EdgeProg: Edge-centric Programming for IoT Applications

Borui Li and Wei Dong
College of Computer Science, Zhejiang University, China, and,
Alibaba-Zhejiang University Joint Institute of Frontier Technologies
Email: {*libr, dongw*}@emnets.org

***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% and 79.41% in terms of execution latency and lines of code, compared with state-of-the-art approaches.

***Index Terms*—IoT, IFTTT, Integer Linear Programming**

I. Introduction

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, resulting in 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 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?
- How to partition the user-perceived program to achieve the best delay performance?
- 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) EdgeProg achieves a 20.96% reduction on average, and up to 99.05% 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) For application runtime, the dynamic linking and loading technique outperforms than design alternatives such as virtual machine (by 9.98X) and scripting languages (by 6.37X). (4) The profiling methods adopted by EdgeProg achieves 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 to minimize the makespan of the task. The partitioning algorithm optimizes the placement of each stage in an application with consideration of both processing and network latency.
- 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.

II. Background and EdgeProg Usage

In this section, we briefly introduce the techniques used in EdgeProg, including the dynamic linking and loading of IoT devices as well as the virtual sensor. Then we present the usage of EdgeProg with a simple smart home application.

A. Background

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 [4] and SELF [5]). 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 [6], [4], [7] and bootloader [8], 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.

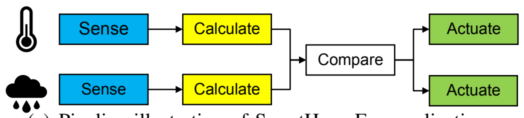
Virtual sensor. Opposite to the physical or hardware sensor, a virtual sensor is a logical entity that abstracts the data sensed by real sensors which could be located at different places. Traditional hardware sensors generally produce raw measurements of physical properties such as the moisture value or light intensity, which are unprofitable unless being transformed into the high-level domain-dependent information. Furthermore, capturing the valuable information usually requires 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, virtual sensors have been proposed. For example, SenseHAR [9] advocates a virtual activity sensor that abstracts the data of several inertial sensors from different devices using a sensor fusion network. Similarly, LiKamWa *et al.* propose a virtual mood sensor named Moodscope [10] to 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.

B. EdgeProg Usage

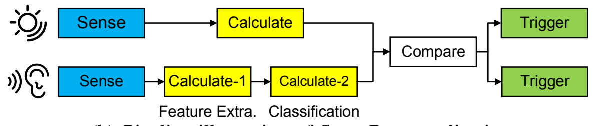
We excerpt a simple smart home project named `SmartHomeEnv` from smarthome.com to illustrate how EdgeProg can be used. As shown in Figure 1(a), `SmartHomeEnv` takes the temperature and humidity data from two IoT nodes as input, turns on the air conditioner and dryer if the two readings exceed fixed thresholds. The two nodes are wirelessly connected to an edge server, which could be a Raspberry Pi.

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



(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 stage processing time on the device (and on the edge node, if necessary) is annotated inside the block.

```
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.

the edge server. Figure 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.

A key feature of EdgeProg is that it can automatically partition the whole application codes to optimize the execution performance, which is increasingly essential for computation-intensive IoT tasks such as speech recognition and video surveillance. Figure 1(b) shows such an example in which the `VoiceRecog` task may be too heavyweight for resource-constrained devices such as TelosB or Arduino. EdgeProg will automatically partition this task to the edge-side if it yields a better performance than placing it on the device.

III. EdgeProg Overview

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

A. 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.
- **Latency-aware.** The timeliness is recognized as a critical performance metric of an edge-device coordinated application. The ability to deliver a time-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.

B. EdgeProg Architecture

In Figure 3, we show a birds-eye view of EdgeProg's system architecture and functional workflow. 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 §IV-A for details). The system takes the user code as input, preprocesses and feeds it into the *code partitioner*. With the help of the *time profile* of each device and our partitioning algorithm, the code partitioner finds the optimal partition and placement of each stage. 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 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 §IV-A.

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 §IV-B.

Time Profiler. The timing information of each stage is one of the critical inputs to the code partitioner. Similar to [2], [11], 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 §V-E.

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 [12] 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

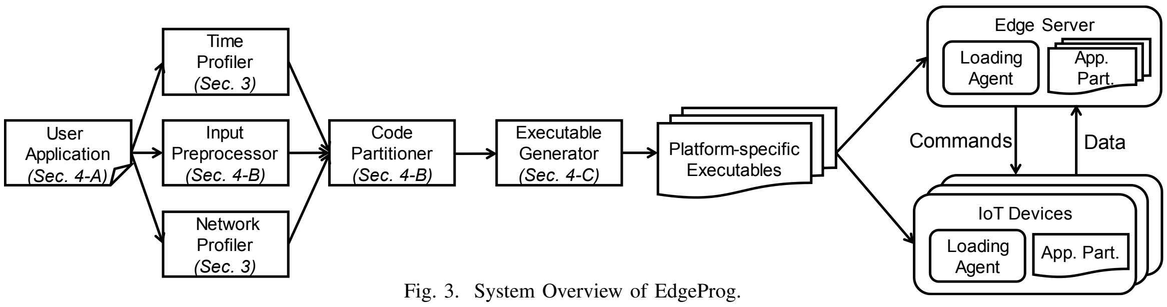

User Application (Sec. 4-A)
Time Profiler (Sec. 3)
Input Preprocessor (Sec. 4-B)
Network Profiler (Sec. 3)
Code Partitioner (Sec. 4-B)
Executable Generator (Sec. 4-C)
Platform-specific Executables
Edge Server
Loading Agent
App. Part.
Commands
Data
IoT Devices
Loading Agent
App. Part.

Fig. 3. System Overview of EdgeProg.

Zigbee network. Raw observations such as the bandwidth and received signal strength indicator (RSSI), which is sampled by the loading agent at 0.1Hz in order not to influence the regular network transmission, are fed into the M-SVR. The predictor outputs the future throughput estimation and per-packet transmission time for further fine-grained time calculation in §IV-B. 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 §IV-C.

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.

IV. 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 EdgeProg obtains the optimal partition of the input IoT application with fully aware of the user-perceived event handling latency, 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.

A. 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 Figure 4, we use the SmartDoor application described in §II-B as an example to illustrate three critical features in the following.

```
 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.

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 [13] and Regiment [14]. 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 [15], [3], [16] 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 Figure 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 nowaday IoT scenario, we enhance our DSL with the virtual sensor. As we described in §II-A, the virtual sensor is an efficient approach for developers to describe complicated data processing algorithms, which are generally organized in two stages: feature extraction and

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

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

classification. As shown in Figure 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 [17], [18]. 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 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 Figure 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 the 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 Figure 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.

B. 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. Thus, as latency-ware is one of the design goals of EdgeProg, we formulate the latency minimizing problem and employ an efficient solver to obtain the optimal placement of each stage.

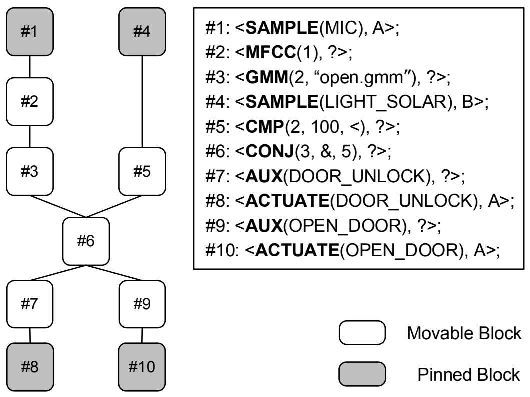
The key insight of our partitioning algorithm is that we push the computation close to the data source as much as possible and make the best use of the computation ability of each device to achieve latency reduction. Moreover, the optimal placement that exhibits favorable computation-transmission tradeoff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

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 Figure 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 Figure 6.

- *Functionality.* To express the functionality, we borrow the idea of tasklet primitives from Tenet [19] such as `SAMPLE`, `ACTUATE` and `CONJ`, which provides 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. 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.

Except for the 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



#1: <**SAMPLE**(MIC), A>;
#2: <**MFCC**(1), ?>;
#3: <**GMM**(2, "open.gmm"), ?>;
#4: <**SAMPLE**(LIGHT_SOLAR), B>;
#5: <**CMP**(2, 100, <), ?>;
#6: <**CONJ**(3, &, 5), ?>;
#7: <**AUX**(DOOR_UNLOCK), ?>;
#8: <**ACTUATE**(DOOR_UNLOCK), A>;
#9: <**AUX**(OPEN_DOOR), ?>;
#10: <**ACTUATE**(OPEN_DOOR), A>;

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

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 stage 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 Figure 6 illustrates.

The ILP problem for optimal partitioning. 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 use a binary indicator $X_{b_is_i}$ to demonstrate the partition result as:

$$X_{b_is_i} = \begin{cases} 1 & \text{logic block } b_i \text{ is assigned to device } s_i \\ 0 & \text{logic block } b_i \text{ is not assigned to device } s_i \end{cases}, \quad (1)$$

where s_i represents the possible placement device of block b_i.

We would like to borrow the existing code partitioning algorithm proposed in Wishbone [2] to solve our problem. *Unfortunately*, Wishbone algorithms are not feasible in our problem mainly due to the following two differences:

- *Node weights.* Consider assigning weights to vertices in the graph which represents the processing time of the corresponding logic block. The weight of movable blocks in our data flow graph is two-fold, indicating the local and edge-server processing time, while each vertex in the Wishbone 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 latency, which makes the Wishbone formulation no longer suitable for our problem.

Different from Wishbone, our objective is to minimize the task execution latency, which leads to minimizing the length of the longest path in the data flow graph. We define the full path as the path from a source vertex to a sink vertex, denoted as p. We use $len(p)$, $\delta(p)$ and $P(G)$ to indicate the length of path p, the number of vertices in path p, and the set of all full paths in graph G. Our optimization goal is thus denoted as $\min\max_{p\in P(G)} len(p)$. Moreover, due to the $len(p)$ is the sum of data processing and transmitting latency across all possible placements, our objective is formulated with the binary placement indicator $X_{b_is_i} \in \{0,1\}$ as:

$$\arg\min_{X} \max_{p\in P(G)} \sum_{i=1}^{\delta(p)} \sum_{s_i\in S_i} X_{b_is_i} T^C_{b_is_i} + \sum_{i=1}^{\delta(p)-1} \sum_{\substack{s_i\in S_i \\ s_{i'}\in S_{i'}}} X_{b_is_i} X_{b_{i'}s_{i'}} T^N_{b_is_is_{i'}} \quad (2)$$

where i, i' are the adjacent vertices in path p (*i.e.*, i'=i+1). S_i denotes the set of all possible placements of the i-th logic block. $T^C_{b_is_i}$ denotes the data processing time of the i-th block on placement s_i, and $T^N_{b_is_is_{i'}}$ represents data transmitting time between block b_i of placement s_i and block $b_{i'}$ of placement $s_{i'}$. We assume that the data transmission time is negligible if the two consecutive logic blocks are placed on the same device. Thus we have:

$$T^N_{b_is_is_{i'}} = \begin{cases} \left\lceil \frac{q_{ii'}}{r_{ii'k}} \right\rceil t_{ii'k} & s_i \neq s_{i'} \\ 0 & s_i = s_{i'} \end{cases}, \quad (3)$$

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 could be 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 §III-B.

Nevertheless, the objective formulation as Equation (2) is a quadratic minimax problem, which is shown to be an NP-hard problem [20]. The state-of-the-arts employ heuristic algorithms to solve it efficiently. For example, the most recent work [21] utilizes a breadth-first greedy search algorithm to solve it. While we prefer to employ a solver that is less prone to local optima. Inspired by McCormick relaxation [22], we re-formulate Equation (2) as an integer programming problem (ILP), which could be efficiently solved by the standard solver, *e.g.*, `lp_solve`. Towards this goal, we first convert the quadratic objective function to a linear one by introducing an auxiliary variable $\varepsilon_{is_is_{i'}}$=$X_{b_is_i}\cdot X_{b_{i'}s_{i'}}$ to replace the quadratic term $X_{b_is_i}\cdot X_{b_{i'}s_{i'}}$ in Equation (2). Moreover, the presence of $\varepsilon_{is_is_{i'}}$ causes the introduction of these constraints:

$$(\forall i\in \delta(p)-1, s_i\in S_i, s_{i'}\in S_{i'}) \quad \varepsilon_{is_is_{i'}} \geq 0 \quad (4)$$

$$(\forall i\in \delta(p)-1, s_i\in S_i, s_{i'}\in S_{i'}) \quad \varepsilon_{is_is_{i'}} \leq X_{b_is_i} \quad (5)$$

$$(\forall i\in \delta(p)-1, s_i\in S_i, s_{i'}\in S_{i'}) \quad \varepsilon_{is_is_{i'}} \leq X_{b_{i'}s_{i'}} \quad (6)$$

$$(\forall i\in \delta(p)-1, s_i\in S_i, s_{i'}\in S_{i'}) \quad \varepsilon_{is_is_{i'}} + 1 \geq X_{b_is_i} + X_{b_{i'}s_{i'}}. \quad (7)$$

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 illustrated as:

$$\text{Objective:} \qquad \underset{X}{\arg\min} \quad z \tag{8}$$

Subject to:

$$z \geq \sum_{i=1}^{\delta(p)} \sum_{s_i \in S_i} X_{b_i s_i} T^C_{b_i s_i} + \sum_{i=1}^{\delta(p)-1} \sum_{\substack{s_i \in S_i \\ s_{i'} \in S_{i'}}} \varepsilon_{i s_i s_{i'}} T^N_{b_i s_i s_{i'}}, \forall p \in G. \tag{9}$$

Furthermore, we add constraints for placement indicator $X_{b_i s_i}$ to ensure all the logic blocks are appointed to a specific device.

$$\sum_{s_i \in S_i} X_{b_i s_i} = 1, \forall i \in p \in G \tag{10}$$

Thus, any optimal solution of Equation (8) subject to (4)-(7), (9) and (10) will be the optimal partition of the input application.

C. Executable Generator

The executable generator 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 edge server (mostly Linux-compatible hardware) as well as sensing devices in a similar manner. Then compile 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.

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 due to the short protothread incurs much process switching overhead, which will also harm the overall makespan.

[1]Contiki supports preemptive multi-threading as a 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.

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. The fragments of each device are obtained by leveraging a depth-first traverse of the graph 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 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.

V. Evaluation

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

A. Experiment Setup

We summarize the five macro-benchmarks to evaluate our system in Table I: two sensing applications and three real-world applications.

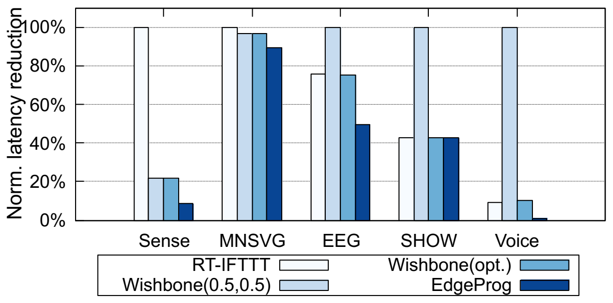
- *Sense.* A common sensing application with outlier detection using algorithms proposed in [23] and data compression using the LEC algorithm [24].
- *MNSVG.* A weather forecast application using an MNSVG model proposed in [3] to predict temperature and humidity values.
- *EEG.* Using the EEG signal to detect seizure [25] , 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 [26].
- *Voice.* Counting the number of speakers with signal processing and clustering algorithms [27].

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.

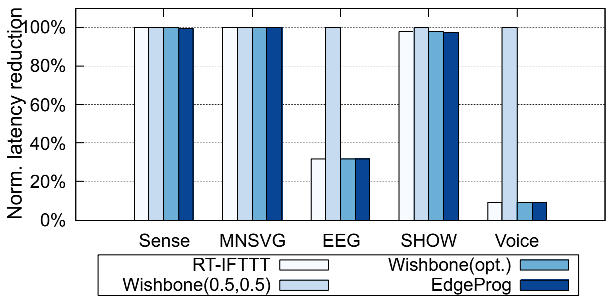
- *RT-IFTTT* [3]. The server does all of the computation. IoT devices only need to report the sensor value or take actions under the server's command.
- *Wishbone(0.5, 0.5)* [2]. Wishbone is a partitioning system for sensornet applications whose goal is to minimize a combined objective of CPU and network workload, which could be formulated with two weights as (α 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 with tuning the parameters with 0.1 step, and record the best performance as this baseline.

TABLE I
IMPLEMENTED BENCHMARK APPLICATIONS.

Name	Application	Sensor	# Operators	Algorithms
Sense	Outlier Detector [23], [24]	Temp., Light	8	Average, Matrix multiplication, LEC compression
MNSVG	Weather Forecasting [3]	Temp., Humidity	4	MNSVG
EEG	Seizure Onset Detect. [25]	EEG	80	Wavelet decomposition, SVM
SHOW	Smart Handwriting [26]	Accel.	13	FFT, Random forest
Voice	Speaker Count [27]	MIC	10	MFCC, Pitch estimation, Unsupervised clustering

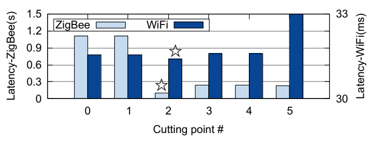


(a) Speedup under Zigbee network.

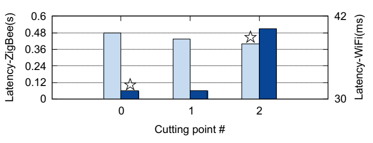


(b) Speedup under WiFi network.

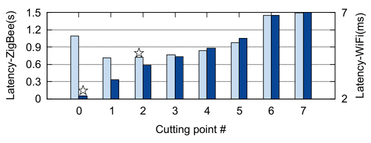
Fig. 7. Latency speedup achieved by EdgeProg normalized to the worst-performed baseline. EdgeProg reduces the task latency by 18.2% compared with WishBone(opt.) and 31.0% with RT-IFTTT on average.



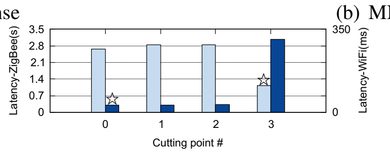
(a) Sense



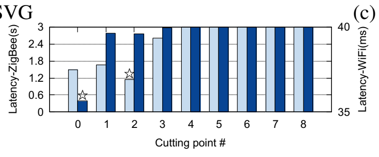
(b) MNSVG



(c) EEG



(d) SHOW



(e) Voice

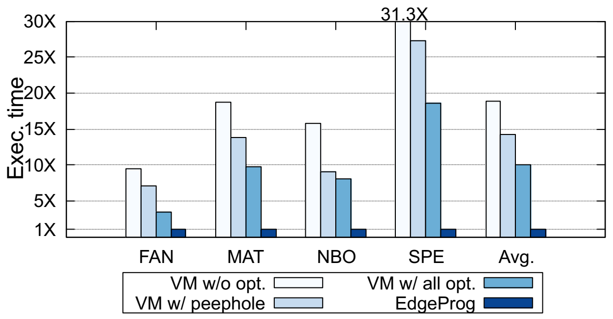
Fig. 8. Latency of each macro-benchmark at available cutting points under both networks. Cutting points are arranged to assure that fewer operators are executed locally when point number gets bigger. We omit the bigger cutting points part of Voice and EEG due to the continuous growth of latency.

B. Latency Reduction

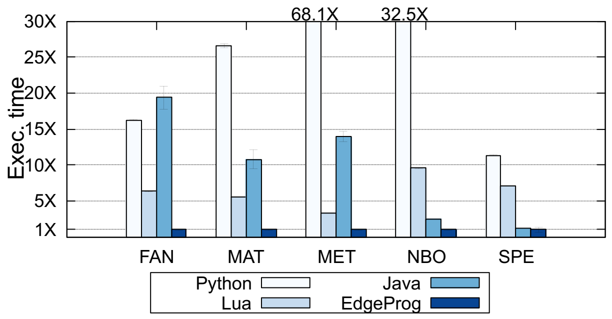
Figure 7 depicts the task makespan of five macro-benchmarks under Zigbee (on TelosB node) and WiFi (on Raspberry Pi) network. We use a laptop with 2.8GHz i7-7700HQ CPU and 16GB memory as our edge server. EdgeProg achieves a 20.96% reduction on average across all settings, and up to 99.05% reduction in 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 for Voice and EEG benchmarks under both settings while falls flat for MNSVG. This variation mainly due to computation complexity and network demands of each benchmark. As illustrated in Table I, 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 than 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. To further study this observation, we established a ground truth by exhaustively running each benchmark at every available cutting points on our testbed. Figure 8 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 the left, which means the decrease of



(a) Compare with virtual machines



(b) Compare with scripting languages

Fig. 9. Run-time efficiency comparison between EdgeProg and design alternatives.

TABLE II
DISSEMINATION SIZE AMONG PLATFORMS (BYTE).

App.	TelosB	MicaZ	Raspberry Pi
Sense	4344	6384	4004
MNSVG	2756	3460	2280
EEG	4500	6276	3920
SHOW	22952	28660	14540
Voice	32076	42416	19336

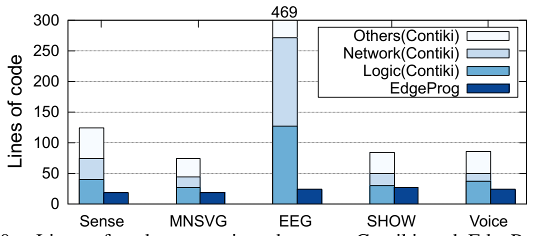


Fig. 10. Lines of code comparison between Contiki and EdgeProg. The "Logic", "Network" and "Others" represent the lines of code for expressing core application logic, inter-device network and others such as definition and included headers in Contiki source code.

optimization space and leads to closer performance among baselines.

C. 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 II.

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 Web Language Benchmark Game (WLBG). WLBG is a language benchmark suite maintained by the Debian community. The five benchmarks we excerpted are Fannkuch problem (FAN), Matrix multiplication (MAT), Meteor predicting (MET), N-Body solution (NBO), and Spectral-Norm calculating (SPE). We use CapeVM [7], a state-of-the-art Java VM developed for lightweight execution on embedded devices, as the representative of 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 native Java, which is used in CapeVM, as our design alternatives of scripting languages.

Figure 9 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 Figure 9(a), the VM method introduces a massive loss of run-time efficiency. VM costs more than EdgeProg when executing the same benchmark by 9.98X on average and up to 31.32X. As for scripting languages and native Java illustrated in Figure 9(b), EdgeProg's dynamic linking and loading technique still outperforms than alternatives. Python incurs the most overhead averaged 30.96X and Lua, being famous for its lightweight, still slows by 6.37X than ours.

D. Programming Language

We intend to compare the lines of code needed to implement the macro-benchmarks described in §V-A between traditional Contiki-style and EdgeProg-style. Figure 10 illustrates the 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 relieve 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

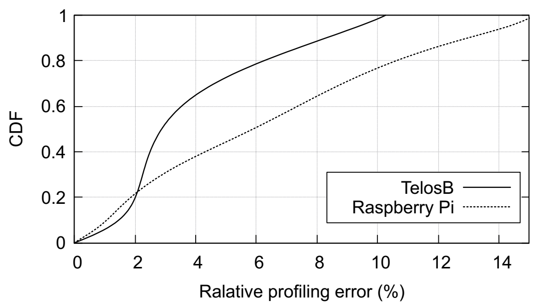


Fig. 11. Profiling accuracy of high-/low-end devices.

consists of 10 EEG devices, and each device owns eight stages. Programming 10 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 due to they need only one sensing device and an edge device.

E. 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 Figure 11. `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.

VI. 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 [28], *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], [15]. 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 [19] 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 a rich literature in designing flexible and efficient VMs on resource-constrained nodes, including Mate [6], CapeVM [7], JVM, *etc*. In addition, a large number of offloading algorithms builds on top of VMs, *e.g.*, Tenet [19], ASVM [29]. Besides VM, there are other more lightweight approaches such as Linux containers, RPC [30], loadable modules [31]. 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 [4], [5].

There is a rich literature in code partitioning and offloading algorithms for performance optimizations. LEO [32] presents an offloading algorithm targets for 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 [30] takes the user-perceived quality of experience (QoE) into offloading decision and makes efforts to achieve the lowest latency. Wishbone [2] presents a code partitioning algorithm among resource-constrained sensor nodes and the server to process data-intensive applications. 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.

Edge computing systems. EdgeProg runs on existing edge platforms, focus on programming IoT nodes connected to the edge. Most existing work of edge computing focuses on how to program the edge itself. In ParaDrop [33], 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.

As for programming the edge-connected nodes, EveryLite [34] proposes a lightweight scripting language (37KB of core runtime size) extended from Lua for developers to build applications. Nevertheless, EdgeProg chooses the native C approach to further reduce the run-time overhead. Furthermore, EveryLight only focuses on programming one node, while EdgeProg also takes the edge device and connected nodes into consideration. Considering the coordinated programming for both the edge and nodes, the most similar and recent work is DDFlow [35]. Its idea borrows from the existing macro-programming approaches [13], [14], [36], 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 with Node-RED. 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.

VII. Conclusion

This paper presents EdgeProg, an edge-centric programming system for relieving developers from detailed implementations by automatically partitions, generates, disseminates and loads the program. 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 most efficient 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 latency reduction. Evaluations show that EdgeProg could reduce the task execution latency by 31.65% for ZigBee networks and 10.26% for WiFi networks. Also, EdgeProg reduces the lines of code by 79.41%.

Acknowledgment

This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700, the National Science Foundation of China under Grant No. 61772465, Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020001. We thank all the reviewers for their valuable comments and helpful suggestions. Wei Dong is the corresponding author.

References

[1] B. Ur *et al.*, “Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes,” in *Proc. of ACM CHI*, 2016.
[2] R. Newton *et al.*, “Wishbone: Profile-based partitioning for sensornet applications.” in *Proc. of USENIX NSDI*, 2009.
[3] S. Heo *et al.*, “RT-IFTTT: Real-Time IoT Framework with Trigger Condition-Aware Flexible Polling Intervals,” in *Proc. of IEEE RTSS*, 2017.
[4] A. Dunkels *et al.*, “Run-time dynamic linking for reprogramming wireless sensor networks,” in *Proc. of ACM SenSys*, 2006.
[5] W. Dong *et al.*, “Dynamic linking and loading in networked embedded systems,” in *Proc. of IEEE MASS*, 2009.
[6] P. Levis and D. Culler, “Maté: A tiny virtual machine for sensor networks,” in *Proc. of ACM ASPLOS*, 2002.
[7] N. Reijers and C.-S. Shih, “CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices,” in *Proc. of ACM SenSys*, 2018.
[8] W. Dong *et al.*, “Elon: enabling efficient and long-term reprogramming for wireless sensor networks,” in *Proc. of ACM SIGMETRICS*, 2010.
[9] J. V. Jeyakumar *et al.*, “SenseHAR: a robust virtual activity sensor for smartphones and wearables,” in *Proc. of ACM SenSys*, 2019.
[10] R. LiKamWa *et al.*, “Moodscope: Building a mood sensor from smartphone usage patterns,” in *Proc. of ACM MobiSys*, 2013.
[11] E. Cuervo *et al.*, “MAUI: making smartphones last longer with code offload,” in *Proc. of ACM MobiSys*, 2010.
[12] M. Sánchez-Fernández *et al.*, “Svm multiregression for nonlinear channel estimation in multiple-input multiple-output systems,” *IEEE transactions on signal processing*, 2004.
[13] R. Gummadi *et al.*, “Macro-programming wireless sensor networks using kairos,” in *Proc. of IEEE DCOSS*, 2005.
[14] R. Newton *et al.*, “The regiment macroprogramming system,” in *Proc. of ACM/IEEE IPSN*, 2007.
[15] B. Ur *et al.*, “Practical trigger-action programming in the smart home,” in *Proc. of ACM CHI*, 2014.
[16] J. Huang and M. Cakmak, “Supporting mental model accuracy in trigger-action programming,” in *Proc. of ACM UbiComp*, 2015.
[17] H. Lu *et al.*, “The jigsaw continuous sensing engine for mobile phone applications,” in *Proc. of ACM SenSys*, 2010.
[18] G. Chen *et al.*, “Small-footprint keyword spotting using deep neural networks,” in *Proc. of IEEE ICASSP*, 2014.
[19] O. Gnawali *et al.*, “The tenet architecture for tiered sensor networks,” in *Proc. of ACM SenSys*, 2006.
[20] R. Eidenbenz and T. Locher, “Task allocation for distributed stream processing,” in *Proc. of IEEE INFOCOM*, 2016.
[21] S. Khare *et al.*, “Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs,” in *Proc. of ACM/IEEE SEC*, 2019.
[22] A. Mitsos *et al.*, “Mccormick-based relaxations of algorithms,” *SIAM Journal on Optimization*, vol. 20, no. 2, pp. 573–601, 2009.
[23] R. Kumar *et al.*, “Harbor: software-based memory protection for sensor nodes,” in *Proc. of ACM/IEEE IPSN*, 2007.
[24] F. Marcelloni and M. Vecchio, “An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks,” *the computer journal*, 2009.
[25] A. Shoeb *et al.*, “Detecting seizure onset in the ambulatory setting: Demonstrating feasibility,” in *Proc. of IEEE EMBS*, 2006.
[26] X. Lin *et al.*, “Show: Smart handwriting on watches,” *Proc. of ACM UbiComp*, 2018.
[27] C. Xu *et al.*, “Crowd++: unsupervised speaker count with smartphones,” in *Proc. of ACM UbiComp*, 2013.
[28] G. Guan *et al.*, “TinyLink: A Holistic System for Rapid Development of IoT Applications,” in *Proc. of ACM MobiCom*, 2017.
[29] P. Levis *et al.*, “Active sensor networks,” in *Proc. of USENIX NSDI*, 2005.
[30] G. Guan *et al.*, “Queec: Qoe-aware edge computing for complex iot event processing under dynamic workloads,” in *Proc. of ACM TURC*, 2019.
[31] C. Cao *et al.*, “TinySDM: Software defined measurement in wireless sensor networks,” in *Proc. of ACM/IEEE IPSN*, 2016.
[32] P. Georgiev *et al.*, “Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources,” in *Proc. of ACM MobiCom*, 2016.
[33] P. Liu *et al.*, “Paradrop: Enabling lightweight multi-tenancy at the network’s extreme edge,” in *Proc. of ACM/IEEE SEC*, 2016.
[34] Z. Li *et al.*, “Everylite: A lightweight scripting language for micro tasks in iot systems,” in *Proc. of IEEE/ACM SEC*, 2018.
[35] J. Noor *et al.*, “Ddflow: visualized declarative programming for heterogeneous iot networks,” in *Proc. of IoTDI*, 2019.
[36] M. Modahl *et al.*, “Mediabroker: An architecture for pervasive computing,” in *Proc. of IEEE PerCom*, 2004.