WIPROG: A WebAssembly-based Approach to Integrated IoT Programming

Borui Li, Wei Dong and Yi Gao
Zhejiang Provincial Key Laboratory of Service Robot, College of Computer Science, Zhejiang University, China
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
Email: {*borui.li, dongw, gaoyi*}@zju.edu.cn

***Abstract*—Programming a complete IoT application usually requires separated programming for device, edge and/or cloud sides, which slows down the development process and makes the project hardly portable. Existing solutions tackle this problem by proposing a single coherent language while leaving two issues unsolved: efficient migration among the three sides and the platform dependency of the binaries.**

We propose WIPROG, an integrated approach to IoT application programming based on *WebAssembly*. WIPROG proposes an edge-centric programming approach that enables developers to write the IoT application as if it runs on the edge. This is achieved by the peripheral-accessing SDKs and annotations specifying the computation placement. WIPROG automatically processes the program to insert auxiliary code and then compile it to WebAssembly. At runtime, WIPROG leverages dynamic code offloading with compact memory snapshotting to achieve efficient execution. WIPROG also provides interfaces for the customization of offloading policies. Results on real-world applications and computation benchmarks show that WIPROG achieves an average reduction by 18.7%~54.3% and 20.1%~57.6% in terms of energy consumption and execution time.

I. INTRODUCTION

Developing an entire IoT application usually involves drastically different technologies and programming techniques at the device, edge and/or cloud sides. This *separated* programming at each side not only slows down the development process, but also makes the project code hardly portable to other applications due to the increasingly heterogeneous IoT platforms.

Therefore, researchers have proposed *integrated* programming for developing IoT applications. For example, TinyLink 2.0 [1] integrates device, cloud and client (mobile end) programming using a single coherent language with a variety of techniques such as TinyApp encapsulation, virtual sensor mechanism, and dynamic energy optimization.

Two significant issues remain unsolved. First, the code cannot seamlessly migrate (code offloading) among the three sides due to different runtime at different sides. For example, the edge and cloud can support Linux OS while the device can only support the Arduino runtime. This limitation not only decreases programming flexibility but also severely limits the possible performance improvements code offloading could offer. Second, the device code is not fully portable. The device code of TinyLink 2.0 can only be portable to platforms that the TinyLink compilation toolchain supports.

We thank all the reviewers for their valuable comments and helpful suggestions. This work is supported by the National Science Foundation of China under Grant No. 62072396 and 61872437, and Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020001. Wei Dong is the corresponding author.

At first glance, these two issues can be easily addressed by installing a common, lightweight runtime engine, e.g., JVM [2], [3] or JavaScript engine [4], [5], at different sides. However, existing runtime engines incur too much overhead for resource-constrained IoT devices. For example, the performance degradation of the JavaScript engine compared with the native approach on IoT devices is over 100× [6].

In this paper, we advocate **WIPROG** – a novel approach to integrated IoT programming motivated by *WebAssembly* [7]. Recently, WebAssembly arises as a promising technology for next-generation Web applications. It is a low-level bytecode format that is designed to be portable across architectures and exhibit near-native performance. The goal of WebAssembly is to serve as a common compilation target for various source-code level languages that can run in a browser. As such, WebAssembly is an ideal solution to address the above-mentioned two issues. It is, however, challenging to adopt WebAssembly into integrated IoT programming directly.

Challenge 1: What is the programming interface we offer to integrated IoT programming? WebAssembly is just an underlying technique to run an intermediate machine code.

Solution 1: WIPROG provides an *edge-centric* approach for integrated programming. The edge-centric means developers could write a monolithic program that contains the whole application logic for device, edge and/or cloud as if it is developed only for the edge. To be more specific, the program is written in native C/C++ language with the peripheral-accessing APIs (for the device-related operations) and attribute-based annotations (to specify the placement of the operation) provided by WIPROG. The annotated program is processed and compiled to WebAssembly modules by the WIPROG LLVM-based toolchain.

Challenge 2: How does the code can be efficiently offloaded among three sides?

Solution 2: WIPROG allows applications to define "remotable" functions (via the `wiprog_remotable` annotation) that could be offloaded among three sides. At runtime, WIPROG leverages the offloading policy and performance profiles (e.g., network condition, execution time on different devices) to make the offloading decision. Once decided to offload, WIPROG breaks the normal execution flow of the WebAssembly module through a *table-based interception* approach, records the necessary information (e.g., program states and function parameters) and calls the remote function by remote procedure call (RPC). Furthermore, WiProg proposes a *compact memory snapshotting* mechanism to reduce the network shipping cost when offloading WebAssembly functions.

Challenge 3: How to enable the customization of offloading policies to meet different demands and facilitate further researches?

Solution 3: WiProg *decouples* the offloading policy from the offloading handling framework. By taking advantage of the dynamic linking technique, the offloading policy is designed as a loadable module of WiProg. The offloading handler is notified with the decision via the module's return value. WiProg supports adding supplementary data sources to meet the diverse requirement of customized policies. Both the built-in profiles (e.g., runtime bandwidth, offline-profiled execution time) and the additional data sources could be accessed via the APIs provided by WiProg.

We implement WiProg and evaluate its performance extensively. Results show that: (1) Compared with the state-of-the-art approach [4], WiProg achieves 18.7%~54.3% and 20.1%~57.6% average performance gain in terms of energy consumption and execution time, respectively. (2) WiProg incurs up to 3.3% runtime overhead, which is relatively low and acceptable. (3) The policy customization interfaces of WiProg could be used to express various offloading policies, regardless of the policy is executed online or offline. WiProg is publicly available on GitHub[1].

II. Background and WiProg Usage

In this section, we introduce the characteristics of WebAssembly that motivate us to explore its potential usage in IoT programming and present the usage of WiProg.

A. WebAssembly Background

Why WebAssembly? WebAssembly [7] is a binary instruction format developed by a group of browser vendors. Among the many properties of WebAssembly, the *portability* and *runtime efficiency* are the most attractive features for IoT applications. The portability of WebAssembly is two-fold. (1) For programming languages, each language with an LLVM frontend could be compiled into WebAssembly, such as C, C++, Rust and Go [8]. (2) For hardware architectures, WebAssembly is designed to utilize the common hardware capabilities available on a wide range of platforms, including edge and IoT devices. For the runtime efficiency, we conducted preliminary experiments comparing WebAssembly with the native and the existing cross-platform languages (C#, Java) used by state-of-the-art computation offloading approaches [3], [9]. We use five benchmarks from the Computer Language Benchmarks Game (CLBG) [10]; Fig. 1 shows the experiment results. We can observe that WebAssembly shows comparable speed (1.47×) against native and outperforms C# and Java by 16.8% and 65.0% on average.

[1]https://github.com/liborui/WiProg

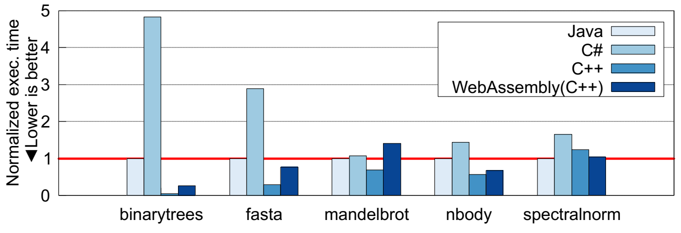


Fig. 1. Execution time of WebAssembly (compiled from C++ source), other cross-platform languages, and native C++ (normalized by Java benchmarks).

```
#include "face_detection.h"
#include "wiprog.h"
using namespace std;

__attribute__((wiprog_device_only))
vector<unsigned char> camCapture(int w, int d){/*Impl.*/}
__attribute__((wiprog_remotable))
vector<unsigned char> cvtColor(vector<unsigned char> raw,
      int type){/*Impl.*/}
__attribute__((wiprog_remotable))
vector<seeta::FaceInfo> faceDetect(vector<unsigned char>
      img){/*Impl.*/}
__attribute__((wiprog_edge_only))
void LOG(vector<seeta::FaceInfo> info){/*Impl.*/}
__attribute__((wiprog_basic))
int main(int argc, char** argv) {
   // Ommited initialization codes for clarity
   // 1. Get input from camera device
   vector<unsigned char> raw_data = camCapture(640, 480);
   // 2. Convert input to grayscale for detection
   vector<unsigned char> img_data = cvtColor(raw_data,
         COLOR_BGR2GRAY);
   // 3. Detection function
   vector<seeta::FaceInfo> faces = faceDetect(img_data);
   // 4. Log the face infomation
   LOG(faces);
}
```

Fig. 2. Code snippets of Face Detection Application.

Preliminaries of WebAssembly. The *memory* model, *table* object, and *import* mechanism of WebAssembly play important roles in our design.

The memory model of WebAssembly is referenced as the *linear memory model*. As the name suggests, it is physically organized as a flat vector of bytes. The data is accessed using the offset from the start of the linear memory at runtime. The WebAssembly module can only access the linear memory of its own, which prevents memory leakage and dangerous operations from the host.

The table object of WebAssembly is a continuous memory segment that saves function pointers, which is separated from the linear memory of the WebAssembly module to keep the fragile function pointers from being tampered at runtime. The function pointer saved in the table could not be modified by the WebAssembly module itself during execution because it can only access the linear memory.

The import mechanism of WebAssembly allows the host environment to pass an object to the module. The imported object can be memory or table, which enables the data sharing between the host and the module. WebAssembly also supports importing a function, which is useful to extend the functionality of WebAssembly.

B. WiProg Usage

We excerpt a face detection application written in C++ from an open-source face recognition system as an example to illustrate how to develop an IoT application with multiple devices in an integrated manner. This application reads raw captures from a camera located on the device, conducts preparatory RGB-to-grayscale conversion to reduce the computation complexity, and calls the `faceDetect()` function for the bounding boxes of detected faces.

With the traditional separated approach, the IoT device facilitated with a camera is programmed to capture pictures and sends them to the server. The edge and cloud server process the images using pre-installed service binaries or containers. Following this separated manner, the partition of the application logic and the data flow between devices should be handled manually by developers.

Fig. 2 shows the WIPROG code snippets of the face detection application. For developers, WIPROG provides decorative attributes (lines 5, 7, 9, 11) to annotate functions that could be dynamically offloaded between the edge and the cloud (`wiprog_remotable`) or should be placed on a specific device (e.g., `wiprog_device_only`). Taking the annotated code as input, the WIPROG generates the necessary glue code and helper functions to perform data transfer and remote execution automatically. Then, the code is compiled to WebAssembly and disseminated to the edge device for execution. On each device, the WIPROG runtime backend is pre-installed to handle the interpretation of WebAssembly and make the offloading decision for each remotable function with the pre-defined offloading policy. If the execution reaches the functions that need to run on other devices (e.g., device-only or remotable functions), the WIPROG backend will intercept the execution and perform remote execution via the RPC. Because the IoT applications are commonly periodically executed, the WebAssembly module is *kept* in the destination device after the migration for further invocation until a new module is received to reduce the migration overhead. Note that developers could also customize the offloading policy with the interfaces provided by WIPROG instead of the two default policies: minimizing task execution time or energy consumption.

Unlike the traditional approach, developers could focus on the overall application logic in an integrated manner without specifying the data flow and the optimal placement of each function explicitly.

III. WIPROG OVERVIEW

Fig. 3 depicts the birds-eye view of WIPROG's system architecture and functional workflow. The whole WIPROG framework consists of two subsystems: WIPROG frontend and backend. WIPROG frontend is designed for doing the preparatory tasks *before* deployment. *After* deployment, WIPROG backends that run on all the devices are responsible for executing the WebAssembly module and handling the runtime computation offloading.

A. WiProg Frontend

As shown in Fig. 3, WIPROG frontend compiles the application code together with the customized offloading algorithms (if exists), and does offline profiling of the application code for the backend.

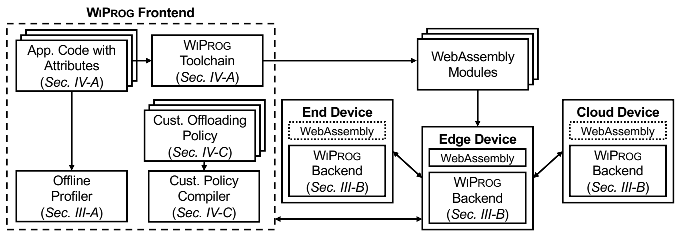

WIPROG Frontend
App. Code with Attributes (Sec. IV-A)
WIPROG Toolchain (Sec. IV-A)
WebAssembly Modules
Cust. Offloading Policy (Sec. IV-C)
Offline Profiler (Sec. III-A)
Cust. Policy Compiler (Sec. IV-C)
End Device
WebAssembly
WIPROG Backend (Sec. III-B)
Edge Device
WebAssembly
WIPROG Backend (Sec. III-B)
Cloud Device
WebAssembly
WIPROG Backend (Sec. III-B)

Fig. 3. System overview of WIPROG.

Application code. The application code is written in C/C++ language with the attribute-based annotation and peripheral-accessing APIs provided by WIPROG to specify whether the function is remotable or must be located on a specific device. The WIPROG toolchain takes the annotated application code as input, generates auxiliary code, and compiles the input to WebAssembly modules. Detailed features of WIPROG annotations and toolchain are described in §IV-A.

Customized offloading policy. WIPROG also provides programming interfaces that allow developers to customize offloading policies. The policy compiler is responsible for compiling the customized offloading policy to a loadable binary module. The module is then transmitted and loaded by the WIPROG backend. We will specify the interfaces and detailed workflows in §IV-C.

Offline profiler. It is used to profile the execution time of each function in the input code. The offline profiler executes the source code with timestamp instruments to acquire the timing data. Collecting data on the real edge or IoT device could be error-prone (e.g., when handling the serial outputs) or sometimes infeasible due to the absence of the hardware. Hence, WIPROG uses near cycle-accurate simulators such as `gem5` [11], [12] for profiling without real devices. Our offline profiler also captures the variation of the input size by profiling with different input sizes and generates a regression model for each function.

B. WiProg Backend

The WIPROG backend, as illustrated in Fig. 4, includes core and customizable modules.

Core modules. Once the WebAssembly module requires to call a remote function, the offloading handler intercepts the execution of the module and snapshots necessary program states. Then the remote execution proxy sends the WebAssembly module together with the snapshot to the target device and calls the remote function via RPC. Breaking the control flow and computing migration with the program states of WebAssembly faces non-trivial challenges, which we will explain in §IV-B.

Customizable modules. Despite providing on-device support for the customized offloading policy, the customization modules of WIPROG backend also employs a built-in runtime profiler to capture the environmental information (i.e., bandwidth, round-trip-time) to facilitate the intelligent decision of offloading policies, which is similar to [13]. Moreover, developers may attach customized data sources such as energy consumption and signal strength indicators to enable various offloading policies [9], [14], which we will describe in §IV-C.

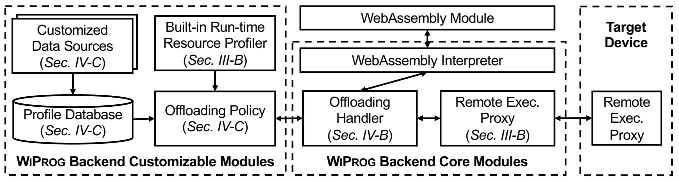

Customized Data Sources (Sec. IV-C)
Built-in Run-time Resource Profiler (Sec. III-B)
Profile Database (Sec. IV-C)
Offloading Policy (Sec. IV-C)
WiProg Backend Customizable Modules
WebAssembly Module
WebAssembly Interpreter
Offloading Handler (Sec. IV-B)
Remote Exec. Proxy (Sec. III-B)
WiProg Backend Core Modules
Target Device
Remote Exec. Proxy

Fig. 4. Illustration of WiProg backend.

IV. WiProg Design

In this section, we will first present the programming interface for developers to write integrated IoT applications in an edge-centric manner. Then we will describe how WiProg achieves runtime offloading and reduces the network shipping cost. Finally, we demonstrate how WiProg supports developers to customize the offloading policy.

A. WiProg Programming Model

Edge-centric programming interface. In order to support the edge-centric programming for integrated IoT applications and reduce the labor work for porting existing codebases, WiProg provides developers *attribute-based annotations* to specify the property (e.g., device-only or remotable) of the functions. Furthermore, to support the programming for IoT devices, WiProg is also equipped with *peripheral-accessing APIs* due to the lack of device-related interfaces in the WebAssembly,

As shown in Fig. 2, the annotations are built on the top of the function attributes (e.g., with `__attributes__` label in C/C++). The reason for using compiler-compatible attributes rather than simple annotations (e.g., `@remote`) as existing works [3], [15] do is simple: it helps us to perform a more fine-grained and reliable source code processing with the help of the token stream generated by the compile. The annotations could be classified into *monopolized-placement* and *shared-placement* attributes.

- *Monopolized-placement attributes.* These attributes (e.g., `wiprog_device_only`) are usually used for the functions that (1) depend on a specific hardware, e.g., cameras on the end device, or (2) output the final result to the desired device.
- *Shared-placement attributes.* The attributes of this kind include `wiprog_remotable` and `wiprog_basic`. The former attribute is used for the functions that could be offloaded between multiple devices. The `wiprog_basic` is the default attribute, which means the corresponding function is neither bond to a dedicated device nor remotable. The functions with the basic attribute will remain as is without being modified for remote execution.

The attribute-based programming provides enough information to enable the WiProg toolchain for further processing.

To compensate for the lack of IoT device-related interfaces in WebAssembly, WiProg provides the peripheral-accessing APIs. According to our preliminary investigation of popular IoT development websites [16]–[18], we recognize three classes of APIs: digital I/O, analog I/O, and communication-related ones (e.g., UART) are commonly used and cover 89.78% of the total APIs we have investigated [19]. Hence, in WiProg, we provide developers with the three classes of APIs for edge-centric programming. The peripheral-accessing APIs are the encapsulation of hardware-specific APIs on different hardware platforms and operate with the function import mechanism of WebAssembly.

Automatic code generation. Taking the annotated code as input, the WiProg toolchain automatically checks the legitimacy of the attributes, completes the functions that are not manually annotated, generates necessary auxiliary code, and compiles the generated code to WebAssembly modules. Hence, besides the compiler, the WiProg toolchain includes an *automatic attribute processor* and a *code generator*.

It is tedious for developers to manually annotate all the functions used in the application, especially for a large project. Incautiously annotated code can incur unnecessary data transmission that triggers performance degradation or even execution failure due to the lack of critical functions. For example, we call it an *illegitimate* assignation if a hardware-related function is assigned with the "edge-only" attribute. The attribute processor checks if existing function attributes are legitimate based on several pre-defined constraints, and assigns attributes to the functions that are not manually annotated, as illustrated in Algorithm 1. Firstly, WiProg generates the call graph of the input code for the parent/child function relationship, which is used in the remainder of the algorithm. Initially, all of the unlabeled functions are assigned to "basic" by default (line 3). The legitimacy-check (lines 4-10) examines the attributes with the following constraints: (1) Functions that access specific features of a device must be pinned to the device (line 5). (2) The child functions of those with monopolized-placement attributes must be pinned to the same device exclusively (lines 6-8). (3) To prevent the nested offloading, the child of remotable functions must not be remotable (line 9). The nested offloading incurs additional transmission overhead [2], which is harmful to the performance of an application.

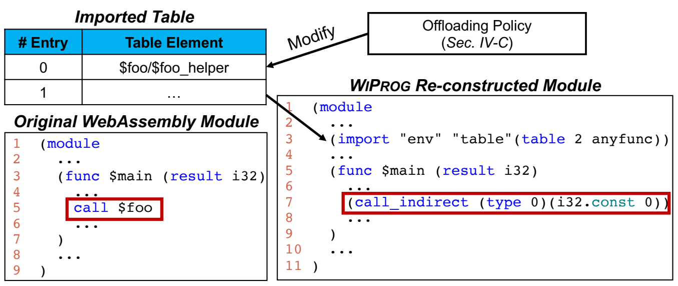
The code generator takes the fully annotated code as input and reconstructs the compilable code for each device. Taking the code for edge device as an example, it includes the functions (and corresponding libraries) with "edge only" and "basic" attributes. WiProg reconstructs the function calls with non-edge attributes as an RPC stub in the generated code and keeps the function bodies as-is for the invocations after WebAssembly migration. For the remotable functions, our code generator constructs a helper function that wraps the context capturing/restoring code and the original function. The detailed design of the helper function and the offloading procedure are presented in §IV-B.

B. Lightweight Offloading Handler

To enable the runtime offloading against the sandboxed execution of WebAssembly, WiProg must be able to address the following questions: (1) How to intercept the execution flow of a WebAssembly module for remote execution? (2)

Algorithm 1 Automatic attribute processing algorithm.

```
Input: The set of functions in the application ($F$), except the
    entry point of the application (e.g., main()).
Output: The set of functions labeled as "device only" ($A_{dev}$),
    "edge only" ($A_{edge}$), "cloud only" ($A_{cld}$), "remotable" ($A_r$),
    and "basic" ($A_{basic}$).
1: Generate the call graph of $F$ for the remaining operation.
2: Initialize $A_{dev}$, $A_{edge}$, $A_{cld}$, $A_r$, $A_{basic}$ using manually
   assigned attributes.
3: $A_{basic} \leftarrow f \cup A_{basic}, \forall f \in \complement_F(A_{dev} \cup A_{edge} \cup A_{cld} \cup A_r \cup A_{basic})$.
4: for $f \in F$ do
5:     return error if $f$ is hardware-dependent but $f \notin A_{dev}$.
6:     return error if $f' \in A_{dev}$ is the parent of $f \in A_{edge} \cup A_{cld}$.
7:     return error if $f' \in A_{edge}$ is the parent of $f \in A_{dev} \cup A_{cld}$.
8:     return error if $f' \in A_{cld}$ is the parent of $f \in A_{edge} \cup A_{dev}$.
9:     return error if $f' \in A_r$ is the parent of $f \in A_r$.
10: end for
11: for $f \in A_{basic}$ do
12:     $A_{dev} \leftarrow f \cup A_{dev}$, if $f' \in A_{dev}$ is the *only* parent of $f$.
13:     $A_{edge} \leftarrow f \cup A_{edge}$, if $f' \in A_{edge}$ is the *only* parent of $f$.
14:     $A_{cld} \leftarrow f \cup A_{cld}$, if $f' \in A_{cld}$ is the *only* parent of $f$.
15: end for
16: return $A_{dev}$, $A_{edge}$, $A_{cld}$, $A_r$, $A_{basic}$.
```



# Entry	Table Element
0	$foo/$foo_helper
1	...

```
(module
  ...
  (func $main (result i32)
    ...
    call $foo
    ...
  )
  ...
)
```

```
1  (module
2    ...
3    (import "env" "table"(table 2 anyfunc))
4    ...
5    (func $main (result i32)
6      ...
7      (call_indirect (type 0)(i32.const 0))
8      ...
9    )
10   ...
11 )
```

Fig. 5. Illustration of the control flow interception approach of WiProg.

How to pack/unpack the states and parameters of a remotable function?

Lightweight execution flow interception. For the first question, we designed a lightweight execution flow interception approach based on two specific features of WebAssembly: *table* and *import*.

As shown in Fig. 5, `foo` is a remotable function, and WiProg generates the helper function of `foo` for capturing and restoring the function context. WiProg firstly reconstructs the compiled WebAssembly binary before the deployment by (1) import a table from the execution host (i.e., WiProg backend) as line 3 of the reconstructed module in Fig. 5, and (2) substitute the direct `call` operation of all the remotable function by `call_indirect` which is a WebAssembly binary instruction that uses the function signature type (type 0 in our example) and the number of table entry (#0 in our example) to call the corresponding function. At runtime, the content of the imported table is modified by the offloading policy, e,g., assigning the table element to `$function_helper` to enable remote execution.

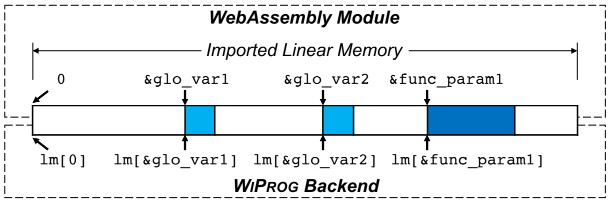


Fig. 6. The compact memory snapshotting approach of WiProg.

With our interception approach, the offloading policy is executed asynchronously and decoupled from the migration procedure. This asynchronism and decoupling enable developers to use complicated offloading algorithms without considering the runtime performance degradation.

Compact memory snapshotting. To achieve remote execution, the most recent work on WebAssembly offloading, MWW [4], migrates the *entire* linear memory to the destination, which dramatically increases the transmission overhead because only a small fraction of the linear memory is necessary for remote execution. Hence, a *compact* memory snapshotting approach is necessary for efficient offloading. We would like to borrow the existing partial runtime snapshotting approaches [2], [20] which directly packs the function stack for offloading. Nevertheless, they are not applicable in WiProg because the function stack is neither stored in the linear memory nor even accessible by the module itself, as we introduced in §II-A.

Hence, we propose a *compact memory snapshotting* approach for WebAssembly to only capture the necessary data when offloading. Firstly, as we stated in §IV-A, the WiProg toolchain performs program analysis to obtain the static information (e.g., type and length) of the global variables and input/output parameters for each remotable function. Beyond the static information, WiProg also logs the dynamic information such as the memory addresses of each variable and sizes of the dynamically allocated variables with the generated code inside the helper function. Hence, with the help of the import mechanism, we build a shared memory region between WiProg backend and the WebAssembly module. At runtime, the WiProg backend creates the snapshot using the necessary global variables and function parameters by accessing the shared memory region with the base address and the offset (i.e., memory addresses inside the WebAssembly module) as shown in Fig. 6. Then, the snapshot is transmitted along with the RPC message and restored to the same offset of the shared memory region on the destination device. Finally, the corresponding function is invoked on the destination and the computation results are packed, transferred and restored to the original device similarly. We will evaluate the reduction of transmitted size and the overhead of our approach in §VI.

C. Extensible Offloading Policies

In this section, we first present the default cost-aware offloading policy of WiProg whose goal is to optimize the efficiency of the generated multi-device application. Thereafter, we describe how WiProg enables developers to customize the policy with their algorithm even additional data sources.

WiProg offloading policy. The default offloading policy of WiProg is to minimize the overall application execution cost. To be more specific, WiProg considers *latency* or *energy* by default when making offloading decisions.

Formally, WiProg formulates a *cost-aware offloading policy* as the 0-1 integer linear programming problem shown below. Binary variable x_{if} represents the offloading decision and I denotes the set of available offloading destination, $i \in I$. $x_{if} = 1$ indicates the function f should be executed on device i, while $x_{if} = 0$ indicates that is not. A_r is the set of remotable functions, as we stated in § IV-A. Hence, our offloading policy could be expressed as:

$$\text{Objective:} \quad \arg\min_{x_{if}} \sum_{f \in A_r} \sum_{i \in I} (C_{if}^{comp} x_{if} + C_{if}^{ntwk} x_{if}), \quad (1)$$

$$\text{Subject to:} \quad \sum_{i \in I} x_{if} = 1, \forall f \in A_r, \quad (2)$$

where C_{if}^{comp} and C_{if}^{ntwk} denote the computational and network cost of executing f on device i. To be more specific, we have $C_{if}^{comp} = t_{if}, C_{if}^{ntwk} = \frac{v_f}{B_i}$ when considering latency as the metric of execution cost, where t_{if} represents the execution time of function f on device i, v_f denotes the sum of snapshot size being transmitted back and forth, and B_i denotes the bandwidth between i and the current device. For minimizing the energy cost, we have $C_{if}^{comp} = t_{if} p_{if}, C_{if}^{ntwk} = \frac{v_f}{B_i} p_i^{ntwk}$, where p_{if} indicates the average power of executing f on device i and p_i^{ntwk} denotes the average transmission power of device i.

The t_{if}, p_{if} and p_i^{ntwk} are obtained by the offline profiler of WiProg frontend shown in Fig. 3. B_i and v_f are measured during the execution by the built-in runtime resource profiler of WiProg backend shown in Fig. 4.

Policy customization. In order to enable the customization of offloading policies, WiProg provides the following supports at system level.

(1) *Decoupled* design between the customizable policy and the WiProg offloading handler. With the help of dynamic linking technique, the offloading policy is designed as a loadable module to the core components. The only restriction of the loadable module is it should expose a core function for WiProg backend to invoke. The function prototype is:

```
vector<int> offloadPolicy(WiProgProfile &profile);
```

At runtime, WiProg mounts the policy by `dlopen`, acquires the symbols by `dlsym`, and calls the `offloadPolicy()` via function pointers. The offloading decision (i.e., x_{if}) is returned to the offloading handler in the form of a vector of integers. This design gives developers a considerable degree of freedom to realize their algorithms even with third-party libraries.

(2) Data-accessing APIs. The data-accessing APIs are member functions of `WiProgProfile` which serve as the input parameter of the core function. The built-in profile includes APIs to retrieve t_{if}, p_{if}, p_i^{ntwk}, B_i and v_f. For example, developers could get the snapshot length of function `foo` via `getSnapshotLen(foo)`. Moreover, the built-in profile also contains the information generated at compile time, e.g., function call graphs. In addition to the intrinsic profiles (e.g., bandwidth, execution time), novel offloading algorithms are more intended to include additional data sources, e.g., [14] involves the received signal strength indicators (RSSI) for more accurate transmission time prediction. Towards this, WiProg incorporates a built-in database that is managed by the backend. Developers could manually capture the necessary data and save the data to the database, and the supplementary data could be accessed easily via the data-accessing APIs presented by WiProg.

We will evaluate the expressiveness of our policy customization through three case studies in §VI-C.

V. Implementation

To validate the cross-platform nature of WiProg, we implemented the building blocks shown in Fig. 3 and 4 on three different ISAs: Xtensa [21] (a post-RISC ISA targeted at IoT devices), Arm AArch64 [22] (generally adopted in edge devices) and x86-64 (dominate in cloud servers). Note that our implementation is extensible to other platforms as our implementation is based on native C++ code and has no dependency on other binary-only libraries that could not be easily ported to a new platform. Altogether, our WiProg prototype consists ~6,000 lines of code.

WiProg frontend. Our WiProg compilation toolchain is built on top of the LLVM project [8]. We modify the `clang` compiler to support our attribute-based programming interface and perform the static analysis to obtain the necessary information for the offloading handler. The customized policy is compiled with `-ldl` to enable the dynamic linking and loading of policy at runtime.

WiProg backend. The remote execution proxy of WiProg is built atop of the gRPC framework [23] which uses protobuf [24] for message serialization. Compared with other alternatives [25], [26], gRPC outperforms in the network aspect due to its compact message format, which is important for WiProg as the snapshot size is generally large.

We leverage an open-source runtime, Wasmer [27], with just-in-time (JIT) compilation to execute WebAssembly and we apply our offloading handler techniques to it on the edge and cloud device. For the IoT devices, we use wasm3 [28] interpreter instead of Wasmer because its JIT compilation is too resource-hungry for resource-constrained IoT devices.

VI. Evaluation

In this section, we evaluate WiProg to answer the following three questions: (1) Does WiProg achieve better performance than existing approaches at runtime? (2) What is the runtime overhead of WiProg? (3) Is the policy customization approach of WiProg flexible enough to implement various offloading algorithms?

A. Runtime performance

Hardware components. We evaluate the runtime performance of WiProg on a variety of devices: we use Xtensa EPS32 as the IoT device, Raspberry Pi 4B+ (RPI for short) as well as NVIDIA Jetson TX2 (TX2 for short) as edge nodes, and an x86_64 server (as the cloud server). The IoT devices are wirelessly connected to the edge devices, then connected to the cloud server through the backbone network.

Benchmarks. To be comprehensive, we use two sets of benchmarks in our evaluation: *application benchmarks* and *micro-benchmarks*. The application benchmarks are used to illustrate how WiProg works in the real-world IoT applications, including a face detection application that we used as the example in §II-B (Fig. 2) and an edge smoothing application based on [29]. For micro-benchmarks, we use the PolyBench/C [30] that contains 29 independent computational kernels to evaluate the performance of WiProg on various popular algorithms.

Baselines definition. Here we introduce commonly found offloading alternatives that we used to compare the performance of the WiProg.

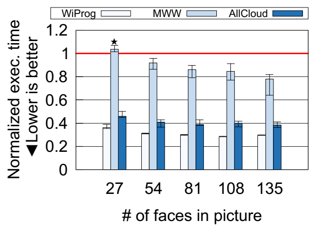
- *AllEdge*: the raw data is collected by the IoT device, then transmitted to the edge device for all the computation.
- *AllCloud*: similar to the AllEdge, while all the computation is performed on the cloud server.
- *MWW* [4]: the most recent work that dynamically offloads the JavaScript and WebAssembly between the edge and cloud. We ported its core WebAssembly migration approach in our framework for a fair comparison.

We use the default offloading policy of WiProg stated in §IV-C. The metrics used for comparison are task execution time and energy efficiency.

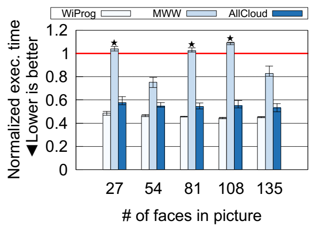
Results of application benchmarks. Fig. 7 and 8 show the results, which are normalized to the AllEdge baseline for clarity. Overall, WiProg achieves 57.6%, 61.4% and 17.8% energy improvement and 18.7%, 20.9% and 26.1% latency speedup on average against MWW, AllEdge and AllCloud, respectively. To be more specific, we have the following observations.

(1) WiProg achieves better performance than the AllEdge and AllCloud baselines by finding the most appropriate placement for remotable functions. For example, WiProg exhibits even lower execution time against both AllEdge and AllCloud in face detection application. This is because the `cvtColor` is assigned to the edge and the `faceDetect` is offloaded to the server. Hence, this placement is superior to the AllEdge by leveraging the immense computing power of the cloud to do heavy-weight `faceDetect`, and shortens the latency against AllCloud by reducing the transmission size by a lightweight `cvtColor` function which grayscales the colored picture to only one channel.

(2) The improvement compared with MWW is mainly achieved by the reduction of transmitted data. The size of transmitted data between the edge and cloud is shown in Table I. We find that WiProg reduces the data by ~95% and

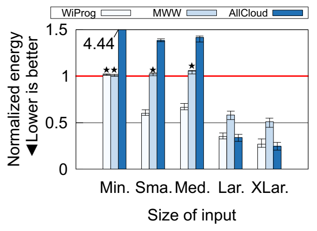


(a) Edge device is Raspberry Pi.

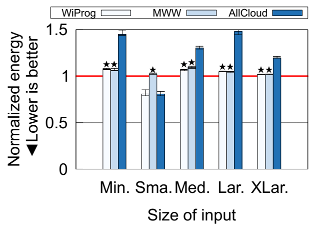


(b) Edge device is Jetson TX2.

Fig. 7. Execution time of face detection application (normalized to AllEdge). The ⋆ implies the benchmark is decided not to offload.



(a) Edge device is Raspberry Pi.



(b) Edge device is Jetson TX2.

Fig. 8. Energy consumption of edge smoothing application (normalized to AllEdge). The ⋆ implies the benchmark is decided not to offload.

~75% on average for the two benchmarks. This is because MWW transmits the *entire* linear memory of WebAssembly while WiProg only transmits the useful portion by leveraging the compact memory snapshotting of WiProg. Except for the transmission, the performance gain is also caused by restoring smaller memory data, which we will further address in §VI-B.

Results of micro-benchmarks. In Fig. 9 and 10, we compare both the energy consumption and execution time of WiProg. In general, WiProg reduces energy consumption by 54.3% and shortens the execution time by 20.1% compared to MWW.

More specifically, we observe that the improvements of WiProg on TX2 (energy: 52.9%, latency: 13.9%) are lesser than those on RPI (energy: 55.5%, latency: 26.2%). The reason is that the WiProg is more intended not to offload for TX2 benchmarks, as the computing power of TX2 is better than RPI. There are 20 out of 29 benchmarks on RPI that are offloaded, but only 15/29 on TX2. Another observation is that WiProg performs better for energy consumption. This is because the performance gain of WiProg mainly comes from the reduction of snapshot transmission time, and the proportion of data transmission cost using the energy metric is higher than that using latency metric as the power consumption during networking is higher than that during computing on the edge device.

B. Overhead of WiProg

The table lookup time of WiProg offloading handler and the memory snapshotting mechanism introduce overhead to the generated IoT applications. The overhead of WiProg could be observed in the benchmarks that decided not to offload, e.g., the benchmark with mini input in Fig. 8(a) and the gesummv in both Fig. 9 and 10. In general, the average

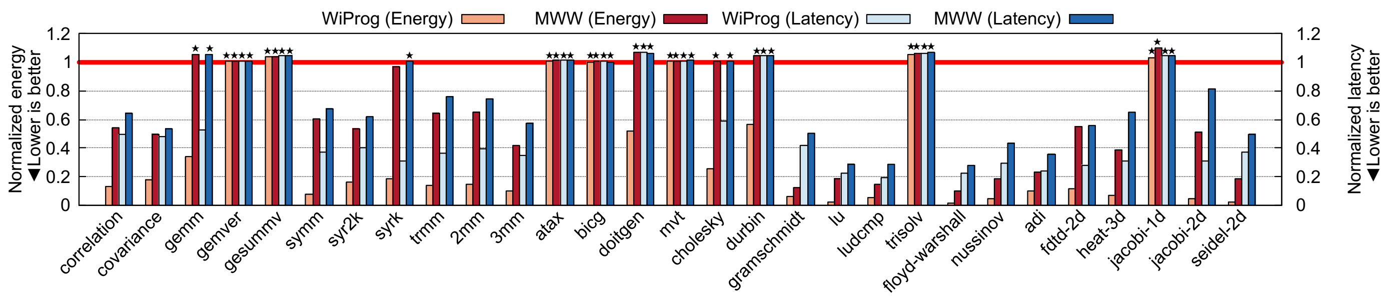


Fig. 9. Runtime performance comparison of micro-benchmarks. The edge device is Raspberry Pi. The ⋆ implies the benchmark is decided not to offload.

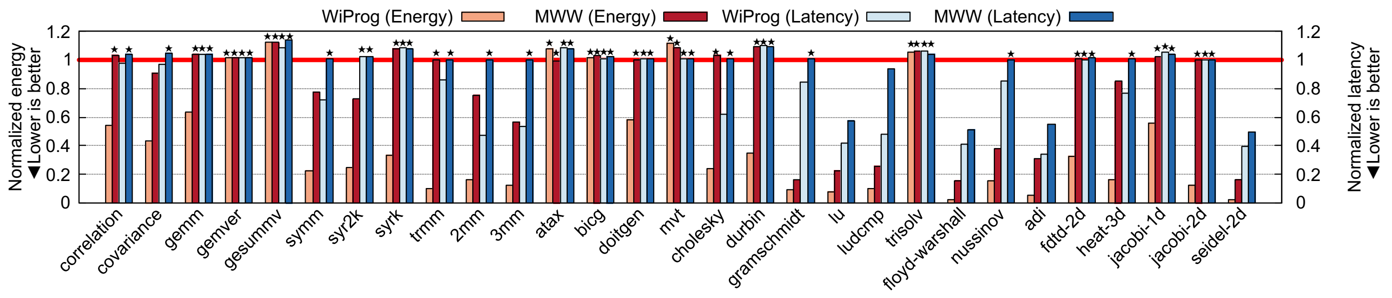


Fig. 10. Runtime performance comparison of micro-benchmarks. The edge device is Jetson TX2. The ⋆ implies the benchmark is decided not to offload.

TABLE I
SIZE OF TRANSMITTED DATA (A.C. STANDS FOR ALLCLOUD).

Face Detection (MB)				Edge Smoothing (MB)			
Size	**WIPROG**	**MWW**	**A.C.**	**Size**	**WIPROG**	**MWW**	**A.C.**
27	0.12	3.41	0.33	**Mini**	1.39	5.53	1.38
54	0.24	5.31	0.69	**Small**	3.17	12.58	3.15
81	0.36	7.01	1.05	**Med.**	8.31	33.18	8.29
108	0.48	8.59	1.41	**Large**	35.41	141.56	35.39
135	0.60	10.42	1.77	**XLarge**	132.73	530.84	132.71

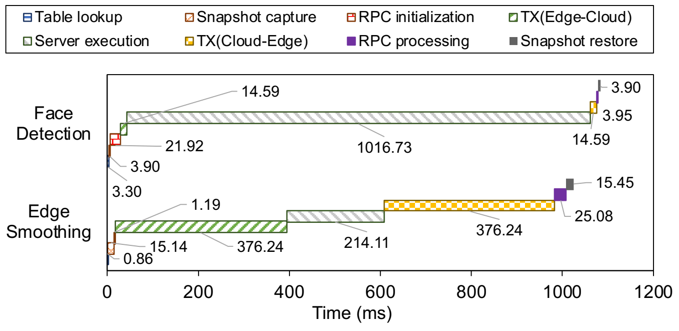


Fig. 11. Execution time breakdown of the offloaded functions.

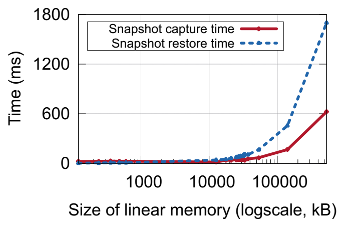


Fig. 12. Memory snapshot overhead.

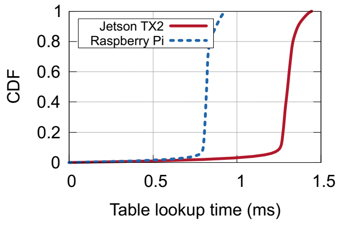


Fig. 13. Table lookup overhead.

```
double rssiEst(double rssi, int txSize){/*Impl.*/}
vector<int> offloadPolicy(WiProgProfile &profile){
   string fName = "faceDetect";
   pair<int, int> len = profile.getSnapshotLen(fName);
   vector<double> tFunc = profile.getFuncExecTime(fName);
   double rssi = profile.getDoubleFromDB("RSSI", "E2C");
   double tTx = rssiEst(rssi, len.first+len.second);
   double tRemote = tFunc[1] + tTx;
   double tEdge = tFunc[0];
   vector<int> dest(NUM_DEV, 0); //NUM_DEV=2
   if(tRemote < tEdge){dest[0] = 0; dest[1] = 1;}
   else{dest[0] = 1; dest[1] = 0;}
   return dest;
}
```

Fig. 14. Example code of customized offloading policy with Kim et al. [14].

runtime overhead is relatively small: only 3.3% and 3.0% on application benchmarks and micro-benchmarks, respectively.

We further investigate the runtime overhead of WIPROG by breaking down the execution time of two application benchmarks with runtime offloading on RPI. We can see from Fig. 11 that the WIPROG overhead, including the time of table lookup, snapshot capture, RPC initialization, RPC processing and snapshot restore, is only a small portion of the entire offloading process. Among the above sources of overhead, only the snapshot capture and restore time varies in different benchmarks. To be more specific, as illustrated in Fig. 12, the capture and restore time grows *linearly* with the size of linear memory, which is reasonable because they need to handle more data when the linear memory grows. Moreover, the table lookup overhead is ~0.8ms on RPI and ~1.2ms on TX2, which is relatively small, as shown in Fig. 13.

C. Flexibility of WiProg offloading policy customization

We demonstrate the flexibility of the WIPROG's policy customization approach by complementing three recent offloading algorithms using our programming interfaces. Note that the potential expressiveness of our customization model is far from exhaustively covered. In the following, we first describe our implementation of Kim et al. [14] as an example, and outline how the other two can be expressed.

Example implementation. Kim et al. [14] makes the offloading decision using an RSSI based model. The core code of our implementation is shown in Fig. 14. We simplify the potential offloading destination to only one edge and one cloud to keep the clarity, and the example WebAssembly

TABLE II
WIPROG POLICY CUSTOMIZATION SUPPORT AND IMPLEMENTATION OUTLINES OF THREE EXISTING OFFLOADING ALGORITHMS.

	Type	Usage of built-in profile	Additional data sources	Algorithm	Required control interface
Kim et al. [14]	Online	TX size, execution time	RSSI	Regression	Decision
LODCO [31]	Online	TX size, execution time	Energy level, power gain	Lyapunov opt.	Decision, CPU freq., TX power
COPMECS [32]	Offline	Execution time, Control flow graph	/	Graph theory	Decision

module we use for illustration in Fig. 2. The RSSI data is measured and saved to the database provided by WIPROG, and retrieved using the database accessing API of WIPROG (line 6 in Fig. 14). The execution time on each device and the snapshot length are accessed from the `WIPROGProfile` (lines 4, 5). Then we implemented the core function of [14] `rssiEst` which takes the RSSI value and data size as input and estimates the transmission time (lines 1, 7). Finally, the offloading decision is returned to the WIPROG backend through our pre-defined interface, namely the vector `dest`.

Case studies. Similar to Kim et al., LODCO [31] and COPMECS [32] could also be implemented with WIPROG. Table II summarizes the necessary components. The Lyapunov optimization based offloading algorithm of LODCO could be implemented with third-party libraries thanks to the flexibility of the dynamic linking technique we used for loading the customized policy. Controlling the CPU frequency and the TX power is beyond the scope of WIPROG, while they could also be implemented manually and called before the return of `offloadPolicy`. The offline decision approach, COPMECS, could also be realized by directly hard-coding the decision made before deployment in `offloadPolicy`.

VII. RELATED WORK

WIPROG builds upon previous research done in integrated programming and code offloading.

Integrated programming for IoT applications. Integrated application programming for multiple devices is a popular topic in IoT researches. The most related and recent work is TinyLink 2.0 [1], which proposes a structured domain-specific language (DSL) for developers to write multi-device applications in one piece of code. An application developed with TinyLink 2.0 DSL consists of three parts: device, client and cloud part. Inside the TinyLink 2.0 system, the three DSL parts are transformed into compilable codes and then compiled to dedicated binaries of each platform. WIPROG differs from TinyLink 2.0 in two aspects. 1) Our approach is to enhance the native programming language with annotations rather than using DSLs to improve the programmability. 2) TinyLink 2.0 only supports static deployment without runtime offloading. Another recent work, EdgeProg [33], leverages an If-This-Then-That (IFTTT) rules based DSL to integrate multi-device programming. Different from EdgeProg, WIPROG takes the advantage of WebAssembly's cross-platform nature to tackle the heterogeneity problem. DDFlow [34] presents a visual programming interface to build collaborative applications across multiple devices. Its idea borrows heavily from existing macro-programming approaches [35], [36] in the wireless sensor network community, which targets to enable the programming in the whole network point-of-view (POV) rather than per-node POV. Compared to the macro-programming methods, WIPROG enables developers to re-use existing code.

Computation migration and code offloading. Several approaches have been proposed to reduce the latency or energy consumption by code offloading.

MWW [4] is the most related offloading framework. It migrates the HTML5 web worker which contains a WebAssembly module and the JavaScript code between a mobile device and edge clouds. MWW achieves runtime migration by the snapshot mechanism of JavaScript, which is not applicable in other languages. WIPROG exhibits better offloading performance because MWW offloads all of the linear memory of the WebAssembly module while WIPROG only transmits the useful portion. Another recent migration approach, Queec [37], offloads the computation-intensive tasks for low-end IoT devices. The offloading decision of Queec is made by considering the device workload and the user-specified QoE requirement of the task. Different from WIPROG, Queec uses shared libraries for offloading, which neglects the heterogeneity between the end device and the edge.

In retrospect, there is a rich literature for computation migration and code offloading. In order to address the heterogeneity problem, other existing works [2], [3], [9] design migrating systems based on cross-platform languages such as Java and C#. CloneCloud [2] and ThinkAir [3] are proposed to offload Java code between the mobile device and servers by migrating Java Virtual Machine (JVM). CloneCloud profiles the application and makes offloading decisions before execution, which may be clumsy to runtime environmental disturbances. ThinkAir focuses on dynamic scaling and parallel execution of server-side JVMs. MAUI [9] achieves code offloading by utilizing the reflection mechanism of C# common language runtime (CLR). Similarly, WIPROG also leverages the cross-platform characteristic of WebAssembly to tackle the heterogeneity problem. Nevertheless, WIPROG achieves better migration performance by using RPC-based offloading other than VM migration as the related works do.

VIII. CONCLUSION

This paper presents WIPROG, an integrated approach for IoT application programming based on WebAssembly. With WIPROG, developers could write the application code in an edge-centric manner. At runtime, WIPROG dynamically offloads the functions which are assigned as "remotable" to achieve runtime efficiency. WIPROG also provides an offloading policy customization approach for developers to implement new offloading algorithms. Results show that WIPROG achieves up to 54.3% and 57.6% average gain compared with the state-of-the-arts in terms of energy consumption and latency reduction, and the policy customization approach indeed facilitates the incorporating of new offloading policies.

References

[1] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, "Tinylink 2.0: integrating device, cloud, and client development for iot applications," in *Proc. of ACM MobiCom*, 2020.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. of ACM EuroSys*, 2011.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. of IEEE INFOCOM*, 2012.

[4] H.-J. Jeong, C. H. Shin, K. Y. Shin, H.-J. Lee, and S.-M. Moon, "Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration," in *Proc. of ACM SoCC*, 2019.

[5] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *Proc. of IEEE ICDCS*, 2018.

[6] Z. Li, X. Peng, L. Chao, and Z. Xu, "Everylite: A lightweight scripting language for micro tasks in iot systems," in *Proc. of IEEE/ACM SEC*, 2018.

[7] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proc. of ACM PLDI*, 2017.

[8] LLVM Project, "LLVM 10 Release Notes," https://releases.llvm.org/10.0.

[9] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. of ACM MobiSys*, 2010.

[10] Debian Community, "The computer language benchmarks game," https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*. IEEE, 2012, pp. 1–7.

[13] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proc. of ACM MobiCom*, 2016.

[14] Y. G. Kim, Y. S. Lee, and S. W. Chung, "Signal strength-aware adaptive offloading with local image preprocessing for energy efficient mobile devices," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 99–111, 2019.

[15] M. Golkarifard, J. Yang, Z. Huang, A. Movaghar, and P. Hui, "Dandelion: A unified code offloading system for wearable computing," *IEEE Transactions on Mobile Computing*, vol. 18, no. 3, pp. 546–559, 2018.

[16] "Arduino," https://www.arduino.cc/reference/en/.

[17] "Espressif," https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html.

[18] "DFRobot Forum," https://www.dfrobot.com/forum.

[19] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng, "TinyLink: A holistic system for rapid development of iot applications," in *Proc. of ACM MobiCom*, 2017.

[20] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing," *IEEE Transactions on Mobile Computing*, vol. 13, no. 11, pp. 2648–2660, 2014.

[21] Tensilica, Inc., *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

[22] Arm Ltd., *ARM A64 Instruction Set Architecture Reference Manual*.

[23] GRPC.io, "grpc: A high-performance, open source universal rpc framework," https://grpc.io/.

[24] "Protocol buffers," https://developers.google.com/protocol-buffers.

[25] "Restful API," https://restfulapi.net/.

[26] Apache, "Thrift," https://thrift.apache.org/.

[27] Wasmer Inc., "Wasmer," https://wasmer.io/.

[28] "wasm3," https://github.com/wasm3/wasm3.

[29] R. Deriche, "Fast algorithms for low-level vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 1, pp. 78–87, 1990.

[30] "PolyBenchC: the polyhedral benchmark suite," http://web.cs.ucla.edu/~pouchet/software/polybench/.

[31] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016.

[32] L. Dong, M. N. Satpute, J. Shan, B. Liu, Y. Yu, and T. Yan, "Computation offloading for mobile-edge computing with multi-user," in *Proc. of IEEE ICDCS*, 2019.

[33] B. Li and W. Dong, "Edgeprog: Edge-centric programming for iot applications," in *Proc. of IEEE ICDCS*, 2020.

[34] J. Noor, H.-Y. Tseng, L. Garcia, and M. Srivastava, "DDFlow: visualized declarative programming for heterogeneous iot networks," in *Proc. of IoTDI*, 2019.

[35] R. Kumar, M. Wolenetz, B. Agarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran, "Dfuse: A framework for distributed data fusion," in *Proc. of ACM SenSys*, 2003.

[36] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Proc. of IEEE DCOSS*, 2005.

[37] G. Guan, W. Dong, J. Zhang, Y. Gao, T. Gu, and J. Bu, "Queec: Qoe-aware edge computing for complex iot event processing under dynamic workloads," in *Proc. of ACM TURC*, 2019.