Automatic Generation of IoT Device Platforms With AutoLink

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

***Abstract*—With the development of the Internet-of-Things (IoT) industry, developers are no longer content with just prototyping a valid system but eager to create a mature IoT system that explores low power consumption or high extensibility instead. In this article, we present AutoLink, an automatic generation system of IoT device platforms. Users may write AutoLink metaprogram with an expressive syntax to specify their diverse requirements (e.g., battery lifetime, interface extensibility, execution time, and cost) of the generated IoT device platform. Taking the metaprogram as an input, AutoLink automatically transforms it into corresponding optimization problems and generates the optimal hardware configuration that meets user requirements best. Toward this, AutoLink also offers a cross-platform, duty cycle-aware power model and a time model to estimate the lifetime and execution period of an IoT system. We implement AutoLink and evaluate its performance using real-world IoT applications. Results show that AutoLink generates the optimal hardware configuration that meets diverse user requirements. Moreover, AutoLink achieves superior power estimation accuracy of IoT device platforms compared with the state-of-the-art approach.**

***Index Terms*—Energy, Internet of Things (IoT), rapid development.**

I. Introduction

THE PAST several years have witnessed the rapid development of Internet of Things (IoT) technologies. IDC forecasts that the number of IoT devices will reach 41.6 billion by 2025 [1]. Nevertheless, it is reported by Garter that IoT adoption is off to a slow start and 75% of IoT projects will take up to twice as long as planned [2]. Among several roadblocks, such as security and budget, the technical complexity of application development is voted as the primary barrier for IoT adoption, according to a recent survey by Microsoft [3].

An IoT application runs on top of the IoT device platform. An IoT device platform is a specially designed embedded system that typically consists of a microcontroller unit (MCU), sensors, communication modules, and many other peripherals. To date, there lacks a general-purpose IoT device platform since the device platforms are usually tightly coupled to their applications [4], [5].

It is challenging to build an appropriate IoT device platform for a specific application, especially for nonexperts in the embedded systems domain. First, the number of device platforms and components has significantly increased recently. For example, besides resource-constrained MCUs like the ATMega series, there exist resource-abundant MCUs like the ones on Raspberry Pi. There are also a large number of peripherals with different characteristics. It is thus difficult to make appropriate design choices from such an ample space of hardware components. Second, application developers may have diverse requirements on the IoT device platform. Low cost is preferred in some applications, while in some other implementations, low energy consumption is preferred.

The above difficulties could be tackled by an *automatic* generation system of IoT devices. Such a system may have a database containing key metrics of various hardware components. Taking the application code as well as the requirements as input, the system can *automatically* generate the *optimal* hardware configuration of the IoT device, i.e., the list of hardware components as well as their connections.

Initial efforts have been spent on the design of such an automatic generation system. Embedded design generation (EDG) approach [6] exploits high-level abstractions to lower the threshold of embedded design. TinyLink [7] advocates a *top-down* approach to make users focus on application logic other than hardware selections. The existing work cannot meet diverse application requirements. For example, EDG only generates valid hardware configuration while TinyLink generates the cost-optimal solution.

In this work, we aim to design an automatic IoT device generation system to meet various application requirements, including cost, lifetime, and extensibility. Nevertheless, designing such a system faces *nontrivial* challenges.

First, how to design an *expressive* language to allow users to express different requirements. For example, the user intends to build a smart houseplant monitoring system powered by the battery and requires that our system could provide an IoT device platform with maximum battery lifetime.

Second, how to incorporate lifetime into consideration. Different from cost, the accurate estimation of lifetime depends on not only the selected hardware components but also the input user code. Our system should consider not only hardware power profiles but also fine-grained code characteristics.

Manuscript received March 4, 2020; revised May 21, 2020 and July 12, 2020; accepted October 16, 2020. Date of publication October 22, 2020; date of current version March 24, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1600700; in part by the National Science Foundation of China under Grant 61772465; and in part by the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under Grant LR19F020001. *(Corresponding author: Wei Dong.)*

The authors are with the College of Computer Science and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University, Hangzhou 310000, China (e-mail: borui.li@zju.edu.cn; dongw@zju.edu.cn).

Digital Object Identifier 10.1109/JIOT.2020.3033130

We propose AutoLink, an automatic IoT device platform generation system, to address the above challenges. With Autolink, users can specify their requirements of the generated device platforms, such as lifetime, cost or real-time constraints in AutoLink syntax. Take the metaprogram as input, AutoLink automatically expresses the objectives and constraints in a mathematical manner and transforms user requirements to an optimization problem. By solving the problem, AutoLink generates the most appropriate device platform. AutoLink mainly targets on the nonexperts to help them develop an optimized IoT application efficiently. Moreover, AutoLink also provides an interactive visualizer, which allows professional IoT developers to gain insight into the performance of their application code and the hardware they selected in terms of power, extensibility, etc., which could be a guidance to improve their software/hardware design.

We implement AutoLink and perform extensive evaluations using benchmarks and real-world IoT applications. Results show that: 1) AutoLink automatically generates the most appropriate hardware configuration that meets various user requirements and 2) the power estimation accuracy of AutoLink exceeds 97%, which is superior compared with the state-of-the-art Amulet [8] and the time estimation accuracy exceeds 90% in the worst cases.

The contributions of this article are summarized as follows.

1) We present AutoLink, an automatic IoT device platform generation system considering multiple criteria, including cost, lifetime, and extensibility. With AutoLink, developers can express diverse application requirements so the resultant IoT device can meet application-specific needs.
2) We formulate the device platform generation problem as an optimization problem. Then, propose a time and a duty cycle-aware power estimation approach that both consider the impact of different device combinations and the dramatic contrast between idle and active power, which is considerable due to the intermittent wake up of typical IoT applications.
3) We implement the AutoLink system and carefully evaluated the performance of the generated device platform with real-world cases. Moreover, we evaluated the time and power estimating accuracy with benchmarks. Results show that AutoLink generates the optimized solution for the IoT device platform, meanwhile satisfying various user requirements.

The remainder of this article is structured as follows. Section II introduces the related works about IoT systems synthesis and energy modeling. Section III introduces the usage of the AutoLink system. Section IV presents AutoLink's formulation of the optimization problem. The essential compounds of AutoLink and the dynamic constraint generation system are detailed in Sections V and VI. In Section VII, we describe the implementation details and show the evaluation results of our device platform generation system. Section VIII discusses some important open issues, and finally, Section IX concludes this article and describes the future directions of AutoLink.

II. Related Work

During the development of IoT applications, the requirement of adequate software and hardware knowledge raises the threshold for interested amateur developers and stalls the widespread IoT deployment. To alleviate the problem, researchers in both industrial and academic communities dedicate themselves to the computer-assisted IoT system synthesis topic.

TinyLink [7], CodeFirst [9], and EDG [6] aim to make hardware synthesis more accessible to nonexperts. TinyLink formulates the synthesis problem as an integer linear programming (ILP) problem and enables developers to explore a more extensive hardware prototype space. CodeFirst uses the dependencies between software libraries and hardware components to infer necessary hardware components. EDG approach uses block diagrams represent software control logic and hardware component, formulates the synthesis problem as a satisfiability problem, and leverages satisfiability modulo theories (SMT) solver, Z3 [10], to solve it.

Outside the area of IoT, a tool that has been developed and adopted by DEC company for configuring computer systems is XCON, which is based on an expert system research project named R1 [11].

Different from the aforementioned approaches, we not only focus on synthesizing a *valid* platform but also a *well-performed* one from the perspective of power consumption, timeliness and extensibility.

Furthermore, many techniques have been proposed to evaluate the performance of embedded or IoT systems. From the perspective of energy modeling and estimating, Quanto [12] and Amulet [8] present their solutions. Quanto introduces a fine-grained energy measurement framework of TinyOS. It instruments the hardware drivers to log system-wide power states and global energy consumption for every power state transition. Then, Quanto obtains the power draw of each state with a series of logs. Amulet wearable platform proposes an energy model to estimate the power consumption of the system for developers to get a complete view of how their application works.

Different from Quanto, AutoLink neither requires modification on hardware drivers nor runs the source code to obtain the power information. Comparing with Amulet, our work obtains a more fine-grained model by considering the impact of power fluctuation when different hardware plugs and co-works with each other.

III. AutoLink Usage

Before describing the details of AutoLink, we first present its usage. AutoLink builds on top of TinyLink [7], a rapid prototyping system for IoT applications. We choose TinyLink mostly because it adopts the Arduino programming style that is popular in IoT prototyping and the entire system is publicly ready-to-use.

Note that AutoLink can also be implemented on other hardware synthesis systems, as long as they own a mathematical formulation of the synthesis problem, such as EDG [6] and CodeFirst [9].

```
void setup() {
    Bluetooth.init("AutoLink");
}
void loop() {
    Temperature.read();
    double temp = Temperature.data();
    Light.read();
    double light = Light.data();
    if(temp>20){
      String sendback = "Light"+String(light);
      Bluetooth.send(sendback);
    }
    Time.delayMillis(5000);
}
```

Fig. 1. Application code using TinyLink Language.

Fig. 1 shows a code example of an IoT application developed with TinyLink for monitoring the temperature and ambient light of a houseplant. Users implement the application logic in two main functions. The `setup()` function is used to initialize the Bluetooth component while `loop()` is used to sample the sensor data and upload the data to the default server. TinyLink can generate the cost-optimal hardware configuration considering user constraints and hardware constraints.

We notice that in many practical scenarios, users not only require a cost-optimal solution but also have many other requirements on the hardware configuration. We can see from the following four cases how AutoLink can satisfy users' diverse requirements.

Case 1: Users require that the IoT device has the minimum cost. Users can write the following AutoLink metaprogram to meet this requirement:

```
min Cost
```

Case 2: Users require that the device platform has the minimum cost and its lifetime must be longer than ten days with an 8000-mAh battery. This requirement could be expressed with the following AutoLink metaprogram:

```
min Cost
Lifetime > 10 days &&
Battery = 8000 mAh
```

We can see that AutoLink's syntax allows users to specify additional constraints for the generated device platform.

Case 3: Users may want to maximize the lifetime with four additional constraints: 1) the cost is less than 100 USD; 2) the MCU board of the system is specified as Arduino Uno. 3) the system has real-time constraints, i.e., the execution time of `loop()` function must be no more than 5100 ms; and 4) the system has good extensibility, i.e., the number of remaining analog pins is no less than 4. The corresponding AutoLink metaprogram is shown as follows:

```
max Lifetime
Cost < 100 USD &&
Mainboard = "Arduino Uno" &&
Time.Loop <= 5100 ms &&
Pin.Analog >= 4
```

We can see that in addition to cost, users can also specify lifetime as the optimization goal. Moreover, AutoLink syntax also supports hardware (line #3), real-time (line #4), and extensibility constraints (e.g., in terms of the number of pins, line #5).

Case 4: Users can also specify logical operations among the constraints. For example, they can specify that: 1) the number of remaining ports should no less than four, and there is at least one analog port or 2) the number of remaining pins should no less than four, and there is at least one analog pin. Along with the other constraints, the corresponding AutoLink metaprogram is

```
min Cost
Lifetime > 10 day &&
Time.Loop <= 5100 ms &&
Port >= 4 && Port.Analog >= 1 ||
Pin >= 4 && Pin.Analog >= 1
```

AutoLink supports multiple conjunctions for users to express the complicated logical relationship between conditions.

Hence, developing an IoT application with AutoLink needs three inputs from users: an AutoLink metaprogram, an application code using the language of hardware synthesis system (e.g., TinyLink language for TinyLink [7] or ModularMiddleware language for CodeFirst [9]) and the user/hardware constraints.

The AutoLink metaprogram and the application code are used to generate AutoLink constraints and optimization criteria (e.g., cost, interface, time, and lifetime). These AutoLink constraints and optimization criteria, along with the user/hardware constraints, are fed into AutoLink solver to obtain the final output. With respect to the user/hardware constraints, for users' convenience, AutoLink automatically generates them with the application code by calling the exposed APIs of TinyLink system by default. Moreover, AutoLink also provides a command-line interface to support users for inputting the user/hardware constraints generated by other synthesis systems.

The results generated by AutoLink contain two parts: hardware configuration and application binary. The hardware configuration is the selected hardware manifest of the uploaded user application, as Fig. 2 shows. With respect to the software binary, in order to set developers apart from complicated cross-compiling environments, AutoLink also automatically compiles user code into the executable binary of the chosen IoT device platform.

IV. Platform Generation Problem

A. IoT Device Platform

A typical IoT device platform (or hardware configuration) consists of many hardware components. According to their functions, the hardware components can be divided into three categories.

1) *Mainboard:* It contains key computational components of a system (e.g., MCU and ROM) and offers interfaces for peripherals.
2) *Shield:* A shield can be plugged into the mainboard to extend the number of interfaces. Moreover, some shields

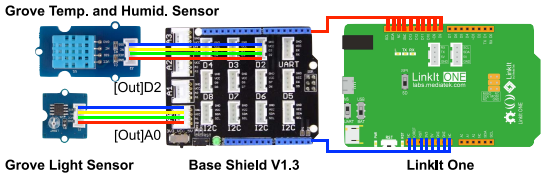


Fig. 2. Hardware connection output of case 2 in Section III.

provide functionalities (e.g., SD card shield) or even provide additional pins or ports with on-shield MCU (e.g., Grove Pi+).

3) *Peripheral:* A peripheral can connect to mainboards or shields through interfaces. Peripherals could be divided into sensing components (e.g., light sensor), communication components (e.g., BLE and WiFi), and controlling components (e.g., LED and relay).

Different hardware components can be connected via different interfaces. The interfaces can be divided into six categories (digital, analog, I2C, UART, PWM, and SPI) according to the communication protocols or two categories (*port* and *pin*) according to the physical appearance. Ports are encapsulations of pins. Hence, AutoLink includes a hardware database containing different kinds of hardware components and their characteristics.

B. Optimization Problem

We first introduce the notations used in this section.

1) $\mathbb{M}, \mathbb{S}, \mathbb{D}$*:* We use them to denote the set of mainboards, shields, and peripherals, respectively.
2) $\mathbb{C}, \mathbb{X}, \mathbb{T}, \mathbb{L}, \mathbb{P}$*:* We use them to denote the cost, extensibility, execution time, lifetime, and average power of the device platform.
3) d_i, c_i*:* We use d_i to denote the selection indicator, $d_i = 1$ indicates hardware component i is selected in the device platform. c_i is the price of component i. $i \in \mathbb{M} \cup \mathbb{S} \cup \mathbb{D}$.
4) $\mathbb{W}, \mathbb{I}$*:* We use $\mathbb{W}$={pin, port} to denote the set of physical encapsulations and $\mathbb{I}$={Analog, Digital, UART, I2C, PWM, SPI} to denote the set of protocols.
5) $\mathbb{X}_{w,I}$, $\mathbb{X}_{mcu,I}$*:* We use $\mathbb{X}_{w,I}$ to denote the remaining number of interface $w \in \mathbb{W}$ with protocol $I \in \mathbb{I}$. Due to the available interface number also restricted by the MCU-supported pin [7], we use $\mathbb{X}_{mcu,I}$ to denote the remaining MCU-supported pin of protocol $I \in \mathbb{I}$. For example, $\mathbb{X}_{pin,I2C}$ and $\mathbb{X}_{mcu,I2C}$ are the number of remaining physical and MCU pins of protocol I2C.
6) $\mathbb{X}^+_{i,w,I}$, $\mathbb{X}^-_{i,w,I}$, $\mathbb{X}^+_{i,mcu,I}$, $\mathbb{X}^-_{i,mcu,I}$*:* We use $\mathbb{X}^+_{i,w,I}$ and $\mathbb{X}^-_{i,w,I}$ to denote the interface number of encapsulation w and protocol $I \in \mathbb{I}$ provided/consumed by hardware i. Similarly, $\mathbb{X}^+_{i,mcu,I}$ and $\mathbb{X}^-_{i,mcu,I}$ denote MCU-supported pin number of protocol $I \in \mathbb{I}$ provided/consumed by hardware i. For example, $\mathbb{X}^+_{i,pin,I2C}$ (or $\mathbb{X}^-_{i,pin,I2C}$) and $\mathbb{X}^+_{i,mcu,I2C}$ (or $\mathbb{X}^-_{i,mcu,I2C}$) are the number of provided(or consumed) I2C pins and MCU-supported I2C pins of hardware component i.
7) U, $E_{battery}$*:* We use U to denote the code set of the input user application and E_{battery} to denote the battery capacity when calculating lifetime.

Problem Formulation: An AutoLink metaprogram is typically transformed into the following optimization problem:

$$\begin{aligned} &\textbf{Find the values of } d_i \ (\forall i \in \mathbb{M} \cup \mathbb{S} \cup \mathbb{D}) \\ &\max \,/\, \min \quad \text{Obj}_1 \\ &\text{s.t.} \quad \begin{cases} \text{User Constraints} \\ \text{Hardware Constraints} \\ \text{Obj}_2 \quad \geq, \leq \quad \text{Req}_2 \\ \dots \\ \text{Obj}_n \quad \geq, \leq \quad \text{Req}_n \end{cases} \end{aligned} \tag{1}$$

where user and hardware constraints are generated according to the constraints in the adopted hardware synthesis technique (TinyLink system in our implementation). $\{\text{Req}_2, \dots, \text{Req}_n\}$ in (1) are specified in the AutoLink metaprogram. Each *Obj* is among the four optimization criteria: {Cost, Extensibility, Time, Lifetime}.

Definition 1 (Cost): We define cost as the sum of each component's price in an IoT device system. For example, objective `min Cost` can be expressed as

$$\min \ \mathbb{C}(d_i) = \sum_{i \in \mathbb{M} \cup \mathbb{S} \cup \mathbb{D}} c_i d_i. \tag{2}$$

Definition 2 (Extensibility): We define extensibility as the remaining interfaces of the generated device platform.

Similar to cost, extensibility could also be expressed with selection indicator d_i since the interface number of component i is constant. For example, the constraint `Pin.I2C>k` could be transformed into two constraints

$$\begin{cases} \mathbb{X}_{pin,I2C} = \sum_{i \in \mathbb{M} \cup \mathbb{S} \cup \mathbb{D}} \left(\mathbb{X}^+_{i,pin,I2C} - \mathbb{X}^-_{i,pin,I2C} \right) d_i > k \\ \mathbb{X}_{mcu,I2C} = \sum_{i \in \mathbb{M} \cup \mathbb{S} \cup \mathbb{D}} \left(\mathbb{X}^+_{i,mcu,I2C} - \mathbb{X}^-_{i,mcu,I2C} \right) d_i > k \end{cases} \tag{3}$$

due to the pin number is restricted by both physical and MCU-supported pin numbers [7].

It is worth noting that remaining memory could also be a measurement of extensibility, and we will consider it in future work.

Definition 3 (Execution Time): We define it as time used for executing one iteration of a specific function.

For example, metaprogram `Time.Loop <= 5100 ms` represents the time of execute `loop()` function once (denoted as $\mathbb{T}$) should be no more than 5100 ms. Apparently, $\mathbb{T}$ not only depends on the hardware d_i but also the input user code U.

Definition 4 (Lifetime): We define the lifetime as the live duration of the IoT device platform powered by a battery with fixed capacity.

For example, objective `max Lifetime` could be converted as

$$\max \ \mathbb{L}(d_i, U) = E_{\text{battery}} / \mathbb{P}(d_i, U). \tag{4}$$

In AutoLink, we set the default E_{battery} as 10000 mAh if users not specified in the AutoLink metaprogram. Hence, the question of estimating $\mathbb{L}$ turns into predicting $\mathbb{P}$. Similar to execution time, power consumption (i.e., lifetime) also not only depends on d_i but also depends on U.

We say both cost and extensibility as *static criteria* since they are independent of the user program. However, it is challenging to express the execution time and lifetime since these

two metrics depend not only on the hardware components but also on the input application code. Take the lifetime as an example, for an application with 99% duty cycle, an MCU with low active power may be selected. In comparison, for a 1% duty-cycled application, an MCU with low sleep power is better. We say, both time and lifetime as *dynamic criteria*.

It is also worth noting that the optimization problem changes drastically with the addition of lifetime and time constraints. Cost, extensibility, and user and hardware constraints can all be expressed as linear functions of d_i, whereas the time and lifetime constraints are nonlinear functions of d_i. Hence, traditional ILP solvers cannot be reused for our problem and how to efficiently solve the complicated problem is another challenge of AutoLink.

We will address the above two challenges in Sections VI and VII-A, respectively.

V. System Overview of AutoLink

In this section, we will give a bird's-eye view of our hardware platform generation and optimization system.

Fig. 3 depicts the system architecture of AutoLink. AutoLink metaprogram serves as the input of two criteria generation systems.

1) In the static criteria generation system, besides the metaprogram, the static criteria generator also takes the cost and number of pins and ports of each component as input. It then generates cost and interface objectives and constraints with (2) and (3).
2) In the dynamic criteria generation system, our estimation model takes time and power profiles of hardware components as well as user code as input, then generates the estimated execution time and power. Combined with metaprogram, the estimated expressions of time and power finally transform into time and power constraints.

Finally, AutoLink solver takes the outputs of the two generation systems as well as user and hardware constraints as input, then generates the device platform (i.e., d_i). Our hardware database contains general information and the necessary properties used in the two systems, namely, the name of the hardware, cost, number of pin/port consumed/provided, and functionality.

VI. Dynamic Criteria Generation System

This section describes how AutoLink transforms dynamic criteria, time and lifetime, in the metaprogram into mathematical expressions.

We first introduce the notations used in this section.

1) t_{API}, t_ℓ, t_{idle}: We use them to denote the time of executing APIs, non-API code, and idle time in the loop() function.
2) f, F, u, ℓ: We use $f \in F$ and $u \in \ell$ to denote an API and a line of non-API code, where F and ℓ denote the set of APIs and non-API code. $F \cup \ell = U$.
3) $\mathbb{S}_f$, $\mathbb{D}_f$: We use $\mathbb{S}_f$ and $\mathbb{D}_f$ to denote the shield and peripheral set that provide API f.
4) β_f, β_u: We use β_f and β_u to represent the path-weighted execution count of f and u, which is designed to capture the influence of different execution path in the time and power estimation model.
5) $t_{i,j,f}$, $t_{i,u}$: We use $t_{i,j,f}$ to denote the execution time of API f on mainboard i and peripheral j. $t_{i,u}$ is the execution time of non-API code u on i.
6) $\mathbb{P}_i(U)$, $\mathbb{P}_{i,j}(U)$: We use $\mathbb{P}_i(U)$ to denote the power consumed by mainboard i, and $\mathbb{P}_{i,j}(U)$ is the power consumed by component j on mainboard i, which reflects the electrical characteristics of different hardware combinations.
7) $k, \mathbb{K}_i, \mathbb{K}_i^{\text{active}}, \mathbb{K}_i^{idle}$: We use $\mathbb{K}_i^{\text{active}}$ and $\mathbb{K}_i^{idle}$ to denote the set of idle and active states, $k \in \mathbb{K}_i = \mathbb{K}_i^{\text{active}} \cup \mathbb{K}_i^{idle}$.
8) μ_i^k, $\mu_{i,j}^k$: We use μ_i^k and $\mu_{i,j}^k$ to denote the duty cycle of component i (or component i on mainboard j) at power state k.
9) P_i^k, $P_{i,j}^k$: We use P_i^k and $P_{i,j}^k$ to denote the power of component i (or component i on mainboard j) at power state k.

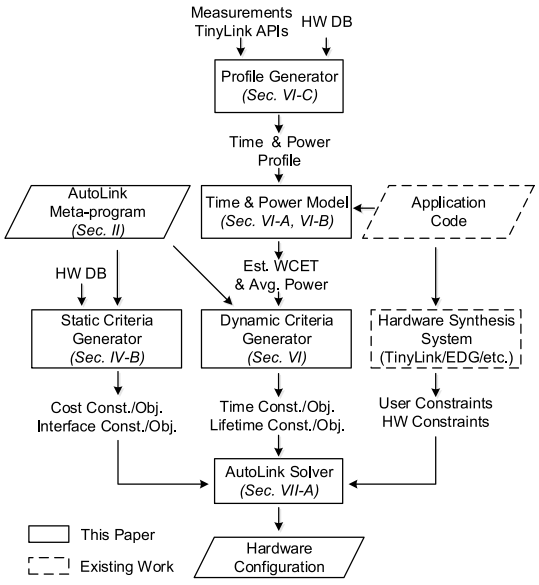


Fig. 3. System architecture of AutoLink.

The execution time could be expressed as the sum of the execution time of active state and idle state. Furthermore, the active time can be divided into the time of executing APIs (t_{API}) and non-API code (t_ℓ). For example, objective `min Time.Loop` is transformed into

$$\min \ \mathbb{T}(d_i, U) = t_{\text{API}}(d_i) + t_\ell(d_i) + t_{\text{idle}}(U). \tag{5}$$

The lifetime could be converted to power consumption with (4). Since different types of mainboards own different electrical characteristics (e.g., resistance), components exhibit different power when they plugged onto different mainboards. Hence, we obtain the average power $\mathbb{P}(d_i, U)$ of an IoT device platform as

$$\mathbb{P}(d_i, U) = \sum_{i \in \mathbb{M}} \mathbb{P}_i(U) d_i + \sum_{i \in \mathbb{M}} \sum_{j \in \mathbb{S} \cup \mathbb{D}} \mathbb{P}_{i,j}(U) d_i d_j. \tag{6}$$

The complete dynamic criteria generation toolchain of AutoLink is illustrated in Fig. 4. In the rest of this section, we, respectively, describe our time and power estimation

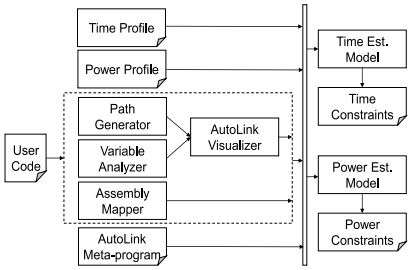


Fig. 4. Dynamic constraints generation toolchain.

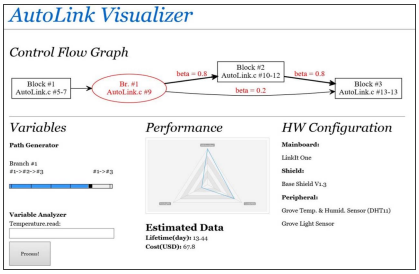


Fig. 5. Screenshot of AutoLink Visualizer.

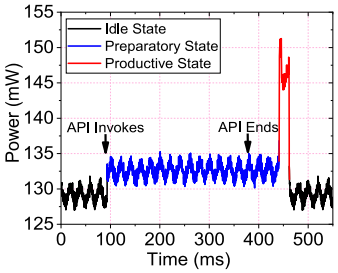


Fig. 6. Illustration of power states of a temperature read API.

model that fully exploits cross-hardware characteristics in Sections VI-A and VI-B, then describe the automatic profile generation system in Section VI-C.

A. Time Estimation Model

AutoLink's duty cycle-aware power model builds upon our time model; hence, we first describe how AutoLink estimates t_{API}, t_ℓ, and t_{idle} in (5). t_{API} and t_ℓ can be expanded with d_i as

$$\begin{cases} t_{\text{API}}(d_i) = \sum_{f\in F}\sum_{j\in\mathbb{S}_f\cup\mathbb{D}_f}\sum_{i\in\mathbb{M}} d_i d_j \beta_f t_{i,j,f} \\ t_\ell(d_i) = \sum_{i\in\mathbb{M}}\sum_{u\in\ell} d_i \beta_u t_{i,u}. \end{cases} \tag{7}$$

β_f and β_u represent the fraction of different execution paths of branches or loops in the user code. Through AutoLink Visualizer, as Fig. 5 shows, developers can adjust the execution ratio of each branch in the whole program control-flow graph which is generated by AutoLink Path Generator, implemented with ANTLR.

$t_{i,j,f}$ is the execution time of API f on mainboard i and peripheral j. It varies among different <i, j, f> tuple mainly due to different hardware pairs and data communication protocols. We will elaborate on the automatic generation of $t_{i,j,f}$ in Section VI-C. Nevertheless, the execution time of some APIs depends on the size of input parameters, such as `Bluetooth.send`. Toward this, we introduce AutoLink Variable Analyzer, a taint analyzer built on the top of Valgrind [13], to mark the parameter as taint sink, propagates backward and calculates the size of the taint source. If the taint source depends on external input (e.g., Fig. 1 line #11), AutoLink utilizes Visualizer to interact with developers. It is worth noting that both the time and power estimating models capture the ideal network conditions without consideration for different connection parameters as well as environmental disturbances by default. AutoLink can be extended to capture different connection parameters. This is because the AutoLink Variable Analyzer can capture the parameters specified in the corresponding TinyLink APIs and use a prebuilt time model to capture the impact of the parameters. We take LoRa, a widely used long-range communication technology, as an example. There are three important parameters that affect the bitrate of LoRa: spreading factor (SF), coding rate (CR), and signal bandwidth (BW). For example, the SF could be set to `x` with API `rf95.setSpreadingFactor(x)`. The AutoLink Variable Analyzer can analyze the specific value of `x`. Finally, the data rate (DR) can be calculated with equation $\text{DR} = \text{SF}(\text{BW}/2^{\text{SF}})\text{CR}$ [14]. The time criterion can thus be calculated according to (7). Similarly, AutoLink can also be applied to other short-range technologies, such as BLE [15] and 802.15.4 [16]. With regard to environmental disturbances, our system allows the developers to specify a network coefficient in the AutoLink Visualizer if the developer has a good estimation of network condition.

$t_{i,u}$ depends on the chosen mainboard due to the different MCU frequency and instruction set. Nevertheless, it is inaccurate to profile and model at the high-level programming languages, such as C due to the vast set of C grammar and versatile semantic. Therefore, different from Amulet [8], we go further to the assembly level. The stiff nature of assembly could increase the granularity of our time model; hence, we implement AutoLink Assembly Mapper for mapping user code to assembly. Then, we model the execution time of non-API code using profiles containing instruction cycles and architecture-specific metrics, such as MCU throughput (MIPS/MHz) and frequency, for each mainboard to address the different performance induced by architectural design.

B. Power Estimation Model

Building upon the time model, we propose our *duty cycle-aware* power estimation model. With fully aware of duty cycle, $\mathbb{P}_i(U)$ and $\mathbb{P}_{i,j}(U)$ in (6) could be expanded as

$$\begin{cases} \mathbb{P}_i(U) = \sum_{k\in\mathbb{K}_i} \mu_i^k(U) P_i^k, & i \in \mathbb{M} \\ \mathbb{P}_{i,j}(U) = \sum_{k\in\mathbb{K}_j} \mu_{i,j}^k(U) P_{i,j}^k, & i \in \mathbb{M}, j \in \mathbb{S}\cup\mathbb{D}. \end{cases} \tag{8}$$

P_i^k (or $P_{i,j}^k$) is the power of mainboard i (or peripheral i on mainboard j) at power state k. Due to our preliminary experiment, as Fig. 6 shows, invoking an API could incur several power states of a specific peripheral. Even if the API ends, the peripheral may still stay in a tail energy state [17]. The same variation is observed in the non-API code. Hence, we argue that considering the variation of power states during an API call could achieve better accuracy in our power model. The time spent of each state, denoted as $\tau_{i,u}^k$ or $\tau_{i,j,f}^k$, is also

automatically generated and profiled in our power database as detailed in Section VI-C.

$\mu_i^k(U)$ (or $\mu_{i,j}^k(U)$) is the quotient of active and total time

$$\mu_i^k(U) = \sum_{u \in \ell} \beta_u \tau_{i,u}^k / \mathbb{T} \tag{9}$$

$$\mu_{i,j}^k(U) = \sum_{f \in \mathbb{F}_i} \beta_f \tau_{i,j,f}^k / \mathbb{T}. \tag{10}$$

Hence, the duty cycle of component i could be expressed as

$$\mu_i = \sum_{k \in \mathbb{K}_i^{\text{active}}} \mu_i^k. \tag{11}$$

C. AutoLink Profile Generation System

AutoLink profile generation system (APGS) automatically generates the profile used in our time and power estimation model. APGS is composed of Time- and Power-APGS.

Time-APGS: Assembly cycle mapping and MCU throughput index is fixed for a specific MCU architecture. We have already obtained the data of four architectures (AVR ATmega, ARM Cortex-A7, A8, and A53), which encompass most of the prevalent MCU architectures today. Developers only need to adjust the MCU frequency if the new mainboard shares the same architecture as existing profiles.

We profile API execution time by automatically instrumenting the API with a timestamp. To achieve better accuracy, we run test cases several times and take the average result into API time profile.

Power-APGS: Power profile records how API changes the power state ($P_{i,j,f}^k$) and the duration of each power state caused by invoking an API ($\tau_{i,j,f}^k$). Note that the endurance time of a specific power state when invoking an API is different from API execution time due to the tail energy period [17], as Fig. 6 shows. For a new hardware i that supports API set F_i, we designed a general benchmarking schema using the power trace obtained with Monsoon Power Monitor, which contains numerous pairs of timestamp and instantaneous power. AutoLink leverages the k-means clustering to automatically separate different power states and record the power and duration of each state. Parameter k could be obtained with the datasheet.

VII. Implementation and Evaluation

In this section, we give a brief description of AutoLink implementation details and present the evaluation results from various angles.

A. Implementation Details

We implemented the complete process of AutoLink illustrated in Fig. 3. In this section, we focus on the implementation of its solver.

Applying the dynamic criteria to the optimization problem presented in Section IV-B makes it as a mixed-integer nonlinear programming (MINLP) problem, where the integer variable stands for the hardware selection vector d_i. Therefore, the ILP solver of TinyLink is no longer sufficient. Various methods are proposed to solve the MINLP problem, such as optimal algorithms (e.g., brute-force and SMT [10] approaches) and heuristic solvers.

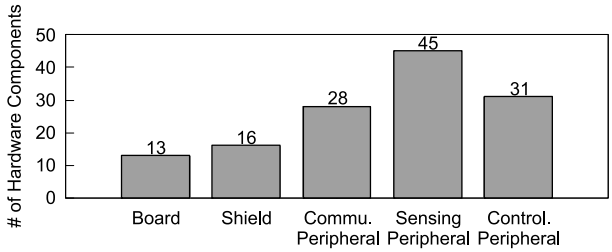


Fig. 7. Number of hardware components in AutoLink database.

Similar to the prior work EDG [6], the SMT solver Z3 [10] seems to be a plausible solver. Hence, we tailor Z3 with a step-by-step approaching strategy to solve our optimization problem. Nevertheless, AutoLink obtains over 200 pieces of hardware constraints and over 100 solution dimensions. The 10^4 solving space will enlarge the solving time to an unacceptable extent because the optimal algorithms endure triple-exponential time complexity. In our experiment, the SMT solution time increases from 0.0119 s to 63.52 min when the solving space grows from 50 to 10^4. Therefore, we adopt the memetic algorithm, a metaheuristic that exhibits high scalability (3.16 s for 10^4 solving space) and less probability to local optima, as the core of AutoLink solver. The algorithm is iterative, and each iteration contains mutation, local search, and recombination. We obtain the following parameters of the algorithms by referring to [18] and fine tuning with experiments: 1000 generations, 100 population size, and 0.01 mutation probability. Furthermore, AutoLink also supports a neural network-based approach to gain the result IoT device platform. Compared with the memetic approach, the neural network approach exhibits a shorter solving time while with lower accuracy. To build our system with more judicious, we use the memetic solver by default, and present a knob for users to switch between the default solver and the quicker neural network solver.

B. Evaluation Setup

Hardware Database: As Fig. 7 shows, we obtained 13 mainboards, including the mainstream ones, such as Arduino Uno (ARD for short), LinkIt One (LIO for short), Raspberry Pi (RPI for short), and BeagleBone (BBB for short), 16 shields, and over 100 peripherals in three categories.

Macrobenchmark: We selected and implemented five real-world IoT projects from popular IoT websites, such as `Hackster.io`, `Maker.pro` and `Instructables.com` as macrobenchmarks to validate the effectiveness of AutoLink. Table I summarizes the project names and main functionalities.

Microbenchmark: We selected five generally used functions in IoT applications [7] as microbenchmarks to evaluate the accuracy of time and power estimation method described in Section VI: 1) Temp. read; 2) Light read; 3) Humidity read; 4) Gyro read; and 5) BLE Send.

Case Study: We use three real-world case studies, which are a long-lasting smart houseplant node described in Section III,

TABLE I
MACROBENCHMARKS TO TEST THE EFFECTIVENESS OF AUTOLINK

#	Project Name	Main Functionalities
1	BLE Controllable RGB Light	BLE, LED
2	Automate Your Home	Relay, PIR, WiFi
3	POI Hunter	GPS, LCD, WiFi
4	Salmon Stream Monitor	Temperature,HCHO,CO_2,WiFi
5	Morse Translator	Button, LCD

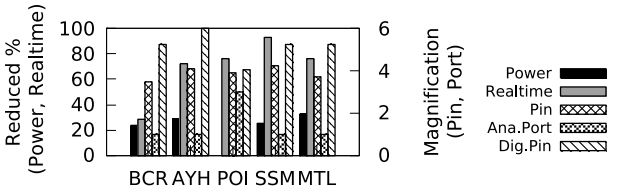


Fig. 8. Relative decreased percentage (for power and real time) and the magnification (for pin and port) compared with TinyLink [7].

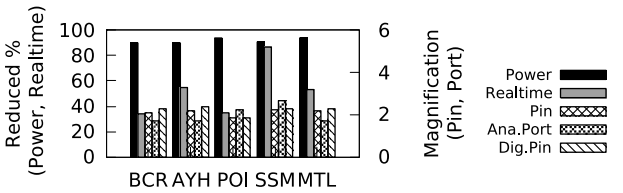


Fig. 9. Relative decreased percentage (for power and real time) and the magnification (for pin and port) compared with CodeFirst [9].

a highly extensible air quality monitoring node and a real-time toxic gas detection node in Section VII-D.

C. Overall Evaluation

To evaluate the overall optimization effectiveness of AutoLink, we rehearse each macrobenchmark with five pieces of AutoLink metaprogram individually, namely: 1) max lifetime; 2) min Time.Loop; 3) max Pin; 4) max Port.Analog; and 5) max Pin.Digital, and compare the performance of the synthesized device platform with TinyLink [7] and CodeFirst [9]. We use Monsoon Power Monitor [19] to measure the power consumption.

Figs. 8 and 9 illustrate the relative decreased percentage (for power and real time) and the magnification (for extensibility) compared to the device platform generated by TinyLink and CodeFirst. Note that the result of CodeFirst for one input is not exclusive. It is mainly due to CodeFirst focuses on generating a valid configuration rather than an optimized one. Each one could be CodeFirst's output if there are multiple possible results available. Hence, in Fig. 9, we averaged the performance of all possible solutions by CodeFirst by modifying the algorithm of CodeFirst to force it into generating all possible configurations. BCR, AYH, POI, SSM, and MTL denote the macrobenchmark #1–#5, respectively. We can observe that AutoLink generates the better-performed solution for the optimization goal specified in the metaprogram for most cases compared with state of the arts. The average percentages of decreased/increased performance among the five benchmarks are 21.99%, 69.21%, 3.89×, 1.4×, and 5.18× when compared with TinyLink, and 91.72%, 52.74%, 2.11×, 2.01×, and 2.21× when compared with CodeFirst. Combining the results shown in Figs. 8 and 9, we notice that the power reduction when compared with CodeFirst is much larger than those with TinyLink, which is mainly due to the multiple results of CodeFirst contain high power consumption, such as BBB and RPI. When minimizing the power of the POI project, the solution generated by AutoLink gains no improvement due to the cost-optimal and power-optimal solution results in the same hardware configuration, which contains the LIO mainboard (GPS and WiFi are onboard modules) with a Grove RGB Backlight LCD.

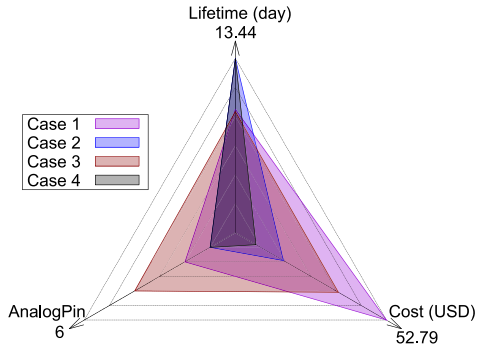


Fig. 10. Radar graph of the illustrated cases in Table I. Vertices of the triangle depicts the *optimal* value generated with the brute-force searching approach.

D. Case Study

Smart Houseplant Node: This case has been described in Section III. The generated hardware configuration is summarized in Table II (the lifetime is normalized with 10 000-mAh battery for comparison). We use a radar graph to illustrate the performance of different objectives as Fig. 10 and all data have been normalized with the maximum of each dimension.

Case 1 generates the *cost-optimal* solution. In the radar graph, the triangle of case 1 reaches the peak in cost dimension. While in lifetime or extensibility (i.e., analog pin) dimension, it is not the best hardware configuration. Case 2 adds a lifetime constraint; hence, the generated platform changes from ARD to LIO due to it exhibits lower power but costs more. Case 3 illustrates the user requires to use ARD and specifies a real-time constraint. Thus, AutoLink chooses Grove Temperature Sensor other than DHT11 to meet the time constraint due to the execution cycle of DHT11 (272 ms) is too long. Case 4 illustrates a more complicated constraint of port number. Compared with case 2, to meet the port number constraint, a digital light sensor is chosen for light function due to the number of analog MCU pin on LIO is only 3.

Compared with TinyLink, AutoLink generates different device platforms that meet user's demands best.

Highly Extensible Air Quality Node: Mosaic [5] is a mobile sensing network system deployed on buses to achieve city-scale air quality sensing. A Mosaic node measures PM2.5, PM10, GPS, temperature, and humidity data, uploads them to the cloud server through GPRS and saves system logs in SD card. Due to the UART and Digital pin extensibility of Arduino is low, Mosaic falls flat while adding some new functionalities

TABLE II
HARDWARE CONFIGURATIONS OF SMART HOUSEPLANT NODE GENERATED BY DIFFERENT AUTOLINK METAPROGRAM

	Case 1	Case 2	Case 3	Case 4
Mainboard	Arduino Uno	Linkit One	Arduino Uno	Linkit One
Shield	Base Shield V1.3	Base Shield V1.3	Base Shield V1.3	Base Shield V1.3
Peipherals	Grove Temp. Sensor	Grove Temp. & Humid. Sensor (DHT11)	Grove Temp. Sensor	Grove Temp. Sensor
	Grove Light Sensor	Grove Light Sensor	Grove Digital Light Sensor	Grove Digital Light Sensor
	Grove BLE V1.0	(built-in BLE module)	Grove BLE V1.0	(built-in BLE module)
Esti. Lifetime/day	9.52	13.44	9.33	13.42
Total Cost/$	52.79	67.8	59.79	71.8

TABLE III
HARDWARE CONFIGURATIONS OF AIR QUALITY NODE

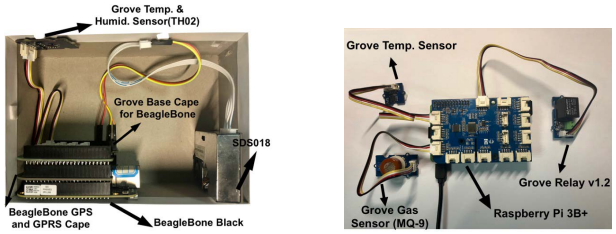
	AutoLink	Mosaic Node v1 [5]
Mainboard	BeagleBone Black	Arduino UNO
Shield	Grove Base Cape for BeagleBone	Base Shield v1.3
	BeagleBone GPS and GPRS Cape	GPRS Shield v2.0
	-	SD Card Shield v4
Peripheral	SDS018	Dust Sensor (PPD42NS)
	Grove Temp. & Humid. Sensor(TH02)	Grove Temp. & Humid. Sensor(DHT11)
	-	Grove GPS (SIM28)
#DigitalPin Left	65	10
#UARTPin Left	8	0

TABLE IV
HARDWARE CONFIGURATIONS OF TOXIC GAS DETECTION NODE

	AutoLink	TinyLink [7]
Mainboard	Raspberry Pi 3B+	Arduino UNO
Shield	GrovePi+	Base Shield v1.3
Peripheral	Grove Gas Sensor(MQ-9)	NGL07 Gas Sensor
	Grove Relay v1.2	Grove Relay v1.2
	Grove Temp. Sensor	Grove Temp. & Humid. Sensor(DHT11)
Loop Time/ms	286.79	1599.46

such as carbon dioxide sensing. Hence, we use a metaprogram as `max Pin.Digital, Pin.UART>3` to avoid the extensibility shortage. Table III lists the resulting hardware configurations and the number of digital and UART pins left of AutoLink and Mosaic. The hardware platform generated by AutoLink uses BeagleBone Black (other than Arduino Uno) mainly due to BBB has 10 UART pins and 69 digital pins, which is much more than ARD (UART 2, digital 14). The assembled AutoLink node is shown in Fig. 11(a).

Real-Time Toxic Gas Detection Node: Toxic gas such as carbon monoxide might be produced when heating with charcoal fire. We intend to create an application that could take immediate actions (e.g., open the windows with electromagnetic relay) when detected a harmful concentration of toxic gas. Hence, we use AutoLink metaprogram: `min Time.Loop`. Table IV lists the resulting hardware configurations of AutoLink and TinyLink. Comparing to the solution of TinyLink, AutoLink selects Raspberry Pi to gain a faster computation speed. The result of AutoLink mainly differs from TinyLink's in selecting Raspberry Pi (other than Arduino) and MQ-9 gas sensor (other than NGL07), which gain a faster computation speed. The toxic gas detecting and processing time reduced from TinyLink's 1599.46 to 286.79 ms. Fig. 11(b) illustrates the AutoLink node.



(a) (b)

Fig. 11. Assembled AutoLink nodes for two real-world case studies. (a) Highly exten. air quality node. (b) Real-time toxic gas dete. node.

E. Dynamic Constraints Estimating Evaluation

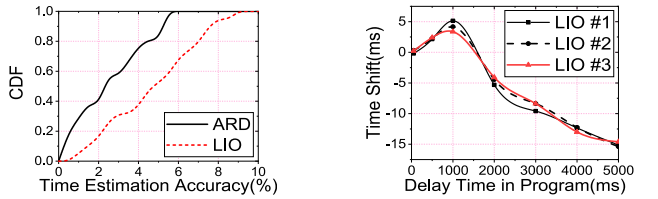
We implement five microbenchmarks in Section VII-B and four cases in Table II on both ARD and LIO mainboards.

Time Estimating Accuracy: As is illustrated in Fig. 12(a), the time estimation error is less than 6% on ARD and 10% on LIO. The extra error of LIO mainly derived from the *time shift* on LIO when invoking a delay API provided by its manufacturer. As Fig. 12(b) shows, the *time shift* relates to the parameter of delay API. The time shift mainly due to the Arduino Porting Layer of LIO builds on the top of its FreeRTOS system, which aims to support the delay API while its execution may be interrupted by RTOS driver.

Power Estimating Accuracy: To address the impact of duty cycle, we adjust delay time from 0 to 5000 ms of the test cases. Fig. 13(a) and (b) illustrates the estimated power and the average error of each benchmark on both mainboards and case studies. Compared to the state-of-the-art method Amulet [8], whose highest error is 9.7% and the average error is 5.8%, AutoLink power estimation exhibits better performance. The maximum error on ARD is less than 5% and the average error of all benchmarks and cases is 2.32%, which is outperformed than LIO (maximum: 7.50% and average: 5.06%). The reason for the improvement is our estimation model treats non-API code differently using Variable Analyzer and Assembly Mapper while Amulet treats it the same, and we model the power fluctuation when different hardware components cowork with each other. We note that estimation error on LIO is generally larger than ARD both in time and power evaluation. The reason is that the MCU autoscaling and background services such as watchdog on LIO leads to the fluctuation of execution time and power.

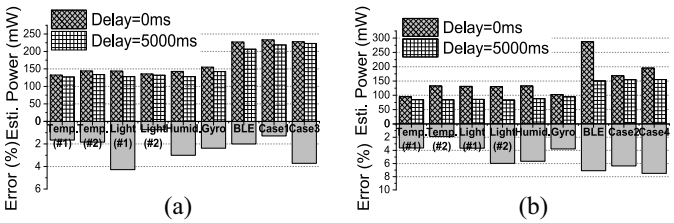
VIII. DISCUSSION

In this section, we discuss several open issues, point out limitations and identify the future work of AutoLink.



(a) (b)

Fig. 12. Time estimating evaluation results. (a) Time estimating accuracy of mainboard ARD and LIO. (b) LIO time shift. We tested on three LIO boards denoted as #1–#3.



(a) (b)

Fig. 13. Power estimating evaluation results. (a) Accuracy on ARD. (b) Accuracy on LIO.

Software Extensibility Modeling: Currently, AutoLink measures extensibility using the remaining number of hardware interfaces. Nevertheless, the software extensibility, i.e., limited ROM and RAM space, remains as an obstacle for IoT applications. For example, Arduino Uno owns only 32-KB flash and 2-KB RAM, which is unable to support an application with both WiFi and SD card saving functionality. Prior studies of RAM usage prediction contain white-box methods [20], [21] and ones facilitated with supervised or unsupervised machine learning algorithms [22], [23]. In order to tackle the ROM and RAM shortage problem in the device platform generation process, we consider obtaining the ROM and RAM usage of user code *without* running or compiling is a possible future work of AutoLink. We consider obtaining the software extensibility *without* compiling or running as a possible future work of AutoLink.

Secondary Development Modeling: When developers continue with a subsequent development with existing hardware and some modified requirements, generating an entirely new device platform means developers would enlarge their expenditure to accommodate with the new platform, which is not acceptable. The importance of secondary development makes it one of the intended extension of AutoLink, and it could be alleviated by taking the existing device platform (i.e., d_i) as an input of AutoLink and considering of the difference between existing and ongoing d_i with cosine similarity or other distance metrics.

Embedded OS support of AutoLink: In general, exploiting an embedded OS generally brings task scheduling and dedicated programming styles to devices, compared with programming on bare-metal mainboards such as Arduino's. Hence, porting an embedded OS to AutoLink mainly targets these two aspects.

1) Task scheduling indicates that our time and power model should adapt to a more complicated scenario than the bare-metal's. Nevertheless, the APGS we introduced in Section VI-C automatically generates the time and power profiles when Contiki OS exists.
2) Another problem is the new programming style. Take Contiki OS as an example, it adopts an event-driven programming style based on C language. First, the setup-loop programming style in AutoLink is easily adopted to Contiki by using the timer instead of Time.sleep() in the loop function. For the native Contiki event-driven programs, we leverage our AutoLink Visualizer to take developer' intuitions to the occurring frequency of the events into consideration. Then, our models could make predictions to execution time and power.

Furthermore, existing dedicated solutions to model time and power for event-driven systems, such as TOSSIM [24] (for time) and Quanto [12] (for power), are also applicable for AutoLink.

Hardware Community Enlightenment: During our development and evaluation of AutoLink, we conduct several suggestions to whom may concern about hardware design. Take the four mainboards we discussed in the evaluation as examples. ARD mainboard owns a limited number of analog pins, which suppresses its appliance in terms of extensibility. While benefited from its low price, ARD could still be chosen in specific scenarios, such as cases 1 and 3 in Table II. LIO occupies a place in the market owing to its low-power nature, but its high price and low MCU performance stall it from dominant in our selection space. Mainboards with more powerful MCUs, such as RPI and BBB, should concentrate on power-saving techniques and lower prices, which will make them a better competitor in the IoT mainboard market.

Bring AutoLink Closer to Commercial Production: In our current implementation, hardware components in AutoLink database are mainly prototyping components, while commercial IoT products generally exhibit well encapsulation such as PCB. In the future, AutoLink could embrace the System-in-Package (SiP) [25] technique that achieves better encapsulation than prototyping and may lead AutoLink closer to manufacturing.

IX. Conclusion

In this article, we advocated AutoLink, an automatic approach to generating an IoT application. We take the lifetime, extensibility, cost, and timeliness of an IoT system into consideration and propose an expressive syntax for users to specify their diverse requirements for IoT device platforms. Furthermore, AutoLink proposes a duty cycle-aware power estimation model. We implemented AutoLink and evaluated its performance using benchmarks and real-world cases. Experiments show that AutoLink can generate the optimal hardware configuration that meets user requirements.

References

[1] *Worldwide Global DataSphere IoT Device and Data Forecast, 2019–2023*, IDC, Framingham, MA, USA, May 2019.
[2] *Predicts 2016: Unexpected Implications Arising From the Internet of Things*, Gartner, Stamford, CO, USA, Jan. 2016.
[3] *IoT Signals: IoT Is Driving Both Opportunity and Revenue*, Microsoft, Redmond, WA, USA, Jul. 2019.

[4] M. Hessar, V. Iyer, and S. Gollakota, "eNABLING on-body transmissions with commodity devices," in *Proc. ACM UbiComp*, 2016, pp. 1100–1111.
[5] Y. Gao *et al.*, "Mosaic: A low-cost mobile sensing system for urban air quality monitoring," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[6] R. Ramesh *et al.*, "Turning coders into makers: The promise of embedded design generation," in *Proc. ACM SCF*, 2017, pp. 1–10.
[7] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng, "TinyLink: A holistic system for rapid development of IoT applications," in *Proc. ACM MobiCom*, 2017, pp. 383–395.
[8] J. D. Hester *et al.*, "Amulet: An energy-efficient, multi-application wearable platform," in *Proc. ACM SenSys*, 2016, pp. 216–229.
[9] D. Graham and G. Zhou, "Prototyping wearables: A code-first approach to the design of embedded systems," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 806–815, Oct. 2016.
[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. ETAPS TACAS*, 2008, pp. 337–340.
[11] J. P. McDermott, "R1: A rule-based configurer of computer systems," *Artif. Intell.*, vol. 19, no. 1, pp. 39–88, 1982.
[12] R. Fonseca *et al.*, "Quanto: Tracking energy in networked embedded systems," in *Proc. USENIX OSDI*, 2008, pp. 323–338.
[13] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," in *Proc. ACM PLDI*, 2007, pp. 89–100.
[14] A. Augustin, J. Yi, T. H. Clausen, and W. M. Townsley, "A study of LoRa: Long range & low power networks for the Internet of Things," *Sensors*, vol. 16, no. 9, p. 1466, 2016.
[15] M. Spörk *et al.*, "Bleach: Exploiting the full potential of ipv6 over ble in constrained embedded IoT devices," in *Proc. ACM SenSys*, 2017, pp. 1–14.
[16] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, "pTunes: Runtime parameter adaptation for low-power MAC protocols," in *Proc. ACM/IEEE IPSN*, 2012, pp. 173–184.
[17] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proc. ACM EuroSys*, 2011, pp. 153–168.
[18] C. Cotta *et al.*, "Memetic algorithms in planning, scheduling, and timetabling," in *Evolutionary Scheduling*. New York, NY, USA: Springer, 2007, pp. 1–30.
[19] *Monsoon Power Monitor*. Accessed: Nov. 4, 2020. [Online]. Available: https://www.msoon.com/
[20] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," in *Proc. ACM POPL*, 2003, pp. 185–197.
[21] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin, "Analysing memory resource bounds for low-level programs," in *Proc. ACM ISMM*, 2008, pp. 151–160.
[22] H. Leather *et al.*, "Automatic feature generation for machine learning–based optimising compilation," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, p. 14, 2014.
[23] K. E. Coons *et al.*, "Feature selection and policy optimization for distributed instruction placement using reinforcement learning," in *Proc. ACM PACT*, 2008, pp. 32–42.
[24] P. Levis *et al.*, "TOSSIM: Accurate and scalable simulation of entire tinyos applications," in *Proc. SenSys*, 2003, pp. 1–9.
[25] K. L. Tai, "System-in-package (SIP): Challenges and opportunities," in *Proc. IEEE DAC*, 2000, pp. 191–196.

Borui Li (Graduate Student Member, IEEE) received the B.S. degree in computer science from Nanjing University of Posts and Telecommunications, Nanjing, China, in 2017. He is currently pursuing the Ph.D. degree with Zhejiang University, Hangzhou, China.

His research interests include Internet of Things and edge computing.

Wei Dong (Member, IEEE) received the B.S. and Ph.D. degrees from the College of Computer Science, Zhejiang University, Hangzhou, China, in 2005 and 2011, respectively.

He is currently a Full Professor with the College of Computer Science, Zhejiang University, where he leads the Embedded and Networked Systems Laboratory. He has published over 100 papers in prestigious conferences and journals, including MobiCom, INFOCOM, ICNP, ToN, and TMC. His research interests include Internet of Things and sensor networks, wireless and mobile computing, and network measurement.

Prof. Dong is a member of ACM.