

TinyLink: A Holistic System for Rapid Development of IoT Applications

<http://tinylink.emnets.org>

Gaoyang Guan, Wei Dong*, Yi Gao, Kaibo Fu, Zhihao Cheng
College of Computer Science, Zhejiang University
Emails: {guangy, dongw, gaoy, fukb, chengzh}@emnets.org

ABSTRACT

Rapid development is essential for IoT (Internet of Things) application developers to obtain first-mover advantages and reduce the development cost. In this paper, we present TinyLink, a holistic system for rapid development of IoT applications. The key idea of TinyLink is to use a *top-down* approach for designing both the hardware and the software of IoT applications. Developers write the application code in a C-like language to specify the key logic of their applications, without dealing with the details of the specific hardware components. Taking the application code as input, TinyLink automatically generates the hardware configuration as well as the binary program executable on the target hardware platform. TinyLink provides unified APIs for applications to interact with the underlying hardware components. We implement TinyLink and evaluate its performance using real-world IoT applications. Results show that: (1) TinyLink achieves rapid development of IoT applications, reducing 52.58% of lines of code in average compared with traditional approaches; (2) TinyLink searches a much larger design space and thus can generate a superior solution for the hardware configuration, compared with the state-of-the-art approach; (3) TinyLink incurs acceptable overhead in terms of execution time and program memory.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Networks** → **Cyber-physical networks**;

KEYWORDS

Internet of Things; Rapid Development

1 INTRODUCTION

Over the past years, the Internet of Things (IoT) [1] has evolved from a vague concept to practical systems connecting lots of network devices, and it is considered as a promising future computing technology [2]. IoT platforms are tightly coupled to

their applications [3–5]. This coupling makes it difficult for general-purpose platforms to address application-specific needs [6].

The lack of general-purpose IoT platforms causes difficulties in developing IoT applications. Developers have to know all aspects of an IoT system including the requirements of target application as well as the hardware components and the software components. They typically follow a *bottom-up* approach from building an appropriate hardware platform (consisting of appropriate microcontrollers and different sensors) to writing application software that can run on the platform [7–9]. This bottom-up approach is time consuming [10]: an individual application developer has to learn and build a hardware platform first, then construct the software libraries, and finally write the application code; an application developer in a team has to wait for others to complete the hardware platform and the software system before he/she can write application code.

In this paper, we advocate a *top-down* approach: an application developer can first write the application code as if there exists a virtual platform which *possesses* all hardware components in the market. An intelligent system takes the application code as input and will: (1) automatically generate concrete configurations for hardware platforms (e.g., which components to select and how they connect); (2) translate the hardware-independent application code into hardware-dependent code which can directly execute on the generated hardware platform. This top-down approach greatly *accelerates* the developing process. While our previous position paper suggested the promise of this top-down approach [11], it fell short of answering several practical challenges, which we address in this paper.

First, how to select the most appropriate hardware components in order to satisfy application requirements? User desired functionalities can be inferred from the application code. How to select the hardware components accomplishing all the desired functionalities? Considering different costs and different characteristics of these hardware components, how to formulate this hardware selection problem and which optimization criterion should be chosen?

Second, how to embrace a diverse set of commercially available hardware components in a compatible manner? Significant hardware innovation has occurred recently, particularly in the Maker domain around Arduino and Raspberry Pi platforms. There has been a move from fixed platforms towards platforms with high extensibility. For example, mainboards already offer many interfaces, e.g., port and physical pin. Shield (a.k.a. expansion board) offers more flexibility for peripherals to interface with the mainboard. How to consider complex connections between peripherals and the mainboard?

* Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MobiCom '17, October 16–20, 2017, Snowbird, UT, USA

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-4916-1/17/10...\$15.00

<https://doi.org/10.1145/3117811.3117825>

Finally, how to express application logic in a hardware-independent manner and translate it into the executable code for the corresponding hardware configuration? Abstracting a unified programming environment is difficult as different platforms support different environments, e.g., Arduino supports AVR programming environment, and Raspberry Pi supports ARM-based Linux programming environment. How to implement a holistic system so that application developers need not care about low-level details such as device drivers, cross-compilers and so on?

In order to address the above challenges, we present TinyLink, a holistic system for rapid development of IoT applications. With TinyLink, developers can specify application logic in a C-like language, without dealing with the details of the underlying hardware, drivers, compilers, etc. Taking the application code as input, TinyLink automatically generates hardware configurations as well as the hardware-dependent code executable on the target hardware platform.

TinyLink includes a hardware database containing commercial-off-the-shelf (COTS) hardware components as well as their characteristics (e.g., costs, interfaces, required voltage, etc.). Based on the hardware database, TinyLink explicitly formulates possible compositions as hardware constraints. TinyLink carefully considers possible connections between ports and physical pins provided by mainboards and shields, and thus can embrace a diverse set of COTS hardware components. TinyLink also extracts the user constraints from the code, e.g., the invocation of a TinyLink API `TL_GPS.read()` means that one GPS sensor is required. TinyLink explores the design space by considering both hardware constraints and user constraints. Currently, TinyLink adopts an optimization criterion to minimize the cost. Other criteria (e.g., extensibility, energy, and performance) can also be adopted. TinyLink seeks the best solution by solving the optimization problem and outputs the hardware configuration (consisting of the set of hardware components and their connections).

TinyLink reuses easy-to-use Arduino-like environment for application programming. TinyLink provides unified APIs for the application to interface with the underlying hardware components. The APIs are fully implemented on four mainboards, including Arduino UNO, Raspberry Pi 2, BeagleBone Black and LinkIt ONE. To minimize the implementation overhead, TinyLink heavily reuses existing libraries. Glue code is implemented in TinyLink, serving as a binding from the APIs to existing libraries TinyLink builds upon.

TinyLink is a holistic system: besides generating hardware configurations and providing easy-to-use APIs, TinyLink also includes: (1) Cloud-based compiling architecture; (2) Web-based application programming interface; (3) commonly used benchmarks for system validation.

We evaluate TinyLink using real-world IoT applications. Results show that: (1) TinyLink achieves rapid development of IoT applications, reducing 52.58% of lines of code in average compared with existing programming approaches for Arduino UNO, Raspberry Pi 2, BeagleBone Black and LinkIt ONE; (2) TinyLink searches a much larger design space and thus can generate a superior solution for the hardware configuration, compared with CodeFirst [12]; (3) TinyLink incurs acceptable overhead in terms of execution time and program memory. Moreover, we present three concrete case

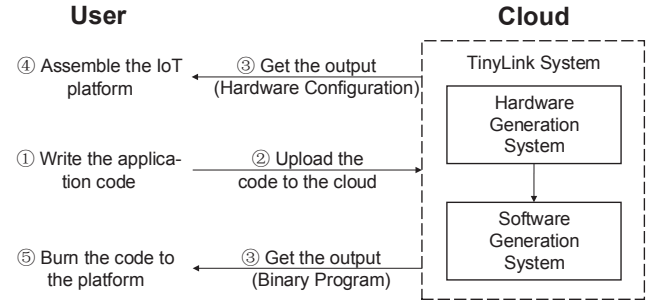


Figure 1: Workflow overview of TinyLink.

studies which illustrate how TinyLink can help us quickly develop real-world IoT applications.

The contributions of this work are summarized as below:

- We present TinyLink, a *holistic* system for developing IoT applications. TinyLink transforms IoT development process from *bottom-up* to *top-down*, enabling *rapid* development of IoT applications. TinyLink abstracts away the IoT hardware, so developers can focus on the application logic without experience in embedded systems.
- We formulate the hardware selection problem as an optimization problem. We carefully consider *hardware constraints* and *user constraints* so that a wide range of COTS hardware components can be used in a compatible manner and application requirements can be satisfied.
- We design unified APIs for applications to interface with COTS hardware components. We implement the APIs on four mainstream platforms so that application developers need not deal with low-level hardware details.
- We carefully evaluate TinyLink using benchmarks and three concrete case studies. Results show that TinyLink achieves rapid development of IoT applications and generates optimal solutions for hardware configuration, while incurring acceptable overhead.

The rest of this paper is structured as follows. Section 2 introduces how to use TinyLink. Section 3 shows TinyLink's design overview. Section 4 and Section 5 describe the hardware generation system and the software generation system, respectively. Section 6 shows the evaluation results. Section 7 discusses some important open issues. Section 8 describes the related work, and finally, Section 9 concludes this paper and gives future directions of work.

2 TINYLINK USAGE

In this section, we will describe how to use TinyLink for IoT application development. We first present an application example using TinyLink, followed by describing its syntax details.

2.1 Example

In this example, we present the development of an IoT application using TinyLink for monitoring the moisture and ambient light of a houseplant. As shown in Figure 1, a developer needs to perform the following five steps:

- ① **Write the application code.** Developers can directly write the application code in a *hardware-independent* way. For example,

```

1 void upload();
2 void setup() {
3   TL_WiFi.init();
4   TL_WiFi.join("SSID", "PASSWORD");
5   TL_Light.setMeasuringRange(1,30000, "LUX");
6   TL_Light.setADCResolution(10);
7 }
8
9 void loop() {
10  TL_Light.read();
11  TL_Soil_Moisture.read();
12  upload();
13  TL_Time.delayMillis(60000);
14 }
15
16 void upload() {
17   double light = TL_Light.data();
18   double sm = TL_Soil_Moisture.data();
19   String url = "http://hostname/ul.php?";
20   url += String("light=") + String(light);
21   url += String("&sm=") + String(sm);
22   TL_WiFi.get(url);
23 }

```

Figure 2: Application code example.

Figure 2 shows our implementation. First, the `setup()` function is executed for the initialization of the required modules (i.e., the WiFi module). Second, the `loop()` function is continuously executed for reading the sensor data and uploading the data via the WiFi module. Applications can directly invoke functions of the modules via TinyLink built-in APIs which are prefixed with “TL_”.

② **Upload the code to the cloud.** The TinyLink system runs at the cloud. Taking the application code as input, TinyLink executes multiple tasks and the results will be returned back to the developer.

③ **Get the outputs.** The developer gets the following two outputs: (1) the hardware configuration which includes the selected hardware components as well as their connections. TinyLink visualizes the result so that developers can easily follow the configuration to assemble the hardware platform. For example, Figure 3 shows the visualization of the hardware configuration which includes an Arduino UNO, a Base Shield V1.3, a Grove Light Sensor, a Soil Moisture Analog Sensor and an ESP8266 ESP01 WiFi module; (2) the hardware-dependent code. TinyLink transforms the hardware-independent application code into hardware-dependent binary code which can directly run on the target hardware platform.

④ **Assemble the IoT platform.** Given the visualization of the hardware configuration (e.g., Figure 3), developers can easily assemble the selected components into an IoT platform.

⑤ **Burn the code to the platform.** The developer can then burn the hardware-dependent code onto the hardware platform to execute the desired functionalities.

It is worth noting that this top-down development approach differs drastically from the traditional bottom-up approaches and can greatly accelerate the entire IoT developing process. With TinyLink, developers can easily express the application logic without dealing with hardware details. Also, the hardware engineering efforts can greatly be reduced since TinyLink automatically handles complex hardware compatibility issues.

2.2 TinyLink Programming

We consider ease of programming as a primary design goal of TinyLink. TinyLink borrows the program structure from Arduino

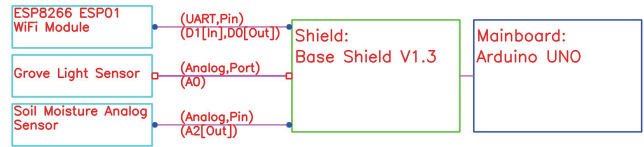


Figure 3: Hardware connection for the example.

programming. This is because: (1) it is fairly simple; (2) it is already very familiar to a large community. As shown in Figure 2, the application code built on top of TinyLink includes two primary functions.

setup(). The `setup()` function runs only once when the program starts. Developers can put the initialization of the required components or other one-shot tasks in this function. As shown in Figure 2, the program first initiates the WiFi module (line 3), and configures it by setting the SSID and the password of the AP (line 4). Developers can also specify additional features of hardware components, e.g., the measurement range (line 5) and the ADC resolution (line 6) of a light sensor.

loop(). The `loop()` function will be continuously executed after the `setup()` function. The `loop()` function usually includes some periodical tasks. As shown in Figure 2, the program reads data from the light sensor and the soil moisture sensor. Then the program invokes the function `upload()` to upload the data to the cloud server. Finally, the program invokes a built-in API to sleep for one minute (line 13) and then the loop restarts again.

TinyLink provides a keyword, `REQUIRE`, for explicitly specifying a required module. In the following, we give two examples.

- (1) `REQUIRE Light;`—This statement specifies that the IoT platform must include the light sensor even if the light sensor is not used in the application code;
- (2) `REQUIRE DBGPrint;`—This statement specifies that IoT platform must include an additional UART interface for printing out debugging information.

TinyLink provides many hardware-independent APIs for developers. The details will be described in Section 5.2.

3 TINYLINK DESIGN OVERVIEW

We describe TinyLink’s design goals and the corresponding approaches we have adopted.

- **Rapid IoT application development.** To achieve this goal, TinyLink: (1) adopts the *top-down* development model for IoT applications; (2) generates the hardware configuration and the binary program *at the cloud* so that there is no installation of local developing environments.
- **Ease of programming.** To achieve this goal, TinyLink borrows the simple program structure of Arduino programming. In addition, TinyLink provides *hardware-independent* APIs to facilitate application programming.
- **Little hardware engineering efforts.** TinyLink incorporates a hardware database with clear specifications of hardware components and their characteristics. The generation of the hardware configuration is formulated as an optimization problem with constraints inferred from the application code. By solving the optimization problem, TinyLink *automatically* generates the best hardware configuration.

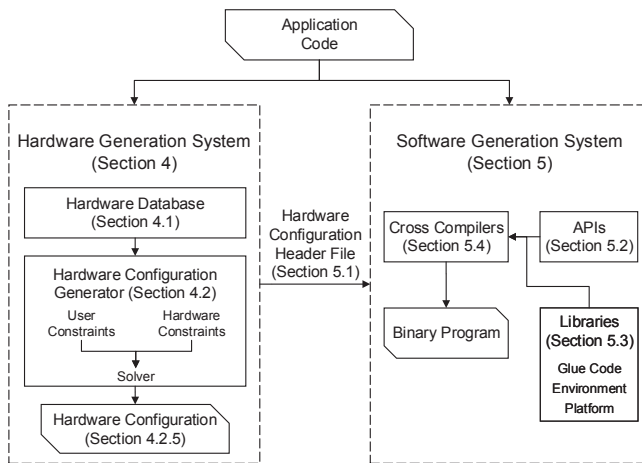


Figure 4: TinyLink system architecture.

- **High extensibility to accommodate many hardware components.** TinyLink is designed to embrace a set of COTS hardware components. Unlike CodeFirst [12], we do not assume a fixed mainboard. In addition, we need to consider complex connections between interfaces on hardware components. Finally, we need to carefully consider the implementations of libraries so that developers can easily interface with the underlying hardware components.

Figure 4 depicts the TinyLink system architecture, including the hardware generation system (Section 4) and the software generation system (Section 5). The application code serves as the input of TinyLink and is passed to both generation systems:

- Inside the hardware generation system, the hardware configuration generator automatically generates the hardware configuration by analyzing the application code and considering the information from the hardware database. The hardware generation system also generates the hardware configuration header file.
- Inside the software generation system, the cross compiler takes three sources as inputs, namely the application code, the generated hardware configuration header file and libraries. Finally, the binary program for the target platform is generated.

4 HARDWARE GENERATION SYSTEM

In this section, we will describe the TinyLink hardware generation system from two aspects, the hardware database (Section 4.1) and the hardware configuration generator (Section 4.2).

4.1 Hardware Database

TinyLink includes a hardware database containing COTS hardware components and their characteristics. This hardware database serves as the key information source to automatically generate hardware configurations. We look for popular and widely-used COTS hardware components for embedded and IoT development and find that many platforms consist of three kinds of components:

- **Mainboard.** A mainboard contains key electronic components of a system, e.g., MCU and memory. It also provides

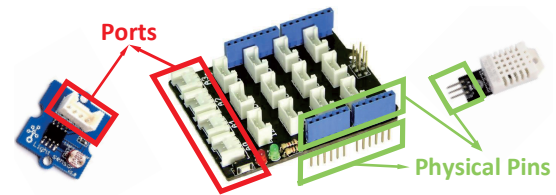


Figure 5: Physical pins and ports.

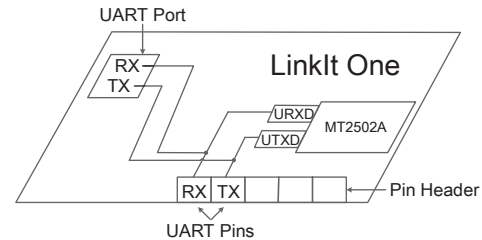


Figure 6: Internal connections of UART pins, UART ports and UART MCU pins on LinkIt One.

interfaces (in the forms of physical pin or port) for other peripherals.

- **Shield.** A shield can be plugged onto the mainboard to transform or extend the types or number of interfaces. Also, shields with built-in modules will consume interfaces and provide functionalities, e.g., WiFi Shield V2.0 consumes UART pins and provides WiFi functionality. Moreover, some shields containing MCUs can provide extra interfaces, e.g., GrovePi+ [13] has an ATmega328 MCU and provides three extra Analog pins for Raspberry Pi.
- **Peripherals.** A peripheral can connect to a mainboard or a shield via interfaces. Peripherals include input devices (e.g., sensors), output devices (e.g., display modules) and input/output devices (e.g., communication modules).

Correspondingly, our database includes three tables. Table 1 shows a simplified table which contains the following key fields: ID, cost, provided interface, consumed interface, provided MCU pin, consumed MCU pin and functionality. For LinkIt One, it: (1) costs 59 dollars; (2) has three Analog pins and two UART pins (the example only shows two types); (4) has one UART port; (5) has exposed three Analog MCU pins and two UART MCU pins; (6) occupies none of the interfaces and MCU pins; (7) has functionalities of GPRS, GPS, SD, WiFi, BLE integrated into the mainboard.

As shown in Figure 5, physical pins and ports are two major kinds of interfaces on IoT components. Besides, MCU or System on Chip (SoC) usually contains from several to dozens of general-purpose input/output pins (GPIO), which we call MCU pins. The physical pins and ports are wiring to MCU pins. For example, in Figure 6, the UART pins (i.e., RX and TX) and the UART port are wiring to the UART MCU pins (i.e., URXD and UTXD).

4.2 Hardware Configuration Generator

IoT has a diverse set of applications for different scenarios. Although some applications can use a powerful mainboard with abundant inferences, others cannot. Considering form factors, cost and other factors, the most appropriate hardware component may be resource constrained. The generation process intends to address the problem

Table 1: Database example.

Component	ID	Cost	Provided Interface				Consumed Interface				Provided MCU Pin		Consumed MCU Pin		Functionality
			Analog Pin	UART Pin	Analog Port	UART Port	Analog Pin	UART Pin	Analog Port	UART Port	Analog	UART	Analog	UART	
Arduino UNO	1	\$21.99	6	2	0	0	0	0	0	0	6	2	0	0	LED
LinkIt One	2	\$59.00	3	2	0	1	0	0	0	0	3	2	0	0	LED, GPRS, GPS, Storage, WiFi, BLE
BetaBlocks	3	\$75.00	0	0	2	1	0	0	0	0	4	2	0	0	LED
WiFi Shield V2.0	4	\$39.95	6	2	1	0	6	2	0	0	0	0	0	2	WiFi
Base Shield V1.3	5	\$8.90	6	2	2	1	6	2	0	0	0	0	0	0	-
Grove Light Sensor	6	\$2.90	0	0	0	0	0	0	1	0	0	0	2	0	Light
Soil Moisture Analog Sensor	7	\$3.98	0	0	0	0	1	0	0	0	0	0	1	0	Soil Moisture
Grove Moisture Sensor	8	\$4.90	0	0	0	0	0	0	1	0	0	0	2	0	Soil Moisture
ESP8266 ESP01 WiFi Module	9	\$6.95	0	0	0	0	0	2	0	0	0	0	0	2	WiFi
Grove UART WiFi Module	10	\$13.90	0	0	0	0	0	0	0	1	0	0	0	2	WiFi

of how to assemble COTS hardware components in an effective and compatible way.

We first describe the conditions for a valid/compatible hardware configuration. We then formulate the hardware selection problem, i.e., which hardware components should be selected to form the IoT platform. Finally, we generate the hardware connections and visualize the final results for the developers.

4.2.1 Valid Hardware Configuration. In TinyLink, it is important to clearly define a valid hardware configuration before we can search for a valid hardware configuration. We first define a valid connection between two hardware components via interfaces. A valid connection should satisfy the following conditions: (1) the two interfaces must be in the same form (i.e., physical pin or port); (2) the two interfaces must be type compatible. Usually, it means that the two interfaces has the same type, e.g., UART. In practice, we also consider the special case in which an interface can have two or more alternative types; (3) the interfaces must be power compatible, i.e., the supplied voltage level of an interface is compatible with the working voltage level of the other interface, and the drawn current should be smaller than the maximal allowable current of both interfaces.

In a valid hardware configuration, all connections must be valid. To simplify the configuration, we first assume that there is only one mainboard. In addition, it should satisfy the following conditions: (1) the configuration must satisfy all the application requirements; (2) a connection is a one-to-one mapping. In other words, an interface cannot connect to multiple interfaces of another component since it will cause signal interference. It also means that the provided number of interfaces must be no smaller than the consumed number of interfaces with respect to a particular interface type (e.g., UART); (3) only one connection can be established on interfaces which are wired to the same MCU pin. As shown in Figure 6, either the UART port or the UART pins can be connected at a time. A detailed example will be described in Section 4.2.4.

Table 2 summarizes the conditions we have considered in the current implementation of TinyLink.

There are several important points worth emphasizing. First, TinyLink has a much larger design space than previous approaches like CodeFirst [12]. For example, CodeFirst only allows connections between ports while TinyLink allows connections between pins which are common cases for many peripherals such as Soil Moisture Analog Sensor and ESP8266 WiFi module. Second, the conditions we have considered for a valid configuration are never exhaustive [14]. It is possible that a valid hardware configuration generated by TinyLink could not work due to conditions we have not considered,

Table 2: Implemented constraints in TinyLink.

Constraint	Description
Mainboard	Only one mainboard exists in a hardware configuration.
Interface	For each type of the interface, the number of consumed ones should be smaller than the number of the provided.
Power Supply	The supplied voltage level should be compatible with the working voltage level.
Drawn Current	The total drawn current of all components should be smaller than the input current; the drawn current of an interface should be smaller than the maximal allowable current.
MCU Pin	The number of consumed MCU pins should be smaller than the number of the provided.
Compatibility	All components should be compatible with each other in terms of form factor, communication, etc.
Alternative Type	All the interfaces which have alternative types can use at most one type at a time.

e.g., incompatible baud rate of UART. Nevertheless, TinyLink is valuable since it help developers avoid numerous invalid hardware configurations. In addition, TinyLink can be easily extended to accommodate additional conditions in the future.

4.2.2 Problem Formulation. We first introduce the following notations.

- M, S, P . We use them denote the sets of mainboard, shield, and peripheral, respectively. In the database example in Table 1, $M = \{1, 2, 3\}$, $S = \{4, 5\}$, $P = \{6, 7, 8, 9, 10\}$.
- U, F . We use F to denote the set of all functionalities. We use U to denote the set of user required functionalities. U can be inferred from the application code. U is included in F .
- i, d_i, c_i, f_i^u . We use d_i to denote the indicator variable. $d_i = 1$ indicates that hardware component i is present in the final IoT platform, where $i \in \{M \cup S \cup P\}$, while $d_i = 0$ indicates that it is not. c_i is the cost of the component i . f_i^u is the boolean value (i.e., 1 and 0) indicating whether the component i has the required functionality of $u \in U$ or not.
- t, T . We use t to denote the type of an interface, where t is in the interface type set $T = \{\text{UART, Analog, SPI, I}^2\text{C, PWM, USB, Digital}\}$.
- $NO_i^+(t), NO_i^-(t)$. $NO_i^+(t)$ represents the number of interfaces with the type t in the form of port provided by the component i , while $NO_i^-(t)$ represents the number of interfaces with the type t in the form of port consumed by the component i .
- $NI_i^+(t), NI_i^-(t)$. Similar to the notations for ports, where NI represents the number of physical pins.
- $NM_i^+(t), NM_i^-(t)$. Similar to the notations for ports and physical pins, where NM represents the number of MCU pins.

With the above definitions, the hardware selection problem can be formulated as follows with the optimization criterion being the total cost.

$$\begin{aligned} & \text{Find the values of } d_i \ (\forall i \in \{M \cup S \cup P\}) \\ & \min \sum_{i \in \{M \cup S \cup P\}} d_i c_i \\ & \text{s.t. } \begin{cases} \text{User Constraints} \\ \text{Hardware Constraints} \end{cases} \end{aligned} \quad (1)$$

Component i is selected iff $d_i = 1$. We call a set of i s as a feasible set if they satisfy all the constraints. In the following, we describe how we carefully setup constraints and conduct validations to ensure that feasible solutions are valid ones.

4.2.3 User Constraint. TinyLink first *preprocesses* the application code and infers the user requirement U . User constraints are automatically generated:

$$\sum_{i \in \{M \cup S \cup P\}} f_i^u d_i \geq 1, \quad \forall u \in U \quad (2)$$

Example. We revisit the example shown in Figure 2. Also assume the hardware database is shown in Table 1. TinyLink automatically infers U as $U = \{\text{Light, Soil_Moisture, WiFi}\}$. Thus three user constraints are generated:

$$\begin{cases} d_6 \geq 1 & (\text{Light}) \\ d_7 + d_8 \geq 1 & (\text{Soil_Moisture}) \\ d_2 + d_4 + d_9 + d_{10} \geq 1 & (\text{WiFi}) \end{cases}$$

4.2.4 Hardware Constraint. While the user constraints are fairly simple, the hardware constraints are much more complicated. Due to space limit, we describe three important constraints.

1) **Port Constraint.** TinyLink allows the use of shields to extend the capability of mainboard. As such, for a particular type, the number of ports provided by the mainboard plus the number of ports provided by the shield, minus the number of ports consumed by the shield, should be no smaller than the number of ports consumed by the peripherals. The port constraint for TinyLink is shown below:

$$\sum_{i \in M} NO_i^+(t) d_i + \sum_{i \in S} (NO_i^+(t) d_i - NO_i^-(t) d_i) \geq \sum_{i \in P} NO_i^-(t) d_i, \quad \forall t \in T \quad (3)$$

2) **Physical Pin Constraint.** TinyLink allows connections via physical pins. Similar to the port constraint, the physical pin constraint for TinyLink is shown below:

$$\sum_{i \in M} NI_i^+(t) d_i + \sum_{i \in S} (NI_i^+(t) d_i - NI_i^-(t) d_i) \geq \sum_{i \in P} NI_i^-(t) d_i, \quad \forall t \in T \quad (4)$$

3) **MCU Pin Constraint.** Nevertheless, the actual number of *usable* ports and physical pins is not as easy as calculating their sum. This is because some ports and physical pins are mapped to the same MCU pins, which means that they actually share the same interface and will interfere with each other if they are used simultaneously. For example, in Figure 6, the UART physical pin RX in the pin header and the pin inside the UART port (i.e., RX pin of the UART port) are *wiring* to the same MCU pin URXD on

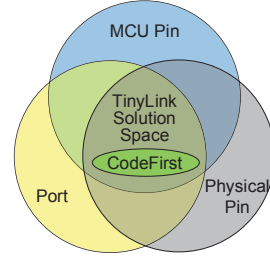


Figure 7: Solution space comparison between TinyLink and CodeFirst.

MT2502A, the SoC of LinkIt One. It is the same case for the UART physical pin TX and the TX pin in the UART port.

As a result, we need to constrain the number of usable interfaces at the MCU pin level. The total number of consumed MCU pins should be no larger than the total number of provided ones. The MCU pin constraint is generated as follows:

$$\sum_{i \in M} NM_i^+(t) d_i + \sum_{i \in S} (NM_i^+(t) d_i - NM_i^-(t) d_i) \geq \sum_{i \in P} NM_i^-(t) d_i, \quad \forall t \in T \quad (5)$$

Example. We revisit the example in Figure 2. TinyLink generates the following constraints:

$$\begin{cases} 0d_1 + 0d_2 + 2d_3 + (1d_4 + 2d_5 - 0d_4 - 0d_5) \geq \\ \quad d_6 + 0d_7 + d_8 + 0d_9 + 0d_{10} & (\text{Port Cons. for Analog}) \\ 0d_1 + d_2 + d_3 + (0d_4 + d_5 - 0d_4 - 0d_5) \geq \\ \quad 0d_6 + 0d_7 + 0d_8 + 0d_9 + d_{10} & (\text{Port Cons. for UART}) \\ 6d_1 + 3d_2 + 0d_3 + (6d_4 + 6d_5 - 6d_4 - 6d_5) \geq \\ \quad 0d_6 + d_7 + 0d_8 + 0d_9 + 0d_{10} & (\text{PHY Pin Cons. for Analog}) \\ 2d_1 + 2d_2 + 0d_3 + (2d_4 + 2d_5 - 2d_4 - 2d_5) \geq \\ \quad 0d_6 + 0d_7 + 0d_8 + 2d_9 + 0d_{10} & (\text{PHY Pin Cons. for UART}) \\ 6d_1 + 3d_2 + 4d_3 + (0d_4 + 0d_5 - 0d_4 - 0d_5) \geq \\ \quad 2d_6 + d_7 + 2d_8 + 0d_9 + 0d_{10} & (\text{MCU Pin Cons. for Analog}) \\ 2d_1 + 2d_2 + 2d_3 + (0d_4 + 0d_5 - 2d_4 - 0d_5) \geq \\ \quad 0d_6 + 0d_7 + 0d_8 + 2d_9 + 2d_{10} & (\text{MCU Pin Cons. for UART}) \end{cases}$$

Together with the mainboard constraint (i.e., $d_1 + d_2 + d_3 = 1$) and user constraints in Section 4.2.3, we can get 19 feasible sets.

Comparison with CodeFirst. The CodeFirst [12]: (1) does not allow the use of shield; (2) does not allow connections via pins; (3) does not consider the MCU pin constraints.

With the following port constraints,

$$\sum_{i \in M} NO_i^+(t) d_i \geq \sum_{i \in P} NO_i^-(t) d_i, \quad \forall t \in T$$

CodeFirst generates the following concrete constraints for the example shown in Figure 2:

$$\begin{cases} 2d_3 \geq d_6 + 0d_7 + d_8 + 0d_9 + 0d_{10} & (\text{Port Cons. for Analog}) \\ d_3 \geq 0d_6 + 0d_7 + 0d_8 + 0d_9 + d_{10} & (\text{Port Cons. for UART}) \end{cases}$$

Assuming a fixed mainboard of BetaBlocks, CodeFirst gets only one feasible solution $\{3, 6, 8, 10\}$, which is included in the set of TinyLink's feasible solutions.

Figure 7 shows the solution space of TinyLink and CodeFirst. Under the three constraints we have described, the set of all

Table 3: API call frequency summarized from real-world projects.

LED	Temperature	WiFi	Display	Humidity	Motor	Light	Speaker	Distance	Soil Moisture	Accelerometer	BLE	Buzzer	GPS	Switch	Motion	Barometric	Joystick	RFID	Storage	Others
23.11%	10.22%	10.22%	9.78%	4.89%	4.44%	3.56%	2.67%	2.67%	2.22%	2.22%	2.22%	1.78%	1.78%	1.78%	1.78%	1.33%	1.33%	1.33%	1.33%	9.33%

CodeFirst feasible solutions is a subset of TinyLink feasible solutions. The search space of TinyLink is much larger since it allows complex connections between components.

4.2.5 Implementation. The generation of hardware configuration consists of three main steps.

First, we solve the optimization problem described above and look for the most appropriate hardware components. In real implementations, we use complete set of constraints listed in Table 2. We further consider the compatibility constraint. The compatibility constraint: (1) forbids physically incompatible hardware components to be connected, e.g., Base Shield V1.3 and Raspberry Pi 2 are not physically compatible due to their different form factors; (2) restrains hardware components from being connected to those they cannot work with in a compatible way during run time, e.g., a Grove Temperature and Humidity Sensor Pro on a LinkIt One frequently fails to read the temperature data though their connection via ports is valid. Since the formulated problem is a classical integer linear programming problem, we utilize the ILP solver, *lp_solve* [15], to solve this optimization problem.

Second, TinyLink will show developers how to assemble these hardware components after generating the hardware component list. The connection is generated by correctly allocating interfaces on the mainboard and the shields to hardware components. Given the selected mainboard and shields, TinyLink calculates all the available interfaces, and then allocate them by priority. First, TinyLink allocates the pin interfaces occupied by built-in modules on mainboards and shields, e.g., UART pins on WiFi Shield V2.0. Afterwards, TinyLink allocates non-occupied ports and non-occupied physical pins to peripherals, because ports usually occupy consecutive MCU pins shared with physical pins. Based on the allocation, TinyLink utilizes OpenCV to generate a simple visualization for the generated hardware configuration, e.g., Figure 3 visualizes the recommended hardware configuration for the application code shown in Figure 2.

Finally, TinyLink employs an additional validation procedure to verify that the recommended hardware configuration is a valid one, according to definitions described in Section 4.2.1. If the hardware configuration happens to be invalid due to invalid connections (which is rare from our experience), TinyLink invokes the ILP solver again to seek for a different solution which is valid by our definition. In the future, we would like to borrow the crowdsourcing idea for IoT application development. If the current developer selects a hardware configuration that was used and reported successfully by others, the developer has a high chance of getting the correct one.

5 SOFTWARE GENERATION SYSTEM

This section describes how to generate hardware-dependent binary program from the application code. Section 5.1 describes the generation of hardware configuration header file. Section 5.2 and Section 5.3 present TinyLink API design and the library implementations in detail. Section 5.4 describes the cross compilers,

```

1 #include "TL_DeviceID.h"
2
3 #define MAINBOARD ARDUINO_UNO
4 #define WIFI ESP8266_ESP01
5 #define LIGHT GROVE_LIGHT
6 #define SOIL_MOISTURE SOIL_MOISTURE_ANALOG
7
8 #define WIFI_UART_RX 1
9 #define WIFI_UART_TX 0
10 #define LIGHT_ANALOG_OUTPUT A0
11 #define SOIL_MOISTURE_ANALOG_OUTPUT A2

```

Figure 8: Hardware configuration header file.

followed by Section 5.5 which discusses interactive debugging in TinyLink.

5.1 Hardware Configuration Header File

From the perspective of developers, the application code is hardware-independent. However, from the perspective of cross compilers, the code should be hardware-dependent, e.g., link to the correct libraries corresponding to the underlying hardware. Therefore, we translate the hardware configuration information (see Figure 3) to the hardware configuration header file in C language (see Figure 8).

The hardware configuration header file is composed of three sections. The first section includes other important header files (line 1), such as the device ID header file. The second section describes which hardware components are used via macros (line 3-6). In each line, an abstract component (e.g., MAINBOARD in line 3) is defined by a specific hardware component (e.g., ARDUINO_UNO in line 3), which relates the hardware-independent functionality to the hardware-dependent components. The third section describes the hardware connection information (line 8-11). This hardware configuration header file is generated by the hardware generation system and is essential for compiling the application code for the target platform.

5.2 API Design

Unlike traditional IoT programming, programming with TinyLink is hardware-independent, i.e., the developers are unaware of the underlying hardware components. APIs play an important role in relating high-level semantics to low-level details. TinyLink provides generalized APIs to facilitate application programming.

In order to provide developers with widely-used APIs, we investigate 116 projects from popular IoT online forums [16, 17] and websites [18, 19]. We extract 225 commonly used APIs from them, and group them into 39 categories by the functionalities they provide. As shown in Table 3, we list the twenty mostly used categories and their call frequency, where the most popular one is the LED, accounting for about 23%. In the current implementation of TinyLink, we encapsulate and implement 24 categories, covering 89.78% of the total APIs we have investigated. The remaining 10.22% (i.e., uncovered) of the APIs are left unimplemented, including those controlling DIY devices, transmitting and receiving infrared signals, etc. Given a limited development time, we have decided to implement the most important and commonly used APIs while skipping those infrequently invoked ones.

We say that TinyLink supports a project if all the APIs used in this project have corresponding TinyLink APIs. 67.24% of the projects are supported by TinyLink and can be ported to TinyLink with little modifications. The remaining 32.76% of the projects have at least one uncovered API and thus cannot be easily ported to TinyLink in the current stage. We plan to implement more TinyLink APIs to support more IoT applications in the future.

How to exchange messages over unreliable networks in an energy-efficient manner is an important issue in IoT development. Message Queue Telemetry Transport (MQTT) protocol is a lightweight messaging protocol [20] which builds on top of the TCP protocol. It has been adopted as a primary protocol for IoT communications by major IoT cloud providers (e.g., IBM, Amazon and Microsoft). To support this widely-used protocol in IoT development, we provide MQTT APIs in TinyLink. Once a TCP connection is established, a TinyLink node with MQTT can connect to a broker (i.e., MQTT client at the server side) upon it, and then publish/subscribe their information. We have tested our TinyLink MQTT APIs on IBM Bluemix and China Mobile IoT Open Platform. All four mainboards, i.e., Arduino UNO, LinkIt One, Raspberry Pi 2 and BeagleBone Black, can connect to MQTT brokers of the two cloud platforms using ESP8266 WiFi module and Grove UART WiFi module, as well as the built-in WiFi module of LinkIt One and the built-in Ethernet modules of Raspberry Pi 2 and BeagleBone Black.

In addition, we provide basic debugging APIs for developers. They can write `REQUIRE DBGPrint` to indicate that they require printing out debugging information. In this case, the TinyLink hardware system will occupy an additional serial interface on the mainboard. All debugging information will be transmitted via this serial interface. Developers can use `TL_Debug.print()` to print out their own debugging information.

5.3 Library Implementation

Libraries are the actual implementations of TinyLink APIs. We are facing practical implementation challenges since each library implementation requires developing efforts and the number of required implementations may grow rapidly with increasing number of hardware components. We address these challenges in the following ways.

First, we implement the most commonly used libraries according to our experience so that we can benefit more number of users given a limited development time. With the increasing use of TinyLink, we could get more feedbacks about which libraries should be implemented with a higher priority.

Second, we try to reuse existing libraries and only add *glue* code to implement TinyLink APIs. Writing glue code includes abstracting the unified input/output interfaces, replacing hardware-dependent code with macros, adding conditional syntax in the API header file and so on.

Figure 9 illustrates how we implement the TinyLink API `TL_LED.turnOn()` by reusing libraries on four mainboards. For these four mainboards, we use the function `pinMode()` to configure LED pin behavior and use `digitalWrite()` to turn the LED on. These APIs are implemented differently for different platforms. On Arduino and LinkIt platforms (Arduino programming environment and VM on Linux), they can be easily implemented using the

```
void TL_LED::turnOn() {
    pinMode(LED_DIGITAL_OUTPUT, OUTPUT);
    digitalWrite(LED_DIGITAL_OUTPUT, HIGH);
}
```

Figure 9: Glue code example of implementing an LED function.

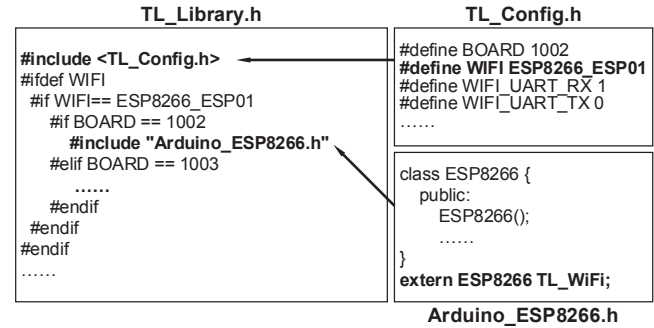


Figure 10: Including libraries using header files.

system library. On Raspberry Pi and BeagleBone platforms (Debian-based Linux operation systems), they can be implemented using third-party libraries, e.g., wiringPi [21] and wiringBone [22]. The implementation overhead for glue code is relatively small when we carefully choose the existing libraries.

Finally, we will allow for crowdsourcing the library implementations by application developers. We will release TinyLink libraries to be open source online, and in the long term, we hope that more and more IoT developers will join us to implement the useful libraries which can be shared among other developers. As an initial effort, we provide component templates written in C++ for developers to implement TinyLink libraries for a new device. For example, in order to implement a library for a simple sensor, a developer can use the sensor template we have developed, inherit the Sensor class and implement abstract interfaces like `virtual int _read()=0`.

It is also worth noting that it is possible for TinyLink to generate a hardware configuration which misses some software implementations. To address this issue, we provide two options for the developers: (1) provide the best IoT hardware configuration and the application developers have to implement the required library code; (2) provide the suboptimal solution with complete software implementations.

5.4 Cross Compiler

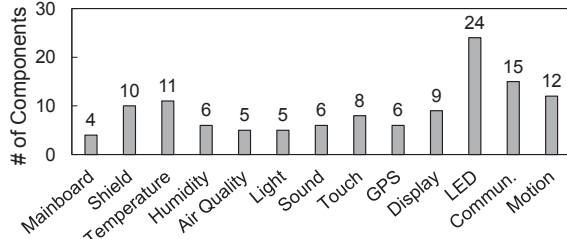
The preprocessed code needs to be cross compiled on the cloud to the binary code for the target platform. There are two key problems that TinyLink needs to address: (1) Which libraries to link? (2) Which cross compiler to use?

TinyLink relies on a header file `TL_Config.h` which encodes the hardware configuration. The type of mainboards defined in `TL_Config.h` determines the specific cross compiler.

Figure 10 shows how TinyLink links hardware-independent application code with hardware-dependent libraries. TinyLink libraries include two headers files, which are `TL_Library.h` and `<mainboard>_<peripheral>.h`, for transforming hardware-independent APIs to hardware-dependent APIs. For the example shown in Figure 10, `TL_Library.h` will include `TL_Config.h` to include necessary function modules, e.g., `ESP8266_ESP01`, which

Table 4: Cross compilers of four mainboards.

Mainboard	MCU Architecture	Cross Compiler
Arduino UNO	AVR ATmega	avr-gcc, avr-g++
LinkIt One	ARM7 EJ-S	arm-none-eabi-gcc, arm-none-eabi-g++
Raspberry Pi 2	ARM Cortex-A7	arm-linux-gnueabi-gcc, arm-linux-gnueabi-g++
BeagleBone Black	ARM Cortex-A8	arm-linux-gnueabi-gcc, arm-linux-gnueabi-g++

**Figure 11: Number of components in TinyLink database.**

again includes `Arduino_ESP8266.h` according to the hardware configuration defined in `TL_Config.h`.

We list cross compilers for four mainboards in Table 4. The Arduino UNO based on AVR ATmega utilizes `avr-gcc` and `avr-g++` compilers. LinkIt One, which is based on ARM7, leverages `arm-none-eabi-gcc` and `arm-none-eabi-g++` compilers. The Raspberry Pi 2 and BeagleBone Black, with ARM Cortex-A7 and Cortex-A8 respectively, both use `arm-linux-gnueabi-gcc` and `arm-linux-gnueabi-g++` compilers.

5.5 Interactive Debugging

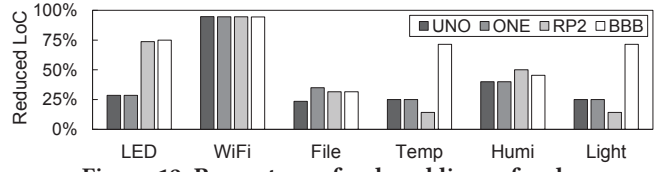
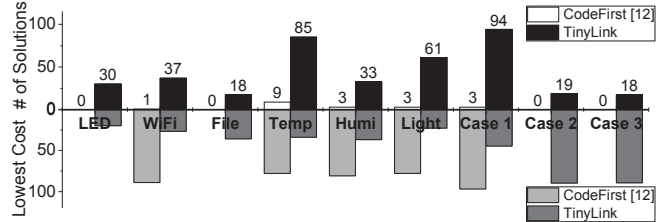
TinyLink supports interactive debugging if the native system supports it. In the current stage, TinyLink supports interactive debugging using `gdb` for the Debian systems on Raspberry Pi and BeagleBone Black. If `gdb` debugging is required by writing `REQUIRE_GDB`, TinyLink will compile the application code to include necessary debugging information. In addition, developers are provided with the source code of the whole project, including high-level application code as well as low-level hardware-dependent library code. Based on these, developers can log into the platform and debug the source code with functions provided by `gdb`, e.g., breakpoints, watchpoints and stepped execution.

6 EVALUATION

In this section, we present the evaluation of TinyLink. Section 6.1 presents the experiment setup. Section 6.2 shows the overall evaluations in terms of ease of programming, feasible solutions and overhead. We study three real-world cases using both TinyLink and CodeFirst [12] in Section 6.3.

6.1 Experiment Setup

Hardware Component. In Figure 11, we list the number of components in our hardware database grouped by functionalities. There are four mainboards, ten shields and more than one hundred peripherals recorded in our database. The four mainboards include Arduino UNO (UNO for short), LinkIt ONE (ONE for short), Raspberry Pi 2 (RP2 for short) and BeagleBone Black (BBB for short). The characteristics (e.g., resolution and sensing range) of these components are also stored in our database.

**Figure 12: Percentage of reduced lines of code.****Figure 13: Number of solutions and the lowest cost.**

Benchmark. We use six frequently used benchmarks according to Table 3 for evaluation, which are: (1) LED Write (LED for short), which turns on an LED; (2) WiFi GET (WiFi for short), which sends an HTTP GET request to a website in the local area network and retrieves the response; (3) File Write & Read (File for short), which writes data to a file on a storage module and then reads the data; (4) Temperature Read (Temp for short), which reads temperature data from a sensor; (5) Humidity Read (Humi for short), similar to Temp; (6) Light Read (Light for short), similar to Temp.

Case Study. We use three real-world case studies, which are an indoor smart houseplant node which has been described in Section 2, an air quality monitoring node and a voice controlled LED lamp node. More details are presented in Section 6.3.

6.2 Overall Evaluation

6.2.1 Ease of Programming. First, we compare lines of code needed to implement the benchmarks between using TinyLink APIs and using original APIs (e.g., APIs provided by hardware manufacturers). We implement the six benchmarks on four mainboards and report the reduced lines of code. In Figure 12, we can see that using TinyLink reduces the lines of code by 14.29% to 94.67%. The average percentage of reduced lines of code on four mainboards is 52.58%. In particular, for implementing the WiFi GET benchmark, using TinyLink APIs reduces more than 94.40% of lines of code on all mainboards. This is due to the simple API design and a large amount of glue code implementations for the WiFi library.

Second, we compare the lines of code between TinyLink and CodeFirst [12]. CodeFirst uses 27 lines of code to implement an example of reading data from a temperature sensor, displaying data on an LCD screen and transmitting data via Bluetooth. This is the only example which has the source code in CodeFirst. In TinyLink, developers can implement an application with the same functionalities only using 15 lines of code.

6.2.2 Feasible Solution. TinyLink can generate many feasible solutions after exploring the hardware database and choose the best one according to the given optimization criterion (e.g., the total cost). A larger searching space will provide more opportunities to generate more feasible solutions, as well as a better solution. We evaluate the feasible solutions for six small benchmarks and three case studies, and compare the results with CodeFirst [12]. Figure 13

Table 5: Relative execution overhead.

API From Benchmark	LED Write	UART Write	UART Read	File Write	File Read	Temp Read	Humi Read	Light Read
UNO	13.79%	0.00%	0.00%	1.10%	1.26%	0.01%	0.01%	7.56%
ONE	6.25%	0.02%	0.63%	0.38%	0.48%	0.02%	0.04%	1.15%
BBB	10.86%	0.76%	2.73%	15.68%	14.16%	0.03%	0.04%	8.90%
RP2	12.86%	10.74%	7.50%	0.71%	1.94%	0.43%	0.48%	2.13%

Table 6: Program space overhead (Bytes).

Benchmark	LED	WiFi	File	Temp	Humi	Light
UNO	166	184	2178	374	278	414
ONE	4372	2780	31000	3328	3320	3636
BBB	78224	63668	82311	79428	79428	77504
RP2	30713	11039	33317	27016	27016	26332

Table 7: Memory space overhead (Bytes).

Benchmark	LED	WiFi	File	Temp	Humi	Light
UNO	21	170	56	23	23	23
ONE	1272	1384	21384	1200	1208	1248
BBB	82591	99511	85193	82756	82627	82257
RP2	30713	11039	33317	27016	27016	26332

shows the results. For each benchmark and case study, we count the number of feasible solutions, as well as the lowest cost in the set of the feasible solutions. We can observe that TinyLink generates far more feasible solutions than CodeFirst and much lower total cost for each benchmark and case study. In the LED write and the File Write & Read benchmarks, CodeFirst fails to generate any feasible solutions, because it only use ports which are usually not supported by peripherals with the storage functionality.

6.2.3 Overhead. The implementations of glue code on existing libraries introduce overhead to the generated IoT applications. In the following, we evaluate the execution overhead, program space overhead and memory space overhead.

Execution Overhead. Most implementations of unified TinyLink APIs are the encapsulations of existing ones, and the overhead comes from the glue code. We take experiments on the execution time of API calls in our benchmarks. Table 5 shows the relative execution overhead of using TinyLink APIs compared with using original APIs. Considering the magnitude of the execution overhead is μs (e.g., $0.75\mu s$ for LED benchmark on UNO), the overhead is acceptable.

Program Space Overhead. The program space overhead is proportional to the amount of implemented glue code. Due to different MCU architectures, we take different methods to measure the program space. For the UNO with an AVR MCU, the program space is calculated by the sum of .text and .data segments. For the other three with ARM MCU, the program space is the size of their cross compiled binary programs.

Table 6 shows the program space overhead. The program space overhead of UNO and ONE is no more than 3KB and 31KB respectively. Considering their program flash are 32KB and 16MB respectively, the overhead is small and acceptable. The reasons why the overhead of BBB and RP2 is much larger is that they run on a Debian Linux system and their binary programs are in ELF format. But considering their program flash are at least 4GB, the overhead within 100KB is negligible.

Memory Space Overhead. The measured memory space overhead is composed of the overhead from static memory overhead (i.e., .text, .data, etc.) and run-time memory overhead (i.e., stack and heap). Table 7 shows that the memory space overhead is small

Table 8: Hardware configurations of smart houseplant nodes.

	TinyLink	CodeFirst [12]
Mainboard	Arduino UNO	BetaBlocks
Shield	Base Shield V1.3	-
Peripheral	Grove Light Sensor	Grove Light Sensor
	Soil Moisture Analog Sensor	Grove Moisture Sensor
	ESP8266 ESP01 WiFi Module	Grove UART WiFi Module
Total Cost	\$44.72	\$96.70

Table 9: Hardware configurations of air quality nodes.

	TinyLink	Mosaic Node V1 [23]	CodeFirst [12]
Mainboard	LinkIt ONE	Arduino UNO	No Solutions
Shield	Base Shield V1.3	Base Shield V1.3	
	-	GPRS Shield V2.0	
	-	SD Card Shield V4	
Peripheral	Dust Sensor (PPD42NS)	Dust Sensor (PPD42NS)	
	Grove Temperature and Humidity Sensor (DHT11)	Grove Temperature and Humidity Sensor (DHT11)	
	-	Grove GPS (SIM28)	
Total Cost	\$89.70	\$150.99	N/A

for UNO and is negligible for the other three. For UNO, the overhead of all benchmarks are within 171B, 8.35% of its RAM size (i.e., 2KB). For ONE, the overhead of all benchmarks is within 21KB, 1.03% of its RAM size (i.e., 2MB). For BBB and RP2, the overhead of benchmarks is within 100KB, which are negligible since their RAM are 512MB and 1GB, respectively.

6.3 Case Study

Smart Houseplant Node. This case has been described in Section 2. Figure 2 shows the application code. Under the criterion of the lowest cost, the best hardware configurations for both TinyLink and CodeFirst [12] are shown in Table 8. In the TinyLink solution, the Base Shield V1.3 is selected, because it can provide ports (required by the Grove Light Sensor) which UNO does not contain. The fixed platform of CodeFirst, BetaBlocks, contains ports so that it does not need shields. It is obvious that the cost of TinyLink's solution is much lower than the cost of CodeFirst's solution. This is because BetaBlocks's characteristic of only using ports limits the searching space for feasible solutions. The assembled node of TinyLink is shown in Figure 14(a).

Air Quality Node. Mosaic is a mobile sensor network system for city scale sensing [23]. The requirement of a mosaic node is measuring PM2.5, PM10, position, temperature and humidity every 30 seconds. Then the sampled data are uploaded to a cloud server via GPRS. Besides, output log data are stored in the SD card for further debugging and developing.

We intend to build such nodes using both TinyLink and CodeFirst. The generated hardware configurations are shown in Table 9. Mosaic Node V1 is the original node used in [23], which is very expensive. TinyLink generates a hardware configuration that reduces about 41% of the total cost, due to the built-in components on LinkIt One. In Table 10, we can observe that TinyLink achieves all functionalities provided by Mosaic Node V1. CodeFirst cannot generate a feasible solution, since there are no suitable interfaces for storage modules. The assembled node of TinyLink is shown in Figure 14(b).

Voice Controlled LED Lamp Node. TinyLink is suitable for building smart home applications. We intend to control home appliances (e.g., LED lamps) via smart devices which take voice commands as input. We have implemented a voice controlled LED

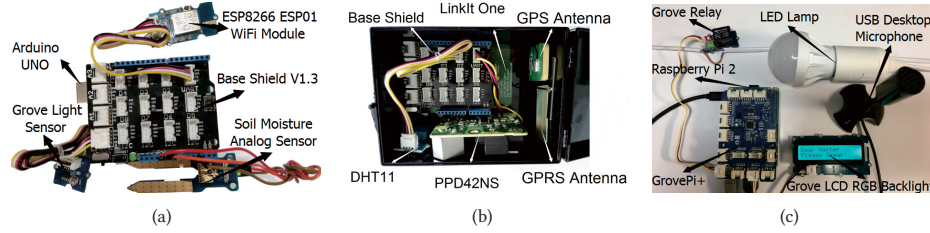


Figure 14: Assembled TinyLink nodes for three real-world case studies.

Table 10: Functionalities of implemented air quality nodes.

Functionality	TinyLink	Mosaic Node V1 [23]
PM2.5 Monitoring	✓	✓
PM10 Monitoring	✓	✓
Temperature Monitoring	✓	✓
Humidity Monitoring	✓	✓
GPS Sampling	✓	✓
Storing Data to the SD Card	✓	✓
Uploading Data to the Cloud	✓	✓
Periodically Executing Tasks	✓	✓

Table 11: Hardware configurations of voice controlled LED lamp nodes.

	TinyLink	CodeFirst [12]
Mainboard	Raspberry Pi 2	No Solutions
Shield	GrovePi+	
Peripheral	Grove Relay	
	USB Desktop Microphone	
	Grove LCD RGB Backlight	
Total Cost	\$89.20	N/A

lamp node via TinyLink. The source code only contains 51 lines of code. The requirements extracted from the code are recording user's voice periodically using a sampling rate of 16kHz, recognizing voice commands (e.g., "Light On" and "Light Off"), printing commands on an LCD screen and controlling the corresponding home appliance.

Table 11 lists the generated hardware configurations. The hardware configuration for TinyLink contains a USB desktop microphone directly attached to a USB port on a Raspberry Pi 2, a Grove relay module and a Grove LCD RGB backlight module which are attached to ports on a shield, GrovePi+. Figure 14(c) shows the assembled node. There are no solutions for CodeFirst because its mainboard lacks interfaces for voice recording modules.

The library implementations for the recording API and the voice recognition API are based on PocketSphinx, a CMU's speaker-independent continuous speech recognition engine [24].

7 DISCUSSION

In this section, we discuss several important open issues.

Optimization Criteria. Currently, TinyLink's optimization criterion is to minimize the cost. However, other optimization criteria (e.g., extensibility, energy consumption, etc.) can also be adopted. For extensibility, it is possible to define an extensibility metric as the number of non-occupied interfaces which could be used in the future. For energy consumption, however, it is a non-trivial task to estimate accurately for a particular platform during design-time given the application code. One possible approach for optimizing energy consumption is described as follows. First, we should estimate the execution time and energy consumption of each TinyLink API for different components on different platforms by well-designed benchmarks (or values reported in

datasheets). Second, we should carefully analyze the application code for information such as which TinyLink APIs it has invoked. Using the above information, we can roughly estimate the energy consumption of the entire code for different components on different platforms. Finally, we can look for the platform with minimum energy consumption by setting energy consumption as an optimization criterion and other factors like the cost as constraints.

TinyLink's Extensibility. Due to TinyLink's high level abstraction, the application code is hardware-independent. TinyLink can incorporate other MCUs by including their features into the hardware database as well as glue code for the new platforms. Therefore, other MCUs (e.g., MSP430) can also be supported by TinyLink in the future. TinyLink uses COTS hardware components and deals with the issues on how to assemble them in a compatible way. In the current stage, we think that TinyLink is primarily targeting for rapid prototyping. We think that the TinyLink's approach for rapid prototyping can have important implications and guidance to the final production.

Code Migration. For developers who are familiar with embedded systems (especially the hardware), code transformation is indeed a good feature for porting their existing code from one platform to another. For other developers, we believe that TinyLink is a good approach since it significantly reduces the developing efforts. With TinyLink abstraction, the code is hardware-independent and is also easy to migrate between platforms with the cross-compilation on the cloud. Moreover, TinyLink reuses the Arduino programming style heavily to prevent a steep learning curve. Since code transformation is another direction of work, we will consider it as possible future work of TinyLink.

8 RELATED WORK

TinyLink borrows heavily from three large areas of prior work: hardware platform design, hardware/software co-design and IoT rapid development. Each of these topics has made key implications which TinyLink incorporates into its design and structure.

Hardware Platform Design. With the development of MEMS technology, we have seen significant advances in the hardware industry. Early in 2003, the well-known sensor network platform, Telos, was released along with the TinyOS operating system by researchers at UC Berkeley [25].

The most important considerations for a sensor network platform are energy efficiency and small form factor. The design of such dedicated, small platform is difficult and requires expert knowledge. For example, Dutta *et al.* present a building block approach [6] to hardware platform design based on a decade of collective experience, arriving at an architecture in which general-purpose modules that incorporate commonly-used functionality are integrated with application-specific carriers that satisfy the

unique sensing, power supply, and mechanical constraints of an application.

Andersen *et al.* present the design of a new hardware and software platform [26] by bringing together Mobile, Wearable, Maker and Wireless Sensor Network technologies to achieve a high degree of synergy and energy efficiency.

Sutton *et al.* present a design methodology [27] for adaptive event-triggered acoustic sensing systems, with the careful consideration for adaptability, responsiveness, and energy efficiency. All the work aims to design an energy-efficient hardware platform with good system performance. However, *expert knowledge* on the hardware experience and embedded software have created a barrier for upper-layer application development. TinyLink intends to remove this barrier by incorporating hardware knowledge into our system design.

Recent years have also witnessed significant innovations in the Maker domain around Arduino [28] and Raspberry Pi [29]. These platforms are characterized by powerful processors with rich interfaces. This development trends have important implications for our designs: (1) the COTS hardware components and their compositions have already enabled a wide range of innovative applications; (2) we need to carefully consider the complex connections among hardware components and the API implementations on different platforms.

Hardware and Software Co-design. Hardware/software co-design approaches use synthesis strategies to generate efficient hardware and software partitions.

The most relevant to our work is Platform Based Design (PBD) [30]. The goal of PBD is to select and compose a subset of available components (i.e., the configuration), from the design space of possible platform configurations. A good and implementable configuration satisfies the system constraints as well as all component-imposed assumptions. PBD is a meet-in-the-middle process which maps an abstracted top layer description to a more detailed lower layer implementation as well as building a platform by defining the library that allows performance estimation.

Peter *et al.* propose a component-based description language CoDeL [31] that allows system designers to express components as reusable building blocks of the system with parameterizable models, properties, and interconnectivity. The automatic mapping from the higher layer to the lower layer is the process of exploration of complex design spaces which is frequently fostered by satisfiability (SAT) [32] solvers or Integer Linear Programming (ILP).

Similar to PBD, TinyLink also adopts an automatic mapping process from the application logic to a concrete hardware configuration. While PBD addresses a general problem, TinyLink addresses specific and practical challenges such as hardware-independent APIs, compatibility among COTS hardware components, implementations of glue code on different platforms.

IoT Rapid Development. The rapid development of IoT systems have attracted much research attentions from both industrial and academic communities.

Shayne Hodge proposes a general idea [33] which is to build an architecture like that of a modern web application. At a very high level in this model, the IoT hardware sends data over a network (or the Internet) to the backend software for storage and processing. The user interface is built as a web application, which allows it to

be viewed on a broad range of devices. The details on how to build the IoT hardware, however, are not given.

The Matrix project [34] was launched on Kickstarter in 2015. The platform contains up to 15 sensors, so that users only need to create IoT Apps, without caring about the hardware. However, the platform is fixed, it is not flexible to create customized IoT Apps.

IOTIFY [35] is the virtualization engine for the IoT to help build IoT applications. IOTIFY reduces the complexity of dealing with hardware, and large networks by virtualizing IoT endpoints in the cloud. Different from IOTIFY, TinyLink generates real hardware components and application software.

.NET Gadgeteer [36] is an integrated platform from Microsoft to support developing with customized components. Developers can quickly build devices with the help of Gadgeteer modules, .NET Micro Framework and 3D design tools. Wio Link [37] is a similar approach which builds upon the Wio Link board. TinyLink focuses on how to rapidly develop IoT applications using a variety of COTS components, instead of using customized components.

A recent approach is the code-first design to prototype wearable devices [12]. The hardware configuration would be generated from the analysis of the application code, because one software module is directly mapped to one hardware component and hardware dependencies can be determined by analyzing software dependencies. However, it assumes a fixed mainboard, limiting its capability to embrace a diverse set of COTS components.

The design of TinyLink is much more complex since it supports different mainboards and provides pin level abstractions with a careful consideration for shields. TinyLink also includes software libraries to facilitate upper layer application programming.

9 CONCLUSION

In this paper we present TinyLink, a holistic system for rapid development of IoT applications. TinyLink uses a top-down approach for both hardware and software designs. Developers only need to specify key application logic with TinyLink APIs, without dealing with the underlying hardware. TinyLink takes their code as input, automatically generates the hardware configuration and the binary program executable on the target platform. We implement TinyLink and evaluate its performance using benchmarks and three real-world case studies. Results show that TinyLink achieves rapid development of IoT applications, reducing 52.58% of lines of code in average for implementations, while incurring acceptable overhead.

The future work of TinyLink includes two directions. First, we will further extend the supported hardware components of TinyLink. Second, we will extend TinyLink to other optimization goals, e.g., energy, performance and extensibility.

ACKNOWLEDGMENTS

This work is supported by the National Basic Research Program of China (No. 2015CB352400), the National Science Foundation of China (No. 61472360 and No. 61502417), Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang Provincial Key Research and Development Program (No. 2017C02044), and the Fundamental Research Funds for the Central Universities (No. 2017FZA5013). We thank all the reviewers and the shepherd for their valuable comments and helpful suggestions.

REFERENCES

- [1] Michael Chui, Markus Löffler, and Roger Roberts. The internet of things. *McKinsey Quarterly*, 2(2010):1–9, 2010.
- [2] Rui Luis Aguiar, Nora Benhabiles, Tobias Pfeiffer, Pablo Rodriguez, Harish Viswanathan, Jia Wang, and Hui Zang. Big data, iot,.... buzz words for academia or reality for industry? In *Proc. of ACM MobiCom*, 2015.
- [3] Biyi Fang, Qiumin Xu, Taiwoo Park, and Mi Zhang. Airsense: an intelligent home-based sensing system for indoor air quality analytics. In *Proc. of ACM UbiComp*, 2016.
- [4] Hsin-Ruey Tsai, Shih-Yao Wei, Jui-Chun Hsiao, Ting-Wei Chiu, Yi-Ping Lo, Chi-Feng Keng, Yi-Ping Hung, and Jin-Jong Chen. ikneebraces: knee adduction moment evaluation measured by motion sensors in gait detection. In *Proc. of ACM UbiComp*, 2016.
- [5] Mehrdad Hesar, Vikram Iyer, and Shyamnath Gollakota. Enabling on-body transmissions with commodity devices. In *Proc. of ACM UbiComp*, 2016.
- [6] Prabal Dutta, Jay Taneja, Jaemin Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensor networks. In *Proc. of ACM SenSys*, 2008.
- [7] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proc. of USENIX NSDI*, 2017.
- [8] Yi Gao, Wei Dong, Kai Guo, Xue Liu, Yuan Chen, Xiaojin Liu, Jiajun Bu, and Chun Chen. Mosaic: A low-cost mobile sensing system for urban air quality monitoring. In *Proc. of IEEE INFOCOM*, 2016.
- [9] Lan Zhang, Xiang-Yang Li, Wenchao Huang, Kebin Liu, Shuwei Zong, Xuesi Jian, Puchun Feng, Taeho Jung, and Yunhao Liu. It starts with igaze: Visual attention driven networking with smart glasses. In *Proc. of ACM MobiCom*, 2014.
- [10] Steve Hodges, Stuart Taylor, Nicolas Villar, James Scott, Dominik Bial, and Patrick Tobias Fischer. Prototyping connected devices for the internet of things. *IEEE Computer*, 46(2):26–34, 2013.
- [11] Gaoyang Guan, Wei Dong, Yi Gao, and Jiajun Bu. Towards rapid and cost-effective prototyping of iot platforms. In *Proc. of IEEE ICNP Workshop on Hot Topics in Internet of Things (Hot-IoT)*, 2016.
- [12] Daniel Graham and Gang Zhou. Prototyping wearables: A code-first approach to the design of embedded systems. *IEEE Internet of Things Journal*, 3(5):806–815, 2016.
- [13] Port Description of GrovePi+. <https://www.dexterindustries.com/GrovePi/engineering/port-description/>.
- [14] Rohit Ramesh and Prabal Dutta. Embedded development tools revisited: Verification and generation from the top down. *GetMobile: Mobile Computing and Communications*, 20(2):5–10, 2016.
- [15] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. Ip_solve 5.5, open source (mixed-integer) linear programming system, 2004.
- [16] Arduino Forum. <https://forum.arduino.cc/>.
- [17] Raspberry Pi Forum. <https://www.raspberrypi.org/forums/>.
- [18] Instructables. <http://www.instructables.com/>.
- [19] Electronic Hub. <http://www.electronicshub.org/iot-project-ideas/>.
- [20] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.
- [21] Gordon Henderson. Wiring Pi GPIO Interface library for the Raspberry Pi. <http://wiringpi.com/>.
- [22] Abhraneel Bera. Wiring/Arduino style library for BeagleBone Black Platform. <http://beagleboard.org/project/wiringBone/>.
- [23] Wei Dong, Gaoyang Guan, Yuan Chen, Kai Guo, and Yi Gao. Mosaic: Towards city scale sensing with mobile sensor networks. In *Proc. of IEEE ICPADS*, 2015.
- [24] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alexander I Rudnick. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing*, 2006.
- [25] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. 2005.
- [26] Michael P. Andersen, Gabe Fierro, and David E. Culler. System design for a synergistic, low power mote/ble embedded platform. In *Proc. of ACM/IEEE IPSN*, 2016.
- [27] Felix Sutton, Reto Da Forno, David Gschwend, Tonio Gsell, Roman Lim, Jan Beutel, and Lothar Thiele. The design of a responsive and energy-efficient event-triggered wireless sensing system. In *Proc. of ACM EWSN*, pages 144–155, 2017.
- [28] Arduino. <https://www.arduino.cc>.
- [29] Raspberry Pi. <https://www.raspberrypi.org>.
- [30] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.
- [31] Steffen Peter and Tony Givargis. Component-based synthesis of embedded systems using satisfiability modulo theories. *ACM Transactions on Design Automation of Electronic Systems*, 20(4):49:1–49:27, 2015.
- [32] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
- [33] Shayne Hodge. A Rapid IoT Prototyping Toolkit, 2016.
- [34] MATRIX. <https://www.kickstarter.com/projects/2061039712/matrix-the-internet-of-things-for-everyonetm>.
- [35] IOTIFY. <http://iotify.io/>.
- [36] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. .net gadgeteer: a platform for custom devices. In *Proc. of Pervasive Computing*, 2012.
- [37] Wio Link. <https://iot.seeed.cc/>.