Spinal Code: Automatic Code Extraction for Near-User Computation in Fogs

Bongjun Kim
POSTECH
Pohang, Republic of KOREA
bong90@postech.ac.kr

Seonyeong Heo
POSTECH
Pohang, Republic of KOREA
heosy@postech.ac.kr

Gyeongmin Lee
POSTECH
Pohang, Republic of KOREA
paina@postech.ac.kr

Seungbin Song
Yonsei University
Seoul, Republic of KOREA
seungbin@yonsei.ac.kr

Jong Kim
POSTECH
Pohang, Republic of KOREA
jkim@postech.ac.kr

Hanjun Kim*
Yonsei University
Seoul, Republic of KOREA
hanjun@yonsei.ac.kr

ABSTRACT

In the Internet of Things (IoT) environments, cloud servers integrate various IoT devices including sensors and actuators, and provide new services that assist daily lives of users interacting with the physical world. While response time is a crucial factor of quality of the services, supporting short response time is challenging for the cloud servers due to a growing number and amount of connected devices and their communication. To reduce the burden of the cloud servers, fog computing is a promising alternative to offload computation and communication overheads from the cloud servers to fog nodes. However, since existing fog computing frameworks do not extract codes for fog nodes fully automatically, programmers should manually write and analyze their applications for fog computing. This work proposes Spinal Code, a new compiler-runtime framework for near-user computation that automatically partitions an original cloud-centric program into distributed sub-programs running over the cloud and fog nodes. Moreover, to reduce response time in the physical world, Spinal Code allows programmers to annotate latency sensitive actuators in a program, and optimizes the critical paths from required sensors to the actuators when it generates the sub-programs. This work implements 9 IoT programs across 4 service domains: healthcare, smart home, smart building and smart factory, and demonstrates that Spinal Code successfully reduces 44.3% of response time and 79.9% of communication on the cloud compared with a cloud-centric model.

CCS CONCEPTS

• **Hardware** → **Emerging languages and compilers**; • **Software and its engineering** → *Distributed programming languages.*

*The corresponding author

CC '19, February 16–17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6277-1/19/02…$15.00
https://doi.org/10.1145/3302516.3307356

KEYWORDS

Internet of Things, IoT, Fog Computing

ACM Reference Format:
Bongjun Kim, Seonyeong Heo, Gyeongmin Lee, Seungbin Song, Jong Kim, and Hanjun Kim. 2019. Spinal Code: Automatic Code Extraction for Near-User Computation in Fogs. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307356

1 INTRODUCTION

Interacting with the physical world through IoT devices such as sensors and actuators, the Internet of Things (IoT) inaugurates new services such as healthcare and smart home. Figure 1 shows a quality management example in a smart factory service. In the example, sensors measure features of each product and send the measured values ($L_{1...N}$) to a cloud server. The cloud server calculates and logs the average and maximum errors (E_a, E_m) between the measured values and their target values (L_T), and recalibrates the machine. If E_m is larger than a threshold (E_{th}), the server also sends an error notification to registered mobile phones. Here, some of the actuators are latency sensitive. While the logging latency is hidden to users, the recalibration latency from the measurement affects manufacturing throughput and product quality of the factory. Therefore, reducing response time especially for the latency sensitive actuators is important to improve quality of the services (QoS).

While response time is a crucial factor for IoT services to satisfy the users, supporting low response time is challenging for IoT frameworks. Recent IoT frameworks such as Samsung SmartThings [30], Microsoft Flow [26] and IFTTT [15] are cloud-centric like Figure 1, having their own cloud servers that integrate IoT devices in user environments like thermometers, hygrometers, bulbs and mobile phones. Although the cloud-centric IoT frameworks make IoT programming easy by liberating programmers from connectivity management of the devices, communication between a cloud server and the devices through the Internet may cause unpredictable delays, harming QoS especially for latency sensitive actuators like the machines in the smart factory example. Moreover, due to its centralized architecture and a growing number of connected devices, computation and communication overheads on their cloud servers are huge and increasing.

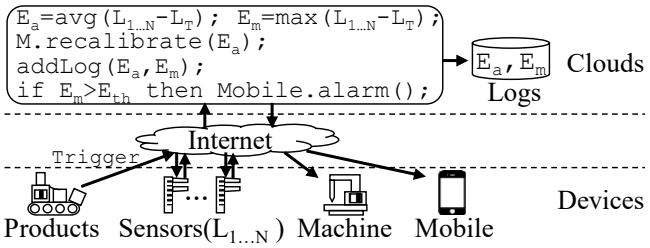


Figure 1: A smart factory service example

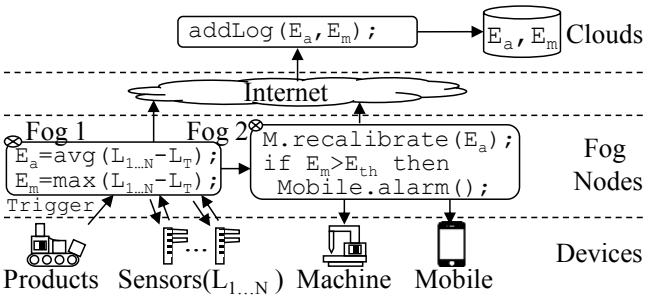


Figure 2: Fog computing of the smart factory example in Figure 1

Fog computing [2, 33, 34, 37, 38, 40, 41] can be a promising alternative to cloud computing that allows fog nodes near user environments to provide parts of an IoT service. Since the IoT devices communicate with nearby fog nodes instead of a distant cloud server, the fog computing can tolerate unpredictable latency of communications through the Internet and support stable QoS especially for critical paths from sensors to latency sensitive actuators. Moreover, since fog nodes execute parts of the IoT services and directly communicate with the devices, the fog computing can offload computation and communication overheads from the cloud onto fog nodes, reducing the centralized burdens of the cloud servers.

However, writing an IoT program for fog computing requires a huge amount of programmers' efforts. Programmers should analyze the program to find fog-executable codes and their communication costs, and manually partition the program into multiple sub-programs for each fog node. To reduce the programming burden, recent papers propose programming models for fog computing [7, 8, 14, 27, 35], but they require the programmers to explicitly describe the fog-executable codes or data flows in the program. Therefore, the programmers still need to manually analyze their IoT programs to exploit fog nodes. Moreover, none of them optimize a critical path to latency sensitive actuators.

This work proposes Spinal Code, a new compiler-runtime framework that automatically partitions a cloud-centric IoT program into multiple distributed sub-programs running over the cloud and fog nodes. Given an IoT program with annotations about latency sensitive actuators, the Spinal Code runtime collects online profiling information such as response time of the latency sensitive actuators and communication costs between fogs, and dynamically invokes re-compilation of the cloud-centric program. The Spinal Code compiler automatically analyzes data flows and critical paths of the program, and divides the critical paths into multiple sub-programs with minimal communications. Then, the compiler allocates non-critical instructions into the sub-programs with minimal communications after the critical instructions. Finally, the runtime deploys the partitioned programs to the cloud server and fog nodes.

Since most cloud-centric IoT frameworks [15, 26, 30] are closed source, this work implements a SmartThings-like cloud-centric IoT framework and the Spinal Code framework on top of the LLVM C++ compiler infrastructure [18]. For 9 cloud-centric IoT programs in 4 IoT services such as healthcare, smart home, smart building and smart factory, the Spinal Code framework automatically generates programs for fog computing only with 2 annotations at most, and reduces response time of the latency sensitive actuators and communication amount of the cloud by 44.3% and 79.9% on average respectively.

The contributions of this paper are:

- The Spinal Code compiler-runtime cooperative framework that automatically generates and deploys sub-programs for fog computing from cloud-centric programs
- The Spinal Code compiler that automatically analyzes data flows of IoT programs with expected communication overheads
- The critical path-aware partitioning that reduces response time to latency sensitive actuators while optimizing communication among fog nodes and the cloud

2 MOTIVATION

As an IoT service directly interacts with the physical world, response time is one of the crucial features that affect QoS. For example, since machine recalibration affects production quality and throughput of the factory in the smart factory example in Figure 1, some operations in IoT services are latency sensitive. Thus, it is important for IoT frameworks to reduce response times, especially for the latency sensitive operations in order to increase the QoS.

However, since many IoT frameworks [15, 26, 30] are cloud-centric as Figure 3(a), supporting a stable and fast response time is challenging due to unpredictable Internet communication delays and centralized computation and communication overheads on cloud servers. Since all the devices are connected to a cloud server, and the server executes a whole IoT service program, it is easy for programmers to write an IoT service program and manage device connectivities in the cloud-centric frameworks. However, Network condition between the cloud and the devices is hardly predictable due to a large physical distance between them and unstable Internet communication latency. Moreover, since the cloud server is solely responsible for executing IoT programs and managing numerous connected devices, this centralized processing gives large computation and communication burdens to the cloud server. Therefore, the cloud-centric IoT frameworks have difficulty in assuring a fast and stable response time for users, degrading QoS.

Fog computing [2, 33, 34, 37, 38, 40, 41] can be a good alternative system model to the cloud computing. As Figure 3(b) illustrates, in the fog computing environments, fog nodes are located between the cloud server and IoT devices, and execute parts of the programs.

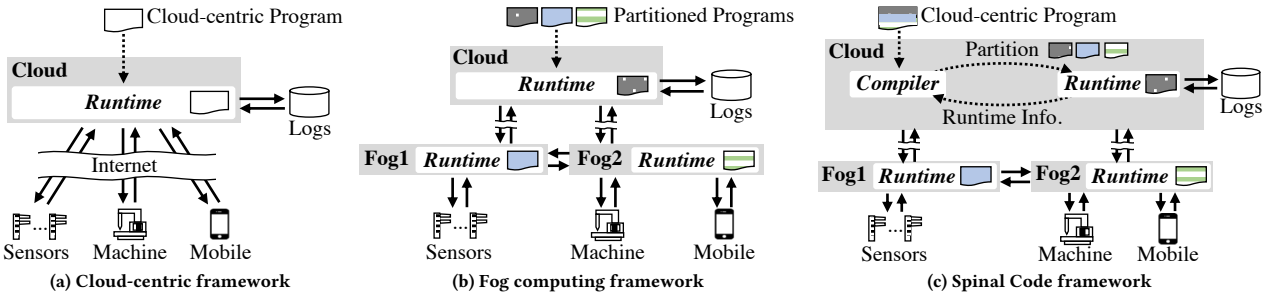


Figure 3: Different structures of IoT frameworks

Table 1: Comparison of fog computing frameworks

Framework	Data flow analysis	Partitioning	Critical path awareness
Mobile Fog [14]	Manual	Manual	×
Giang et al. [7, 8]	Manual	Manual	×
Wishbone [27]	Manual	Automatic	×
Kairos [12]	Manual	Automatic	×
Szydlo et al. [35]	Manual	Automatic	×
Spinal Code	Automatic	Automatic	✓

The fog computing efficiently reduces the response time by distributing parts of programs to fog nodes that are located near end devices. Figure 2 briefly shows that how the fog computing reduces the response time and the overheads of the cloud server for the smart factory example in Figure 1. A fog node `Fog 1` calculates average and maximum errors, and directly sends the errors to `Fog 2` that recalibrates the machine. Since devices communicate with nearby fog nodes instead of the distant cloud server, there is no long and unpredictable Internet communication from sensing errors to recalibration, and response time becomes more stable and faster. Furthermore, the cloud server receives only average and maximum errors instead of all the sensor values ($L_{1...N}$), and executes only the logging operation, so fog computing reduces computation and communication overheads of the cloud server.

However, manually implementing fog computing requires a lot of programmers' efforts. First, programmers should decide how to partition an entire program into multiple sub-programs and where to locate each sub-program to fully exploit the fog nodes. For example, programmers divide the smart factory program in Figure 1 into multiple sub-programs like Figure 2, and deploy the sub-programs at appropriate locations. Programmers should also synchronize sub-programs for the cloud and the fog nodes like the communication between `Fog 1` and `Fog 2` in Figure 2. Moreover, since dynamic runtime conditions such as network bandwidth can affect optimal partitioning, it is difficult for manual partitioning to reflect runtime information on its partitions.

Moreover, fog computing should be aware of critical paths of IoT programs. In an IoT program, only parts of actuators are latency sensitive while the others not. In the smart factory example, the recalibration latency from the sensors to the machine directly affects manufacturing throughput while logging average and maximum errors not. Since fog nodes have limited computation power, fog nodes should execute critical paths first that affect response time of the latency sensitive actuators like Figure 2.

Recent works [7, 8, 14, 27, 35] try to reduce the programming burden with new programming models for fog computing. Mobile Fog [14] and Giang et al. [7, 8] propose APIs and a distributed data flow programming model respectively that allow programmers to specify fog-executable sub-programs and their communication. Although the APIs and the distributed data flow programming model simplify fog computing programming, programmers should write multiple sub-programs with explicit communication. Wishbone [27], Szydlo et al. [35] and Kairos [12] propose frameworks that automatically transform a single data flow program into multiple sub-programs. However, the frameworks still require programmers to analyze and describe data flows of their programs. Moreover, none of them optimizes a critical path to latency sensitive operations. Table 1 summarizes the features of the fog computing frameworks. Section 7 describes more existing works about programming models for fog computing and automatic code partitioning for heterogeneous environments.

3 OVERVIEW: SPINAL CODE FRAMEWORK

Spinal Code is a compiler-runtime cooperative framework that automatically partitions a cloud-centric IoT program into multiple distributed sub-programs running over the cloud and fog nodes. Figure 3 (c) illustrates the overall structure of the Spinal Code framework. The framework consists of the Spinal Code compiler that runs on only the cloud server and the Spinal Code runtime that runs on the cloud server and fog nodes.

The Spinal Code framework allows IoT service programmers to write their IoT programs as a single cloud-centric program without considering fog nodes. The Spinal Code framework uses a similar programming model with Samsung SmartThings [30] where all the devices are connected to a cloud server and represented as objects. A program can communicate data and commands with the devices by invoking member functions of their corresponding objects. When executing the program, the Spinal Code framework provides mappings between the language-level objects and the physical devices. For example, in the smart factory program of Figure 1, the `Mobile` object represents the mobile phone, and the program can send an alarm to the mobile phone by calling `Mobile.alarm()`.

Given the cloud-centric program, the Spinal Code compiler automatically analyzes the program and transforms the program

into multiple sub-programs for the cloud server and fog nodes. To guarantee the correct program order and the consistency among sub-programs across the cloud and fog nodes, the compiler analyzes dependence among instructions in the original program, and makes partitions with communication codes reflecting the dependence. Moreover, the compiler allows programmers to optionally give priority to latency sensitive operations, and makes the operations to precede in advance of other non-critical instructions in the partitioned codes. Here, the compiler exists only at the cloud server because the cloud server has the original cloud-centric program and enough computation power to support data flow analysis and distributed code generation for heterogeneous devices. Section 4 describes the Spinal Code compiler in detail.

The Spinal Code runtime collects performance metrics and manages deployment and communication of sub-programs that the Spinal Code compiler generates. While executing IoT programs, the cloud server and fog nodes silently collect performance metrics such as response time of the programs and network bandwidth and latency between the cloud and fog nodes. The cloud periodically decides whether to re-compile the original or partitioned programs based on the collected metrics, enabling the Spinal Code framework to reflect dynamically changing network environments. Moreover, the runtime copies partitioned sub-programs from the cloud to fog nodes, and executes the new sub-programs by supporting communication among the cloud and fogs. Finally, the runtime also manages device connectivity, and maps objects in IoT programs onto physical devices. Section 5 describes the Spinal Code runtime in detail.

4 SPINAL CODE COMPILER

This section describes how the Spinal Code compiler automatically partitions an original cloud-centric program considering critical paths of the program. Figure 4 illustrates the overall compilation process of the compiler. Given re-compilation requests from the Spinal Code runtime, the Spinal Code compiler takes a target program, device locations and performance metrics from online profiling such as response times and communication latencies as its inputs. With the inputs, the compiler marks device method calls with their anchor locations or criticality (Section 4.1). Then, the compiler analyzes dependences among instructions, estimates communication costs for each dependence edge, and generates weighted program dependence graph (PDG) (Section 4.2). Finally, the compiler makes efficient partitions based on the weighted PDG, and inserts communication code for the partitions (Section 4.3).

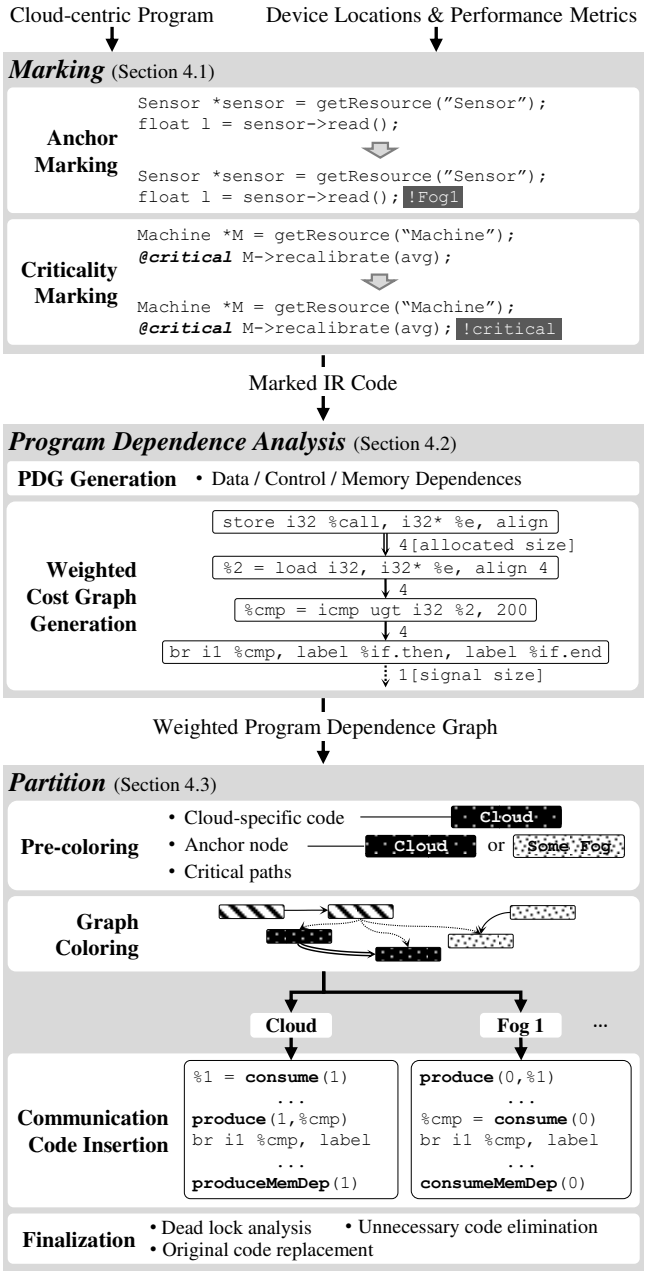


Figure 4: Overall compilation process of Spinal Code

4.1 Marking

In the marking step, the Spinal Code compiler first finds device method calls and `@critical` annotated method calls. First, given device locations from the runtime, the compiler marks the device method calls with the locations of their corresponding devices called `anchors`. An `anchor` can be a cloud server or a fog node if a method call should pass through the cloud or the fog. The compiler will use the anchor to reflect the communication delay from the device method. Second, if a method call is annotated as `@critical`, indicating a latency sensitive operation, the compiler also marks the call as time-critical. The compiler will analyze critical paths from required sensors to the anchors of the time-critical calls, and make partitions that execute the critical paths as soon as possible.

4.2 Weighted Program Dependence Analysis

With the marked program, the Spinal Code compiler analyzes program dependences and communication costs among instructions, and generates a weighted PDG. First, the compiler analyzes control and data dependences with several known algorithms [6, 22, 28, 29], and generates a PDG. To reduce false positive memory dependences

```
1  void verify() {
2   float Esum = 0.0;
3   float Emax = 0.0;
4   for(int i=0; i<N; i++) {
5    l = sensors[i].read();
6    e = abs(l-l_target);
7    Esum = Esum + e;
8    if (e > Emax)
9     Emax = e;
10  }
11  float Eavg = Esum / N;
12  @critical M.recal(Eavg);
13  addLog(Eavg, Emax);
14  if (Emax > Eth)
15   Mobile.alarm("Error");
16 }
```

(a) Pseudo code

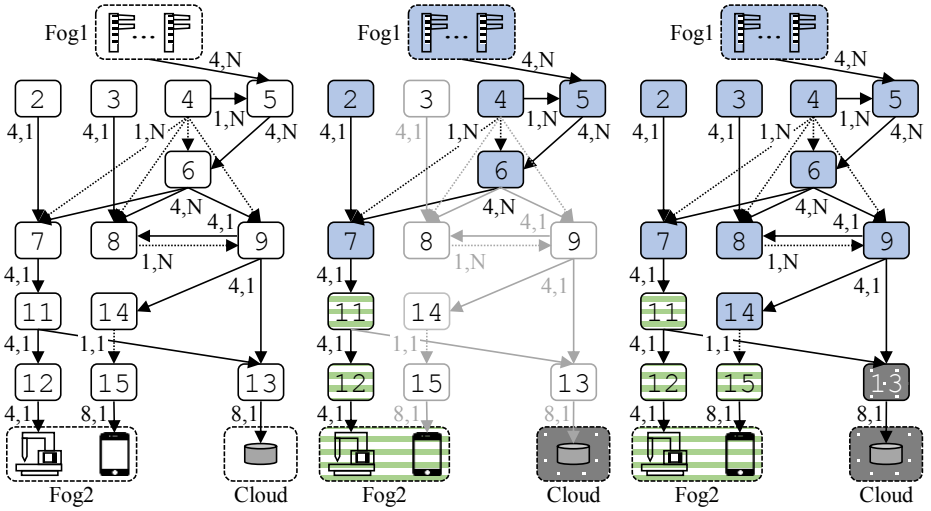


Figure 5: Pseudo code of the smart factory example in Figure 1 and its partitioning process

in the PDG, the compiler also executes flow-sensitive data dependence analysis [13] on memory operations, so creates a memory dependence edge considering a program execution flow.

Second, the compiler estimates communication costs among instructions and marks the costs on the edges of the PDG. A dependence edge between two instructions in the PDG means that communication is required if the two instructions are placed in different nodes like a cloud server and a fog node. Thus, the compiler estimates communication costs from the data size, the invocation counts of the edges and online profiling results such as communication latencies and bandwidths between nodes from the Spinal Code runtime. Figure 5(b) shows that how the compiler estimates and marks the communication costs of the smart factory program in Figure 5(a). Since the program repeats a loop for N times, the weights of the edges within the loop become a multiple of N.

4.3 Partitioning

From the weighted PDG, the Spinal Code compiler marks anchors on instructions with fixed locations (pre-coloring), assigns locations of the other instructions considering critical paths and communication costs (coloring), makes partitions with communication code insertion for partitioned edges (communication code insertion), and executes sanity checks for the partitioned code (finalization).

Pre-coloring: In the pre-coloring step, the compiler assigns *colors* to anchors and cloud-exclusive codes. A *color* is a unique ID assigned to the cloud and fog nodes. Through pre-coloring, the compiler can fix a location of particular code. Since anchors are already marked with their locations in the marking step (Section 4.1), the compiler just assigns the colors of the anchor locations to them. Here, cloud-exclusive codes are instructions such as I/O and locks that should be executed on a cloud server. For example, if two different programs share a variable and access it atomically, programmers should use atomic regions and locks in the original programs, and the compiler marks the atomic regions and locks as cloud-exclusive codes and executes them in the cloud server.

Coloring: The compiler colors instructions in two phases: (i) coloring instructions in critical paths and (ii) coloring the others. First, from the time-critical calls that are marked at the marking step, the compiler finds all the required instructions for the calls by recursively tracing sources of dependences to the time-critical calls, and constructs critical paths. Then, the compiler colors the critical paths with a coloring algorithm in Algorithm 1. Figure 5(c) shows the PDG after the compiler colors critical paths. After coloring the critical paths, the compiler colors the other instructions with the same algorithm. Figure 5(d) shows the colored PDG. In this way, the compiler makes partitions while giving higher priority to the critical paths.

The coloring algorithm in Algorithm 1 calculates expected `losses` when each color is assigned to an instruction, and chooses the color that minimizes the `loss` for each instruction. Here, a `loss` is the sum of expected data communication latencies from/to other locations reflecting network latencies and bandwidth. Here, the compiler allows system administrators to adjust a burden of the cloud by multiplying a weight to the loss value of the cloud.

Communication code insertion: After coloring instructions, the compiler makes partitions for the cloud and fogs, and inserts communication code to respect dependences between partitions. According to colors, the compiler assigns instructions into their corresponding partitions. If a source and a destination of an edge in the colored PDG have different colors, the compiler inserts communication codes such as `produce` and `consume` at the source and the destination codes. Figure 6 shows how the compiler inserts communication code to each partition. For example, in Figure 5(d), there is a data dependence edge from Instruction 11 (Colored as Fog 1) to Instruction 13 (Colored as Cloud). Therefore, the compiler inserts a `produce` function call into the partition of Fog 1 that

Algorithm 1: The Coloring Algorithm

Input: A cost graph $G = (V, E)$, the number of colors N, a cloud burden weight w, and the network bandwidth information between nodes
Output: A coloring table C that contains the assigned colors of all nodes in G

while *there exists an uncolored node in G* **do**
  $v \leftarrow$ Find an uncolored vertex with the largest number of colored neighbors in G ;
  color $\leftarrow 1$;
  minLoss $\leftarrow 0$;
  for $i \leftarrow 1$ **to** N **do**
    loss $\leftarrow 0$;
    if $i = \text{color}_{cloud}$ **then** $\lambda \leftarrow w$;
    else $\lambda \leftarrow 1$;
    foreach *edge* $e = (v, v')$ *or* (v', v) *where* v' *is a colored vertex in* V **do**
      $\text{color}' \leftarrow C[v']$;
      if $i \neq \text{color}'$ **then**
        $\mu \leftarrow$ the network bandwith between node_i and $\text{node}_{\text{color}'}$;
        $\text{loss} \leftarrow \text{loss} + \lambda \cdot \frac{e.weight}{\mu}$;
      end
    end
    if loss < minLoss **then**
      color $\leftarrow i$;
      minLoss $\leftarrow$ loss ;
    end
  end
  $C[v] \leftarrow$ color ;
end

sends `Eavg` to Cloud, and inserts a `consume` function call into the partition of Cloud that receives `Eavg` from Fog 1.

Finalization: In the final step, the compiler checks sanity of the partitioned sub-programs, and replaces the original code into the partitioned ones. The compiler analyzes the `produce` and `consume` pairs and control flows of partitioned programs, and checks whether deadlock and data races can occur across partitions. Then, the compiler replaces the original code into the new partitioned programs by sending a deploy request to the Spinal Code runtime.

5 SPINAL CODE RUNTIME

To support efficient execution of partitioned sub-programs, the Spinal Code runtime monitors online profiling results, dynamically invokes re-compilation of the original cloud-centric program, deploys the partitioned sub-programs, manages device connectivity and supports produce and consume communications among the partitioned programs. The Spinal Code runtime operates in two phases: pre-execution and program execution. Section 5.1 and Section 5.2 describe each phase in detail.

```
1  void verify_fog1() {
2   // lines 2-10 in the original code
3   produce(fog2, Esum);
4   produce(cloud, Emax);
5   bool isExec = Emax > Eth;
6   produce(fog2, isExec);
7  }
8
9  void verify_fog2() {
10  float Esum = consume(fog1);
11  float Eavg = Esum / N;
12  M.recal(Eavg);
13
14  produce(cloud, Eavg);
15  bool isExec = consume(fog1);
16  if(isExec)
17    Mobile.alarm("Error");
18 }
19
20 void verify_cloud() {
21  float Eavg = consume(fog2);
22  float Emax = consume(fog1);
23  addLog(Eavg, Emax);
24 }
```

Figure 6: Partitions of pseudo code in Figure 5

5.1 Pre-execution Phase

The pre-execution phase of the Spinal Code runtime includes online profiling, re-compilation invocation and deploying re-compiled partitioned programs to fog nodes.

Online profiling: To optimize a program reflecting dynamic execution environments, the runtime has a performance monitor that silently measures communication costs among the cloud and fog nodes and response time of each program. As in Figure 7, the performance monitor has two tables such as a communication cost table and a program table. The communication cost table contains network conditions such as network latencies among the cloud and the fog nodes, and the runtime periodically measures and updates the network conditions in the table. The program table maintains profiled response time of an original program and partitioned sub-programs for each program. In the program table of Figure 7, a highlighted entry represents the target program to be partitioned. For the program 'verify' that is already running as a cloud-centric version, the performance monitor can invoke re-compilation of the program to partition the program into multiple sub-programs (Step ⓪).

Re-compilation invocation: The re-compilation can be invoked directly by a user or by the performance monitor in the runtime based on the collected response time of each program. When re-compilation is invoked, the runtime passes device locations in the device manager and the communication cost table in the performance monitor to the Spinal Code compiler (Step ①). Then, the compiler generates newly partitioned sub-programs from the original cloud-centric program (Step ②). Here, the Spinal Code

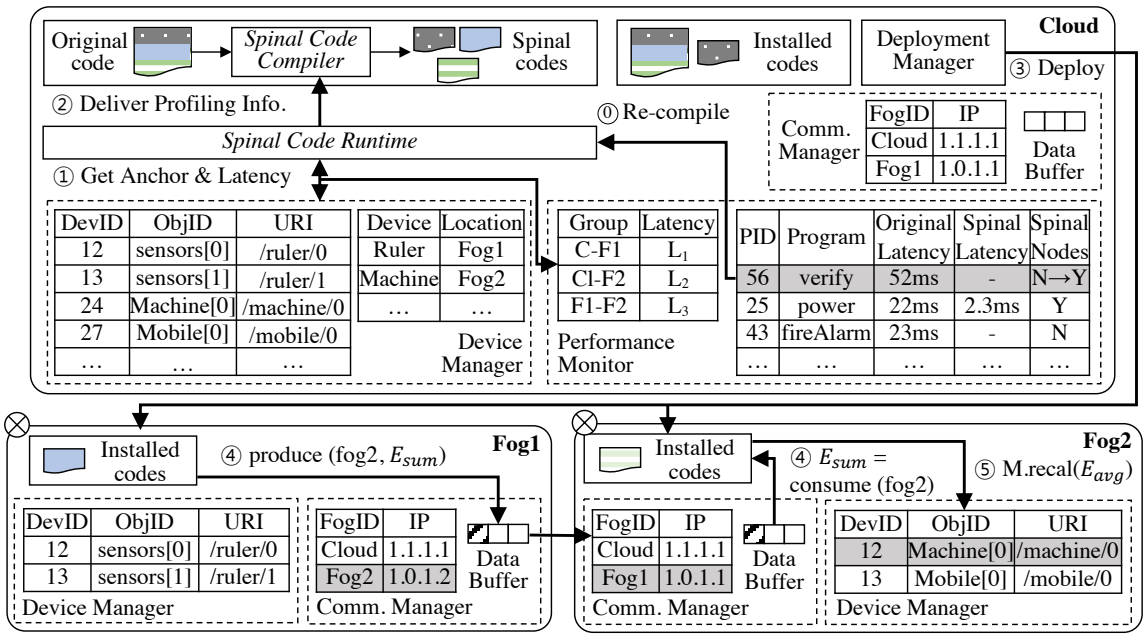


Figure 7: The overall structure of the Spinal Code runtime

compiler partitions a program with default runtime costs in case of the initial compilation.

Deployment: Given a deployment request after the re-compilation, the deployment manager in the cloud server installs the new partitioned sub-programs to their corresponding fog nodes (Step ③). In order to allow fog nodes to interact with the connected devices, the runtime copies device information entries from the cloud to the fog nodes before executing the programs.

5.2 Program Execution Phase

To enable correct and efficient execution of the partitioned sub-programs, the Spinal Code runtime mediates data and signal communication among fog nodes, and maintains connectivity information of IoT devices.

Communication management: To guarantee the correct execution of the partitioned sub-programs, the runtime mediates data and signal communication among the cloud and fog nodes. For example in Figure 6, `Fog 1` should produce `Esum` for Fog 2 to respect the data dependence between Instruction 7 and Instruction 11 (Step ④). When the produce function is called, the runtime stores the target data to the data buffer in the communication manager, and sends the data to the target fog. Here, to amortize the communication overheads, the runtime batches multiple data entries and sends them together.

Connectivity management: As the programming model of the Spinal Code framework represents a physical IoT device as an object, the runtime should hold mapping information between the object at the language level and the device Uniform Resource Identifier (URI) at the system level. The device managers in the cloud and fog nodes assign a unique device ID for each device, and keep the mapping information and the device ID at their device tables. When an IoT program registers a new device, the device manager creates a new device ID and stores the ID, the URI and its corresponding object in the program into the device table. Moreover, the device manager immediately synchronizes the device table with the cloud and the fog nodes to make the new device accessible by the others. Then, the cloud maintains all the IoT devices that are connected to the Spinal Code framework. When a sub-program tries to communicate with an IoT device, the runtime translates the communication request into a system level communication using the URI in the device table.

6 EVALUATION

This work implements a prototype Spinal Code framework on the LLVM compiler infrastructure [18]. This work designs and implements 9 cloud-centric IoT programs in 4 service domains: healthcare, smart home, smart building, and smart factory. Table 2 briefly summarizes the IoT programs in each service domain. All the services are implemented in the C++ programming language, extending the exisiting benchmarks used in Esperanto framework [19].

In the evaluation, the Spinal Code framework uses one cloud running on an Amazon EC2 instance and multiple fog nodes running on desktop servers. Also, to set up more realistic evaluation environment, the IoT programs exploit various third-party IoT devices ranging from small development boards with sensors to desktop servers. Those third-party devices are connected to either the cloud or some fog node according to prescribed connection topologies for each service. Table 3 describes the detailed device specifications for each IoT service. Moreover, sensors and actuators are connected to public wireless APs, and fog nodes are connected to different public gateways in different rooms in the same building.

Table 2: Evaluated program description (CP: the number of critical paths, F: the number of fogs)

Service	Program	CP	F	Description
Healthcare	`Sleep`	1	1	Track sleep state and change bulb's brightness
	`Heart`	1	1	Check heart rate and notify heart attack to a hospital
Smart Home	`Baby`	1	1	Monitor a baby and notify when baby is crying
	`Pet`	1	1	Track pet's location and notify when pet goes too far
	`Weather`	1	1	Check outdoor humidity and blink a bulb
Smart Building	`Security`	1	1	Capture and recognize face and send the image to mobile
	`Fire`	2	1	Recognize a fire and notify people and a hospital
Smart Factory	`Quality`	1	2	Check quality of products and recalibrate a machine
	`Power`	1	1	Meter power usage of a factory and turn off at emergency

6.1 Benchmark Description

This section describes the scenarios of 9 IoT programs used for evaluation of the Spinal Code framework. Table 2 briefly summarizes the IoT programs in each service domain.

Healthcare includes two programs, `Sleep` and `Heart`, which help users with health caring or health information tracking. `Sleep` periodically tracks sleep state of a user through a smart watch (Fog) and stores logs in a remote drive (Cloud). If the program recognizes a change in the user's sleep state, it adjusts the brightness of a bulb (Fog, Critical). Similarly, `Heart` periodically tracks heart rate of a user through a smart watch (Fog) and stores logs in a remote drive (Cloud). If the program detects abnormal heart rate, it sends a notification message to the nearest hospital server (Cloud, Critical). Its lines of code for original and manual versions are 122 and 188 respectively.

Smart Home includes three programs, `Baby`, `Pet`, and `Weather`, which provide various services in living environments. `Baby` implements a baby monitor service where an IP camera (Fog) takes a picture of a baby every minute and checks its situation. If an emergency situation occurs, the program sends an alarm to a mobile (Fog, Critical) and stores the picture in a remote drive (Cloud). `Pet` periodically tracks the location of a pet (Fog). If the program finds that the pet goes too far from home, it sends an alarm and a picture of the pet's nearby scenery to a mobile (Fog, Critical). `Weather` is a simple program that notifies a user when a weather board (Fog) senses rain by blinking a bulb (Fog, Critical) of the user. Its lines of code for original and manual versions are 124 and 158 respectively.

Smart Building includes two programs, `Security` and `Fire`, to make a building secure and safe. `Security` uses an IP camera (Fog) to capture an image of a visitor's face. Using face recognition, it detects an unregistered visitor and sends a picture of the visitor to a mobile (Fog, Critical). `Fire` provides a fire alarm service that checks whether a fire occurs using weather boards (Fog). If a fire is detected on one of the weather boards, it sends an alarm to a mobile (Fog, Critical) and notifies the nearest hospital server (Cloud, Critical). Unlike other programs, `Fire` has two critical paths towards alerting the mobile or notifying the hospital server.

Table 3: IoT device specification for each service

Service	Device	Specification
Common Devices	Cloud	AWS EC2 t2.micro instance
	Fog	Desktop Server A, B (Intel Xeon CPU E5-2637 v4, 64GB)
Healthcare	Hospital Server	Desktop Server C (Intel Core i7-6700, 16GB)
	Remote Drive	Desktop Server D (Intel Core i7-6700, 16GB)
	Bulb	Philips Hue
	Smart Watch	Xiaomi Mi Band
Smart Home	IP Camera	ODROID-XU4 with USB-CAM 720P (Samsung Exynos 5422, 2GB)
	Mobile	Samsung Galaxy S5 (Qualcomm Snapdragon 801, 3GB)
	Weather Board	ODROID-XU4 with Weather Board 2
	Remote Drive	Desktop Server D
	Bulb	Philips Hue
Smart Building	IP Camera	ODROID-XU4 with USB-CAM 720P
	Weather Board	ODROID-XU4 with Weather Board 2
	Hospital	Desktop Server C
	Mobile	Samsung Galaxy S5
Smart Factory	Power Meter	ODROID-XU4 with Smart Power 2
	Remote Drive	Desktop Server D
	Machine	ODROID-XU4
	Mobile	Samsung Galaxy S5
	Weather Board	ODROID-XU4 with Weather Board 2

Its lines of code for original and manual versions are 186 and 196 respectively.

Smart Factory includes two programs, `Quality` and `Power`, which automate factory management. `Quality` implements the smart factory example in Figure 1. As Section 1 describes, the program collects the measured values of sensors (Fog 1) and calculates the average and maximum errors. With the average error, it sends a recalibration request to a machine (Fog 2, Critical). Also, if the maximum error exceeds a threshold, it sends a message to a mobile (Fog 2). The program records the average and maximum errors in a remote drive (Cloud). `Power` manages power usage of a factory by measuring power usage with power meters (Fog) and logging it in a remote drive (Cloud). When the total power usage exceeds a threshold, it sends a power interruption command to a certain power meter (Critical). Its lines of code for original and manual versions are 113 and 161 respectively.

6.2 Response Time Analysis

This work measures the response time of 9 cloud-centric IoT programs for three models: 'Cloud', 'Spinal', and 'Manual' as Figure 8 shows. 'Cloud' is a model where the cloud executes all operations of an original cloud-centric program without partitioning. 'Spinal' is the model of the Spinal Code framework where the Spinal Code compiler automatically partitions the original program into multiple sub-programs, and the cloud and fog nodes collaboratively execute the entire program operations. 'Manual' is a model where a programmer manually partitions the original program for fog computing. The response time of a program is measured from the start to the end of the critical path in the program. Especially, `Fire` contains two critical paths, so the response time is measured for each critical path, `Fire(alarm)` and `Fire(noti)`.

On average, the 'Spinal' model reduces the response time by 44.3% compared with the 'Cloud' model. Except `Heart` and `Fire`,

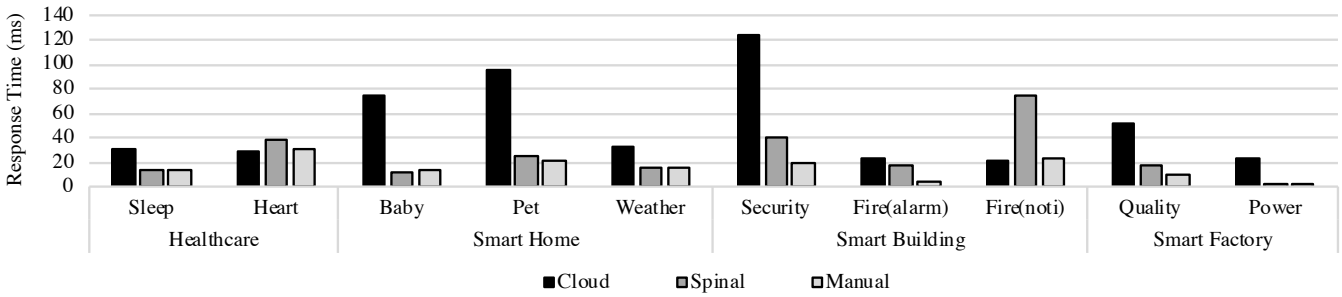
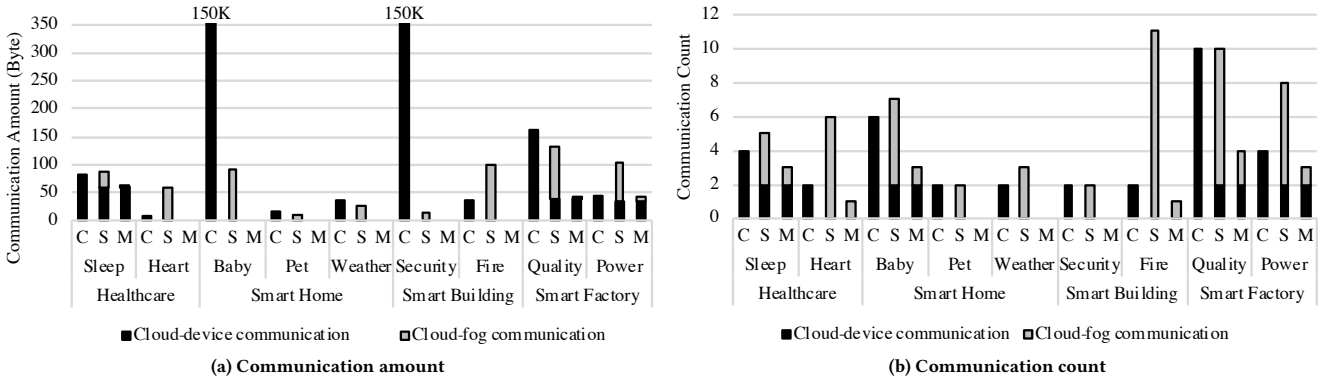
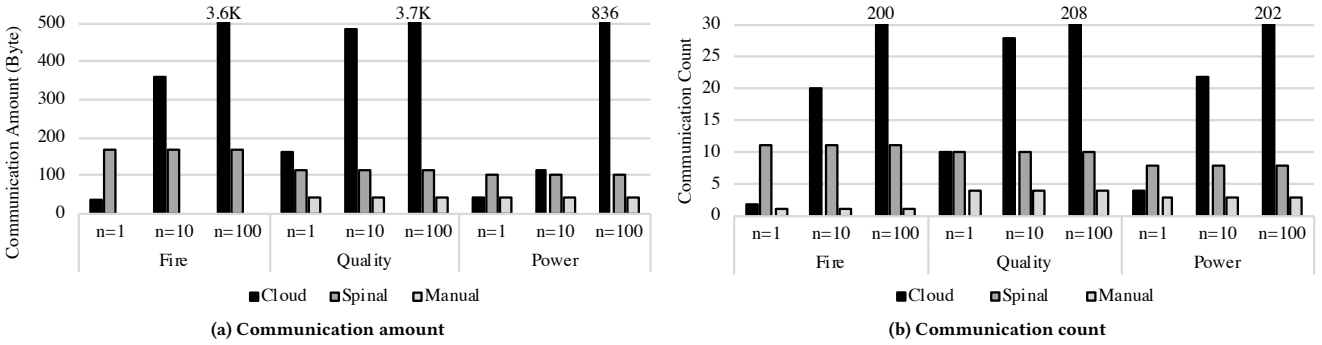


Figure 8: Response time of 9 IoT programs



(a) Communication amount **(b) Communication count**

Figure 9: Communication amount and count of 9 IoT programs



(a) Communication amount **(b) Communication count**

Figure 10: Communication amount and count over the different numbers of devices

the 'Spinal' model always outperforms the 'Cloud' model. In case of `Heart`, the program requires mandatory communication with the cloud, thus the Spinal Code framework cannot fully exploit the advantages of fog computing for `Heart`. This problem is also inevitable in manual partitioning, so 'Manual' shows worse response time than 'Cloud' for `Heart`. In case of `Fire`, the program has two different critical paths and the compiler more focuses on optimizing one of the critical paths. Therefore, a little performance degradation occurs on the other critical path.

For several programs such as `Quality`, the 'Spinal' model shows fairly worse response time than the 'Manual' model, which performs (almost) ideal partitioning. This is because the Spinal Code compiler conservatively analyzes memory dependence, causing unnecessary communication among partitions. For example, for the smart factory code in Figure 5(a), the compiler conservatively inserts communication code for the global variable `Eth` because the variable might be changed by others. On the other hand, a programmer knows that `Eth` is not changed, and does not insert communication code for `Eth` in the 'Manual' model.

In Baby, Spinal Code obtains better performance than manual partitioning because the 'Manual' model is manually partitioned in the C++ language level rather than the instruction level and the parts of non-critical instructions are executed before critical paths.

6.3 Communication Analysis

To evaluate how effectively the Spinal Code framework reduces the communication burden of the cloud server, this work also measures communication counts and amounts of the cloud server for the 9 IoT programs. Figure 9(a) shows the communication amount of the cloud server for the three models such as 'Cloud', 'Spinal', and 'Manual'. Compared with the 'Cloud' model, the Spinal Code framework reduces 79.9% of the communication amount of the cloud server on average. The Spinal Code framework reduces more than 99.9% of the communication amount of the cloud server for Baby and Security because the fog node sends an image file to a mobile without passing through the cloud server in the 'Spinal' model.

While the 'Manual' model always reduces the communication amount of the cloud server, the 'Spinal' model increases the communication amount for four programs such as for 4 programs such as Sleep, Heart, Fire, and Power. Unlike the 'Manual' model that executes only required communication, the Spinal Code compiler conservatively analyzes memory dependences and inserts unnecessary communication codes like the Eth case in Figure 5(a). With more precise static analysis, the compiler can further reduce the communication amount, and the 'Spinal' model can outperform the 'Cloud' model.

Figure 9(b) shows the communication count of the cloud server for the three models. Although the 'Manual' model always outperforms the 'Cloud' model, the 'Spinal' model communicates more than the 'Manual' model. The additional communication count of the 'Spinal' model consists of unnecessary signal communication and data communication because of conservative static analysis of the Spinal Code compiler. Similar to the communication amount, more precise static analysis can reduce the communication count between the cloud server and fog nodes.

6.4 Scalability Analysis

Although this work evaluates the Spinal Code framework with less than 10 sensors and actuators, a real-world IoT framework should support hundreds and thousands of sensors and actuators. This work evaluates scalability only for the smart building and smart factory services because they use a large number of sensors such as weather boards and power meters. By simulating Fire, Quality and Power with 10 and 100 sensors, this work measures communication amounts and counts of the cloud server. Figure 10 shows the communication amounts and counts of the cloud server when 1, 10 and 100 sensors are connected. Since the cloud server directly communicates with sensors in the 'Cloud' model, the communication amount and count of the 'Cloud' model linearly increase as the number of sensors increases. However, in the 'Spinal' and 'Manual' models, since all the sensors are connected to fog nodes, and the fog nodes pre-process the sensor values and deliver the pre-processed results to the cloud server, the number of connected sensors does not affect the communication amount and count of the server.

7 RELATED WORK

Programming models for fog computing: Fog computing [2, 33, 34, 37, 38, 40, 41] is an emerging system model where a cloud outsources distributed fog nodes locating closer to end-devices. Fog computing shares the concept with Cloudlet [31] and Tenet [9]; Spinal Code resembles concepts and enables near-user computation with fog nodes. However, since the fog computing model requires laborious programming for heterogeneous and distributed devices, existing works [7, 8, 14, 21, 35] propose programming models for fog computing to reduce the programming burden.

Mobile Fog [14] is a high-level programming model that provides hierarchical abstraction of devices in a platform. By splitting or merging the geospatial coverage of an overloaded process, Mobile Fog supports dynamic workload scaling across devices. However, Mobile Fog forces programmers to describe fog-executable codes through their APIs, so programmers should analyze and write fog-executable parts of the program. On the other hand, Spinal Code automatically analyzes and extracts fog-executable codes from the program, thus fully reducing the programming burden.

Distributed data-flow models [7, 8, 27, 35] enable integrated programming of heterogeneous devices where each device takes part of a sub-flow in the entire service flow. Especially, these models are suitable for IoT applications, the primary applicable domain of fog computing. Some models [7, 8] require programmers to specify which device will run on which sub-flow, but Wishbone [27], Kairos [12] and Szydlo et al. [35] propose automatic decomposition of a service flow to move processing from the cloud to the fog nodes. However, in the data-flow models, programmers still have to specify data flows among nodes. Unlike these models, Spinal Code shifts the analysis and programming burden to the compiler, which finds data dependence through static analysis and automatically partitions in view of communication cost. Moreover, none of them optimizes a critical path to latency sensitive actuations.

Microsoft Azure IoT Edge [25] is a framework that provides programming APIs for modules to be cooperated with Microsoft Azure IoT Hub. Azure IoT Hub allows to offload some parts of a service logic to an Azure IoT Edge device. With the Microsoft Azure IoT Edge framework, a service programmer can build an efficient IoT service that can save time and the amount of data transferred by filtering data sent to the cloud or deploying machine learning modules on the edge. However, the programmer needs to manually develop edge modules or possibly partition an original service logic to utilize Azure IoT Edge.

Automatic code partitioning for heterogeneous environments: Spinal Code automatically partitions an original cloud-centric program into multiple sub-programs to offload computation and communication overheads from cloud servers to fog nodes. While Spinal Code is a new framework that supports automatic code partitioning for fog computing, automatic code partitioning for heterogeneous environments has been widely discussed especially for mobile-cloud computing [1, 4, 5, 10, 11, 17, 19, 20, 27, 39]. The mobile-cloud offloading frameworks offload computation overheads from mobile devices to the cloud servers, and reduce energy

consumption and computation costs of mobile devices. However, all the frameworks assume that a mobile device is connected to a single cloud server. On the other hand, Spinal Code offloads computation from a cloud server to multiple fog nodes that are located near multiple sensors and actuators, supporting multi-site automatic partitioning.

J-Orchestra [36] is an application partitioning system for the Java language. J-Orchestra offers a profiler and a classifier that provide information about interdependent relationships among program objects and classification of platform-specific codes. While J-Orchestra leaves decisions of allocating the objects to users, the Spinal Code framework determines appropriate locations of the objects by automatically generating a cost graph.

There exist several works [3, 16, 23, 24, 32] that model a fog computing system and try to solve a computation offloading problem with theoretical analysis such as game theory or queuing theory. By adjusting the models to fit the Spinal Code framework, this work can utilize the algorithms to improve the partitioning algorithm of the Spinal Code framework considering various factors such as energy consumption.

8 CONCLUSION

Spinal Code is a new compiler-runtime framework that fully automatically and efficiently transforms a cloud-centric program into multiple distributed sub-programs for near-user computation in fog nodes. Moreover, Spinal Code allows programmers to annotate latency sensitive operations in a program, optimizes the critical paths from required sensors to the operations when generating the sub-programs, and reduces response time of the operations. This work implements 9 IoT programs across 4 service domains: healthcare, smart home, smart building and smart factory, and demonstrates that Spinal Code successfully reduces 44.3% of response time and 79.9% of network traffics on the cloud compared with a cloud-centric model.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This research was supported by NRF-2017R1C1B3009332, NRF-2015M3C4A7065646, IITP-2017-0-00195 and IITP-2018-0-01392 through the National Research Foundation of Korea (NRF) and the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. This research was also partly supported by the Yonsei university research fund of 2018.

REFERENCES

[1] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*.

[3] X. Chen, L. Jiao, W. Li, and X. Fu. 2016. Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing. *IEEE/ACM Transactions on Networking* (2016).

[4] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.

[5] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*.

[6] Ron Cytron, Jeanne Ferrante, and V. Sarkar. 1990. Compact Representations for Control Dependence. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*.

[7] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. 2015. Developing IoT applications in the fog: a distributed dataflow approach. In *Proceedings of 2015 International Conference on the Internet of Things (IOT '15)*.

[8] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M Leung. 2015. Distributed Data Flow: A Programming Model for the Crowdsourced Internet of Things. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference (Middleware Doct Symposium '15)*.

[9] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. 2006. The Tenet Architecture for Tiered Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*.

[10] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*.

[11] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[12] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming Wireless Sensor Networks Using Kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*.

[13] B. Hardekopf and C. Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*.

[14] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. 2013. Mobile Fog: A Programming Model for Large-scale Applications on the Internet of Things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing (MCC '13)*.

[15] IFTTT 2018. https://ifttt.com.

[16] Ajay Kattepur, Harshit Dohare, Visali Mushunuri, Hemant Kumar Rath, and Anantha Simha. 2016. Resource Constrained Offloading in Fog Computing. In *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets (MECC '16)*.

[17] B. Kim, S. Heo, G. Lee, S. Park, H. Kim, and J. Kim. 2016. Heterogeneous Distributed Shared Memory for Lightweight Internet of Things Devices. *IEEE Micro* 36, 6 (2016), 16–24.

[18] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*.

[19] Gyeongmin Lee, Seonyeong Heo, Bongjun Kim, Jong Kim, and Hanjun Kim. 2017. Integrated IoT Programming with Selective Abstraction. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017)*.

[20] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. 2015. Architecture-aware Automatic Computation Offload for Native Applications. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[21] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr. 2017. Coding for Distributed Fog Computing. *IEEE Communications Magazine* (2017).

[22] Yuan Lin and David Padua. 2000. Compiler Analysis of Irregular Memory Accesses. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*.

[23] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi. 2017. Multi-objective Optimization for Computation Offloading in Fog Computing. *IEEE Internet of Things Journal* (2017).

[24] X. Meng, W. Wang, and Z. Zhang. 2017. Delay-Constrained Hybrid Computation Offloading With Cloud and Fog Computing. *IEEE Access* (2017).

[25] Microsoft Azure IoT 2018. https://azure.microsoft.com/en-us/.

[26] Microsoft Flow 2018. https://flow.microsoft.com.

[27] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*.

[28] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*.

[29] Radu Rugina and Martin Rinard. 2000. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the ACM SIGPLAN*

2000 Conference on Programming Language Design and Implementation (PLDI '00).

[30] Samsung SmartThings 2018. http://www.smartthings.com.

[31] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* (2009).

[32] K. Sinha and M. Kulkarni. 2011. Techniques for Fine-Grained, Multi-site Computation Offloading. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

[33] I. Stojmenovic. 2014. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *2014 Australasian Telecommunication Networks and Applications Conference (ATNAC)*.

[34] I. Stojmenovic and S. Wen. 2014. The Fog computing paradigm: Scenarios and security issues. In *2014 Federated Conference on Computer Science and Information Systems*.

[35] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. 2017. Flow-Based Programming for IoT Leveraging Fog Computing. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*.

[36] Eli Tilevich and Yannis Smaragdakis. 2002. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*.

[37] Luis M. Vaquero and Luis Rodero-Merino. 2014. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *SIGCOMM Comput. Commun. Rev.* (2014).

[38] P. Varshney and Y. Simmhan. 2017. Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*.

[39] Cheng Wang and Zhiyuan Li. 2004. Parametric Analysis for Adaptive Computation Offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*.

[40] S. Yi, Z. Hao, Z. Qin, and Q. Li. 2015. Fog Computing: Platform and Applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*.

[41] Shanhe Yi, Cheng Li, and Qun Li. 2015. A Survey of Fog Computing: Concepts, Applications and Issues. In *Proceedings of the 2015 Workshop on Mobile Big Data (Mobidata '15)*.