Integrated IoT Programming with Selective Abstraction

Gyeongmin Lee Seonyeong Heo Bongjun Kim Jong Kim Hanjun Kim
POSTECH, Republic of Korea
{paina, heosy, bong90, jkim, hanjun}@postech.ac.kr

Abstract

The explosion of networked devices has driven a new computing environment called the Internet of Things (IoT), enabling various services such as home automation and health monitoring. Despite the promising applicability of the IoT, developing an IoT service is challenging for programmers, because the programmers should integrate multiple programmable devices and heterogeneous third-party devices. Recent works have proposed integrated programming platforms, but they either require device-specific implementation for third-party devices without any device abstraction, or abstract all the devices to the standard interfaces requiring unnecessary abstraction of programmable devices. To integrate IoT devices with selective abstraction, this work revisits the object oriented programming (OOP) model, and proposes a new language extension and its compiler-runtime framework, called Esperanto. With three annotations that map each object to its corresponding IoT device, the Esperanto language allows programmers to integrate multiple programmable devices into one OOP program and to abstract similar third-party devices into their common ancestor classes. Given the annotations, the Esperanto compiler automatically partitions the integrated program into multiple sub-programs for each programmable IoT device, and inserts communication and synchronization code. Moreover, for the ancestor classes, the Esperanto runtime dynamically identifies connected third-party devices, and links their corresponding descendent objects. Compared to an existing approach on the integrated IoT programming, Esperanto requires 33.3% fewer lines of code to implement 5 IoT services, and reduces their response time by 44.8% on average.

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

Keywords Internet of Things, IoT, Integrated programming model, Esperanto

1. Introduction

According to Gartner, 26 billion devices will be interconnected through the Internet by 2020, inaugurating a new era of the Internet of Things (IoT) [13]. In the era of the IoT, various networked devices collaboratively provide a service with their specific features. Figure 1 illustrates a baby monitor application as an IoT service example. A custom embedded board with an IP camera monitors a baby. If the baby cries, the IP camera notifies parents by blinking smartbulbs and sending a message to mobile phones. The parents can watch the baby by connecting to the IP camera with their mobiles, and keep some video clips to the cloud server. Like the baby monitor example, the IoT environment enables new promising services by integrating various networked devices ranging from sensors and actuators like IP cameras and smartbulbs to smartphones and cloud servers.

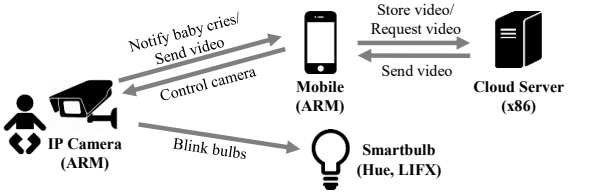


Figure 1. An IoT service example: Baby monitor

Despite the promising applicability of the IoT, building an IoT application is challenging for programmers due to multiple programmable devices and heterogeneous APIs of third-party devices. To integrate multiple programmable IoT devices, programmers should write multiple disjoint sub-programs for each device, and explicitly manage communication among the devices. Moreover, since different vendors adopt different APIs for their devices, programmers should add device-specific implementation for similar third-party devices like Hue and LIFX smartbulbs. Thus, to simplify IoT programming, an IoT programming platform should integrate multiple programmable IoT devices while abstracting various APIs of similar third-party devices into common APIs.

Though recent works [4, 11, 14, 15, 19, 22, 24, 30, 31, 33–35] have proposed various IoT platforms that integrate IoT devices, but none of them can fully solve the challenges. Protocol integration platforms [11, 19, 35] unify communication protocols across IoT devices, but they still require programmers to write and synchronize multiple sub-programs without a holistic view of an IoT application. Device integration platforms [4, 15, 30, 33, 36] integrates multiple IoT devices with a holistic view of an application. However, they either require device-specific implementation for third-party devices without abstracting heterogeneous APIs into a common API [4, 15, 30], or require unnecessary abstraction of programmable devices to the standard interfaces considering all the devices as third-party devices [33, 36].

To integrate multiple programmable IoT devices and selectively abstract heterogeneous APIs of third-party devices, this work revisits the object oriented programming (OOP) model, and proposes a new language extension and its compiler-runtime framework, called Esperanto. With three annotations that express the correspondence between an object and a thing, the Esperanto language allows programmers to write only one object oriented program for multiple programmable IoT devices. Moreover, exploiting

LCTES'17, June 21–22, 2017, Barcelona, Spain
ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081031

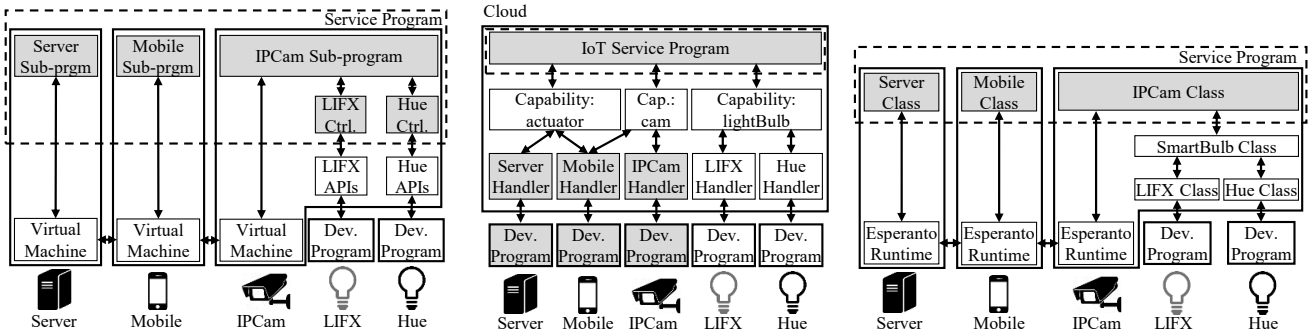


Figure 2. The overall structures of the baby monitor program (Figure 1) in different IoT frameworks. Here, the baby monitor program executes customized sub-programs for multiple programmable devices such as the IP camera, the server and the mobile, exploiting third-party devices such as Hue and LIFX smartbulbs. Gray boxes are programs that programmers need to write, and dashed boxes are integrated programming views that each framework provides.

the inheritance of the OOP model, the Esperanto language supports selective abstraction of third-party devices into a single ancestor class. Given the annotations, the Esperanto compiler automatically partitions the integrated object oriented program into multiple sub-programs for each IoT device, and inserts communication and synchronization code. The Esperanto runtime dynamically binds the ancestor class to its descendant objects reflecting connected third-party devices at run-time.

This work implements the Esperanto compiler-runtime framework on top of the LLVM C++ compiler infrastructure [25]. With a video demo [12], this work shows that the Esperanto compiler framework correctly transforms the integrated baby monitor program in the Esperanto language into multiple IoT sub-programs for each device. To further evaluate the Esperanto language and the compiler framework, this work also implements 14 events of 5 IoT services such as baby monitor, fitness tracking, taxi application, heart attack detection, and fire alarm application in the Esperanto language. Compared to an existing device integration approach, the Esperanto language and compiler framework require 33.3% fewer lines of code with 44.8% shorter response time on average.

The contributions of this paper are:

- The syntax and semantics of the Esperanto language that allows programmers to write one integrated program for multiple programmable IoT devices and to selectively abstract heterogeneous APIs of third-party devices into common interfaces
- The Esperanto compiler that automatically partitions an integrated IoT program into multiple sub-programs for programmable devices
- The Esperanto runtime that dynamically identifies connected IoT devices and links the common interfaces into their corresponding third-party devices

2. Motivation

While the IoT integrates heterogeneous networked devices and provides new services, developing an IoT program requires a huge amount of programmers' efforts due to the distributed programmable devices and heterogeneous APIs of third-party devices.

Distributed programmable devices: In the IoT environments, multiple different devices cooperatively execute an IoT application. Since the devices are distributed and heterogeneous, programmers write multiple disjoint sub-programs for each device, and integrate the sub-programs into one service through explicit communication management. For example, to develop the baby monitor application in Figure 1, programmers should write different sub-programs for an IP camera, mobile phones, bulbs and a server with explicit communication codes for the notification and video clips.

System	Integrated Programming Model	Programmable Device Integration	Device Abstraction
Eclipse SmartHome [11]	×	✓	×
The Thing System [35]	×	✓	✓
IoTivity [19]	×	✓	✓
Dist. Node-RED [4, 15, 30]	✓	✓	×
SmartThings [33]	✓	×	✓
Esperanto [This paper]	✓	✓	✓

Table 1. Comparison of existing IoT programming approaches

Heterogeneous APIs of third-party devices: Since third-party IoT devices provide a vast range of features that one manufacturer cannot solely provide, exploiting third-party devices is inevitable to enrich the functionality of IoT services. However, different vendors adopt different APIs for the same type of devices, requiring device-specific implementation for each device and making IoT programming difficult. For example, though Hue and LIFX are smartbulbs with similar features, they support different communication protocols and APIs. As a result, programmers should write different binding codes to support Hue and LIFX as Figure 2(a) illustrates.

Recent works [4, 11, 14, 15, 19, 22, 24, 30, 31, 33–36] have proposed various IoT platforms that integrate heterogeneous IoT devices, but none of the platforms fully integrate programmable and third-party IoT devices as Table 1 shows.

Protocol integration platforms: Protocol integration platforms such as The Thing System [35], IoTivity [19] and Eclipse SmartHome [11] unify communication protocols across different IoT devices, so the heterogeneous IoT devices can communicate with each other in the standard protocols. However, the platforms do not integrate programming environments of each device, so programmers should develop multiple sub-programs of the devices without a holistic view of an application and explicitly manage communication between the devices.

Programmable device-centric integration: Programmable device-centric integration platforms [4, 14, 15, 30] integrate programming environments of multiple programmable IoT devices. As Figure 2(a) shows, while the platforms allow programmers to write an IoT application with a holistic view, they execute each sub-program of the application directly on their corresponding devices. However, although the platforms integrate multiple programmable

Challenges of IoT Programming		Esperanto
Device Integration	Device registration	Object creation
	Identification method	Memory address
	Communication	Method call
Device Abstraction	Common interface	Ancestor class
	Device-specific implementation	Descendant class

Table 2. Features of the Esperanto language that address the challenges in Section 2

devices, the existing device-centric integration platforms do not abstract heterogeneous APIs of third-party IoT devices, requiring programmers to develop multiple device-specific codes for various third-party devices like the gray boxes in Figure 2(a). Moreover, the platforms [4, 15, 30] require virtualization of underlying devices to seamlessly integrate heterogeneous architectures.

Device integration with abstraction (cloud-centric): Cloud-centric IoT platforms [33, 36] integrate IoT devices with device abstraction. With abstract interfaces instead of multiple vendor-customized interfaces, programmers can control diverse underlying devices like Figure 2(b). However, the existing platforms enforce the programmers to use standard interfaces for all the devices including custom programmable devices, making custom device management difficult. For example, as Figure 2(b) illustrates, to integrate programmable devices such as the IP camera, the mobile phone and the server, programmers should map the devices to some of the existing standard interfaces, and implement device handlers on the cloud and device programs on each programmable device. If a custom device includes a new feature that does not exist in the capability list, programmers should categorize the device into a general sensor or actuator like the mobile phone and the server in Figure 2(b). Moreover, since the custom device is categorized into general sensors and actuators, programmers need to write additional codes in the cloud to find the custom device from others.

Esperanto: Figure 2(c) illustrates the Esperanto compiler framework that this work newly proposes. The Esperanto framework integrates multiple programmable devices without abstraction like programmable device-centric integration platforms while selectively abstracting heterogeneous APIs of third-party devices into a common API. Therefore, with the Esperanto framework, programmers can write an IoT application without device-specific implementation of third-party devices and unnecessary abstraction of programmable devices.

3. Esperanto Language

The Esperanto language integrates multiple programmable IoT devices while selectively abstracting third-party devices. Section 3.1 describes the design principles of the language, Section 3.2 introduces syntax and semantics of the proposed Esperanto primitives, and Section 3.3 demonstrates how the language integrates multiple programmable devices and abstracts third-party devices with an IoT application example.

3.1 Design Principle

Simple but powerful programming language: To effectively reduce the burden of IoT programmers, the proposed language should be simple and easy for programmers to learn and use, and also be powerful enough to solve the challenges in Section 2. To achieve the goal, this work revisits the object oriented programming (OOP) model that most programmers are familiar with. In the OOP model, objects correspond to things in the real world, and a parent class abstracts its children classes. Exploiting the correspondence between objects and things and the inheritance feature of the OOP language, this work integrates multiple programmable devices and selectively abstracts heterogeneous APIs of third-party devices to their parent classes with a minimal extension of an existing OOP language, three new annotations in total.

Syntax of the Esperanto primitives	
Physical Device Declaration	`#pragma` **`EspDevDecl`**`(devID,main)`
Device-Object Mapping	`#pragma` **`EspDevice`**`(devID,(conditions))` `class className;`
Third-party Device Import	`#pragma` **`EspImport`**`(className,funcName)`
Runtime variable examples in `condition`	
`TYPE`	Device type (e.g. Server, Mobile, Bulb, Watch, ...)
`VENDOR`	Vendor of device
`MODEL`	Model name of device
`ARCH`	Processor architecture of device
`OS`	Operating system of device

Table 3. Esperanto syntax

Integrated programming with a single machine view: An IoT service is composed of things and communications among things. Similarly, an OOP program is composed of objects and describes interactions between objects. Based on the correspondence between objects and things, the Esperanto language integrates multiple sub-programs of IoT devices in an IoT service into a single integrated OOP program. As an OOP programmer manages multiple objects in a single OOP program, an Esperanto programmer manages multiple IoT devices with a single Esperanto program. According to the Esperanto primitives, the Esperanto compiler partitions the integrated program into multiple sub-programs, and inserts communication codes for method calls at different objects. Table 2 summarizes how the Esperanto language integrates multiple programmable devices.

Device abstraction: The concept of inheritance and polymorphism gives "is-a" relationships between objects and allows polymorphic behaviors while providing a common interface to objects. Exploiting this concept, the Esperanto language abstracts similar types of devices to have a common interface and binds device-specific implementation of the devices to the interface. For the abstract interface, the Esperanto compiler dynamically links its corresponding object implementation reflecting the execution environment. Table 2 summarizes how the Esperanto language abstracts heterogeneous APIs of third-party devices.

3.2 Syntax and Semantics

This work proposes the Esperanto language by extending the existing C++ language with three annotations for the easy and powerful integrated IoT programming with selective abstraction. Though the Esperanto language extends the C++ language, the proposed syntax and semantics are not tied to C++ because the proposed language does not require any C++ specific feature. Table 3 and Figure 3 show the three annotations and the Esperanto codes for the baby monitor example in Figure 1.

`EspDevDecl` declares a programmable device with its name (`devID`), constructor and destructor. The constructor and the destructor will be invoked at the beginning and the end of the sub-program of the device. For example, Figure 3 declares two programmable devices such as `Cam` (IP Camera) and `Phone` with their constructors and destructors (Lines 3-4). Here, `EspDevDecl` is only allowed for programmable devices because the Esperanto compiler generates sub-program binaries for the `EspDevDecl` annotated devices.

`EspDevice` maps its annotated class to the declared device. For example, the Esperanto programmer can install the `IPCamera`

```
1  /* *** BabyMonitor.h/cpp *** */
2  // Declare all the programmable devices
3  #pragma EspDevDecl(Cam, cam_ctor, cam_dtor)
4  #pragma EspDevDecl(Phone, m_ctor, m_dtor)
5
6  // Generate an import function for 3rd-party devices
7  #pragma EspImport(SmartBulb, getBulbs)
8
9  // Map IPCamera class to device Cam
10 #pragma EspDevice(Cam)
11 class IPCamera {
12 private:
13   void onBabyCry();
14 };
15
16 // Map Mobile class to device Phone
17 #pragma EspDevice(Phone)
18 class Mobile {
19 public:
20   void alarm(string msg);
21 };
22
23 IPCamera* cam;
24 List<Mobile*> m_list;
25 SmartBulb** bulbs;
26 int num_bulbs = 0;
27
28 // Work as main function of Cam device
29 void cam_ctor(){
30   cam = new IPCamera();
31   // Bind all the SmartBulb instances
32   bulbs = getBulbs(&num_bulbs);
33 }
34
35 void IPCamera::onBabyCry(){
36   for(size_t i=0;i<m_list.size();i++){
37     // Send a message through a function call
38     m_list[i]->alarm("Baby is crying");
39   }
40   // Device abstraction for SmartBulbs
41   for(size_t i=0;i<num_bulbs;i++) bulbs[i]->blink();
42 }
43
44 // A mobile phone registers itself to a m_list
45 void m_ctor(){
46   Mobile* m = new Mobile();
47   m_list.push_back(m);
48 }
```

Figure 3. Esperanto pseudo codes of the baby monitor application in Figure 1

class at the device `Cam` with the `EspDevice` annotation at line 10 in Figure 3. Here, the programmer needs to insert the annotation only for code that should be executed in the device. Given the annotation, the Esperanto compiler will automatically map non-annotated instructions to appropriate IoT devices based on performance estimation results.

`EspDevice` can optionally pass device conditions as an argument to specify its target physical device. The Esperanto compiler framework requires hardware description of each device such as its device type, vendor, model, architecture and operating system. The Esperanto runtime dynamically checks the description, and maps the object to the appropriate device. For example, Figure 4 shows that `Bulb` device programmers annotate their classes with `VENDOR` and `MODEL` runtime variables (Line 2 in Hue.h and 2 and 9 in LIFX.h). According to the annotated condition, the runtime maps `Hue`, `LIFX`, and `LIFXZ` classes to their corresponding physical devices such as Hue, LIFX and LIFXZ smartbulbs.

`EspImport` and its import function allow IoT application programs to exploit non-programmable third-party devices like Hue and LIFX bulbs. `EspImport` generates an import function that binds all the `className` devices (Line 7), and the import function returns `className` objects of all the connected devices (Line 32). Programmers can specify a certain type of devices by passing its corresponding class type as the `className` argument. For example, if a programmer uses `LIFX` instead of `SmartBulb` as the first argument, the import function returns connected `LIFX` and `LIFXZ` objects but does not return `Hue` objects.

`Runtime variables` are hardware and system information of connected devices such as their device type, vendor, architecture and operating system. With the runtime variables, programmers can specify a target device that an object is mapped on.

```
1  /* *** SmartBulb.h *** */
2  #pragma EspDevice(Bulb, TYPE==BULB)
3  class SmartBulb {
4  public:
5    virtual bool connect() = 0;
6    virtual void blink() = 0;
7  };
```

```
1  /* *** Hue.h *** */
2  #pragma EspDevice(Bulb, VENDOR==PHILIPS)
3  class Hue : SmartBulb {
4  public:
5    bool connect();
6    void blink();
7  private:
8    char bridge[23];
9  };
```

```
1  /* *** LIFX.h *** */
2  #pragma EspDevice(Bulb, VENDOR==LIFX)
3  class LIFX : SmartBulb {
4  public:
5    bool connect();
6    void blink();
7  };
8
9  #pragma EspDevice(Bulb, MODEL==LIFXZ)
10 class LIFXZ : LIFX{
11 public:
12   void blink(int idx);
13 };
```

Figure 4. SmartBulb class and its descendant classes

3.3 Device Integration and Abstraction

Exploiting the existing features of the OOP model with the proposed primitives, the Esperanto language effectively integrates multiple programmable devices and selectively abstracts heterogeneous APIs of third-party devices.

Multiple programmable device integration: Considering an object as a thing, the Esperanto programmers write an integrated object oriented program for one IoT service. The Esperanto language allows programmers to register a device by allocating an object instance. Since the Esperanto compiler framework provides a unified virtual address view across the IoT devices, each object instance has a unique memory address in an IoT service. By binding the memory address of an object instance with the IP address of the corresponding device, the Esperanto compiler can identify IoT devices for object instances, and transform a method call of other objects (remote call) like line 38 in Figure 3 to the communication between devices. For example, whenever a new mobile phone allocates a `Mobile` object at line 46 in Figure 3, the Esperanto compiler framework maps the memory address of the object instance to the IP address of the physical mobile phone. When an IP camera accesses a mobile object instance, the Esperanto runtime translates the memory address of the instance to the IP address of the mobile, and sends the message to the corresponding mobile device.

Device abstraction: With the inheritance feature, the Esperanto language abstracts heterogeneous APIs of third-party devices, supporting device agnosticism. For example, in Figure 4 the `SmartBulb` class defines an interface of a smartbulb device, and the `Hue` and `LIFX` classes that inherit the `SmartBulb` class

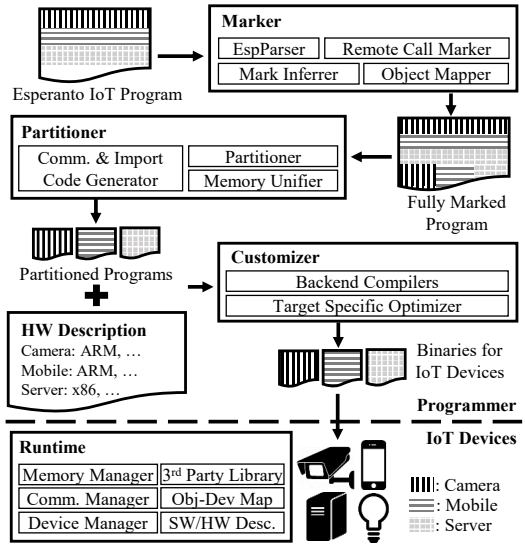


Figure 5. The overall structure of the Esperanto compiler

implement functionalities of each device. The Esperanto language allows programmers to write their applications with the interface of the `SmartBulb` class regardless of underlying devices (Line 41). The Esperanto runtime dynamically finds and returns appropriate objects reflecting execution environments for `getBulbs`, so the program invokes `blink` functions of `Hue` and `LIFX`.

4. Esperanto Compiler

To support the Esperanto language, this work proposes a new compiler-runtime cooperative framework. The Esperanto compiler partitions an integrated Esperanto program into multiple sub-programs for IoT devices (Section 4.1), and inserts device abstraction management code (Section 4.2). The Esperanto runtime manages communication among the IoT devices and device abstraction reflecting connected devices at run-time. Section 5 will describe details of the runtime system.

4.1 Device Integration Management

The Esperanto compiler transforms an integrated Esperanto program into multiple sub-programs for IoT devices. Figure 5 shows the overall structure of the Esperanto compiler framework. The Esperanto compiler consists of a marker, a partitioner and a customizer. The marker marks all the instructions in a program with their target devices. The partitioner divides the marked instructions into multiple sub-programs, and inserts communication instructions. The customizer keeps a set of back-end compilers, and customizes each sub-program for each IoT device.

Marker: The marker parses the Esperanto syntax and maps all the instructions in a program to their target devices. The parser recognizes the Esperanto syntax, and marks all the instructions in a device-annotated class as their target devices. For example, the parser marks the `IPCamera` class and its member functions as the `Cam` device.

The mark inferrer marks all the non-annotated instructions. Since a programmer does not annotate the instructions to a specific device, the instructions can be placed on any device. The mark inferrer estimates performance of possible target devices, and annotates the instructions as the optimal device. For further performance optimization, dynamic deployment [8, 10, 16, 37] can be applied to the non-annotated instructions.

```
1  /* *** Sub-program for IPCamera *** */
2  // Generate an import function for 3rd-party devices
3  //#pragma EspImport(SmartBulb, getBulbs)
4  SmartBulb** getBulbs(int &size) {
5    return _getSmartBulbs(size);
6  }
7  ...
8
9  // Work as main function of Cam device
10 // void cam_ctor(){
11 void main(){
12   cam = new IPCamera();
13   mapObjDev(cam, getIPaddress());
14
15   // Bind all the SmartBulb instances
16   bulbs = getBulbs(&num_bulbs);
17 }
18
19 void IPCamera::onBabyCry(){
20   for(size_t i=0;i<m_list.size();i++){
21     // Send a message through a function call
22     // m_list[i]->alarm("Baby is crying");
23     send(&m_list[i], ALARM, "Baby is crying");
24   }
25   // Device agnosticism for SmartBulbs
26   for(size_t i=0;i<num_bulbs;i++) bulbs[i]->blink();
27 }
```

```
1  /* *** Sub-program for Phone *** */
2  ...
3  void alarm(string msg);
4  ...
5  void communicationHandler(){
6    calledFcnID = recv();
7    switch(calledFcnID){
8      case ALARM:
9        arg = recv();
10       alarm(arg);
11   }
12 }
```

Figure 6. Partitioned sub-programs of the baby monitor code example in Figure 3

If the caller and callee instructions are marked as different devices, the remote call marker annotates the caller as a remote function call that requires network communication (Lines 22-23 in in Figure 6).

The object mapper inserts a call instruction of `mapObjDev` where an `EspDevDecl`-annotated object is allocated (Line 13 in Figure 6). The `mapObjDev` function registers the pointer of the newly allocated memory object and the current network IP address into the object-device map in the runtime. Using the object-device map, the runtime can find a correct target device from the memory object pointer. If an object is not device-annotated, the object mapper does not insert a call instruction of `mapObjDev`.

Partitioner: Given marked instructions, the partitioner automatically partitions the Esperanto program into multiple sub-programs for each IoT device, and inserts communication instructions. Figure 6 shows how the partitioner partitions the Esperanto program into multiple partitioned sub-programs. First, the partitioner replaces all the remote function call sites with the network communication `send` function calls. (Line 23 in Figure 6). The partitioner passes the memory address of callee objects as the first argument of the `send` function. Since `mapObjDev` registers the object address with its IP address, the Esperanto runtime can find the target IP address from the object address (Figure 8). Then, the partitioner generates multiple sub-programs and erases instructions that are not marked for each sub-program. Finally, the partitioner unifies heterogeneous memory layouts such as struct alignment, pointer size and endianness across different IoT devices [23].

Customizer: The customizer keeps a set of back-end compilers that the Esperanto compiler framework supports, and customizes

```
/* *** Bulb Library *** */
SmartBulb** __getSmartBulbs(size_t &size) {
  size = getNumDev(TYPE==BULB);
  SmartBulb** bulbs = malloc(...);
  for(i=1:size) {
    if(getRuntimeVar(i, VENDOR)==PHILIPS) {
      bulbs[i] = new Hue();
    } else if(getRuntimeVar(i, VENDOR)==LIFX) {
      if(getRuntimeVar(i, MODEL)==LIFXZ)
        bulbs[i] = new LIFXZ();
      else
        bulbs[i] = new LIFX();
    }
  }
  return bulbs;
}
...
LIFX** __getLIFX(size_t &size) { ... }
...
```

Figure 7. Abstraction management code example of SmartBulb classes in Figure 4

each sub-program for each IoT device. Given description of target machines as a compilation input, the Esperanto compiler framework picks appropriate back-end compilers from the set, and compiles the sub-programs with the back-end compilers. If there exists a code that works only for a specific hardware, the customizer optimizes the code with the back-end compiler.

4.2 Device Abstraction Management

The Esperanto compiler inserts abstraction management code for third-party devices and `EspImport` annotations, Figure 7 shows how the Esperanto compiler transforms an abstract interface of third-party devices such as the `SmartBulb` class into an import function (`__getSmartBulb`) that dynamically binds its child object reflecting connected devices at run-time. The compiler analyzes each `EspDevice` annotated class with the annotated conditions, and writes its class hierarchy in the SW description file (Figure 8). Based on the analyzed class hierarchy, the compiler generates an import function for each `EspDevice` annotated class. The import function dynamically reads HW descriptions of connected devices, compares the descriptions with conditions of the annotated class and its descendant classes, and allocates corresponding objects. The Esperanto compiler transforms the `EspImport` annotation in the application program into a wrapper function of the import function. As a result, programmers can use third-party devices in a device agnostic way.

5. Esperanto Runtime

The Esperanto runtime manages all the connected IoT devices at the user environments, and supports multiple programmable device integration and selective abstraction. The runtime consists of three modules such as a device manager, a communication manager, and a memory manager.

The device manager manages all the connected IoT devices and supports abstraction of third-party devices. When a new IoT device is connected to the user environment, the device manager collects HW description about the device such as IP address, device type, vendor, architecture, and operating system (Steps 1 to 3 in Figure 8). Here, the Esperanto runtime periodically checks connected IoT devices to find the third-party IoT devices that do not include the Esperanto runtime. When an Esperanto program searches devices with a runtime condition such as `TYPE==BULB`, the device manager searches corresponding devices from the HW description.

To support device abstraction, the device manager also keeps SW description that includes all the ancestor classes of a connected device. If an Esperanto program imports one of the ancestor

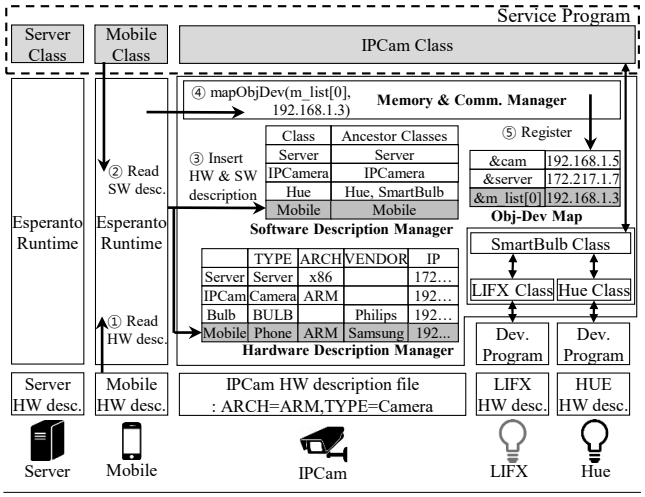


Figure 8. The overall structure of the Esperanto runtime

classes using `EspImport`, the runtime checks the SW description and returns the connected devices. Then, the Esperanto program creates appropriate objects for the devices with their HW description. For example, the `__getSmartBulbs` function in Figure 7 requests the Esperanto runtime to search third-party devices that have `SmartBulb` as its ancestor class. The Esperanto runtime finds descendent classes of `SmartBulb` from the SW description and returns their connected device lists from the HW description such as `Hue`, `LIFX` and `LIFXZ`. As Figure 7 illustrates, the `__getSmartBulb` function creates and returns corresponding descendent objects for the devices in the lists.

The communication manager maps objects in an Esperanto program to a physical device, and manages communication among objects. The Esperanto compiler inserts an object-device mapping function call (`mapObjDev`) for every `EspDevice`-annotated object allocation. When `mapObjDev` is invoked, the communication manager inserts a new element to the object-device map. Steps 4 and 5 in Figure 8 show how the Esperanto runtime maps a newly created device object to a connected physical device.

The Esperanto compiler transforms remote function call instructions on the objects into the communication function calls (`send`) passing the object memory pointer as an argument. If `send` is invoked, the device manager finds a corresponding IP address with the memory address of the passed-in object from the object-device map, and translates the memory address to the IP address. Then, the communication manager sends the message to the target IP address.

The memory manager in the Esperanto runtime manages memory coherence across IoT devices at run-time. Since the memory unifier in the Esperanto compiler unifies heterogeneous memory layouts across heterogeneous IoT devices as one layout, the memory manager in the runtime only needs to support memory coherence without worrying about memory translation. Therefore, the existing distributed shared memory systems [5, 32] can be used for the runtime.

6. Evaluation

This work implements the Esperanto framework on the LLVM compiler infrastructure [25]. To evaluate the Esperanto language, compiler and runtime, this work designs 5 IoT services in the Esperanto language, and deploys the services on heterogeneous IoT devices ranging from embedded systems such as ODROID-XU4 and ODROID-C0 [17] to a Samsung Galaxy S5 mobile phone and a desktop server. The mobile runs the Android 4.4.2 (KitKat),

Benchmark	Device	Specification
Baby Monitor	IP Camera	ODROID-XU4 with USB-CAM 720P (Samsung Exynos 5422, 2GB)
	Mobile	Samsung Galaxy S5 (Qualcomm Snapdragon 801, 3GB)
	Server	Desktop Server (Intel Core i7-6700, 16GB)
	Bulb	Philips Hue and LIFX
Fitness Tracking	Server	Desktop Server
	Mobile	Samsung Galaxy S5
	SmartScale	ODROID-C0 (Amlogic ARM Cortex-A5, 1GB)
	SmartBand	Gear Fit and Mi Band
Taxi App.	Server	Desktop Server
	Driver	ODROID-XU4
	Customer	Samsung Galaxy S5
Heart Attack	Gateway	Desktop Server
	Mobile	Samsung Galaxy S5
	Hospital	Desktop Server
	SmartBand	Gear Fit and Mi Band
Fire Alarm	Server	Desktop Server
	Mobile	Samsung Galaxy S5
	Thermometers	ODROID-XU4 with a weather board
	Bulb	Philips Hue and LIFX

Table 4. IoT Service and device specification

the desktop server runs Ubuntu 16.04, and the embedded systems run Ubuntu MATE 1.10.2. Here, to evaluate programmability on a device-specific custom hardware, this work installs a camera hardware module and a weather board on the embedded systems. Third-party devices such as smartbulbs (Hue and LIFX) and smart bands (Gear Fit and Mi Band) are used to evaluate the third-party device management in a device agnostic way. All the ODROID devices are wirelessly connected with 144Mbps maximum bandwidth (802.11n), and the mobile is connected with 844Mbps (802.11ac). Table 4 describes devices used in each service.

6.1 The Evaluated IoT Services

To evaluate the Esperanto language and framework, this work designs 5 IoT services such as baby monitor, fitness tracking, taxi application, heart attack detection and fire alarm application, which support 14 events in total. Table 5 briefly describes each event.

`Baby Monitor` monitors and records a baby's condition with an IP camera, mobile phones, a server and bulbs. An IP camera periodically captures an image of the baby and sends the image frame to the internal server via the mobile phones (Event `Cam`). If the IP camera notices the change in the baby's state, such as the situation that the baby starts to cry, the IP camera makes connected bulbs blink and sends an alarm message to the mobiles (Event `Alarm`).

`Fitness Tracking` is a personal health reporting system based on collected user data. A user registers a mobile device into a server to use the fitness tracking service (Event `Register`). The mobile gathers health information through several devices like a smart scale and a smart band (Event `Scale`). Then, the mobile updates the raw information on the server (Event `Update`). After analyzing the collected data, the server offers a personal health report on user's request (Event `Report`).

`Taxi Application` simulates an online taxi-rider matching, composed of a server, drivers and customers. In the initialization phase, a driver sends a sign-in request to the server, and receives a unique ID. (Event `Register`). Each driver periodically updates its location to the server with the unique ID (Event `Loc`). When a customer requests for a ride, the server searches nearby drivers and forwards the request to the drivers. Then, the server notifies the customer about the matching result, whether one driver accepts the request or none of the drivers is available (Event `Call`).

`Heart Attack` tracks a person's heart rate and detects an abnormal heartbeat state. This program consists of a gateway, hospitals, mobiles and smart bands. Each hospital registers its server and location to the gateway (Event `Register`). A mobile checks heart rates through a smart band regularly. If the heart rate goes too high or too low, the mobile sends a notification to the gateway, and the gateway notifies the nearest hospital in the hospital list about the emergency (Event `Alarm`). The hospital gets an authority to make a direct connection with the mobile, and the mobile sends required information to the hospital (Event `SendRate`).

`Fire Alarm` collects temperatures of a building and detects fire. Thermometers periodically measure temperatures in the building and send the temperatures to a server (Event `Info`). If the server detects a fire from the temperatures, the server notifies registered mobile phones and smartbulbs about the fire (Event `Alarm`).

Benchmark	Event	Description
Baby Monitor	Alarm	Notifies the baby cry to bulb and mobile
	Cam	Send an image frame to mobiles and upload the frame to the server
Fitness Tracking	Register	Register a mobile device into the server
	Scale	Send weight and body fat to mobiles
	Update	Update tracking and weight history
	Report	Generate an analysis report
Taxi App.	Register	Register a taxi driver into the server
	Loc	Update the location of a taxi
	Call	Call a nearby taxi
Heart Attack	Register	Register a hospital server into the server
	Alarm	Notify heart attack to a nearby hospital
	SendRate	Send heart rates to history
Fire Alarm	Info	Send temperatures to the server
	Alarm	Notify a fire alarm to mobiles

Table 5. Evaluated event description

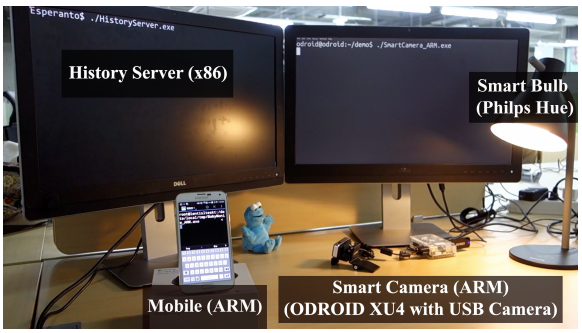


Figure 9. A snapshot of the video demo [12]

6.2 Programmability of Esperanto

To evaluate programmability of the Esperanto language, this work implements all the 5 services in Section 6.1 in the Esperanto language, and successfully compiles and executes the services. All the Esperanto programs are written in an integrated programming way with a single machine view. The Esperanto programs are written in a device agnostic way, and correctly support different third-party devices such as smartbulbs (Hue and LIFX) and smart bands (Gear Fit and Mi Band). In the fitness tracking and the heart attack programs, the Galaxy S5 and the smart bands have step counters and beat sensors, and provide the step counting and the heart rates checking features.

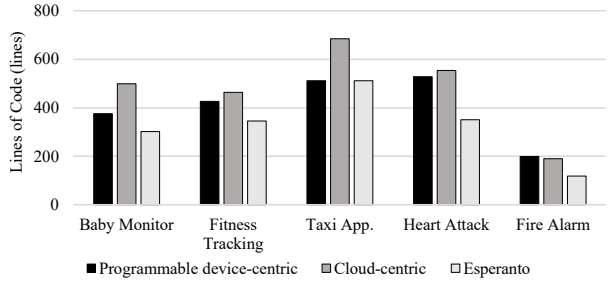


Figure 10. Lines of code of the IoT programs

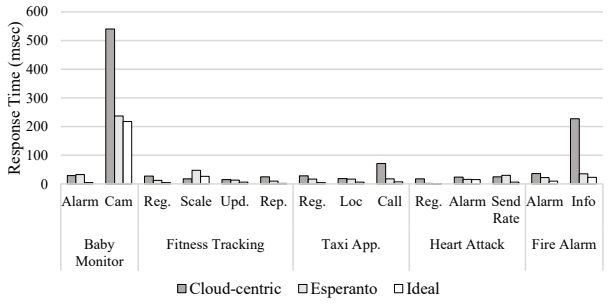
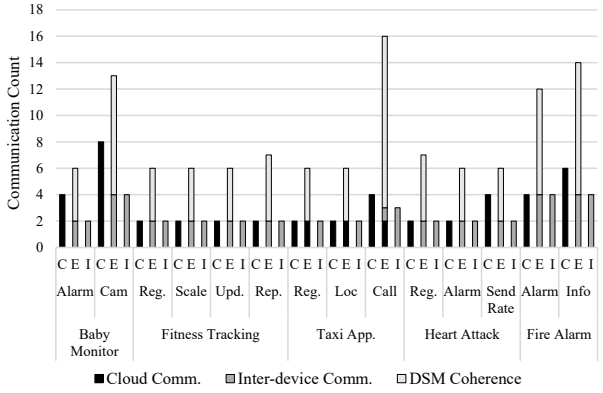


Figure 11. Response time of each event in the IoT programs

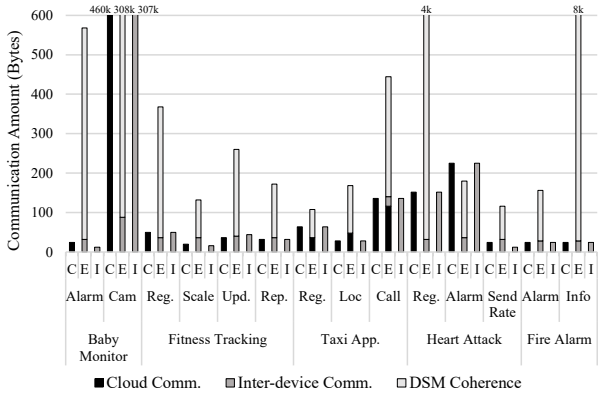
With a video demo [12], this work demonstrates that the Esperanto compiler framework correctly transforms the integrated baby monitor program in the Esperanto language into multiple IoT sub-programs for each device. The demo also shows that the baby monitor program successfully blinks a smartbulb Hue though the program is written in a device agnostic way. Figure 9 shows a snapshot of the video demo.

To compare the programmability of the Esperanto language with other IoT programming models, this work also implements the same services in programmable device-centric and cloud-centric integration approaches. The programmable device-centric approach integrates multiple programmable devices into one programming environment like Esperanto, but it does not support any device abstraction requiring device-specific implementation for each third-party devices. The cloud-centric integration approach integrates IoT devices with standard abstraction, so it requires unnecessary device handlers and device programs for programmable devices. Here, to eliminate performance effects from the underlying platforms, all the services are written in C++.

Figure 10 shows that Esperanto successfully simplifies programming IoT services. For the 5 IoT service, the average lines of the Esperanto programs are 23.8% and 33.3% shorter than lines of programmable device-centric and cloud-centric programs while the programs provide the same IoT services. The lines of code consist of third-party device management, device handlers and service algorithm. All the programs do not include any device registration, identification and explicit communication code among devices because the three approaches support integrated IoT programming with a holistic programming view. Since Esperanto supports device agnosticism without requiring device-customized code for third-party devices, the Esperanto programs have fewer lines of code than the programmable device-centric ones. Here, if the programs support more third-party devices, the programmable device-centric programs will have more lines of code because the lines of device-customized code is proportional to the number of third-party devices. Moreover, the Esperanto programs do not include



(a) Communication Count



(b) Communication Amount

Figure 12. Communication count and amount of each event in the IoT programs. C, E and I represent Cloud-centric, Esperanto and Ideal programs respectively.

device handlers for programmable devices, so the Esperanto programs have also fewer lines of code than the cloud-centric ones. As a result, Esperanto requires only service algorithm codes and effectively reduces the burden of programmers.

6.3 Response Time Analysis

This section evaluates performance of the cloud-centric integration approach and Esperanto. This work does not evaluate the programmable device-centric programs because the programmable device-centric programs and the Esperanto program have the same communication topology. Instead of the programmable device-centric programs, this work manually implements the ideal IoT services that do not include any platform overheads with optimal communication to analyze the maximum achievable performance of the services.

Figure 11 shows the response time of each event in the IoT programs. The results are the average response time of ten invocations per event. Since Esperanto allows IoT devices to directly communicate with each other without passing through the cloud server, Esperanto shows average 44.8% shorter response time than cloud-centric integration approach. Compared to the ideal programs, the Esperanto programs suffer from average 12.56 milliseconds and up to 28.14 milliseconds (`Baby Monitor Alarm`) latency overheads that are negligible enough for people not to recognize [29].

Since most of the response time consists of communication overheads among devices, to deeply analyze the communication

overheads, this work measures the communication counts and amounts of each event in the cloud-centric, Esperanto and ideal programs. Figure 12 (a) shows the Esperanto programs suffer from additional communication counts from the DSM coherence. It is because while the cloud-centric and ideal programs send a signal and data in one message, the Esperanto programs invoke remote functions in a message passing way and transfer shared data through the DSM system. The Esperanto DSM system minimizes the DSM coherence costs by adopting the lazy release consistency protocols [2, 20, 40]. In the protocol, an IoT device synchronizes shared data at the beginning and the end of the remote function invocation at caller and callee sides if there is no volatile variable nor misprefetched shared data. `Baby Monitor Cam`, `Taxi App. Call`, `Fire Alarm Alarm` and `Fire Alarm Info` events suffer from a relatively large number of DSM coherence counts because the events invoke remote functions in a nested way. For example, in the baby monitor program, an IP camera uploads an image to a server via a mobile phone, so the `Cam` event has more DSM coherence counts than others.

Figure 12 (b) shows the amount of communication among IoT devices. Since the Esperanto programs communicate shared data through the DSM system, the DSM coherence takes a large portion of the communication amount. Moreover, since the DSM coherence shares page tables and the object-device table map, registering an IoT device causes additional communication amount like `Reg.` in `Fitness Tracking`, `Taxi App.` and `Heart Attack.Alarm` in `Baby Monitor` also suffers from relatively high communication amount because the IP camera newly allocates a large amount of memory for an image, and synchronizes its page table with other devices.

Here, although the Esperanto programs have more communication counts and amount than the cloud-centric programs, the Esperanto programs are faster than the cloud-centric programs because the latency of inter-device communication is much shorter than the latency of cloud-device communication.

7. Related Work

Integrated Programming Support: To reduce the burden of IoT programmers, previous works [1, 6, 9, 14, 15, 18, 22, 24, 30, 31, 33, 34, 37] have proposed integrated programming models and platforms for distributed systems including IoT and wireless sensor networks. Like Esperanto, the programming models and platforms integrate programming environments of heterogeneous devices and provide a holistic view of an application to programmers.

SmartThings [33] is a programming platform for IoT that encapsulates a physical device as a composition of its capabilities. A capability is composed of commands (actions that things can do) and attributes (states that things can be). For example, a colorControl capability has a command that sets its color, and an attribute that represents its hue value. Therefore, the SmartThings capability model allows an application programmer to write an IoT service program in a device agnostic way by providing device management in capability granularity. However, since the framework only supports standard capabilities, it limits its applicability for custom programmable devices. Unlike SmartThings, Esperanto allows the programmer to directly manage programmable devices without the standardization of the devices.

Node-RED [30] and its extensions such as distributed Node-RED [15] and glue.things [24] provide a graphical user interface for integrated IoT programming. With the proposed systems, programmers can design an IoT system by graphically combining devices and specifying data flows. However, due to their lack of support for third-party integration in a device agnostic way, the programmers should implement different binding codes to control various third-party devices. On the other hand, this work supports device agnosticism by exploiting the inheritance and polymorphism features of the object oriented programming model.

Open-Remote [31] also offers graphic-based IoT programming environment for residential and commercial building automation. Open-Remote abstracts detailed low-level implementation by integrating a variety of protocols. Since Open-Remote allows any vendor or integrator to write plug-ins, the platform can easily extensible to support various third-party devices. However, its high level of abstraction leads to relatively limited programming flexibility.

nesC [14], Eon [34] and Mace [22] are extended C/C++ languages that support a holistic design of networked embedded systems. These languages provide abstraction with high level objects and connect components with their interfaces. Although they simplify complex event handling implementation, the languages have little consideration for device agnosticism since they do not target IoT environments.

Programming models on computation offloading [7, 21, 37, 38] allow programmers to integrate programs across distributed systems into one program. However, the proposed models manage only computation tasks that are independent from underlying devices. This work integrates heterogeneous IoT devices into one computing system, providing a holistic programming view.

Protocol Integration System: IoTivity [19], the Thing System [35] and Eclipse SmartHome [11] are frameworks that help to build IoT solutions. The frameworks offer integrated methods for discovery, connection and communication of things, whether the things use different protocols or standards. For example, IoTivity enables seamless device-to-device connectivity with high level APIs for resource discovery, data transmission and device management. The Thing System provides a core middleware called steward, and clients can get resources or communicate with other devices through the steward. Eclipse SmartHome provides an event bus and each device can send or receive events through the bus.

Although the systems successfully integrate heterogeneous devices, programmers still need to write explicit code to make devices interact each other. For example, using IoTitivty, programmers should make explicit API calls according to their interface to request resources. In case of the Thing System, clients should use WebSocket to send a specific message to the steward. Also, with Eclipse SmartHome, each device should define a publisher or subscriber to send or receive events. In contrast, Esperanto performs code instrumentation to provide communication abstraction. Therefore, with Esperanto, programmers only insert a function call rather than specify the entire send-receive procedure.

Distributed Shared Memory: To provide a shared memory view across distributed IoT devices, the Esperanto runtime relies on a heterogeneous distributed shared memory system [39]. Inspired by Native Offloader [26] and Reflex [28], the Esperanto compiler integrates heterogeneous memory layouts and marks synchronization points to reduce the coherence overheads. To reduce the communication amount and counts for the DSM coherence, the Esperanto runtime adopts a relaxed consistency model, called lazy release consistency [2, 20, 23, 40]. While the current implementation adopts only one consistency model for the whole Esperanto program, the Esperanto runtime can reduce the overheads further with multiple consistency protocols for user-annotated shared data [5]. Moreover, like Ivy [27] that is the first page-based software DSM system, the Esperanto runtime manages shared data coherence in a page level granularity. Since this work revisits the object oriented programming model, shared data coherence in an object granularity is possible like Orca [3].

8. Conclusion

This work proposes a new integrated IoT programming framework with selective abstraction, called Esperanto. Esperanto allows pro-

grammers to write one object oriented program for multiple programmable devices with three annotations that bind an object and a thing, and abstract similar third-party devices into their common ancestor classes with the inheritance feature of the OOP model. With multiple programmable device integration and selective abstraction, Esperanto reduces average 33.3% of lines of code and 44.8% of response time for 5 IoT services compared with a cloud-centric integrated programming approach.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank the CoreLab and HPC Lab for their support and feedback during this work. This work is supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1403-04.

References

[1] Google Web Toolkit. https://developers.google.com/web-toolkit/.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 1996.

[3] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE transactions on software engineering*, 1992.

[4] M. Blackstock and R. Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*, 2014.

[5] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 1995.

[6] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[7] H.-h. Chu, H. Song, C. Wong, S. Kurakake, and M. Katagiri. Roam, a seamless application framework. *Journal of Systems and Software*, 2004.

[8] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, 2011.

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, 2006.

[10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[11] Eclipse SmartHome. http://eclipse.org/smarthome.

[12] Esperanto Demo. https://youtu.be/BhrmU5KjX9M.

[13] Gartner. Gartner says the Internet of Things installed base will grow to 26 billion units by 2020. http://www.gartner.com/newsroom/id/2636073.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[15] N. K. Giang, M. Blackstock, R. Lea, and V. C. Leung. Developing IoT applications in the fog: a distributed dataflow approach. In *Proceedings of International Conference on the Internet of Things*, 2015.

[16] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[17] Hardkernel:ODROID. https://www.hardkernel.com.

[18] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[19] IoTivity Project. https://www.iotivity.org.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[21] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*. 2012.

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[23] B. Kim, S. Heo, G. Lee, S. Park, H. Kim, and J. Kim. Heterogeneous distributed shared memory for lightweight internet of things devices. *IEEE Micro*, 2016.

[24] R. Kleinfeld, S. Steglich, L. Radziwonowicz, and C. Doukas. Glue.Things: A mashup platform for wiring the Internet of Things with the Internet of Services. In *Proceedings of the 5th International Workshop on Web of Things*, 2014.

[25] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[26] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th IEEE/ACM International Symposium on Microarchitecture*, 2015.

[27] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 1989.

[28] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[29] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of AFIPS Fall Joint Computer Conference*, 1968.

[30] Node-RED. http://nodered.org.

[31] OpenRemote. http://www.openremote.org.

[32] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[33] SmartThings. http://www.smartthings.com.

[34] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, 2007.

[35] The Thing System. http://thethingsystem.com.

[36] xively by LogMein. http://www.xively.com.

[37] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[38] K. Zhang and S. Pande. Efficient application migration under compiler guidance. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[39] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[40] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996.