applications for specific services, and cannot support dynamic service provisioning. In this paper, we propose BOAT, a TC-based block-streaming application execution scheme. By leveraging BOAT, lightweight IoT devices can dynamically load and locally execute the necessary parts of the requested application without installing the whole application. In such a way, dynamical service provisioning on lightweight IoT devices can be achieved efficiently. The experimental results demonstrate that the proposed approach can efficiently improve the scalability of resource-constrained IoT devices. In our future work, we aim to improve the performance of BOAT from the following two perspectives. To mitigate the impacts of loading delay caused by unstable wireless communications, it is necessary to study an effective prefetching strategy based on user action preferences. Moreover, cache management strategies can be carefully studied to cooperate with the prefetching strategy to further improve the quality of experience.

References

[1] Y. Zhang *et al.*, "A survey on emerging computing paradigms for big data," *Chin. J. Electron.*, vol. 26, no. 1, pp. 1–12, Jan. 2017.
[2] Y. Xu and A. Helal, "Scalable cloud–sensor architecture for the Internet of Things," *IEEE Internet Things J.*, vol. 3, no. 3, pp. 285–298, Jun. 2016.
[3] Y. Kawamoto *et al.*, "A feedback control-based crowd dynamics management in IoT system," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1466–1476, Oct. 2017.
[4] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato, "A survey on network methodologies for real-time analytics of massive IoT data and open research issues," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1457–1477, 3rd Quart., 2017.
[5] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through VM migration and transmission power control," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 810–819, May 2017.
[6] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. ACM 1st Ed. MCC Workshop Mobile Cloud Comput.*, Helsinki, Finland, 2012, pp. 13–16.
[7] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.
[8] Y. Zhang and Y. Zhou, "Transparent computing: Spatio-temporal extension on von Neumann architecture for cloud services," *Tsinghua Sci. Technol.*, vol. 18, no. 1, pp. 10–21, Feb. 2013.
[9] Y. Zhang and Y. Zhou, "TransOS: A transparent computing-based operating system for the cloud," *Int. J. Cloud Comput.*, vol. 1, no. 4, pp. 287–301, 2012.
[10] *IoT Developer Trends 2017 Edition*. Accessed: Mar. 15, 2018. [Online]. Available: https://ianskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/
[11] *Salesforce Platform*. Accessed: Mar. 15, 2018. [Online]. Available: http://www.salesforce.com
[12] R. Attebury, J. George, C. Judd, B. Marcum, and N. Montgomery, "Google docs: A review," *Against Grain*, vol. 20, no. 2, p. 9, 2013.
[13] L. Wu, S. K. Garg, S. Versteeg, and R. Buyya, "SLA-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *IEEE Trans. Services Comput.*, vol. 7, no. 3, pp. 465–485, Jul./Sep. 2014.
[14] A. Bujari, M. Massaro, and C. E. Palazzi, "Vegas over access point: Making room for thin client game systems in a wireless home," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 2002–2012, Dec. 2015.
[15] G. R. James, *Citrix XenDesktop Implementation: A Practical Guide for IT Professionals*. Amsterdam, The Netherlands: Elsevier, 2010.
[16] J. Ventresco, *Implementing VMware Horizon 7*. Birmingham, U.K.: Packt, 2016.
[17] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-user computation partitioning for latency sensitive mobile cloud applications," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2253–2266, Aug. 2015.
[18] A. Bourdena, C. X. Mavromoustakis, G. Mastorakis, J. Rodrigues, and C. Dobre, "Using socio-spatial context in mobile cloud offload process for energy conservation in wireless devices," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2015.2511736.
[19] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Trans. Services Comput.*, vol. 8, no. 2, pp. 175–186, Mar./Apr. 2015.
[20] M. A. Hassan, M. Xiao, Q. Wei, and S. Chen, "Help your mobile applications with fog computing," in *Proc. IEEE 12th Annu. Int. Conf. Sens. Commun. Netw. Workshops (SECON Workshops)*, Seattle, WA, USA, 2015, pp. 1–6.
[21] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3702–3712, Dec. 2016.
[22] J. Ren, H. Guo, C. Xu, and Y. Zhang, "Serving at the edge: A scalable IoT architecture based on transparent computing," *IEEE Netw.*, vol. 31, no. 5, pp. 96–105, Aug. 2017.
[23] D. Zhang, R. Shen, J. Ren, and Y. Zhang, "Delay-optimal proactive service framework for block-stream as a service," *IEEE Wireless Commun. Lett.*, to be published, doi: 10.1109/LWC.2018.2799935.
[24] J. Ren *et al.*, "Lifetime and energy hole evolution analysis in data-gathering wireless sensor networks," *IEEE Trans. Ind. Informat.*, vol. 12, no. 2, pp. 788–800, Apr. 2016.
[25] J. Ren, Y. Zhang, N. Zhang, D. Zhang, and X. Shen, "Dynamic channel access to improve energy efficiency in cognitive radio sensor networks," *IEEE Trans. Wireless Commun.*, vol. 15, no. 5, pp. 3143–3156, May 2016.
[26] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*. London, U.K.: Pearson Educ., 2014.

[27] S. Cao, X. Tao, Y. Hou, and Q. Cui, "An energy-optimal offloading algorithm of mobile computing based on HetNets," in *Proc. IEEE Int. Conf. Connected Veh. Expo (ICCVE)*, Shenzhen, China, 2015, pp. 254–258.

[28] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, Orlando, FL, USA, 2012, pp. 945–953.

[29] H. Flores *et al.*, "Mobile code offloading: From concept to practice and beyond," *IEEE Commun. Mag.*, vol. 53, no. 3, pp. 80–88, Mar. 2015.

[30] Y. Li and W. Gao, "Minimizing context migration in mobile code offload," *IEEE Trans. Mobile Comput.*, vol. 16, no. 4, pp. 1005–1018, Apr. 2017.

[31] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proc. OSDI*, vol. 12, 2012, pp. 93–106.

[32] S. Deng, L. Huang, J. Taheri, and A. Y. Zomaya, "Computation offloading for service workflow in mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3317–3329, Dec. 2015.

[33] W. Zhang, Y. Wen, and D. O. Wu, "Collaborative task execution in mobile cloud computing under a stochastic wireless channel," *IEEE Trans. Wireless Commun.*, vol. 14, no. 1, pp. 81–93, Jan. 2015.

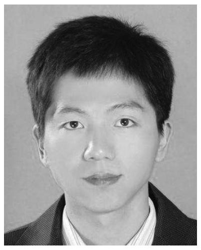
[34] S. E. Mahmoodi, R. N. Uma, and K. P. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2016.2560808.

[35] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.

[36] X. Liu *et al.*, "Java virtual machine based infrastructure for decent wireless sensor network development environment," in *Proc. IEEE 9th Int. Conf. Ubiquitous Intell. Comput. 9th Int. Conf. Auton. Trusted Comput. (UIC/ATC)*, Fukuoka, Japan, 2012, pp. 120–127.

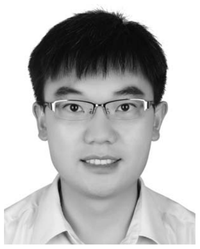
[37] *Executable and Linking Format (ELF) Specification v1.2*, Accessed: Mar. 15, 2018. [Online]. Available: http://refspecs.linuxbase.org/elf/elf.pdf

[38] *ELF for the ARM Architecture*. Accessed: Mar. 15, 2018. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf



Xuhong Peng (S'17) received the B.Sc. degree in electronic science and technology from the China University of Mining and Technology, Xuzhou, China, in 2010, and the M.Sc. degree in communication engineering from Hunan University, Changsha, China, in 2013. He is currently pursuing the Ph.D. degree at the Department of Computer Science, Central South University, Changsha, China.

His current research interests include Internet-of-Things, wireless communication, transparent computing, and edge computing.



Ju Ren (S'13–M'16) received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Central South University, Changsha, China, in 2009, 2012, and 2016, respectively.

From 2013 to 2015, he was a visiting Ph.D. student with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. He is currently a Professor with the School of Information Science and Engineering, Central South University. His current research interests include Internet-of-Things, wireless communication, transparent computing, and cloud computing. He has authored or co-authored over 40 papers in prestigious international journals and conferences including the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and IEEE INFOCOM in the above areas.

Dr. Ren serves/has served as an Associate Editor for the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY and *Peer-to-Peer Networking and Applications*, a leading Guest Editor for *IEEE Network*, and a Technical Program Committee member of many international conferences, including IEEE INFOCOM18, Globecom17, WCNC17, and WCSP16. He also served as the Track Co-Chair for IEEE VTC17 Fall and IEEE I-SPAN 2018, the Special Session Co-Chair of IEEE VCIP17, and an active Reviewer for over 20 international journals. He is a member of the ACM.

Liang She (S'17) received the B.S. and M.S. degrees in computer science from Central South University, Changsha, China, in 2002 and 2011, respectively, where he is currently pursuing the Ph.D. degree in computer science.

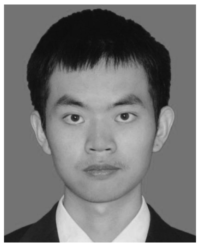
He is a Lecturer with the School of Computer and Information Engineering, Hunan University of Commerce, Changsha. His current research interests include Internet-of-Things, data mining, and personalized recommender systems.

Mr. She is a member of the ACM and CCF.

Deyu Zhang (S'13–M'17) received the B.Sc. degree in communication engineering from PLA Information Engineering University, Zhengzhou, China, in 2005 and the M.Sc. degree in communication engineering and Ph.D degree in computer science from Central South University, Changsha, China, in 2012 and 2016, respectively.

He is currently an Assistant Professor with the School of Software and a Post-Doctoral Fellow with the Transparent Computing Laboratory, School of Information Science and Engineering, Central South University. He has been a Visiting Scholar with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada, from 2014 to 2016. His current research interests include stochastic resource allocation transparent computing, edge computing, and Internet of Things.

Dr. Zhang is a member of the CCF.



Jie Li received the B.Sc. degree in computer science from Central South University, Changsha, China, in 2016, where he is currently pursuing the M.Sc. degree in computer science.

His current research interests include Internet-of-Things, transparent computing, and edge computing.

Yaoxue Zhang received the B.Sc. degree from the Northwest Institute of Telecommunication Engineering, Xi'an, China, in 1982, and the Ph.D. degree in computer networking from Tohoku University, Sendai, Japan, in 1989.

He is currently a Professor with the School of Information Science and Engineering, Central South University, Changsha, China, and also a Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He has authored or co-authored over 200 technical papers in international journals and conferences, as well as 9 monographs and textbooks. His current research interests include computer networking, operating systems, ubiquitous/pervasive computing, transparent computing, and big data.

Dr. Zhang is the Editor-in-Chief of the *Chinese Journal of Electronics*. He is a Fellow of the Chinese Academy of Engineering.