Edge-Centric Programming for IoT Applications With Automatic Code Partitioning

Borui Li, *Student Member, IEEE* and Wei Dong, *Member, IEEE*

Abstract—IoT application development usually involves separate programming at the device side and server side. While separate programming style is sufficient for many simple applications, it is not suitable for many complex applications that involve complex interactions and intensive data processing. We propose *EdgeProg*, an edge-centric programming approach to simplify IoT application programming, motivated by the increasing popularity of edge computing. With EdgeProg, users could write application logic in a centralized manner with an augmented If-This-Then-That (IFTTT) syntax and virtual sensor mechanism. The program can be processed at the edge server, which can automatically generate the actual application code and intelligently partition the code into device code and server code, for achieving the optimal latency. EdgeProg employs dynamic linking and loading to deploy the device code on a variety of IoT devices, which do not run any application-specific codes at the start. Results show that EdgeProg achieves an average reduction of 20.96%, 27.8% and 79.41% in terms of execution latency, energy consumption, and lines of code compared with state-of-the-art approaches.

Index Terms—Edge computing, IoT, programming language

1 Introduction

Internet of Things (IoT) application development usually involves separate programming at the device side and server side. For example, consider a smart plant application. Users can program an IoT node like Arduino to sense the soil humidity of a plant. The sensing data can then be transmitted to the back-end server for further analysis.

This separate programming style is sufficient for many simple applications. However, it is not suitable for many complex applications that involve complex interactions and intensive data processing.

Complex Interactions. Consider the following application: a user wants to turn on an LED when a sensor attached to a door detects an open event. With the traditional programming style, the application logic would be scattered among different sensor nodes. Developers should cope with the complex data stream and network interactions between sensor nodes, which leads to increased system complexity and reduced manageability.

Intensive Data Processing. Consider a speech recognition application. A simple way of designing such a system would deliver all the sensor data to the server running the sophisticated recognition algorithm. This approach may consume excessive energy due to a large number of transmissions. A different approach is to run the recognition algorithm on the IoT device. This approach, however, may cause excessive delays due to the insufficient computation power of the device. Separate programming requires the programmer to make proper decisions, which is quite difficult.

We advocate here a different programming approach, motivated by the increasing popularity of edge computing. In the edge computing paradigm, a number of IoT nodes can perform sensing and actuation. These nodes are connected to a local edge that can perform sophisticated computation. Moreover, edge servers usually have power supplies and are less constrained by energy. Edge computing can offer low processing delay and better privacy.

Taking advantage of the edges, we have developed EdgeProg—a new programming style and software architecture to greatly simplify IoT application programming, resulting in a generic IoT system that can be reprogrammed for a variety of applications without significant loss of overall system efficiency.

To use EdgeProg, developers write a program in a high-level language integrating the whole application logic of an IoT application. This program can further be processed at the edge server, which can automatically generate the actual application code and intelligently partition the code into device code and server code. We call this approach *edge-centric* since developers can regard the program as if it runs on the edge. More importantly, ordinary IoT nodes do not run any application-specific codes at the start. When the program is first executed, the device code will be automatically loaded onto the memory of IoT nodes. Nevertheless, this edge-centric programming process raises some challenges:

- How to design an edge-centric language that could support multi-device interaction and data-intensive computation?

- *The authors are with the College of Computer Science, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University, Hangzhou 310027, China. E-mail: {libr, dongw}@emnets.org.*

Manuscript received 25 Feb. 2021; revised 26 Oct. 2021; accepted 31 Oct. 2021. Date of publication 19 Nov. 2021; date of current version 8 Sept. 2022.
This work was supported by the National Science Foundation of China under Grant 62072396, and Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under Grant LR19F020001.
(Corresponding author: Wei Dong.)
Recommended for acceptance by J. Cao.
Digital Object Identifier no. 10.1109/TC.2021.3129367

- How to partition the user-perceived program to achieve the best delay performance or save most energy?
- How to design a mechanism so that heterogeneous sensor nodes can dynamically load the device-side code and execute it in an efficient manner?

In order to support edge-centric programming and speed-up the application development process, we design a coherent language for specifying the multi-device interaction based on the widely-adopted programming model, IFTTT (IF-This-Then-That) [1]. To further enhance the expressiveness and adopt the data-intensive computation, we extend the traditional IFTTT syntax with the virtual sensor, which accelerates developers to design their own data processing logic with machine learning techniques.

EdgeProg conducts automatic code partitioning which fully leverages the computation ability of each device and achieves optimal end-to-end latency. We abstract the user-written program as a data flow graph, formulate the partitioning problem as an integer programming (ILP) problem and leverage the efficient solver `lp_solve` to obtain the optimal partition.

We implement EdgeProg with Contiki OS for its cross-platform support and the ability to load the optimized executable at runtime with dynamic linking and loading technique. An alternative approach to change the application logic during its execution is exploiting virtual machines (VMs) or using a scripting language. Nevertheless, we do not adopt the alternatives due to they introduce considerable overhead than dynamic linking and loading.

We implement EdgeProg and evaluate its performance extensively. Results show that: (1) EdgeProg programming language can express diverse IoT application logic and reduces the lines of code needed by 79.41% on average. (2) For the execution time of generated applications, EdgeProg achieves a 20.96% reduction on average, and up to 99.05% latency reduction across the five real-world applications under all settings compared with state-of-the-art partitioning systems such as Wishbone [2] and RT-IFTTT [3]. (3) The partitioned application generated by EdgeProg saves 14.8% and 40.8% energy on average compared to Wishbone and RT-IFTTT. (4) For application run-time, the dynamic linking and loading technique outperforms than design alternatives such as virtual machine (by 9.98×) and scripting languages (by 6.37×). (5) The profiling methods adopted by EdgeProg achieve 90%+ and 85%+ accuracy for over 98% test cases. The contributions of this work are summarized as below:

- We present EdgeProg, an *edge-centric* programming system for IoT applications. The EdgeProg language relieves developers from scattered application logic and enables them to express their logic in an easy-to-use way.
- We formulate the code partitioning problem as an ILP problem to minimize the makespan of the task or the energy consumption. The partitioning algorithm optimizes the placement of each stage in an application with consideration of both processing and network cost.
- We implement EdgeProg and evaluate EdgeProg massively with real-world applications and benchmarks. Results show that EdgeProg achieves better latency reduction compared with state-of-the-art approaches and fewer lines of code.

Compared to the conference version of EdgeProg [4], this journal version contains the following important extensions.

- We add more detailed descriptions in Section 2 to better illustrate the usage of EdgeProg.
- Besides latency, we facilitate EdgeProg with the ability to optimize the energy consumption of the edge-device integrated system. Based on the efforts, we reconstruct Section 4.2 with the new analytical formulation of the energy optimization problem and its solution. Also, we present descriptions of the building blocks towards energy optimization in Section 3.
- We add more details about how EdgeProg generates the binaries with Contiki OS in Section 4.3.
- We present a much more detailed evaluation on EdgeProg, especially on the energy optimization performance, in Section 5.
- We add Section 6 to discuss several important issues about EdgeProg.

The rest of this article is structured as follows. Section 2 describes the background and usage of EdgeProg. Section 3 overviews the design goals and building blocks. Section 4 presents the design details. Section 5 shows the evaluation results. Section 6 discusses several important issues about EdgeProg and Section 7 introduces the related work. Finally, Section 8 concludes the paper.

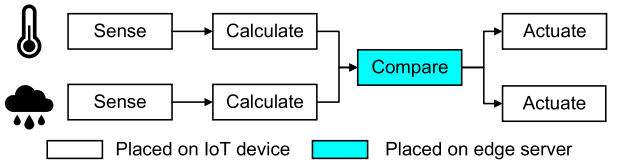
2 Background and EdgeProg Usage

In this section, we briefly introduce the background of the dynamic linking and loading technique of IoT devices used in EdgeProg. Then we present the usage of EdgeProg with a simple smart home application.

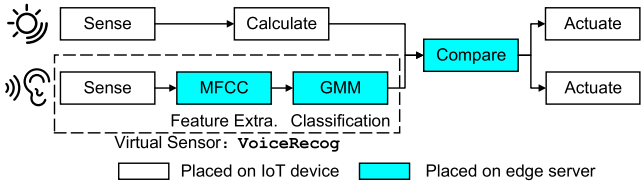
2.1 Dynamic Linking and Loading of IoT Devices

Dynamic linking and loading is one of the over-the-air reprogramming techniques for IoT devices. As its name suggests, reprogramming with dynamic linking and loading technique owns a linking phase and a loading phase. In the linking phase, the on-device reprogrammer first parses the structured information of a file in standard executable and linkable format (ELF) or its variants (e.g., CELF [5] and SELF [6]). Then the reprogrammer allocates ROM and RAM for the data and text segment in the ELF file and performs relocation. The relocation is to patch the data and text segment with real in-memory addresses of the symbols, which are found in the symbol table or calculated using the relocation information in the ELF. Once the linking phase is complete, the reprogrammer writes text segments to the allocated ROM and copies data segments to the RAM, which is called the loading phase. So far, the binary is loaded and ready to be executed.

Compared with the alternatives such as virtual machine [5], [7], [8] and bootloader [9], dynamic linking and loading obtains several inherent merits. (1) High long-term efficiency because it runs native code rather than virtual machine code. (2) Reboot-less update, which is also energy-saving. The recent container technology is also a



(a) Pipeline illustration of `SmartHomeEnv` application.



(b) Pipeline illustration of `SmartDoor` application.

Fig. 1. Illustration of execution stages of the two examples. Each block represents one stage, and data exchange occurs between consecutive stages.

potential alternative to dynamic linking and loading. However, it heavily relies on Linux services such as `cgroup` and `namespace`, which is not available in the resource-constrained IoT devices such as Arduino, TelosB and STM32.

2.2 EdgeProg Usage

We excerpt a simple smart home project named `SmartHomeEnv` from smarthome.com to illustrate how EdgeProg can be used. `SmartHomeEnv` takes the temperature and humidity data from two IoT nodes as input, turns on the air conditioner and dryer if the two readings exceed fixed thresholds, as shown in Fig. 1a. The two nodes are wirelessly connected to an edge server, which could be a PC or other devices that own strong computing ability.

In the traditional approach, two sensors are pre-installed with an application-specific code with functions like periodically transmitting sensor values to the edge server. The edge server further processes these readings and interacts with the sensors with pre-defined interfaces.

With EdgeProg, in contrast, the two sensors are pre-installed with an "idle" program without any application-specific logic. The whole application logic is expressed in an enhanced IFTTT-like language, which is interpreted and processed at the edge server. Fig. 2 shows an EdgeProg application of `SmartHomeEnv`. Lines 2-4 describe the devices (`A`, `B`, `E`) and their interfaces (e.g., the `HUMIDITY` of device `B`) used in this application with keyword `Configuration`. With the information above, lines 5-6 specify the application logic following the IFTTT manner. The edge server automatically partitions the codes into two components, i.e., device-side components and edge-side components. The former are compiled to a loadable module and dispatched to the sensor nodes. Once notified, the "idle" program in the IoT node can dynamically load application-specific module for execution via dynamic linking and loading technique.

EdgeProg also supports building complex applications targeting intensive data processing with the virtual sensor. Fig. 1b illustrates a `SmartDoor` application with several steps for voice recognition. With EdgeProg, the latter steps could be expressed by a virtual sensor `VoiceRecog`. Opposite to the physical or hardware sensor, a virtual sensor is a

```
1 Application SmartHomeEnv{
2    Configuration{TelosB A(TEMPERATURE);
3       TelosB B(HUMIDITY);
4       Edge E(turnOnAC, turnOnDryer);}
5    Rule{IF (A.TEMPERATURE > 30 && B.HUMIDITY > 70)
6       THEN (E.turnOnAC && E.turnOnDryer)}
7 }
```

Fig. 2. Code snippets of `SmartHomeEnv` application.

logical entity that abstracts the data sensed by real sensors which could be located at different places, which we will further describe in Section 4.1.

A key feature of EdgeProg is that it can automatically partition the whole application code to optimize the execution performance, which is increasingly essential for computation-intensive IoT tasks such as speech recognition and video surveillance. As illustrated in both figures of Fig. 1, EdgeProg places each stage on appropriate devices for best performance (i.e., execution time or energy consumption). The MFCC and GMM algorithms in Fig. 1b may be too heavyweight for resource-constrained IoT devices such as TelosB or Arduino. Hence, EdgeProg will automatically partition this task to the edge-side if it yields better performance than placing it on the device.

3 EdgeProg Overview

In this section, we first discuss the design goals of EdgeProg, overview our system design, and introduce some essential components.

3.1 Design Goals

- *Edge-centric.* Compared to the traditional scattered programming manner, EdgeProg should provide users with an edge-centric approach to create the application, which indicates that users need not to break down the application logic into pieces during development.
- *Cost-aware.* The timeliness and energy consumption are recognized as critical costs of an edge-device coordinated application. The ability to deliver a cost-optimal solution of a given input is one of the requirements in EdgeProg's design.
- *Automatic.* By automatic, we mean that EdgeProg should conceive details which have no benefit for users to express their ideas and removes human from the loop to simplify and accelerate the application development.

3.2 EdgeProg Architecture

In Fig. 3, we show a birds-eye view of EdgeProg's system architecture and functional workflow, considering a developing phase, a profiling phase, a binary generation phase, and an execution phase. Users can directly write the application code in an edge-centric manner, i.e., without following the distributed programming style or considering the physical placement of each stage (see Section 4.1 for details). The system takes the user code as input, preprocesses and feeds it into the *code partitioner*. With the help of the *time, energy and network profile* of each device or application, the code partitioner finds the optimal partition and placement

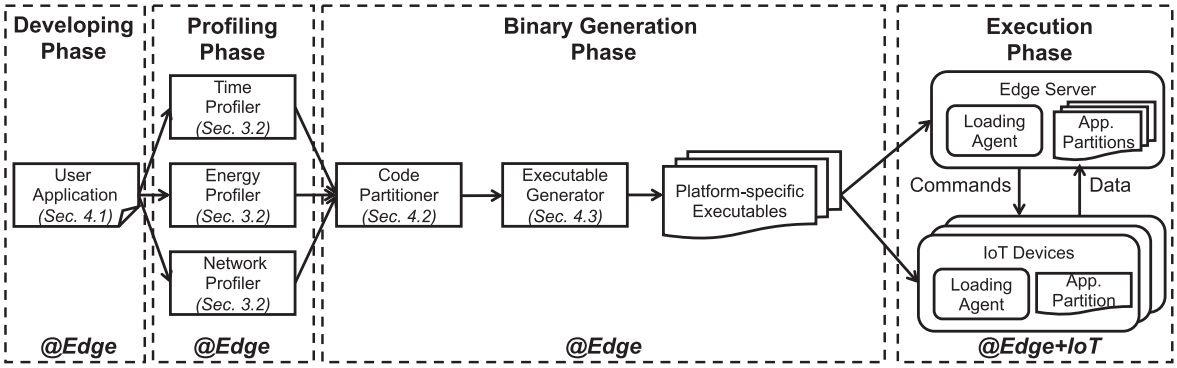


Fig. 3. System overview of EdgeProg.

of each stage using our partitioning algorithm. Processed by the *code generator*, the user-written code is then transformed into the compilable code and compiled to executable or loadable binary by the *code compiler*. Finally, the executables are disseminated to the devices over the air or deployed on the edge device if necessary.

User Input. The user input is written in the EdgeProg programming model, which centers around the notion of `Rules` that specifies the application logic with sensor data, actuator presented by the devices, specified by `Interfaces`, or virtual sensors' output, specified by `Implementation`. Detailed features of the EdgeProg programming language are specified in Section 4.1.

Code Partitioner. The code partitioner is responsible for generating the optimal partition of the user-input applications. We will further give a detailed description in Section 4.2.

Time Profiler. The execution time of each stage on different devices represents the different computation capabilities of each device, which is one of the critical inputs to the code partitioner. Similar to [2], [10], EdgeProg leverages a profiling phase to obtain the execution time on different platforms. For the low-end sensor nodes, we exploit the cycle-accurate simulators such as `MSPsim` for MSP430-based nodes (e.g., TelosB) and `Avrora` for AVR-based nodes (e.g., MicaZ) to get the timing information. For the high-end devices such as Raspberry Pi, profiling it with simulators will be less accurate than the low-end ones mainly due to these powerful devices employ automatic frequency scaling strategy, which reduces the accuracy of a simulator. However, executing on the real device and collect raw timing data is painful and sometimes infeasible due to the hardware interface limit of edge servers. Hence, we choose a near cycle-accurate simulator named `gem5` for profiling high-end devices. We will evaluate the profiling accuracy in Section 5.6.

Energy Profiler. In EdgeProg, an energy profiler is necessary when the objective is to minimize energy consumption. Hence, we build and maintain an energy profile of each device. To be more specific, the profile mainly contains the power under idle state, productive state and network TX/RX. We adopt an automated hardware knowledge based generation approach based on weak supervision learning [11], [12] to generate the energy profile of each device. This learning-based approach reduces the instability and randomness brought by human error, which is more scalable when generating profiles for new devices.

Network Profiler. Network condition (e.g., bandwidth) is also a critical metric being fed into the partitioner. In order to predict the network condition when the application is deployed, we leverage the multiple-output support vector regression (M-SVR) algorithm [13] since it generates a series of prediction results representing the future network condition in a sequence of intervals. In our temporary implementation, the network profiler contains the prediction of the WiFi and Zigbee network. Raw observations such as the bandwidth and received signal strength indicator (RSSI), which is sampled by the loading agent every 60s in order not to influence the regular network transmission, are fed into the M-SVR. Furthermore, when the IoT device is deployed with an application and periodically uploads the data or receiving the commands, the network profiler piggybacks the measurement data with the regular sensor data or commands to further reduce the energy overhead. The predictor outputs the future throughput estimation and per-packet transmission time for further fine-grained time calculation in Section 4.2. Here, since the predicting algorithm acts as a black-box in our system, EdgeProg can use other prediction models instead of the M-SVR model.

Code Generator. The generated optimal partition is processed by the code generator to translate the high-level EdgeProg code into the compilable C code, detailed in Section 4.3.

Code Compiler. Fed by the compilable code, the code compiler generates the executables for the target platform and starts dissemination. In our current implementation, EdgeProg supports four MCU architectures (ATmega, MSP, ARM and x86) with four platforms.

Loading Agent. At the very beginning of our system, there is no application-specific logic running on the node except a loading agent. The loading agent periodically communicates with the edge server for new loadable applications. Once the application is compiled by the compiler and starts dissemination, the loading agent on the deployment destination detects, verifies and receives the executable and dynamically runs it. Moreover, using the wireless channel to dispatch the applications may be unstable due to the existence of wireless interference. Hence, we also advocate a wired loading agent to support disseminating the binaries through USB (for TelosB) and Ethernet (for Raspberry Pi).

4 System Design

In this section, we will first present the design of EdgeProg programming language and highlight the features which enable integrated development. Then we will describe how

```
1  Application SmartDoor{
2     Configuration{
3        RPI A(MIC, DOOR_UNLOCK, OPEN_DOOR);
4        TelosB B(LIGHT_SOLAR);
5     }
6     Implementation{
7        VSensor VoiceRecog("FE, ID"){
8           VoiceRecog.setInput(A.MIC);
9           FE.setModel("MFCC");
10          ID.setModel("GMM", "open.gmm");
11          VoiceRecog.setOutput(<string_t>,"open");
12       }
13    }
14    Rule{
15       IF(VoiceRecog=="open" && B.LIGHT_SOLAR<100)
16       THEN(A.DOOR_UNLOCK && A.OPEN_DOOR)
17    }
18 }
```

Fig. 4. Code snippets of the SmartDoor application.

EdgeProg obtains the optimal partition of the input IoT application with full awareness of the user-perceived event handling latency or energy consumption, including details about the problem formulation and its solution algorithm. Finally, we demonstrate how EdgeProg generates the application code to be disseminated to both devices and edge servers.

4.1 EdgeProg Programming Language

In order to tackle the problem of existing scattered programming style and accelerate the application development process, EdgeProg adopts a rule-based domain-specific language (DSL) for developers to build their applications. An EdgeProg application is typically organized as three parts: *configuration*, *implementation* and *rule*. As shown in Fig. 4, we use the SmartDoor application described in Section 2.2 as an example to illustrate three critical features in the following. More examples could be found in [14].

Edge-Centric Programming Model. In order to achieve the edge-centric design goals of EdgeProg, our programming model should focus users more upon the global behavior other than implementation details. Hence, EdgeProg enables developers to organize their application centered with the overall application logic using keyword `Rule`. There exist several DSLs enabling developers to focus on upper logic, as known as the macro-programming model, in sensornet researches such as Kairos [15] and Regiment [16]. Nevertheless, existing works fall flat nowadays due to the constraint on application portability or lack of actuation. IFTTT programming shows its simpleness and effectiveness in existing researches [3], [17], [18] when expressing the high-level application logic, and this programming approach is widely adopted in state-of-the-art industrial solutions such as Samsung SmartThings and Microsoft Flow. By early 2017, the website ifttt.com had gathered over 320,000 IFTTT programs [1] and the numbers are still increasing dramatically. Therefore, we leverage an IFTTT-like grammar for enabling users to express their idea in a unified and explicit manner, as illustrated in lines 14-17 of Fig. 4. Moreover, we augment the IFTTT grammar with `Configuration` and `Implementation` to make users express the detailed definition and specification of necessary components used in the `Rule` part.

Full Support of Virtual Sensor. In order to accommodate the intensive data processing in the nowaday IoT scenario, we enhance our DSL with the *virtual sensor*. Traditional hardware sensors generally produce raw measurements of physical properties such as the moisture value or light intensity, which are unprofitable unless being transformed into high-level domain-dependent information. Furthermore, capturing valuable information usually requires the coordination of multiple hardware sensors, e.g., detecting fire hazards with both temperature and smoke sensor. In order to tackle the limitations above and make sensor data processing more flexible, existing works combine the readings of multiple sensors for event detection. For example, SenseHAR [19] advocates an activity recognition system that abstracts the data of several inertial sensors from different devices using a sensor fusion network. Similarly, LiKamWa *et al*. [20] measure the user's mental state based on the interactions with the smartphone. Virtual sensors act as a black-box providing the indirect measurements or events, which are typically physically immeasurable, by combining sensed data from several hardware sensors with data processing algorithms. EdgeProg embraces this technique as one of the extensions to standard IFTTT syntax to provide easy-to-use yet expressive handling for intensive data processing.

```
1 VSensor VoiceRecog(AUTO){
2    VoiceRecog.setInput(A.MIC, A.Accel_x, A.Accel_y
        , A.Accel_z, B.Light, B.PIR);
3    VoiceRecog.setOutput(<string_t>,"open", "close"
        );
4 }
```

Fig. 5. An example implementation code snippet of an algorithm-agnostic virtual sensor.

As shown in Fig. 4, lines 4-12 list the configuration of a virtual sensor, `VoiceRecog`, to recognize whether the input voice fragment produced by interface `A.MIC` stands for "open" or not. This virtual sensor is a pipeline of two stages: `FE` and `ID`. The algorithms employed by each stage, specified by the keyword `setModel()`, are MFCC (Mel Frequency Cepstral Coefficient) and GMM (Gaussian Mixture Model), which are commonly used by voice recognition systems [21], [22]. Currently, we implement 17 data processing algorithms, including 12 for feature extraction and 5 for classification. Although `FE` and `ID` are the compositions of the typical pipeline, applications with more stages and parallel stages are also supported in our system, such as the EEG seizure onset detection application described in [2].

Furthermore, there still a lot of complexity for green-handed developers due to they may have no idea of which sensors are strongly related to the expected output and how they are related. To relieve this, we propose the inference-agnostic virtual sensor. To construct it, developers could merely provide the set of possibly related sensors and the expected output of the virtual sensor, as Fig. 5 shows. EdgeProg will first generate a simple sampling application, and developers should record the events they desired with it to obtain enough training data. Then EdgeProg will train an inference model which reflects the relationship between the input sensors and the recorded events. Finally, the trained model is partitioned and disseminated, similar to the other virtual sensors.

Explicit Data Flow. According to our analysis on 101 commonly-used IoT applications from several popular development websites such as DFRobot and Hackster.io, we find

that about 45% lines of code in these projects are written for data flow construction and interaction, which is a considerable proportion and increases the project complexity. Furthermore, multi-device interaction makes the data flow more complicated due to it is conceived in the network packet construction.

In a typical IoT application, data flow starts from the production of sensor data, processed by several algorithms, then finally saved in the database or turned into a command back to the actuator IoT node. Hence, we make the data flow explicit in these three steps. For data production and final actuation, as illustrated in lines 2-4 of Fig. 4, developers specify the data and available actions as interfaces. For example, line 3 illustrates that three interfaces (microphone sampling, door unlocking and door opening) of a Raspberry Pi named A are used in this application. The available interfaces of specific hardware are determined by its vendor or prototype developer. For data processing, virtual sensors and rules directly use or call the interfaces, which results in a unified and explicit data flow.

4.2 Code Partitioning

The goal of EdgeProg's code partitioning sub-system is to divide the user input into appropriate stages and to obtain the optimal placement of each stage. To accomplish them, we first preprocess the user input application into *logic blocks*, which represent the computation stages, and generate a data flow graph of the rules to obtain a full view of the user logic as well as the stage dependency. Afterward, as *cost-aware* is one of the design goals of EdgeProg, we both formulate the latency minimizing problem and energy saving problem into a mathematical expression, then we employ an efficient solver to obtain the optimal placement of each stage.

The key insight of our partitioning algorithm is that we push the computation close to the data source as much as possible and make the best use of the computation ability of each device to achieve latency reduction. Moreover, the optimal placement that exhibits favorable computation-transmission tradeoff could be obtained by EdgeProg benefited from the intrinsic global view of our programming language.

4.2.1 Logic Blocks and Data Flow Graph Construction

Due to the compact nature of our programming language, there are mainly two gaps that prohibit us from further implementation and optimization. (1) Some stages may be implicitly defined and used in the application. For example, in Fig. 4, the interface `LIGHT_SOLAR` of device B is referenced in the rule. Thus the stage of sensing it is necessary but being conceived from the application. (2) The topological information is necessary for optimization, which is also implied in the application.

To fill up the gaps, we construct a *data flow graph* of an application whose nodes are represented with *logic blocks*. A logic block is supposed to be expressive enough as an independent building block of the application, i.e., it should contain adequate information such as placement, algorithm and necessary parameters for time profiling as well as its input source for code generation. Hence, the logic block is defined as a tuple $<$*functionality*, *placement*$>$, as shown in Fig. 6.

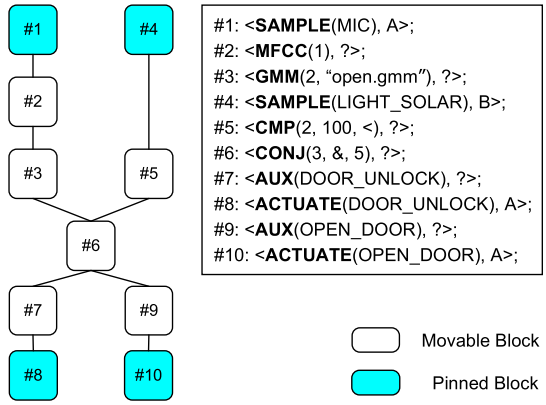


Fig. 6. An illustration of EdgeProg logic flow and logic block of SmartDoor application.

- *Functionality*. To express the functionality, we borrow the idea of tasklet primitives from Tenet [23] such as `SAMPLE`, `ACTUATE` and `CONJ`, which provide building blocks for a wide range of data acquisition and processing tasks. Nevertheless, we further add the algorithms as primitives (e.g., `GMM`) to accommodate the virtual sensor deployment. The data source of a logic block is declared as the first argument of the primitive.
- *Placement*. There are two kinds of code blocks in EdgeProg: *pinned* and *movable*. The pinned blocks are generally physical-constrained functionalities. For example, `SAMPLE` must be placed on the device. Moreover, there are also logical-constrained functionalities. For example, the `CONJ` is pinned to edge server to avoid unnecessary device-to-device traffic, which will lead to sub-optimal partition. Hence, the placement is fixed for a pinned block, and we use its corresponding device alias in the logic block. The placement of a movable block, which is potentially deployed on the device or edge server, is denoted with the question mark to express the uncertainty.

Generally, the logic blocks could be inferred from the EdgeProg program. Taking the program in Fig. 4 as an example, each stage of the VSensor in the implementation part (i.e., `FE`, `ID`) is transformed to a logic block. Except for these explicitly declared logic blocks, some blocks are also necessary for a complete graph but implicitly conceived in the user application. In order to complete the data flow graph with the intrinsic blocks, we analyze all the rules defined in the `Rule` part with the following strategies:

- For conditions exploiting virtual sensors in the IF statement, we refer to the `Implementation` part to obtain the staging pipeline and insert `SAMPLE` blocks for the input.
- For conditions that only compare sensor values, we convert it into two stages: `SAMPLE` and `CMP`.
- We use a `CONJ` block representing the conjunction of all the conditions in the IF statement.
- For each action in the THEN statement, we use two blocks: an auxiliary movable block `AUX` representing

it is edge-triggered or local-triggered and a pinned block `ACTUATE` representing the action.

Then the data flow graph could be constructed as a directed acyclic graph (DAG) $G(V, E)$ whose vertices represent the logic blocks and edges represent there exist a data flow, as Fig. 6 illustrates.

4.2.2 Formulation of Optimal Partitioning Problem

With the help of the data flow graph $G(V, E)$, we formulate the optimal partitioning problem as a numerical optimization problem. The resulting optimal partition could be viewed as assigning each logic block to its most preferable computational device. We would like to borrow the existing code partitioning algorithm proposed in Wishbone [2] to solve our problem.

- *Node weights.* The weights of vertices in the graph represent the processing time or energy consumption of the corresponding logic block. In EdgeProg, the weight of a movable block is two-fold: the local and edge-server processing time/energy. While in Wishbone, each vertex in the graph only has one weight.
- *Optimization goal.* The optimization goal of Wishbone is minimizing the sum of computational budgets and network bandwidth. Nevertheless, EdgeProg focuses on the cost of executing the application (i.e., latency or energy), which makes the Wishbone formulation no longer suitable for our problem.

In EdgeProg, we support two optimization goals: execution time or energy consumption of an edge-device integrated application, and users could choose one of the goals on their demand. We next present the formulation of latency and energy optimization, respectively.

Optimizing Execution Time. Minimizing the task execution latency leads to minimizing the length of the longest path in the data flow graph. We define a full path of the data flow graph $G(V, E)$ as the path from a source vertex to a sink vertex, denoted as π. We use $len(\pi)$, $\delta(\pi)$ and $\Pi(G)$ to represent the length of path π, the number of vertices in path π, and the set of all full paths in graph G. Thus, our optimization goal is thus denoted as

$$\min \max_{\pi \in \Pi(G)} len(\pi). \tag{1}$$

In order to further demonstrate $len(\pi)$, we first introduce a binary indicator $X_{b_i s}$ to demonstrate the placement as

$$X_{b_i s} = \begin{cases} 1 & \text{logic block } b_i \text{ is assigned to device } s \\ 0 & \text{logic block } b_i \text{ is not assigned to device } s \end{cases}, \tag{2}$$

where $s \in S_i$, and S_i denotes the set of all possible devices that could place the ith logic block (i.e., b_i). Thus, the sum of computing and transmitting latency across all possible placements of a full path, $len(\pi)$, could be expressed as

$$len(\pi) = \sum_{i=1}^{\delta(\pi)} \sum_{s \in S_i} X_{b_i s} T^C_{b_i s} + \sum_{i=1}^{\delta(\pi)-1} \sum_{s \in S_i s' \in S_{i'}} X_{b_i s} X_{b_{i'} s'} T^N_{b_i s s'}, \tag{3}$$

where i, i' are the adjacent vertices in path p (i.e., $i' = i + 1$). We use $T^C_{b_i s}$ to denote the data processing cost of the ith block on device s, and $T^N_{b_i s s'}$ to represent the data transmission time between block b_i of device s and block $b_{i'}$ of device s'. We assume that the data transmission time is negligible if the two consecutive logic blocks are placed on the same device. Thus we have

$$T^N_{b_i s s'} = \begin{cases} \left\lceil \frac{q_{ii'}}{r_{ii'k}} \right\rceil t_{ii'k} & s \neq s' \\ 0 & s = s' \end{cases}, \tag{4}$$

where $q_{ii'}$ denotes the data size being transmitted on edge (i, i'). $r_{ii'k}$ is a protocol-specific metric representing the maximum packet payload of protocol k, e.g., the $r_{ii'k}$ of 6LowPAN network is 122 bytes. Furthermore, the per-packet transmission time is given by $t_{ii'k}$, which is profiled and predicted by our network profiler detailed in Section 3.2.

Optimizing Energy Consumption. Different from the latency formulation, optimizing energy consumption should consider the costs of all the edges and vertices rather than consider the longest path only. Thus, the optimization goal is formulated as

$$\arg\min_X \sum_{i=1}^{|V|} \sum_{s \in S_i} X_{b_i s} E^C_{b_i s} + \sum_{i=1}^{|V|} \sum_{s \in S_i s' \in S_{i'}} X_{b_i s} X_{b_{i'} s'} E^N_{b_i s s'}, \tag{5}$$

where $|V|$ stands for the number of vertices in $G(V, E)$. The data processing energy $E^C_{b_i s}$ and transmission energy $E^N_{b_i s s'}$ could be derived by

$$\begin{cases} E^C_{b_i s} = T^C_{b_i s} P^C_s \\ E^N_{b_i s s'} = T^N_{b_i s s'} (P^{TX}_s + P^{RX}_{s'}) \end{cases}, \tag{6}$$

where P^C_s stands for the average power (in mW) of device s for computation. P^{TX}_s and $P^{RX}_{s'}$ represent the average power for TX operation of device s and RX operation of device s', respectively. It is worth noting that, we only consider the energy consumption of IoT devices. The energy consumed by edge devices are ignored (i.e., P^C_s, P^{TX}_s and $P^{RX}_{s'}$ are set to 0) due to the edge devices are mostly AC-powered.

4.2.3 Solution of Optimal Partitioning Problem

The objective formulations of EdgeProg optimization problem (i.e., Equs. (3) and (5)) are *quadratic programming* (QP) problems, which are shown to be NP-hard [24]. The state-of-the-arts employ heuristic algorithms to solve it efficiently. For example, the most recent work [25] utilizes a breadth-first greedy search algorithm to solve it. While we prefer a solving method, which is less prone to local optima and the method's scalability against problem size is also a necessary property.

Inspired by McCormick Envelopes relaxation [26], we reformulate the objectives and constraints of EdgeProg to conform to the formulation of integer linear programming problem (ILP), which could be efficiently solved by the standard solver, e.g., `lp_solve`. We compare the solving time of the QP and ILP formulations in our technical report [14]. Results show that the ILP formulation is more scalable than QP in terms of solving time.

Take the latency optimization problem (Equ. (3)) of EdgeProg as an example. We first convert the quadratic objective function to linear one by introducing an auxiliary variable $\epsilon_{iss'} = X_{b_i s} \cdot X_{b_{i'} s'}$ to replace the quadratic term $X_{b_i s} \cdot X_{b_{i'} s'}$ in Equ. (3). Moreover, the presence of $\epsilon_{iss'}$ causes the introduction of these constraints

$$(\forall i \in \delta(p)-1, s \in S_i, s' \in S_{i'}) \quad \epsilon_{iss'} \geq 0, \tag{7}$$

$$(\forall i \in \delta(p)-1, s \in S_i, s' \in S_{i'}) \quad \epsilon_{iss'} \leq X_{b_i s}, \tag{8}$$

$$(\forall i \in \delta(p)-1, s \in S_i, s' \in S_{i'}) \quad \epsilon_{iss'} \leq X_{b_{i'} s'}, \tag{9}$$

$$(\forall i \in \delta(p)-1, s \in S_i, s' \in S_{i'}) \quad \epsilon_{iss'} + 1 \geq X_{b_i s} + X_{b_{i'} s'}. \tag{10}$$

It can be observed that all the above four constraints are linear. Whereas our objective function is still in a minimax shape, which needs further transformation. We thus introduce another auxiliary variable z and convert the inner max function to a set of constraints to make it follow standard ILP formulation. The rewritten ILP objective function is

$$\textbf{Objective:} \quad \arg\min_{X} \quad z \tag{11}$$

Subject to:

$$z \geq \sum_{i=1}^{\delta(\pi)} \sum_{s \in S_i} X_{b_i s} T^C_{b_i s} + \sum_{i=1}^{\delta(\pi)-1} \sum_{s \in S_i s' \in S_{i'}} \epsilon_{iss'} T^N_{b_i ss'}, \forall \pi \in G. \tag{12}$$

Furthermore, we add constraints for $X_{b_i s}$ to ensure each logic block is appointed to a specific device.

$$\sum_{s \in S_i} X_{b_i s} = 1, \forall i \in G. \tag{13}$$

Thus, any optimal solution of Equ. (11) subject to (7), (8), (9), (10), (12) and (13) will be the optimal partition of the input application.

Similarly, the objective of energy optimization problem (Equ. (5)) is transformed as

$$\arg\min_{X} \sum_{i=1}^{|V|} \sum_{s \in S_i} X_{b_i s} E^C_{b_i s} + \sum_{i=1}^{|V|} \sum_{s \in S_i s' \in S_{i'}} \epsilon_{iss'} E^N_{b_i ss'}, \tag{14}$$

along with the formulations of $E^C_{b_i s}$ and $E^N_{b_i ss'}$ (Equ. (6)), and the constraints similar to Equs. (7), (8), (9), (10) and (13).

4.3 Executable Generator

The executable generation process in EdgeProg contains two steps: (1) constructing pieces of compilable code from the optimal partition and the logic blocks, and (2) compiling the code to platform-specific executables.

Benefited by the cross-platform nature of Contiki OS , we could generate the code for the edge server (mostly Linux-compatible hardware) as well as sensing devices in a similar manner. Then EdgeProg compiles them using the platform-specific toolchains provided by Contiki based on `msp430-gcc` for TelosB and `gcc-linaro-arm` for Raspberry Pi. The only difference our generator should take care of is the different libraries included and sampling APIs used for distinct platforms. Hence, we focus on how to generate compilable code that runs efficiently.

```
1  PROCESS_THREAD(cond1_process, ev, data){
2      static struct etimer et_cond1;
3      PROCESS_BEGIN();
4      etimer_set(&et_cond1, COND1_INTERVAL);
5      while(1){
6          PROCESS_YIELD();
7          if (etimer_expired(&et_cond1)){
8              (do the jobs of logic blocks)
9              process_post(&send_process, send_evt, &
                   to_send1);
10             etimer_reset(&et_cond1);
11         }
12     }
13     PROCESS_END();
14 }
```

Fig. 7. Code snippets of the functioning process of EdgeProg.

As we mentioned in the last section, the logic blocks are designed to be expressive enough to act as a building block of an application, and hence they are transformed to a function into the final compilable code. The most difficult issue is how to organize the function calls in the generated code. The intuitive approach to accommodate the event-driven kernel and the protothread technique of Contiki OS is to arrange all the logic blocks assigned to the same placement in a protothread and send/receive data if the next block is assigned to another device. This simple design raises performance drawbacks. The generated protothread could be too long with this design, which degrades the system performance due to the non-preemptive scheduling of Contiki.[1] Generating one protothread of one block is also not efficient because short protothread incurs much process switching overhead, which will also harm the performance.

Our approach is based on a code template of Contiki necessaries and a send thread with receive callback. The functioning protothreads are generated from graph fragments of the optimized DAG, and the code snippet is illustrated in Fig. 7. The fragments of each device are obtained by leveraging a depth-first traverse of the logic blocks of the DAG which ends at the placement-changing point. Then we assemble a protothread with one fragment by calling functions of the logic blocks. At the end of a thread, it issues an event to the send thread (e.g., line 9 of Fig. 7) for data transmission and yields for other threads. Moreover, based on our time profiling, the graph fragments could be further segmented if it contains several time-consuming tasks for system health.

5 Evaluation

In this section, we evaluate the performance of EdgeProg in various aspects.

5.1 Experiment Setup

First, we introduce the benchmarks we used and baselines we compared in our evaluation.

1. Contiki supports preemptive multi-threading as an optional library, while it requires additional multiple stack allocation which is stressful for low-end devices such as TelosB. Hence we do not adopt this scheme.

TABLE 1
Implemented Benchmark Applications

Name	Application	Sensor	# Operators	Algorithms
Sense	Outlier Detector [27], [28]	Temp., Light	8	Average, Matrix multiplication, LEC compression
MNSVG	Weather Forecasting [3]	Temp., Humidity	4	MNSVG
EEG	Seizure Onset Detect. [29]	EEG	80	Wavelet decomposition, SVM
SHOW	Smart Handwriting [30]	Accel.	13	FFT, Random forest
Voice	Speaker Count [31]	MIC	10	MFCC, Pitch estimation, Unsupervised clustering

Benchmarks. We summarize the five macro-benchmarks to evaluate our EdgeProg in Table 1: two sensing applications and three real-world applications. The #operators column in Table 1 indicates the number of operational logic blocks of each benchmark.

- *Sense.* A common sensing application with outlier detection using algorithms proposed in [27] and data compression using the LEC algorithm [28].
- *MNSVG.* A weather forecast application using an MNSVG model proposed in [3] to predict temperature and humidity values.
- *EEG.* Using the EEG signal to detect seizures [29], taken from Wishbone [2]. It employs ten parallel channels to process the EEG signal with seven order wavelet decomposition in each channel.
- *SHOW.* Detecting and classifying the trajectory of the device with IMU information and random forest algorithm [30].
- *Voice.* Counting the number of speakers with signal processing and clustering algorithms [31].

Baselines Definition. Here we describe the state-of-the-art edge(cloud)-device interactive system alternatives that we use to illustrate the advantages of EdgeProg.

- *RT-IFTTT* [3]. The server does all of the computation. IoT devices only need to report the sensor value or take actions under the server's command.
- *Wishbone(0.5, 0.5)* [2]. Wishbone is a partitioning system for sensornet applications whose goal is to minimize a combined objective of CPU and network workload, which could be formulated with two weights as ($\alpha CPU + \beta Net$). Here (0.5, 0.5) stands for $\alpha = \beta = 0.5$, which indicates CPU and network are of equal importance in this baseline.
- *Wishbone(opt.).* During our preliminary experiment, we notice that better latency performance could be achieved by altering the α and β parameters. Hence, we conduct evaluations by tuning the parameters with 0.1 step, and record the best performance as this baseline.

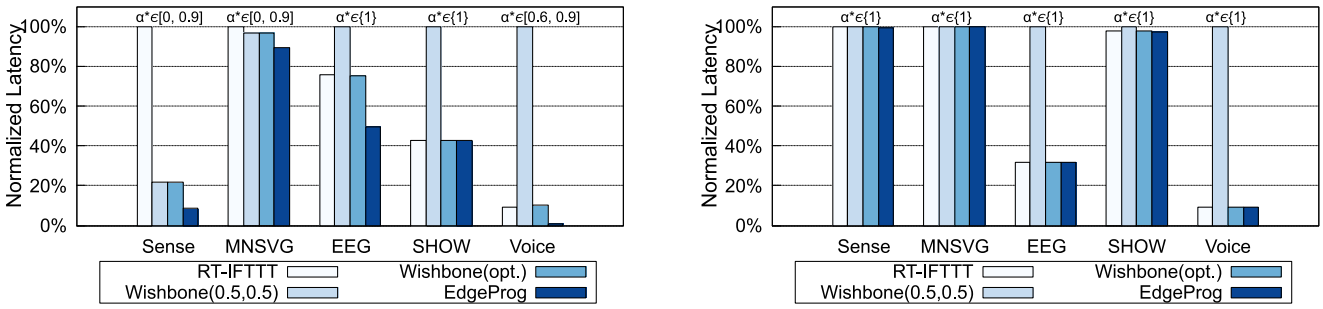
5.2 Latency Reduction

Fig. 8 depicts the task makespan of five macro-benchmarks under Zigbee (on TelosB node) and WiFi (on Raspberry Pi) network. We use a laptop with a 2.8 GHz i7-7700HQ CPU and 16 GB memory as our edge server. EdgeProg achieves a 20.96% reduction on average across all settings, and up to 99.05% reduction in Voice benchmark compared with Wishbone(0.5, 0.5). Moreover, we have two main observations according to the results:

(1) Speed up percentage varies considerably among benchmarks. For example, EdgeProg surpasses the baselines for Voice and EEG benchmarks under both settings while falls flat for MNSVG. This variation mainly due to the computation complexity and network demands of each benchmark. As illustrated in Table 1, EEG is the most complex one with 80 operators, which promises a larger optimization space to reduce the latency. Furthermore, each order of its wavelet decomposition halves input data, which reduces the transmission time of its output and makes it more profitable to local execution. Nevertheless, EdgeProg struggles against SHOW with 13 operators under WiFi, mainly due to the parallel layout of its operators, which leads to fewer valid cut points to partition. As for MNSVG, a small number of its operators results in its available cut points is only three. Under this circumstance, EdgeProg still captures the best cut point for ZigBee, which is neglected by baseline methods. In summary, data-reduction algorithms contribute more to latency reduction.

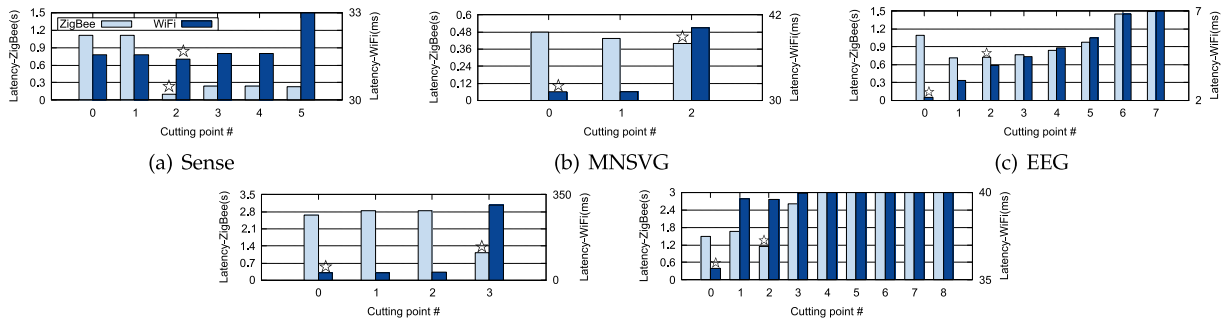
(2) EdgeProg under ZigBee network outperforms under WiFi. Under the ZigBee network, EdgeProg reduces the makespan by 30.96%, 45.80% and 18.19% compared with three baselines, individually. Nevertheless, reduction percentages drop to 0.07%, 30.58% and 0.13% when using WiFi. This is because the WiFi network is much faster than Zigbee, which leads to a short networking time, and the data processing time/energy becomes the dominant fraction in the algorithm. Furthermore, the IoT device we used for WiFi (Raspberry Pi) has better computing power than the device we used for Zigbee (TelosB), which leads to a smaller difference in the data processing performance between the two partitions. Hence, among the benchmarks, both the networking time and the computing time become closer to the sub-optimal ones under WiFi. Therefore, the optimization space of EdgeProg becomes smaller under WiFi network, which finally causes a smaller performance gain under WiFi than Zigbee. It is worth noting that although the performance of each approach is close to each other under WiFi, EdgeProg always obtains the optimal partition for each benchmark.

To further study the above observation, we established a ground truth by exhaustively running each benchmark at every available cutting points on our testbed. Fig. 9 illustrates the results. The star icons indicate EdgeProg's choice for the best cutting points. We can infer from the figures that as the network speed grows, data transmission time decreases and data processing time becomes dominant. Hence, optimization algorithms prefer to offload tasks at early stages, which could be deduced from that the star icons on WiFi bars are more to the left than ZigBee ones. Consequently, the dominant strategies are more concentrated on



(a) Execution time under Zigbee network. (b) Execution time under WiFi network.

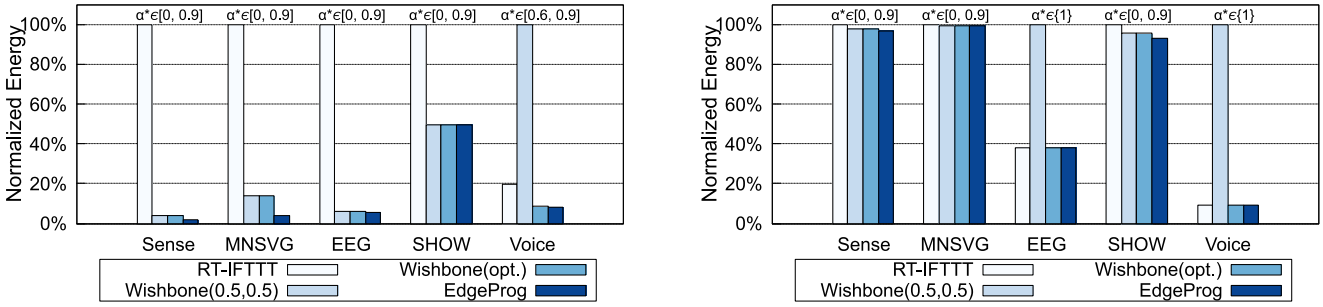
Fig. 8. Latency measurements normalized to the worst-performed baseline. EdgeProg reduces the task latency by 18.2% compared with Wishbone (opt.) and 31.0% with RT-IFTTT on average. The optimal range of α of Wishbone(opt.) for each benchmark is labeled on the top.



(a) Sense (b) MNSVG (c) EEG

(d) SHOW (e) Voice

Fig. 9. Latency of each macro-benchmark at available cutting points under both networks. Cutting points are arranged to assure that fewer operators are executed locally when point number gets bigger. We omit the bigger cutting points part of Voice and EEG due to the continuous growth of latency.



(a) Energy consumption under Zigbee network. (b) Energy consumption under WiFi network.

Fig. 10. Energy consumption comparison results (normalized to the worst-performed baseline). Compared with Wishbone(opt.) and RT-IFTTT, EdgeProg saves the energy by 14.8% and 40.8% on average. The optimal range of α of Wishbone(opt.) for each benchmark is labeled on the top.

the left, which means the decrease of optimization space and leads to closer performance among baselines.

5.3 Energy Saving

Besides latency, EdgeProg could also optimize the energy consumption. The experiment setup is similar to the ones in Section 5.2, and we use the Monsoon Power Monitor to measure the energy consumption. As we specified in Section 4.2.2, we do not consider the power consumed by edge devices because they are mostly AC-powered. Fig. 10 illustrates the evaluation results under Zigbee and WiFi networks. EdgeProg achieves 31.48% overall energy saving on average across all settings, and up to 98.38% reduction in Sense benchmark compared with RT-IFTTT under Zigbee network. We also observed that EdgeProg performs better under Zigbee network (51.60% average reduction) than WiFi (11.37%), and the reason is similar to the latency in Section 5.2.

Nevertheless, there are also some situations that EdgeProg seems to make no optimization to the latency or energy (e.g., the SHOW benchmark in Fig. 8a). The reason of EdgeProg achieves no optimization in certain benchmarks is that the comparing baselines have already achieved the optimal performance (i.e., these methods partition the application in the optimal way) in those situations. However, EdgeProg shows its generality for achieving optimal solutions under different setups (i.e., different optimization goals, application structures, etc.). For example, although the baselines reach the optimal partition for the SHOW benchmark under Zigbee network, only EdgeProg achieves the optimal solution for SHOW benchmark under WiFi network, as shown in Figs. 8b and 10b.

TABLE 2
Dissemination Size Among Platforms (Byte)

App.	TelosB	MicaZ	Raspberry Pi
Sense	4,344	6,384	4,004
MNSVG	2,756	3,460	2,280
EEG	4,500	6,276	3,920
SHOW	22,952	28,660	14,540
Voice	32,076	42,416	19,336

Furthermore, according to the results, we found that the optimal range of α (i.e., α^*) for Wishbone(opt.) varies among benchmarks and optimization goals (latency, energy). Wishbone [2] claims that its objective $\alpha CPU + \beta Net$ could be a proxy for meaningful objectives such as energy, but the per-benchmark variation of α^* makes it difficult to take advantage of Wishbone in practice. More specifically, besides the optimization goal, the α^* is influenced by the task type and device characteristic. 1) *Influence of tasks type*. For example, in Fig. 8a, the α^*_{Sense} (α^* of Sense benchmark) tend to be small while the α^*_{EEG} is 1 when minimizing latency. EEG is a computational-intensive benchmark that contains 80 operators of complex algorithms such as Wavelet decomposition (see Table 1), while Sense is a network-intensive application whose computations are simple (e.g., average). Hence, the large α^*_{EEG} and small α^*_{Sense} are reasonable because the computation of EEG is important while network is vital for Sense. 2) *Influence of device characteristics*. The α^*_{EEG} changes from small in Fig. 10a to big in Fig. 10b. This is because the network change from Zigbee to WiFi significantly reduces the inter-block transmission time of EEG, which makes the computation the dominant factor in the optimization. Unlike Wishbone, EdgeProg provides optimization goals with clear physical meaning which stay unchanged whatever the optimization task is, which makes EdgeProg more useful in practice.

5.4 Overhead

Dissemination Overhead. The dynamic linkable and loadable binary sizes of the macro-benchmarks on three platforms: TelosB (TI MSP430), MicaZ (AVR ATMega128) and Raspberry Pi 3B+ (ARM Cortex-A53) supported by EdgeProg is summarized in Table 2.

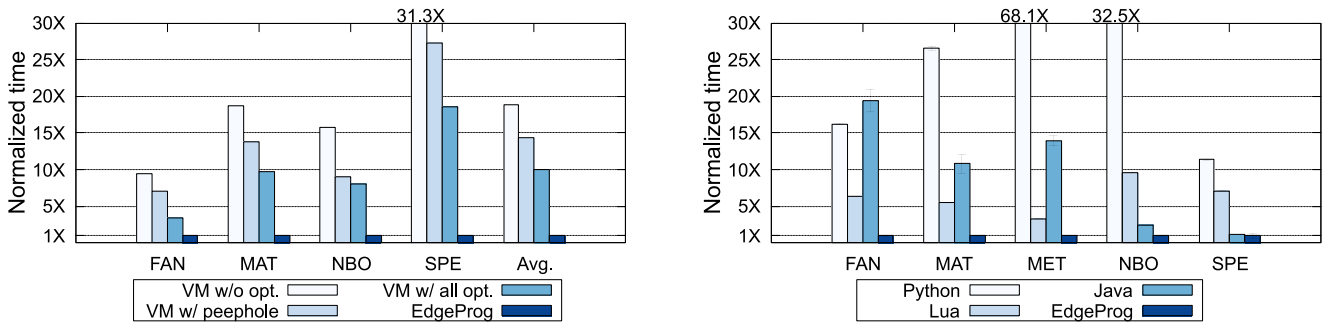
We can see from the data that the binary size of SHOW and Voice is much bigger than other benchmarks, which is mainly due to the complexity of the algorithms they adopted such as FFT, MFCC. Nevertheless, EEG has a smaller size compared with its large number of operators, which is mainly due to each of its tunnels shares the same procedures, and each procedure mainly contains one algorithm, wavelet decomposition, with different parameters.

Run-Time Efficiency. In this section, we compare the run-time efficiency of the dynamic linking and loading technique with its alternatives: virtual machines (VMs) and scripting languages. To eliminate the inherited overhead brought by different implementations, we use five micro-benchmarks from Computer Language Benchmark Game (CLBG). CLBG is a language benchmark suite maintained by the Debian community. The five benchmarks we excerpted are Fannkuch problem (FAN), https://github.com/Byron/benchmarksgame-cvs-mirror Matrix multiplication (MAT), http://attractivechaos.github.io/plb/ Meteor predicting (MET), N-Body solution (NBO), and Spectral-Norm calculating (SPE). We use CapeVM [8], a state-of-the-art Java VM developed for lightweight execution on embedded devices, as the representative of the VM technique. CapeVM proposes various optimization strategies to accommodate different applications, and we set up the experiment with three settings: no optimization, only peephole optimization and all optimizations. Moreover, we choose two scripting languages: Python (for popular) and Lua (for lightweight) along with Java, which is used in CapeVM, as our design alternatives of scripting languages.

Fig. 11 illustrates the experiment result. Due to CapeVM do not support multidimensional arrays and floating points, the MET benchmark could not be implemented with CapeVM. As shown in Fig. 11a, the VM method introduces a massive loss of run-time efficiency. VM costs more than EdgeProg when executing the same benchmark by 9.98× on average and up to 31.32×. As for scripting languages and native Java illustrated in Fig. 11b, EdgeProg's dynamic linking and loading technique still outperforms than alternatives. Python incurs the most overhead averaged 30.96× and Lua, being famous for its lightweight, still slows by 6.37× than ours.

5.5 Programming Language

In order to compare the reduction of lines of code via EdgeProg, we compare the lines of code of the macro-benchmarks described in Section 5.1 written in traditional Contiki-style and EdgeProg-style. Fig. 12 illustrates the



(a) Compare with virtual machines (b) Compare with scripting languages

Fig. 11. Run-time efficiency comparison between EdgeProg and design alternatives.

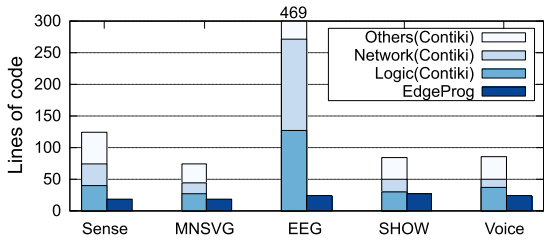


Fig. 12. Lines of code comparison between Contiki and EdgeProg. The "Logic", "Network" and "Others" represent the lines of code for expressing core application logic, inter-device network and others such as definition and included headers in Contiki source code.

comparison results. Note that due to EdgeProg provides several data processing algorithms in advance to simplify the development procedure, we omit the lines of code for implementing the algorithms in Contiki-syle source code to achieve fair comparison and focus more on how EdgeProg helps for complex device interactions. We can observe that (1) EdgeProg reduces the lines of code by 79.41% on average. This is because EdgeProg relieves users of writing complex inter-device interactions and other grammar necessaries. Moreover, the virtual sensor and IFTTT abstraction contribute to the lines of code reduction for application logic. (2) EdgeProg reduces the development complexity, especially for applications with more devices. For example, the 80 stages of EEG application consists of 10 EEG devices, and each device owns eight stages. Programming ten devices increases the lines of code multiple times. While the relatively low reduction percentage of MNSVG (75.68%), SHOW (67.86%) and Voice (72.94%) applications are partly because they need only one device and an edge device.

5.6 Profiling Accuracy

The correctness and accuracy of EdgeProg's latency-effective partition depend on the profiling method. In this subsection, we evaluate the accuracy of profiling methods for both high- (e.g., Raspberry Pi) and low-end (e.g., TelosB) devices that we employ in EdgeProg.

We use `mspsim` to profile the applications of TelosB, and a near cycle-accurate simulator `gem5` for modern platforms such as Raspberry Pi. For `gem5`, we use the system call emulation (SE) mode with the compiled binary as input to avoid the additional overhead of its full-system mode. The results are shown in Fig. 13. `mspsim` could achieve 90%+ accuracy over 97.6% of test cases. Nevertheless, only 87.1% cases of `gem5` reach 90%+ accuracy, which is mainly due to the frequency fluctuation of CPUs and background processes of Raspberry Pi.

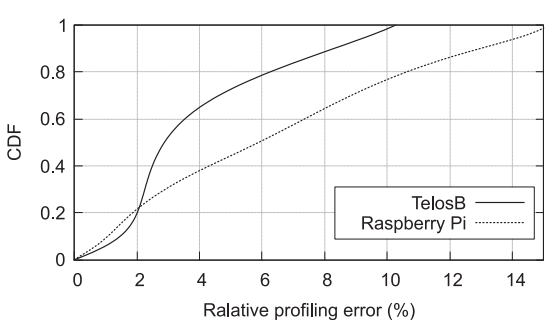


Fig. 13. Profiling accuracy of high-/low-end devices.

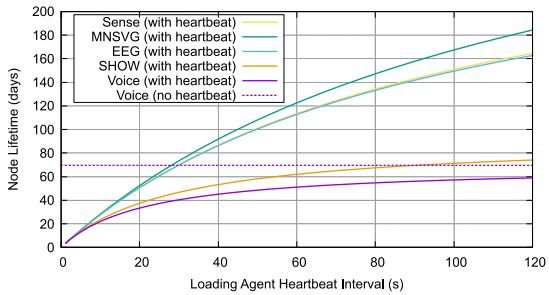


Fig. 14. Relationship between node lifetime against heartbeat interval.

6 Discussion

In this section, we discuss several important open issues of EdgeProg.

Dynamic Evolving Scenario of EdgeProg. Partitioning the application is not a one-shot job in real-world deployments. The optimal partition may change due to the disturbance of wireless network or even device breakdown.

EdgeProg supports the dynamic partition update during run-time. The environmental variation is captured by our network profiler deployed on the edge device (Section 3.2). EdgeProg periodically checks if the environmental variation leads to sub-optimal performance for a certain length of time (i.e., tolerance time), EdgeProg will start the partition updating process. The updating process includes compiling the platform-specific binary and dispatching binaries to devices for reprogramming. Users could adjust the environmental sensitivity of EdgeProg by setting the appropriate tolerance time in case of frequent updating that brings high reprogramming overhead.

Energy Drain of the Loading Agent. The loading agent in EdgeProg communicates with the edge server periodically (i.e., heartbeat) to check if there is a new binary to load and execute. When there is a new binary, the loading agent downloads the binary and loads it. Hence, the energy consumption of the loading agent is two-fold: periodical heartbeat and binary load. Inspired by [32], we build an analytical model and illustrate the energy impact of the loading agent in Fig. 14. We leave the model formulation and to our technical report [14] due to the limited space. We set the battery capacity to 2200 mAh, and assume new binaries are generated every ten days. We can see from the figure that the heartbeats indeed affect the battery lifetime. The loading agent leads to a 14.5% and 26.1% decrease of Voice benchmark when the heartbeat interval is 120s and 60s, respectively. Hence, considering the tradeoff between timeliness of binary loading and the energy drain, we set the heartbeat interval to 60s in our implementation by default, and we allow users to modify the interval to meet their individual needs.

Limitations of EdgeProg Language. Although EdgeProg language shows its simplicity and expressiveness in our evaluation, it still owns two limitations:

(1) Limited support for implementing new peripheral libraries. The software library (e.g., the Light_Solar library of TelosB in line 4 and GMM library in line 10 of Fig. 4) is one of the key components of EdgeProg's language and block-based data flow representation. In the current implementation of EdgeProg, the common peripheral libraries

and 17 data processing algorithms for virtual sensors on four platforms (TelosB, MicaZ, RPI and PC) are included in EdgeProg as we noticed in Section 4.1. We are working on the porting tutorials and code templates for implementing new libraries for peripherals and platforms. (2) Algorithms with feedback. The applications written with EdgeProg language finally transformed into a DAG for optimization, which introduces a restriction that the algorithms with feedback could not be expressed by EdgeProg language. The rationale behind this limitation is the feedback may incur directed cyclic graph (DCG), which is not solvable under our current formulation. We do not take the DCG into consideration temporarily is because: *a)* the DAG representation is widely adopted by the state-of-the-art distributed computation systems (e.g., Apache Storm), and *b)* considering DCG would incur additional solving overhead, which will affect the run-time efficiency of EdgeProg. We consider generalizing the formulation of EdgeProg to support more kinds of application topologies as our future work.

Time and Energy Profiling. In EdgeProg, we gather the time and energy information of each logic block by leveraging cycle-accurate simulators. This method achieves fair accuracy (results shown in Fig. 13) when the hardware parameters (e.g., frequency and existing workload of MCU) of the deployment environment is similar to the simulation, which is common in IoT scenario because the devices mostly operate under a certain performance level [33].

Nevertheless, the auto-scaling technique in the modern high-end edge server brings hundreds or thousands of performance levels, which makes it painful to profile the performance under each hardware parameter setting. Towards this situation, we consider generating the full profile with incomplete data through the efficient learning-driven prediction algorithm [34]. Moreover, we also consider integrating the time estimation against different current workloads proposed in [35] on edge devices to further improve the profiling accuracy of EdgeProg.

7 Related Work

EdgeProg borrows heavily from existing works. In the following paragraphs, We discuss three main categories: IoT application programming, code partitioning and offloading, as well as edge computing.

IoT Application Programming. The traditional approach for IoT programming is device-centric [36] , i.e., the application logic resides on the IoT devices. For example, developers may write application-specific sensor data processing or multi-hop forwarding based on IoT operating systems such as TinyOS or Contiki OS.

To simplify application programming for multi-device interaction, developers can adopt trigger-action programming like IFTTT on edge/cloud servers so that the whole app logic resides on the server. The IoT nodes perform general functions like sensor data sampling and data transmissions. IFTTT programming is widely adopted in the industry, such as Samsung SmartThings and Microsoft Flow. It also attracts a lot of research attention from academia [3], [17]. For example, a recent work, RT-IFTTT [3], enhances the traditional IFTTT syntax. RT-IFTTT's key idea is to dynamically adjust the sensor data polling intervals to satisfy both energy and real-time constraints. EdgeProg inherits from IFTTT's server-centric programming model but differs from existing works in two important ways. First, we enhance the IFTTT syntax with special consideration on data-intensive computation. Second, we enable much more flexible server-device cooperation by supporting code partitioning and dynamic code loading on the device, compared with RT-IFTTT which only supports adjusting data sampling intervals.

In retrospect, a similar work to ours is Tenet [23] in the sensor network literature. Tenet assumes a two-tier network architecture consisting of ordinary sensor nodes and master nodes. Tenet's principle is to place the application-specific logic on the master tier using a dataflow program. The master nodes can dynamically task sensor nodes to process data locally. In EdgeProg, the edge server plays an equivalent role to the master nodes. EdgeProg differs from Tenet in the language design, device-side system support, and performance optimizations.

Code Partitioning and Offloading. Code offloading to heterogeneous IoT nodes needs system support at the device-side. A virtual machine is a common approach to mask heterogeneity. There is rich literature in designing flexible and efficient VMs on resource-constrained nodes, including Mate [7], CapeVM [8], JVM, *etc*. In addition, a large number of offloading algorithms builds on top of VMs, e.g., Tenet [23], ASVM [37]. Besides VM, there are other more lightweight approaches such as Linux containers, RPC [38], loadable modules [39]. We adopt the loadable module approach in EdgeProg. This is because execution efficiency is critical for energy-constrained IoT nodes and native code runs much faster than VM instructions [5], [6].

There is rich literature in code partitioning and offloading algorithms for performance optimizations. LEO [40] presents an offloading algorithm targeting mobile sensing applications. LEO makes use of domain specific signal processing knowledge to smartly distribute the sensor processing tasks across the broader range of heterogeneous computational resources of high-end phones (CPU, co-processor, GPU and the cloud). LEO achieves fine-grained energy control by exposing internal pipeline stages to the scheduler. Queec [38] takes the user-perceived quality of experience (QoE) into offloading decision and makes efforts to achieve the lowest latency. EdgeProg shares similarities with many existing algorithms to optimize performance metrics such as latency or energy. However, EdgeProg uses a different formulation considering multiple rules execution, cached values, and concurrent execution on different IoT nodes.

Also, there are a variety of efforts concerning the partitioning and deployment of DAG-represented applications. Wishbone [2] presents a code partitioning algorithm among resource-constrained sensor nodes and the server to process data-intensive applications by cutting the unnecessary edges of the DAG. P-EDF-omp [41] proposes a DAG partitioning algorithm to schedule tasks across multiple processors while keeping the hard real-time guarantee for OpenMP applications. Moreover, Storm is a widely used analytic framework, and its application is constructed using DAG. HeteroEdge [42] focuses on partitioning and scheduling the Storm applications between CPUs and GPUs to achieve

better latency. Different from the above literature, the DAG partitioning algorithm of EdgeProg considers the weights on both vertices and edges, which further leads to a different formulation and solution.

Edge Computing Systems. EdgeProg runs on existing edge platforms and focuses on programming IoT nodes connected to the edge. Most existing work [43], [44] of edge computing focuses on how to program the edge itself. In ParaDrop, the edge service deployment is initiated and controlled by a cloud server. ParaDrop employs the container technology for the concurrency and isolation between edge services.

Considering the coordinated programming for both the edge and nodes, the most similar and recent work is DDFlow [45]. Its idea borrows from the existing macro-programming approach [15], which aim to build applications in the whole network point-of-view (POV) rather than per-node POV. DDFlow presents a visual programming interface for developers to state their application as a task graph. EdgeProg employes a more declarative way with a domain-specific language rather than graphical programming, and achieves the lowest latency even in the multi-rule situation while DDFlow only considers optimizing one application per time.

8 Conclusion

This paper presents EdgeProg, an edge-centric programming system with automatic code partitioning. In EdgeProg, we provide developers, especially non-experts, with an easy-to-use yet expressive programming language. Build upon the global view of our language, the code partitioner finds the best placement for each part of the application through an ILP formulation, which could be efficient and optimally solved. The key insight is that we make the best use of the computation ability of each device to achieve better performance. Evaluations show that EdgeProg could reduce the task execution latency by 31.65% for ZigBee networks and 10.26% for WiFi networks. For energy, EdgeProg saves 14.8% and 40.8% on average compared with state-of-the-arts. Also, EdgeProg reduces the lines of code by 79.41%.

Acknowledgments

We thank all the reviewers for their valuable comments and helpful suggestions.

References

[1] B. Ur *et al.*, "Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2016, pp. 3227–3231.

[2] R. Newton *et al.*, "Wishbone: Profile-based partitioning for sensornet applications," in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, pp. 395–408.

[3] S. Heo, S. Song, J. Kim, and H. Kim, "RT-IFTTT: Real-time IoT framework with trigger condition-aware flexible polling intervals," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 266–276.

[4] B. Li and W. Dong, "EdgeProg: Edge-centric programming for IoT applications," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 212–222.

[5] A. Dunkels *et al.*, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proc. 4th Int. Conf. Embedded Netw. Sensor Syst.*, 2006, pp. 15–28.

[6] W. Dong, C. Chen, X. Liu, J. Bu, and Yunhao Liu, "Dynamic linking and loading in networked embedded systems," in *Proc. IEEE 6th Int. Conf. Mobile Adhoc Sensor Syst.*, 2009, pp. 554–562.

[7] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *Proc. 10th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2002, pp. 85–95.

[8] N. Reijers and C.-S. Shih, "CapeVM: A safe and fast virtual machine for resource-constrained Internet-of-Things devices," in *Proc. 16th ACM Conf. Embedded Netw. Sensor Syst.*, 2018, pp. 250–263.

[9] W. Dong *et al.*, "Elon: Enabling efficient and long-term reprogramming for wireless sensor networks," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2010, pp. 45–60.

[10] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 49–62.

[11] L. Hsiao *et al.*, "Automating the generation of hardware component knowledge bases," in *Proc. 20th ACM SIGPLAN/SIGBED Int. Conf. Lang. Compilers Tools Embedded Syst.*, 2019, pp. 163–176.

[12] L. Hsiao *et al.*, "Creating hardware component knowledge bases with training data generation and multi-task learning," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 6, pp. 1–26, 2020.

[13] M. Sánchez-Fernández, M. de-Prado-Cumplido, J. Arenas-Garcia, and F. Perez-Cruz, "SVM multiregression for nonlinear channel estimation in multiple-input multiple-output systems," *IEEE Trans. Signal Process.*, vol. 52, no. 8, pp. 2298–2307, Aug. 2004.

[14] B. Li and W. Dong, "Technical report: Edge-centric programming for IoT applications with EdgeProg," 2021, *arXiv:2111.15098*.

[15] R. Gummadi *et al.*, "Macro-programming wireless sensor networks using kairos," in *Proc. Int. Conf. Distrib. Comput. Sensor Syst.*, 2005, pp. 126–140.

[16] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proc. 6th Int. Symp. Inf. Process. Sensor Netw.*, 2007, pp. 489–498.

[17] B. Ur *et al.*, "Practical trigger-action programming in the smart home," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, 2014, pp. 803–812.

[18] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2015, pp. 215–225.

[19] J. V. Jeyakumar *et al.*, "SenseHAR: A robust virtual activity sensor for smartphones and wearables," in *Proc. 17th Conf. Embedded Netw. Sensor Syst.*, 2019, pp. 15–28.

[20] R. LiKamWa *et al.*, "MoodScope: Building a mood sensor from smartphone usage patterns," in *Proc. 11th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2013, pp. 389–402.

[21] H. Lu *et al.*, "The jigsaw continuous sensing engine for mobile phone applications," in *Proc. 8th ACM Conf. Embedded Netw. Sensor Syst.*, 2010, pp. 71–84.

[22] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2014, pp. 4087–4091.

[23] O. Gnawali *et al.*, "The Tenet architecture for tiered sensor networks," in *Proc. 4th Int. Conf. Embedded Netw. Sensor Syst.*, 2006, pp. 153–166.

[24] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[25] S. Khare *et al.*, "Linearize, predict and place: Minimizing the makespan for edge-based stream processing of directed acyclic graphs," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, 2019, pp. 1–14.

[26] A. Mitsos *et al.*, "Mccormick-based relaxations of algorithms," *SIAM J. Optim.*, vol. 20, no. 2, pp. 573–601, 2009.

[27] R. Kumar, E. Kohler, and M. Srivastava,"Harbor: Software-based memory protection for sensor nodes," in *Proc. 6th Int. Symp. Inf. Process. Sensor Netw.*, 2007, pp. 340–349.

[28] F. Marcelloni and M. Vecchio,"An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks," *The Comput. J.*, vol. 52, no. 8, pp. 969–987, 2009.

[29] A. Shoeb, J. Guttag, S. Schachter, D. Schomer, B. Bourgeois, and S. T. Treves, "Detecting seizure onset in the ambulatory setting: Demonstrating feasibility," in *Proc. IEEE 27th Annu. Conf. Eng. Medicine Biol.*, 2006, pp. 3546–3550.

[30] X. Lin *et al.*, "SHOW: Smart handwriting on watches," *Proc. ACM Interactive Mobile Wearable Ubiquitous Technol.*, vol. 1, 2018, Art. no. 151.

[31] C. Xu *et al.*, "Crowd++: Unsupervised speaker count with smartphones," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2013, pp. 43–52.

[32] W. Dong *et al.*, "Optimizing relocatable code for efficient software update in networked embedded systems," *ACM Trans. Sensor Netw.*, vol. 11, no. 2, 2015, Art. no. 22.
[33] B. Li and W. Dong, "Automatic generation of IoT device platforms with autolink," *IEEE Internet Things J.*, vol. 8, no. 7, pp. 5893–5903, Apr. 2021.
[34] M. Hu, L. Zhuang, D. Wu, Y. Zhou, X. Chen, and L. Xiao, "Learning driven computation offloading for asymmetrically informed edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1802–1815, Aug. 2019.
[35] B. Li*et al.*, "Queec: QoE-aware edge computing for IoT devices under dynamic workloads," *ACM Trans. Sensor Netw.*, vol. 17, no. 3, pp. 1–23, 2021.
[36] G. Guan *et al.*, "TinyLink: A holistic system for rapid development of IoT applications," in *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, 2017, pp. 383–395.
[37] P. Levis *et al.*, "Active sensor networks," in *Proc. 2nd Conf. Symp. Netw. Syst. Des. Implementation*, 2005, pp. 343–356.
[38] G. Guan *et al.*, "Queec: QoE-aware edge computing for complex IoT event processing under dynamic workloads," in *Proc. ACM Turing Celebration Conf.*, 2019, pp. 1–5.
[39] C. Cao, L. Luo, Y. Gao, W. Dong, and C. Chen, "TinySDM: Software defined measurement in wireless sensor networks," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw.*, 2016, pp. 1–12.
[40] P. Georgiev *et al.*, "LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, 2016, pp. 320–333.
[41] Y. Wang, X. Jiang, N. Guan, Z. Guo, X. Liu, and W. Yi, "Partitioning based scheduling of OpenMP task systems with tied tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1322–1339, Jun. 2021.
[42] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1270–1278.
[43] P. Liu, D. Willis, and S. Banerjee, "ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *Proc. ACM/IEEE Symp. Edge Comput.*, 2016, pp. 1–13.
[44] Z. Li, X. Peng, L. Chao, and Z. Xu, "EveryLite: A lightweight scripting language for micro tasks in IoT systems," in *Proc. ACM/IEEE Symp. Edge Comput.*, 2018, pp. 381–386.
[45] J. Noor *et al.*, "DDFLOW: Visualized declarative programming for heterogeneous IoT networks," in *Proc. Int. Conf. Internet Things Des. Implementation*, 2019, pp. 172–177.

Borui Li (Student Member, IEEE) received the BS degree in computer science from the Nanjing University of Posts and Telecommunications, China, in 2017. He is currently working toward the PhD degree at Zhejiang University, China. His research interests include Internet of Things and edge computing systems.

Wei Dong (Member, IEEE) received the BS and PhD degrees from the College of Computer Science, Zhejiang University, China, in 2005 and 2011, respectively. He is currently a full professor with the College of Computer Science, Zhejiang University, China, where he leads the Embedded and Networked Systems Laboratory. He has published more than 100 papers in prestigious conferences and journals, including MobiCom, INFOCOM, ToN, and TMC. He is a member of the ACM.