Thread-Aware Area-Efficient High-Level Synthesis Compiler for Embedded Devices

Changsu Kim
POSTECH
Pohang, Republic of Korea
kcs9301@postech.ac.kr

Shinnung Jeong
Yonsei University
Seoul, Republic of Korea
shin0403@yonsei.ac.kr

Sungjun Cho
POSTECH
Pohang, Republic of Korea
allencho1222@postech.ac.kr

Yongwoo Lee
Yonsei University
Seoul, Republic of Korea
dragonrain96@yonsei.ac.kr

William Song
Yonsei University
Seoul, Republic of Korea
wjhsong@yonsei.ac.kr

Youngsok Kim
Yonsei University
Seoul, Republic of Korea
youngsok@yonsei.ac.kr

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

2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-7281-8613-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CGO51591.2021.9370341

***Abstract*—In the embedded device market, custom hardware platforms such as an application specific integrated circuit (ASIC) and a field programmable gate array (FPGA) are attractive thanks to their high performance and power efficiency. However, its huge design costs make it challenging for manufacturers to timely launch new devices. High-level synthesis (HLS) helps significantly reduce the design costs by automating the translation of service algorithms into hardware logics; however, current HLS compilers do not fit well to embedded devices as they fail to produce area-efficient solutions while supporting concurrent events from diverse peripherals such as sensors, actuators and network modules. This paper proposes a new thread-aware HLS compiler named DURO that produces area-efficient embedded devices. DURO shares commonly-invoked functions and operators across different callers and threads with a new thread-aware area cost model, and thus effectively reduces the logic size. Moreover, DURO supports a variety of device peripherals by automatically integrating peripheral controllers and interfaces as peripheral drivers. The experiment results of six embedded devices with ten peripherals demonstrate that DURO reduces the area and energy dissipation of embedded devices by 28.5% and 25.3% compared with the designs generated by the state-of-the-art HLS compiler. This work also implements FPGA prototypes of the six devices using DURO, and the measurement results show 65.3% energy saving over Raspberry Pi Zero with slightly better computation performance.**

I. INTRODUCTION

Developing an embedded device is challenging due to its increasing performance requirements within a limited chip area and power budget. As an embedded device becomes multipurposed supporting network connection, various sensors and actuators [1]–[4], the embedded device needs to manipulate *concurrent events* from various peripherals while meeting its quality of service (QoS). Multi-thread programming is widely used in building a concurrent embedded system [5]–[7]. For instance, Figure 1 shows an example weatherboard that integrates a control network, temperature sensor, and UV sensor as device peripherals. The weatherboard needs to support multiple threads in parallel, each of which executes the event handler of each individual device peripheral. General-purpose hardware platforms such as Raspberry Pi and Arduino

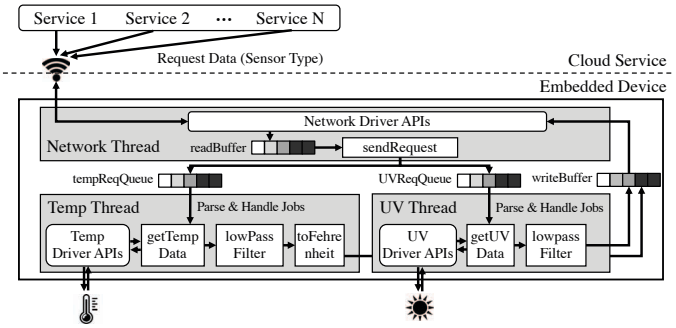


Fig. 1. A weatherboard device example with Temp and UV sensors.

can be handy solutions for manufacturers to build embedded devices, but they do not provide desirable power efficiency or performance for target QoS. A custom hardware platform such as an application specific integrated circuit (ASIC) and a field programmable gate array (FPGA) is an attractive alternative as an embedded hardware platform to satisfy both performance and power constraints that area traded with increased design time and efforts.

High-level synthesis (HLS) [8] can greatly reduce design burdens by automatically translating service algorithms into hardware logics. Unfortunately, current HLS compilers [9]–[32] fail to produce efficient embedded devices due to the following three key limitations. First, the HLS compilers generate an HDL program using a fixed topology to build circuit architectures without considering the target logic area sizes. Nested and flat topologies are two popular topologies to build circuit architectures in HLS. The nested topology privatizes invoked functions and operators in each caller hardware module [14], [15], [33]. The privatization saves execution time by creating independent hardware logics, the duplication of commonly-invoked functions and operators increases the area size. On the other hand, the flat topology shares all the functions through interconnection arbiters [34], [35]. However,

unnecessary sharing of some modules inevitably increases in area. Second, the HLS compilers do not efficiently support multi-threaded programs for embedded devices that generate concurrent events from peripherals. Prior work [15], [36]–[41] tackled the problem by utilizing general-purpose processors such as ARM or MIPS CPUs to perform serial portions of programs and thread management. However, the use of general-purpose processors incurs undesirable area and power increases. Third, the HLS compilers do not support a variety of peripherals used in embedded devices such as sensors, actuators, and network modules. Developers are solely responsible for manually creating peripheral controllers and interface logic to drive peripheral modules.

This work proposes a new thread-aware area-efficient C-to-Verilog HLS compiler for embedded devices, called DURO. Given a multi-threaded C program and target hardware configuration, DURO generates a synthesizable Verilog program for the target. This work introduces new thread-aware area cost models that analyze area costs of a logic when it is privatized, shared within a thread, or shared across multiple threads. Based on the cost models, DURO adopts a flexible topology that shares functions, operators and memory blocks if profitable. Moreover, DURO provides a POSIX thread API for manufacturers, automatically transforms the software threads into hardware thread modules with a thread manager, and inserts arbiters to shared functions and operators. Finally, to support various hardware peripherals without causing manufacturing burden, DURO provides peripheral APIs and drivers for manufacturers as an interface between the service logics and the peripherals.

This work implements six embedded devices such as a weatherboard, a gas alarm, a gyroscope, a step counter, a cardiotachometer and an IP camera using DURO , Vivado HLS [14], and LegUp [15] compiler. Compared to the designs generated by Vivado HLS, the architecture of DURO reduces the area and energy dissipation by 28.5% and 25.3% without any performance degradation. Moreover, this work implements FPGA prototypes using DURO. Compared to Arduino Uno, a low-power hardware platform, the FPGA prototype reduces energy consumption and execution latency by 51.11% and 62.93% on geomean. Compared to Raspberry Pi Zero, a performance oriented hardware platform, the FPGA prototype reduces energy consumption and latency by 65.30% and 0.10% on geomean. The source code of the DURO compiler is available at github [42].

The contributions of this paper are:

- the thread-aware area-efficient high-level synthesis compiler for embedded devices named DURO,
- the circuit topology to generate an HDL program that efficiently shares commonly invoked functions, operators and memories with new thread-aware area cost models,
- automatic hardware thread and peripheral driver generation from POSIX thread and peripheral APIs without additional manufacturing burden,
- and real world evaluation using FPGA prototypes and general-purpose hardware platforms.

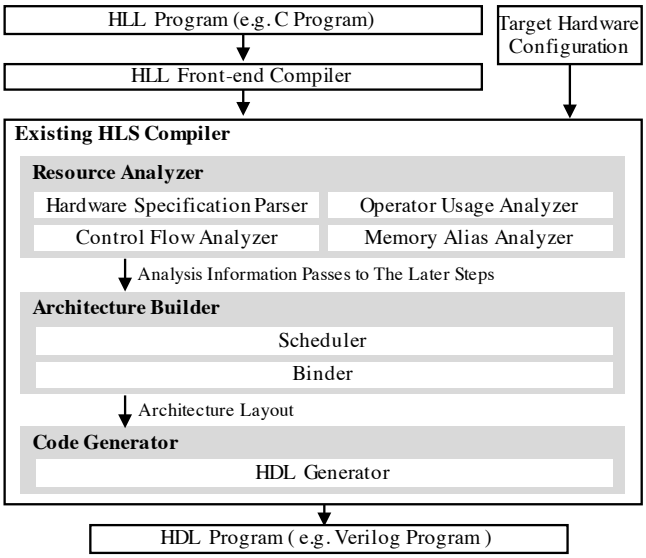


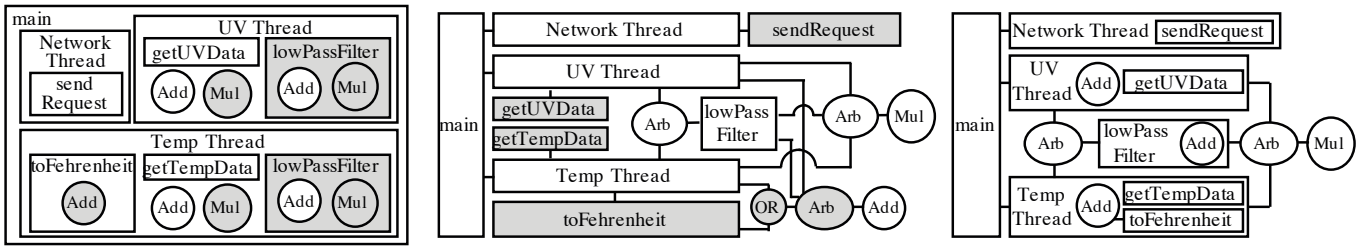
Fig. 2. Overall Process of Existing HLS Compilers

II. BACKGROUND AND MOTIVATION

High-level synthesis (HLS) is a promising approach to design embedded devices. HLS automatically interprets service algorithms written in high-level language (HLL) and generates digital hardware that performs equivalent operations. By using HLS compilers, algorithm developers or software engineers can deploy their designs into hardware platforms without involving low-level data flow languages that are often time-consuming to compose and error prone [43], [44].

Figure 2 illustrates a general HLS process of translating a program written in HLL such as ANSI C into a hardware description language (HDL) program. First, given a HLL program and its target board configuration, the resource analyzer analyzes the control flow, required operators and aliased memory of the program. The resource analyzer draws a call graph and a data path of each function, reserves required operators that are actually used in each data path, and finds types, sizes, uses and aliased pointers of arrays. Second, the architecture builder constructs an equivalent architecture layout for the given HLL program. The architecture builder determines how to instantiate, place and connect function modules and operators respecting the analyzed call graph, data paths and hardware resource specification. The architecture builder allocates memory objects into memory resources such as registers and internal memory (BRAM). The architecture builder generates controllers of the function modules, operators and memory resources. Lastly, the code generator create register-transfer-level (RTL) hardware description language according to the architecture layout.

Though HLS is a promising approach to design embedded devices, the current HLS compilers [9]–[19] do not provide proper designs for the embedded devices due to their area-inefficient architecture layouts and lack of standalone multi-thread and peripheral driver supports.



(a) Nested Topology (b) Flat Topology (c) Proposed Topology

Fig. 3. Nested, flat and proposed circuit topologies. (a) The nested topology (Vivado [14], LegUp [15]) privatizes invoked functions and operators such as the `lowPassFilter` in their caller modules, constructing a hierarchical module structure. (b) The flat topology (J. Choi et al. [34]) shares all the functions and operators via interconnection arbiters that may consume a large area. (c) The proposed topology (DURO) selectively shares commonly invoked functions and operators with thread-aware area cost models. Here, greyed boxes in the nested and flat topologies are area saved in the proposed topology.

First, the compilers do not generate HDL programs having area-efficient architecture layouts. During building an architecture layout, the current HLS compilers adopt one of two popular circuit topologies such as nested and flat ones. The compilers including Vivado [14] and LegUp [15] adopt the nested topology. The nested topology recursively instantiates a callee hardware module inside a caller module, so duplicates and privatizes invoked functions and operators in each caller hardware module. For example, Figure 3 (a) shows a nested topology of the weatherboard device example in Figure 1. In this figure, the nested topology duplicates `lowPassFilter` module in every caller module (i.e., `UV Thread` and `Temp Thread`), and connects each copy of the duplicated modules only to its caller. Though the privatization reduces execution time through independent hardware logics, the duplication increases area. M. Minutoli et al. [35] proposed resource sharing in the nested topology, but programmers should manually annotate the sharing targets and the targets are also limited to functions in the same thread.

On the other hand, the flat topology instantiates each hardware module only once at the same level of hierarchy. Figure 3(b) illustrates the flat topology generated from the example device in Figure 1. The flat topology creates only one instance of the `lowPassFilter` module, and makes other modules such as `UV Thread` and `Temperature Thread` to share the instance at the same level via an arbiter. Sharing the common instances, the flat topology can reduce area costs compared to the nested topology. However, since sharing instances may increase area costs due to the costs of arbiters, a clever strategy is necessary to reduce area costs. J. Choi et al. [34] allowed programmers to choose one of the two topologies (all-privatized and one-for-share) as one option, but the proposed flat topology unnecessarily shares all the commonly invoked functions and operators without considering sharing costs, and thus the generated architecture is not area-efficient.

This work proposes a new flexible topology that efficiently allocates and shares commonly invoked functions, operators, and memories with thread-aware area cost models. Figure 3(c) illustrates how this work generates an area-efficient HDL program for the example device in Figure 1.

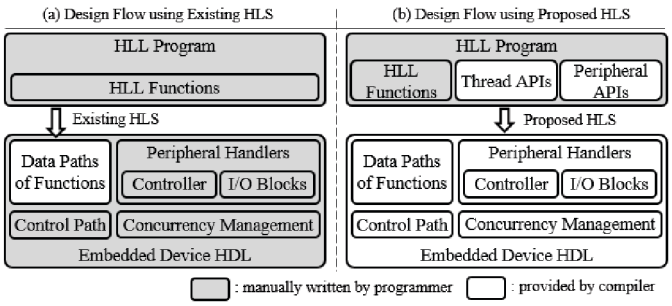


Fig. 4. Design flow using existing HLS and proposed HLS.

Since `lowPassFilter` is more expensive than the required arbiter, the proposed topology creates only one instance of the `lowPassFilter` module, and shares the instance with `UV Thread` and `Temp Thread`. On the other hand, since the `Add` operator is cheaper than the arbiter, the proposed topology duplicates `Add` operators and places the duplicated operators in `UV Thread` and `Temp Thread` in a nested way. Greyed areas in the nested and flat topologies in Figure 3 are the areas that the proposed topology saves.

Second, most compilers do not generate standalone multi-threaded HDL programs. Embedded devices need to manage concurrent events from various peripherals such as temperature and UV sensors of the weatherboard in Figure 1. Some compilers [15], [33], [36]–[41] support thread APIs and generate multi-threaded hardware, but they rely on general-purpose processors such as ARM or MIPS CPUs to manage multiple threads. However, incorporating the general-purpose processors for thread management unnecessarily increases area and power.

Lastly, the HLS compilers do not support various peripherals used in embedded devices such as sensors, actuators, and network modules. The compilers [14], [15] rely on host processors or custom HDL codes to handle communications with peripherals. To build HDL programs supporting concurrent events from device peripherals using the compilers, programmers need to manually write HDL codes as shown in Figure 4(a). In contrast, this work provides programmers with peripheral APIs and drivers that support various protocols

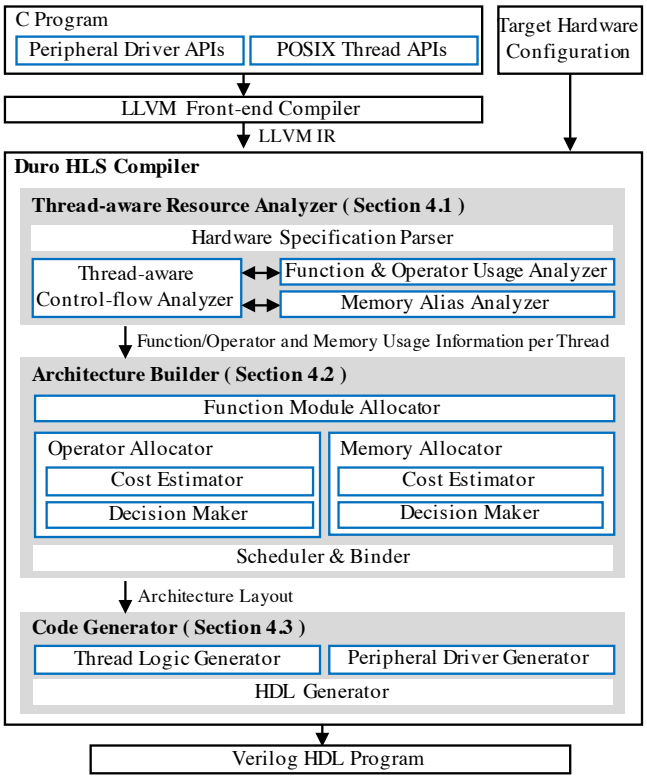


Fig. 5. The overall structure of the DURO HLS compiler. Blue and bold boxes are newly proposed in DURO.

widely used in embedded devices. Figure 4(b) shows that the peripheral APIs and drivers liberate programmers from implementing HDL codes to control peripherals.

III. DURO HLS COMPILER

This work newly proposes a thread-aware area-efficient C-to-Verilog HLS compiler named DURO for embedded devices. Figure 5 illustrates the overall structure of the entire DURO HLS compiler. DURO receives two inputs such as a multi-threaded C program without any annotation and a target hardware configuration including cell library information. The C program includes POSIX thread API function calls that manage software threads and peripheral driver function calls that manipulate hardware peripherals. By sharing commonly invoked operators, functions and memories based on newly proposed hardware cost models, DURO generates a Verilog HDL program that supports multiple threads and various peripherals such as sensors, actuators and network modules.

DURO consists of three parts; a resource analyzer, an architecture builder, and a code generator. The resource analyzer analyzes control flows of each thread, and finds functions, operators and memory objects that each thread uses. The architecture builder first initializes hardware thread modules for software threads. Then, the architecture builder calculates hardware costs of privatizing and sharing functions, operators and memory objects and decides their sharing policies. Finally, the architecture builder allocates functions, operators and memory objects, and generates the architecture layout. The code generator transforms the architecture layout into hardware modules as a Verilog program. Here, the thread logic generator inserts a thread manager module and a mutex manager module, and links the modules with hardware thread modules. The peripheral driver generator translates peripheral API calls into their corresponding hardware protocol logics, generates dedicated controller logics of the protocols, and links the protocol and controller logics to their caller logics.

A. Thread-Aware Resource Analyzer

The thread-aware resource analyzer of DURO traces all the uses of the resources in a program. Resources are operators, function modules, memory blocks and peripherals, and their users are functions and threads. Since contention on shared hardware resources causes unnecessary interconnection costs and increases size of arbiters while also increasing latency, finding resource contention is crucial to reduce area and latency. To find **intra- and inter-thread resource contention**, the resource analyzer analyzes control flows of functions in each thread and their resource usage. In addition, to precisely trace memory objects of which pointers can dynamically change during program execution, the resource analyzer adopts a precise pointer analysis [45].

The resource analyzer stores the analysis results in a resource table. Table I shows parts of the resource table for the weatherboard device example in Figure 1. For example, since `Temp Thread` and `UV Thread` commonly invoke `lowPassFilter`, the resource analyzer marks `Temp Thread` and `UV Thread` as uses of `lowPassFilter`. Here, to precisely store the resource usage information, the resource analyzer marks uses at the leaf nodes. For instance, when `sendRequest` in `Network Thread` uses `tempReqQueue` and `UVReqQueue`, the resource analyzer marks only `sendRequest` as their user.

B. Architecture Builder

The architecture builder of DURO designs an architecture layout of the given program using three allocators; function, operator and memory allocator. The function allocator determines the architectures of function modules based on a call graph, and allocates the function modules and their interconnection. The operator and memory allocators determine the specification, the number and the location of each operator and memory module, calculate sharing costs, and allocate the modules and their interconnection. After designing the architecture layout, the scheduler and binder of the architecture builder design data paths and controllers for functions using the SDC scheduling algorithm [46] and the binding algorithm [47].

1) Function Allocator: The function allocator instantiates, places and connects function modules. In the proposed topology, parts of functions follow nested or flat topologies and the other functions are inlined. The function allocator first computes the area costs of nested and flat topologies for each function, and decides their topology. Algorithm 1 describes the computation and decision algorithm. First, the algorithm marks

TABLE I
PARTS OF THE RESOURCE TABLE OF THE WEATHERBOARD DEVICE IN FIGURE 1

User	Function				Operator		Memory		
	getUV Data	getTemp Data	send Request	lowPass Filter	Add	Mul	readBuffer	tempReqQ	UVReqQ
sendRequest			-					✓	✓
lowPassFilter				-	✓	✓			
Network Thread			✓				✓		
Temp Thread		✓		✓	✓	✓		✓	
UV Thread	✓			✓	✓	✓			✓

Algorithm 1: The Function Allocator Algorithm

```
   Input: G = (V, E): A call graph of the target program
   Input: M: A area cost table for multiplexors
   Input: T: A resource table from the resource analyzer
   Output: G: A new call graph
   Output: Nest: A function set of nested architecture
   Output: Flat: A function set of flat architecture
1  Nest ← φ
2  Flat ← φ
3  foreach v ∈ V do
4      callers ← getUses(v, T)
5      nestCost ← size(callers) * M[getFSMSize(v)]
6      flatCost ←
         M[getFSMSize(v)] + getConnectCost(v, callers)
7      if flatCost < nestCost then
8          Flat ← Flat ∪ {v}
9      else
10         foreach c ∈ callers do
11             nestCost ←
                 M[getFSMSize(v)] + M[getFSMSize(c)]
12             foreach r ∈ getUses(v, T) ∩ getUses(c, T) do
13                 n ← getNumberOfUses(r, v)
14                 n' ← getNumberOfUses(r, c)
15                 nestCost ← nestCost + M[n] + M[n']
16             end
17             c_new ← inline(v, c)
18             inlineCost ← M[getFSMSize(c_new)]
19             foreach r ∈ getUses(v, T) ∩ getUses(c, T) do
20                 n'' ← getNumberOfUses(r, c_new)
21                 inlineCost ← inlineCost + M[n'']
22             end
23             if inlineCost < nestCost then
24                 V ← (V − {v, c}) ∪ {c_new}
25             else
26                 Nest ← Nest ∪ {v}
27             end
28         end
29     end
30 end
```

a function v as the flat topology if function duplication cost is more expensive than connection cost (Line 7-9). Second, if the nested topology for v is estimated as an area-efficient topology, the algorithm either inlines the function into the caller or marks the function as the nested topology depending on their area costs (Line 9-29).

To simplify the binding cost estimation, the algorithm assumes that DURO uses only multiplexors to bind the commonly used operations. The algorithm accumulates area costs (`nestCost`) of multiplexors in the function v and its caller c (Line 11-16), and accumulates area costs (`inlineCost`) of multiplexors in the inlined function c_{new} (Line 17-22). Since multiplexors become larger after inlining due to the increased number of inputs, inlining may increase the overall area costs. If `inlineCost` is smaller than `nestCost`, the algorithm replaces its caller c with the inlined function c_{new}, and updates its call graph G (Line 23-25). If not, the algorithm marks the function v as the nested topology (Line 25-27).

2) Operator Allocator: The operator allocator of DURO calculates area costs of three different operator allocation policies for each operator, and allocates the operator according to the policy with the minimal cost. Figure 6 illustrates the three policies and their corresponding cost functions; i) Private, ii) Sharing-within-thread , and iii) Sharing-between-threads.

First, in the private policy, the operator allocator assigns one operator per a user module. Each user module has their own operator sharing within a data path of a function module, but without sharing with other modules as shown in Figure 6. Its cost function is the multiplication of the area cost (Op) of the operator and the number of their user modules ($\sum_1^N F_i$).

Second, in the sharing-within-thread policy, the operator allocator assigns one operator per thread. User modules within the same thread share the operator, but do not share with other modules in different threads as shown in Figure 6. Since modules that share the operator exist under the same control flow in a thread, the operator allocator can use an OR gate for the shared access management by making all the signals except from one user module zero. Its cost function is the sum of the required operator costs ($Op \times N_{Thread}$) and OR gate costs ($\sum_1^N OR_{F_i}$) to interconnect ports.

Lastly, in the sharing-between-threads policy, the operator allocator assigns only one operator for all the user modules across threads as shown in Figure 6. Since user modules exist in different threads, two modules may use the same operator at the same cycle, or a module may attempt to use the operator that is already occupied by another. To manage the contention, the DURO operator allocator generates an arbiter for the operator, and reflects its area cost to the cost function. The cost function is the sum of one operator cost (Op), OR gate costs ($\sum_1^N OR_{F_i}$) to interconnect within single thread context, and contention managing cost between threads. The contention managing cost includes one arbiter cost (Arb), and additional data paths of user modules ($CM \times \sum_1^N F_i$) to deal with a request and grant signal. Here, a function sometimes exists in multiple thread contexts like `lowPassFilter` in Figure 3. If so, the operator allocator regards the function as a new thread when calculating area costs.

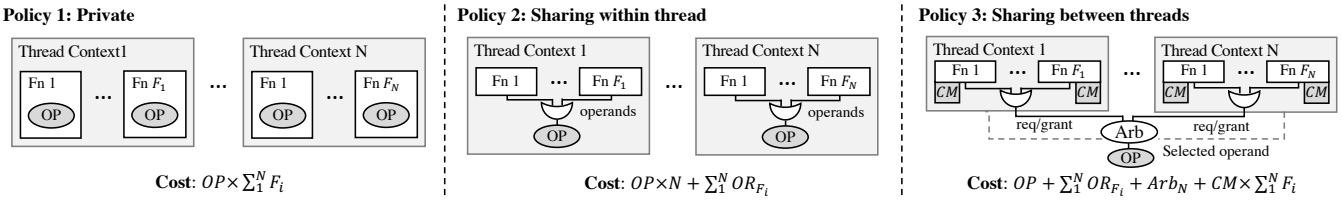
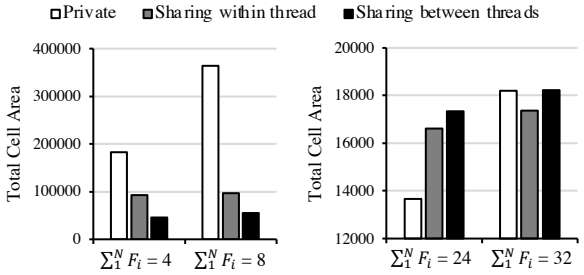


Fig. 6. Three different policies for sharing operators (N: the number of thread contexts, F_i: the number of functions using the same operator in thread context i, CM: contention manager logic that is additionally generated to handle contention)



(a) Area cost for multiply op (b) Area cost for add op

Fig. 7. Cell area of the three policies in Figure 6. The graphs show the required cell area for a `multiply` operator and an `add` operator by the three allocation policies using the 45nm OSU standard cell library [48].

The DURO operator allocator decides the allocation policies for each operator, reflecting the operator type, the number of its user modules and their thread contexts because there is no always-best policy as shown in Figure 7. For example, the private policy is the best for the `add` operator and 24 user modules while the sharing-within-thread policy is the best for 32 user modules. For different operators, different policies become the best. Therefore, the operator allocator should decide the allocation policy in consideration of operator types and the composition of user modules and thread contexts.

3) Memory Allocator: The DURO memory allocator allocates hardware memory modules for each memory object, and connects the allocated memory modules with their user modules in multiple threads. Before allocating the memory modules, the memory allocator decides their module types among the candidates such as a register and SRAM. The memory allocator uses registers for small memory objects because DURO consumes cell area for the SRAM and their controllers.

The memory allocator allocates and connects memory modules either conservatively and aggressively depending on precision of memory analysis results. If the memory analysis results are not precise, the memory allocator conservatively connects all the memory modules to all the memory user modules. To select memory access ports, the memory allocator should know memory operations and their reference memory object sets. Though the DURO resource analyzer analyzes sets of memory objects that each pointer references, the analyzer sometimes fails to calculate the referenced memory objects due to complex pointer arithmetic. If so, the memory allocator chooses the conservative management policy in Figure 8 that defines address spaces for all the memory modules, and makes the generated hardware dynamically select the memory ports according to the given memory address.

Given precise analysis results, the memory allocator first privatizes memory modules that are accessed by a single thread. Figure 8 shows the privatized memory modules as `Private #`. Since a unique thread accesses the memory modules, the memory allocator can save area costs from arbiters. If multiple functions in a thread access the memory module, the memory allocator uses an OR gate for the interconnection like the operator management case. For memory modules shared across multiple threads, the memory allocator adopts either a centralized or a decentralized management policy. The centralized management combines all the shared memory modules, and controls their accesses using one big arbiter and controller. On the other hand, the decentralized management separately allocates memory modules with their own arbiters. The memory allocator calculates area costs of the two management policies according to the cost functions in Figure 8, and allocates the shared memory modules in a minimal cost way.

C. Code Generator

Given an architecture layout from the architecture builder, the DURO code generator generates a Verilog HDL program like other HLS compilers. Moreover, to support concurrent events and various peripherals, the code generator additionally has a thread logic generator and a peripheral driver generator.

1) Thread Logic Generator: The thread logic generator creates and connects the pthreads hardware logics. When a function creates a thread, the thread logic generator creates a thread manager, and inserts `GetThreadID` and `Thread Create` logics into the function. The `GetThreadID` logic requests a unique thread ID to the thread manager that keeps thread IDs. With the thread ID, the `Thread Create` logic directly invokes the thread instance like function invocation. Unlike the function invocation, the caller function module continues next operations without waiting for a return signal from the thread instance. If a function joins a thread, the thread logic generator creates a thread exit listener and inserts a `Thread Join` logic into the function module. The thread exit listener monitors the return signal from the thread instance, and the `Thread Join` logic waits for the return signal if the listener does not yet receive the signal.

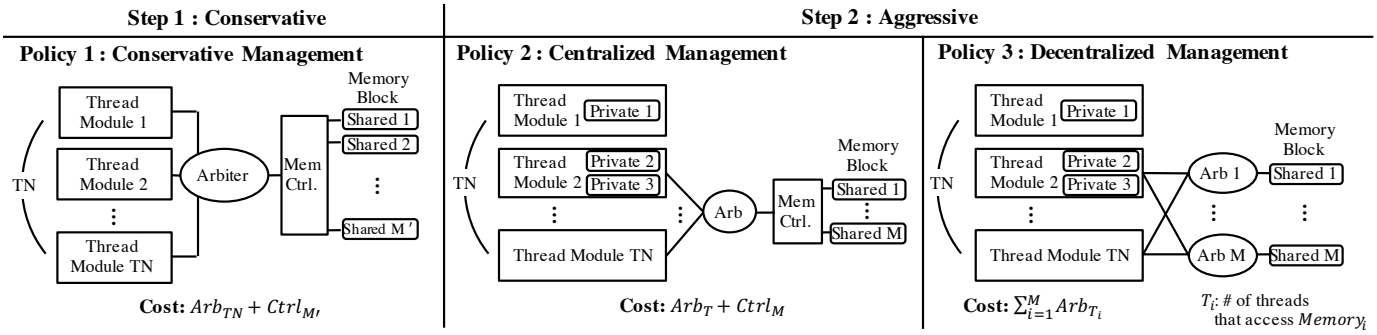


Fig. 8. Three different policies for sharing memory block

TABLE II
PARTS OF PERIPHERAL DRIVER APIS

Protocol	Methods	Description
Digital	digitalRead(pin)	Read digital signal
	digitalWrite(pin,bool)	Write digital signal
Serial	available()	Get # of bytes available to read
	read()	Read first byte of serial data
	readBytes(buf,len)	Read characters into buf
	write(buf,len)	Send characters in buf
I2C	begin(addr)	Join bus as a master or slave
	requestFrom(addr,len)	Send request to slave, and the len of bytes will be sent
	read()	Read a byte that was trasmitted
	write(buf,len)	Write data in buf to the queue
SPI	begin()	Initialize SPI bus
	transfer(buf,len)	Using buf, send&receive bytes

The DURO compiler also supports mutexes for synchronization. If a function uses a mutex, the thread logic generator creates a mutex manager module and inserts `Mutex Init`, `Mutex Lock` and `Mutex Unlock` logics into the function. If a function initializes a mutex, `Mutex Init` requests a mutex ID to the mutex manager, and keeps the ID with the mutex variable. If a function acquires the mutex, the function first loads the mutex ID from the corresponding mutex variable, and requests a lock to the mutex manager with the mutex ID. The mutex manager gives a permission to the function if the mutex is available. If not, the function module waits for the permission. If a function releases the mutex, the function notifies it to the mutex manager with the mutex ID, so the mutex manager can give the permission to the next waiting function module.

2) Peripheral Driver Generator: Hardware platforms communicate with the peripherals through their I/O pins in a promised protocol. For example, the Arduino language, one of the most popular languages for embedded systems, abstracts the protocol as an API, allowing programmers to easily control the I/O pins in a software level. To allow programmers to control peripherals like the Arduino language, DURO provides peripheral driver APIs for various communication protocols such as digital I/O, UART, I2C and SPI that are commonly used in lightweight device communications. Table II shows the APIs for peripherals.

To support a communication protocol through the APIs, the peripheral driver generator installs a peripheral driver on the pins. The peripheral driver consists of mainly two parts; a communication protocol logic and a controller logic. The communication protocol logic is a pre-defined hardware logic in a protocol library that directly communicates with peripherals via I/O pins. The controller logic is an interconnection between the communication protocol logic and a caller function. For example, if a function invokes `Serial.readBytes(userBuffer, length)` that is an UART protocol API, DURO inserts the UART communication protocol logic and its controller logic that read the data in the receive buffer and write them to `userBuffer`.

D. Overall Generated Architecture

The DURO HLS compiler automatically generates a thread-aware area-efficient HDL program integrating management modules of threads and peripherals. Figure 9 illustrates the overall architecture of the program in Figure 1 that DURO generates. DURO generates the thread and peripheral management hardware modules such as `Thread Manager` and `Network Driver`, and connects the modules only to their users. For example, since only `Network Thread` invokes `Network Driver APIs`, DURO connects the `Network Driver` module only to the `Network Thread` module. If multiple user modules invoke a module like a memory controller `Mem Ctrl.` module, DURO inserts an arbiter to control concurrent access requests.

DURO allocates functions, operators and memory in an area-optimized way as described in Section III-B. DURO allocates commonly invoked functions such as `lowPassFilter` shared between thread logics such as `UV Thread` and `Temp Thread` while inlining private functions such as `sendRequest` and `getTempData`. DURO also allocates operators according to three policies in Figure 6. For example, DURO allocates a `multiplier` operator shared between `UV Thread` and `Temp Thread` due to its high area costs, while allocating `adder` operators shared in each thread. To allocate memory modules, DURO calculates its area costs. While decentralizing memory modules such as `writeBuffer tempReqQ` and `UVReqQ` requires three arbiters, combining the memory modules as a centralized policy

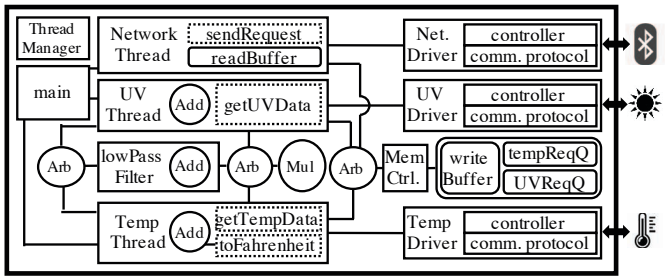


Fig. 9. Overall architecture that DURO generates for the weatherboard example in Figure 1. Dotted boxes are inlined functions.

requires only one arbiter. Thus, DURO adopts the centralized management policy.

DURO transforms the peripheral method calls of the driver APIs in Table II into corresponding hardware logics. For example, the example board in Figure 1 uses `HC-06` (bluetooth module) as a network peripheral. Since `HC-06` uses the UART protocol, programmers can control the bluetooth module using a `Serial` API. DURO generates the `Network Driver` module using pre-defined hardware logics for the UART protocol.

IV. EVALUATION

This work implements the DURO HLS compiler on top of the LLVM compiler infrastructure [49]. Using DURO, this work designs six embedded devices with ten peripherals. Followings briefly describe the six devices.

Weather board (Weather) measures temperature, humidity and UV, and applies the low pass filter algorithm to reduce noise in the data.

Gas alarm (Gas) detects a gas leak. The gas alarm applies the low pass filter to the measured data like the weather board. Unlike the weatherboard, the gas alarm has less threads.

Gyroscope (Gyro) measures angles of the device in three axes, and applies complementary filters to the data.

Step counter (Step) counts steps, and applies kalman filter to the step counting. The counting process consists of 4 steps; i) calculating reference direction of the magnetic field, ii) calculating the gradient decent, iii) calculating the change of quaternion, and iv) integration to yield quaternion.

Cardiotachometer (Cardio) provides encrypted heartbeat data. The sensor measures heartbeat rates and encrypts the rate with AES128 to protect personal information.

IP Camera (Camera) takes a 320 x 240 image and applies convolution to the image. The camera sensor collects one row of the image (320 x 1), applies the convolution filter to the row in a pipeline manner, and repeats the task until all the rows are convoluted.

Each embedded device has at least two peripherals including one network (WiFi or Bluetooth) module and one sensor module. The peripherals use six communication protocols; four protocols such as `Digital`, `I2C`, `SPI` and `UART` that DURO supports via peripheral APIs, and two protocols that require customized communication via the `Digital` API.

TABLE III
IOT DEVICES USED IN THE EXPERIMENTS

Devices	Peripherals	Protocol	Algorithm	# of threads
Weather	DS18B20 (Temperature)	OneWire	Low Pass Filter	5
	DHT11 (Humidity)	Digital		
	VEML6070 (UV)	I2C		
	HC-06 (Bluetooth)	UART	Job Scheduling	
Gas	CCS811 (Gas Detector)	I2C	Low Pass Filter	3
	ESP8266 (WiFi)	UART	Job Scheduling	
Gyro	MPU6050 (MEMS)	SPI	Complementary Filter	3
	HC-06 (Bluetooth)	UART	Job Scheduling	
Step	MPU9250 (MEMS)	I2C	Kalman Filter	3
	HC-06 (Bluetooth)	UART	Job Scheduling	
Cardio	MAX30102 (Heart rate)	I2C	AES	3
	ESP8266 (WiFi)	UART	Job Scheduling	
Camera	OV7670 (Image)	Parallel Comm.	Convolution Filter	3
	ESP8266 (WiFi)	UART	Job Scheduling	

TABLE IV
THE NUMBER OF PRAGMAS USED IN VIVADO HLS AND DURO BASED DESIGN FLOWS

Devices	Vivado	DURO
Weather	25 (`RESOURCE`:15, `ALLOCATION`:8, `INLINE`:2)	0
Gas	16 (`RESOURCE`: 9, `ALLOCATION`: 6, `INLINE`: 1)	0
Gyro	17 (`RESOURCE`: 9, `ALLOCATION`: 8)	0
Step	30 (`RESOURCE`: 16, `ALLOCATION`: 13, `INLINE`: 1)	0
Cardio	30 (`RESOURCE`: 10, `ALLOCATION`: 18, `INLINE`: 2)	0
Camera	51 (`RESOURCE`: 36, `ALLOCATION`: 15)	0

The devices create a thread for each peripheral to manage concurrent events from the peripherals. This work applies six data processing algorithms to the devices; two light-weight (low pass filter and job scheduling), two medium-weight (complementary filter and Kalman filter) and two heavy-weight (AES and convolution filter) algorithms. Table III summarizes the peripherals, protocols and processing algorithms of the embedded devices.

A. ASIC Evaluation

This section compares the proposed topology with the nested and flat topologies shown in Figure 3. In order to compare DURO with a state-of-the-art commercial HLS compiler, this section adds an architecture generated by the Vivado HLS compiler [14] in the evaluation.

Architecture implementation methodology: Using DURO, this work generates embedded device HDL programs based on the proposed, nested and flat topologies. DURO generates the proposed architecture using the proposed area cost estimations and allocation policies. Moreover, DURO generates the nested architecture using nested topology of LegUp [15], and the flat architecture using flat topology of J. Choi et al. [34]. For the Vivado HLS architecture, this work generates each thread module with the Vivado HLS 2019.1 [14], and manually inserts concurrent execution and peripheral driver logics.

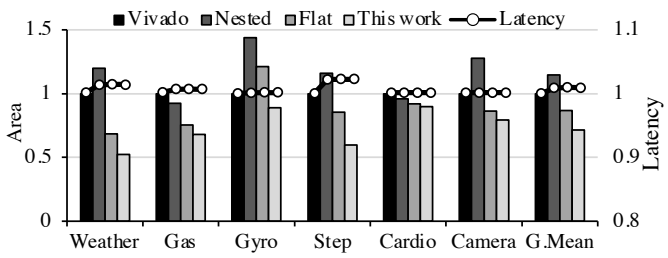


Fig. 10. Latency and area normalized to the Vivado HLS

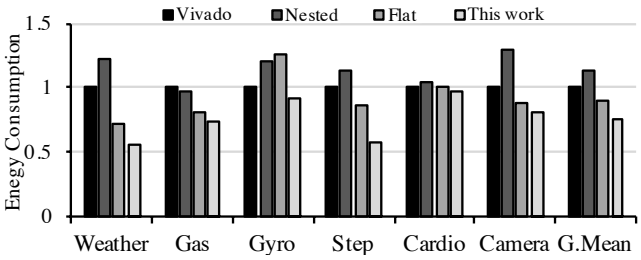


Fig. 11. Energy consumption normalized to the Vivado HLS

To generate low-power and area-efficient hardware designs, this work inserts various pragma annotations in the Vivado design flow. Table IV shows the number of pragma annotations used in the Vivado HLS design flow. The `RESOURCE` pragma specifies a specific library resource for a variable. For example, with the `RESOURCE` pragma, this work makes the Vivado HLS compiler to use a single port RAM for a memory module to reduce area and power. The `ALLOCATION` pragma specifies the number of instances of a function or an operator within a caller function module. Without the pragma, the Vivado HLS compiler may instantiate a function multiple times without sharing it. The `INLINE` pragma makes a function inlined into its caller. Moreover, this work uses configuration options including `config_bind` that allows global sharing of operators. This work takes advantage of the pragmas and configuration options to achieve area-efficient architecture designs in the Vivado HLS design flow. Unlike the Vivado HLS design flow, DURO does not use any annotation for its design flow as it can find optimal sharing and module allocation plans with its area cost models.

Latency, area and power evaluation: This work synthesizes the six devices based on the four architectures using Synopsis Design Compiler with the 45nm OSU standard cell library [48]. This work measures the event handling latency between the reception of a task through the network peripheral and the transmission of its response after finishing the requested task. This work also measures the power consumption for the event using the Synopsis Design Compiler.

Figure 10 and Figure 11 show the execution latency, energy consumption and cell area of the synthesized ASIC designs that are normalized to the architecture generated by the Vivado HLS compiler. While the four architectures show the similar latency for the six devices, the proposed architecture consumes 25.3%, 34.5% and 17.7% less energy than the Vivado-generated, nested, and flat architectures, respectively. Moreover, the proposed architecture requires 28.5%, 37.6% and 17.6% less cell area than the Vivado-generated, nested, and flat architectures, respectively.

Figure 10 shows that the four architectures have similar latencies. The flat architecture is slower than the other architectures when resource contentions on shared operators and function modules occur in a critical path. However, the contentions can be alleviated by pipelining resources that can be a bottleneck. Therefore, waiting cycles for the shared resources are not significant to the overall execution time in the flat and proposed architectures, and the four architectures show very similar total execution time.

The proposed architecture of DURO dramatically reduces area compared to the other architectures for the weatherboard because threads in the weatherboard execute similar tasks. In the weatherboard, multiple threads invoke the same function such as `lowPassFilter`, the same operators such as multipliers and adders, and the same memory modules such as read and write buffers. The flat and proposed architectures reduce their area by sharing commonly invoked function modules and expensive operators like a multiplier. Furthermore, the proposed architecture additionally reduces area by sharing operators like an adder within a thread, inlining callee functions to their callers, and adopting the centralized memory management policy in a low cost.

The proposed architecture reduces a large area for the step counter and the camera even though the devices have only three threads. Since the Kalman filter of the step counter repeatedly invokes the same functions, the proposed and flat architectures can save area compared to the Vivado HLS and nested architectures. Moreover, the proposed architecture of the step counter and the camera can additionally reduce the area costs by sharing operators in the Kalman filter within a thread. Here, the four architectures have similar area costs for the cardiotachometer because most operations of the AES encryption are cheap bitwise operations, and thus the both architectures privatize the cheap operations.

The biggest difference from Vivado HLS in terms of area is that Vivado HLS does not support sharing between threads. Even if a directive such as `ALLOCATION` pragma is used for sharing resources, Vivado HLS abandons to share the resources when dynamic pointer resolution is required. On the other hand, DURO can share the resources through the memory controller that solves the dynamic resolution. Moreover, DURO can recognize the connection cost and suggest an appropriate sharing policy. For this reason, the Vivado HLS-based architecture requires a larger area than the flat and proposed architectures.

Since the event handling latencies are similar in the four architectures, the difference in energy consumption is determined by the power consumption. The consumed power consists of dynamic power that is consumed in event handling and static power that is consumed during the time waiting for the event. Since the four architectures use similar logics for

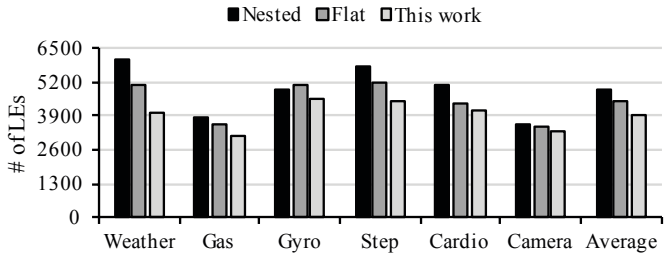


Fig. 12. Required logic elements for FPGA

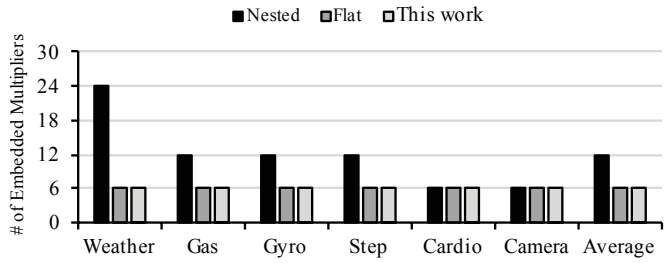


Fig. 13. Required embedded multipliers for FPGA

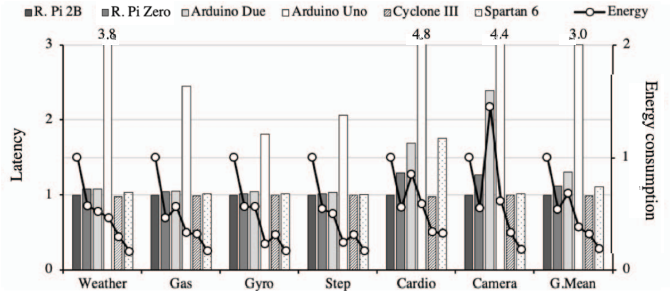


Fig. 14. Latency and energy consumption normalized to Raspberry Pi 2B

the event handling during a short period of time, their dynamic power is similar and less effective, and thus the difference in energy consumption comes from the difference of static power consumption such as leakage power. Therefore, the trend of energy consumption is similar to that of area size as shown in Figure 10 and Figure 11.

This work also analyzes FPGA resource utilization of nested, flat, and our topologies. This work synthesizes the three architectures on the Cyclone III CoreEP3C5 FPGA chip. Figure 12 and Figure 13 shows the required logic elements and multipliers for the three architectures since the chip has embedded multipliers instead of DSP. The trend of resource consumption is similar to ASIC evaluation results.

B. FPGA Prototyping Evaluation

This work implements FPGA prototypes for the six devices in Table III using DURO, and compares them with general-purpose hardware platforms. This work uses three kinds of hardware platforms; i) FPGA prototyping boards implemented with DURO: CoreEP3C5 and CMod S6, ii) high performance boards for execution time comparison: Raspberry Pi 2B and Zero, and iii) low power boards for power consumption comparison: Arduino Due and Uno. Table V lists details about the hardware platforms. This work uses Altera Quartus II 13.0 and Xilinx ISE 14.7 to program the Cyclone III (CoreEP3C5 board) and Spartan 6 (CMod S6), respectively. This work also uses Raspbian 4.14 [50] on the Raspberry Pi platforms, and uses Arduino IDE 1.8.7 for the Arduino platforms. Unlike FPGA and Raspberry Pi, Arduino does not use threads because third party libraries such as FreeRTOS [51] and Arduino Thread [52] degrade the performance compared to the sequential execution. This work measures execution time and the board-level battery consumption of the platforms using the ODROID smart power meter.The battery consumption and execution latencies are normalized to Raspberry Pi 2B.

TABLE V
SPECIFICATION OF THE HARDWARE PLATFORMS

Hardware platform	Clock frequency	Power	# of cores
Raspberry Pi 2B	600MHz - 900MHz	1130mW (w/ Raspbian)	4
Raspberry Pi Zero	700MHz - 1000MHz	548mW (w/ Raspbian)	1
Arduino Due	84MHz	588mW	1
Arduino Uno	16MHz	143mW	1
Cyclone III	50MHz	358mW	-
Spartan 6	8MHz	188mW	-

Figure 14 illustrates that DURO generates power-efficient embedded devices without sacrificing their performance. For example, Spartan6 consumes the least amount of battery among all the evaluated hardware platforms while spending similar execution time. Compared to Arduino Uno that is the most power-efficient general-purpose hardware platform in this evaluation, Spartan6 reduces battery consumption and execution time by 51.11% and 62.93% on geomean. Here, all the hardware platforms except Arduino Uno have similar execution time because peripherals are the performance bottleneck of the platforms. In summary, DURO generates a well-performed hardware that satisfies the performance and power constraints of embedded devices without requiring additional programming burdens.

V. RELATED WORK

Resource sharing: Prior works attempted to share resources at different levels of design flows to reduce the total amount of resources [14], [34], [35], [53]–[59]. Choi et al. [34] suggested a flat architecture that places the entire modules such as functions, operators, and memory modules at the same level of hierarchy for multi-threaded contexts. The flat architecture manually assigns each operator type to user modules. The lack of rules for cost estimation and decision making cannot always guarantee optimal designs with reduced resources. It also does not consider function inlining, sharing operators between threads, and centralized memory management. Vivado HLS provides the `config_bind` option allows sharing operators while it generates a nested architecture [14]. However, Vivado HLS does not support sharing functions and thread contexts,

and Vivado HLS pragmas such as `HLS allocation` or `HLS resource` do not support inter-module sharing. Minutoli et al. [35] proposed a method that supports automated function sharing, but did not consider operator and memory sharing.

Hadjis et al. [58] analyzed the area advantage of resource sharing depending on the type of target FPGA devices, and Cong et al. [56] extracted patterns based on subgraph enumeration and pattern pruning to bind them. The methods in the prior works are able to discern shareable resources and patterns, but they consider resource sharing within the data paths of RTL but not between modules. DURO uses novel allocation policies to efficiently share resources between modules in multi-threaded contexts. If DURO adopts the methods [56], [58] in the architecture building process, DURO can synergistically reduce logic area.

Concurrent executions and thread management: Previous studies [36]–[38], [60], [61] and commercial tools [9], [14], [62], [63] utilize design flows to automate the generation of concurrently operating hardware in FPGAs. An approach to support multi-threading in FPGAs is to use embedded operating systems that run threads on CPUs, and a few prior studies developed hybrid thread APIs that work similar to Pthread [36]–[38]. Stitt et al. [61] proposed a framework that utilizes on-chip CAD tools to generate hardware from Pthread. Leow et al. [60] generated VHDL codes from OpenMP programs that can be directly implemented on FPGAs. The SDSoC [63] and Vivado HLS [14] of Xilinx support generating concurrently operable hardware, but users must manually instantiate the generated hardware since they are not capable of handling threads. Xilinx's SDAccel [62] and Altera's OpenCL SDK [9] support generating pipelined hardware from OpenCL kernels. They both rely on general-purpose processors to manage threads, execute the sequential part of programs, and handle I/O operations. DURO and some other frameworks [15], [33], [34], [64] can compile entire multi-threaded programs into a stand-alone hardware system.

VI. CONCLUSION

This paper proposes a high-level synthesis compiler for embedded devices named DURO. As far as we know, DURO is the first area-efficient HLS compiler with integrated thread management and peripheral driver logic. Based on a thread-aware area cost model, DURO could generate six embedded devices and achieve area and energy saving by 28.5% and 25.3% compared to the designs generated by Vivado HLS.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank the CoreLab members for their support and feedback during this work. This work is supported by IITP-2018-0-01392 and IITP-2020-0-01847 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. *(Corresponding author: Hanjun Kim)*

REFERENCES

[1] G. Laput, Y. Zhang, and C. Harrison, "Synthetic sensors: Towards general-purpose sensing," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 3986–3999.

[2] M. Hayashikoshi, H. Noda, H. Kawai, Y. Murai, S. Otani, K. Nii, Y. Matsuda, and H. Kondo, "Low-power multi-sensor system with power management and nonvolatile memory access control for iot applications," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 784–792, 2018.

[3] M. Simić, G. M. Stojanović, L. Manjakkal, and K. Zaraska, "Multi-sensor system for remote environmental (air and water) quality monitoring," in *2016 24th Telecommunications Forum (TELFOR)*, 2016, pp. 1–4.

[4] C. Anand, S. Sadistap, S. Bindal, and K. S. N. Rao, "Multi-sensor embedded system for agro-industrial applications," in *2009 Third International Conference on Sensor Technologies and Applications*, 2009, pp. 94–99.

[5] W. P. McCartney and N. Sridhar, "Stackless multi-threading for embedded systems," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2940–2952, 2015.

[6] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-E., P. Levis, A. Terzis, and R. Govindan, "TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2009.

[7] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," in *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 2005, p. 2005.

[8] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Dordrecht, The Netherlands: Springer, 2008.

[9] "Altera SDK for OpenCL," Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/downloads.html, 2020.

[10] "Bluespec Inc," Available: http://www.bluespec.com, 2020.

[11] "Catapult High-Level Synthesis," Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/, 2020.

[12] "Cyberworkbench," Available: https://www.nec.com/en/global/prod/cwb/index.html, 2020.

[13] "Impulse Accelerated Technologies," Availalbe: http://www.impulseaccelerated.com, 2020.

[14] "Vivado High-Level Synthesis," Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2020.

[15] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[16] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[17] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 619–622.

[18] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–4.

[19] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 127–134.

[20] B. Egger, H. Lee, D. Kang, M. S. Moghaddam, Y. Cho, Y. Lee, S. Kim, S. Ha, and K. Choi, "A space- and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 197–209.

[21] M. Kudlur, K. Fan, M. Chu, R. Ravindran, N. Clark, and S. Mahlke, "FLASH: Foresighted latency-aware scheduling heuristic for processors

with customized datapaths," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 201–212.

[22] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinle, and J. Burrill, "Compiling for EDGE Architectures," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. IEEE Computer Society, pp. 185–195. [Online]. Available: https://doi.org/10.1109/CGO.2006.10

[23] P. Caldeira, J. C. Penha, L. Bragança, R. Ferreira, J. A. M. Nacif, R. Ferreira, and F. M. Q. Pereira, "From Java to FPGA: An Experience with the Intel HARP System," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 17–24.

[24] K. Fan, M. Kudlur, H. Park, and S. Mahlke, "Cost sensitive modulo scheduling in a loop accelerator synthesis system," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pp. 12 pp.–232.

[25] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, "A computing origami: Folding streams in FPGAs," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. Association for Computing Machinery, pp. 282–287. [Online]. Available: https://doi.org/10.1145/1629911.1629987

[26] J. Liu, A.-A. Kafi, X. Shen, and H. Zhou, "MKPipe: A Compiler Framework for Optimizing Multi-Kernel Workloads in OpenCL for FPGA." [Online]. Available: http://arxiv.org/abs/2002.01614

[27] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth cognizant architecture synthesis of custom hardware accelerators," vol. 20, no. 11, pp. 1355–1371.

[28] S. Mahlke, K. Fan, and M. Kudlur, "Streamroller:: Automatic synthesis of prescribed throughput accelerator pipelines," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pp. 270–275.

[29] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '09. Association for Computing Machinery, pp. 21–30. [Online]. Available: https://doi.org/10.1145/1542452.1542456

[30] R. Schreiber, S. Aditya, B. Ramakrishna Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider, "High-level synthesis of nonprogrammable hardware accelerators," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 113–124.

[31] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. Association for Computing Machinery, pp. 165–176. [Online]. Available: https://doi.org/10.1145/512529.512550

[32] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, pp. 1–12.

[33] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 270–277.

[34] ——, "Resource and memory management techniques for the high-level synthesis of software threads into parallel fpga hardware," in *2015 International Conference on Field Programmable Technology (FPT)*, Dec 2015, pp. 152–159.

[35] M. Minutoli, V. G. Castellana, A. Tumeo, and F. Ferrandi, "Interprocedural resource sharing in high level synthesis through function proxies," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–8.

[36] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006, pp. 89–98.

[37] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid fpga-cpu computational components: a missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, July 2004.

[38] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: a hardware/software co-designed multithreaded rtos kernel," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, Sept 2005, pp. 8 pp.–338.

[39] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–4.

[40] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the nymble system," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, July 2013, pp. 1–8.

[41] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A computational model for reconfigurable devices," in *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–4.

[42] "Duro High-Level Synthesis Compiler," Available: https://github.com/corelab-src/DuroHLS, 2020.

[43] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[44] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[45] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250767

[46] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 433–438.

[47] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *27th ACM/IEEE Design Automation Conference*, June 1990, pp. 499–504.

[48] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, ser. MSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 173–174. [Online]. Available: https://doi.org/10.1109/MSE.2007.44

[49] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[50] "Raspbian Operating System," Available: https://www.raspbian.org, 2020.

[51] "Arduino FreeRTOS Library," Available: https://github.com/feilipu/Arduino_FreeRTOS_Library, 2020.

[52] "Arduino Thread," Available:https://github.com/ivanseidel/ArduinoThread, 2020.

[53] D. Chen, J. Cong, Y. Fan, and J. Xu, "Optimality study of resource binding with multi-vdds," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 580–585.

[54] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *IEEE/ACM International Conference on Computer Aided Design*, ser. ICCAD '06, 2006, pp. 702–708.

[55] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave, "High-level synthesis: an essential ingredient for designing complex asics," in *IEEE/ACM International Conference on Computer Aided Design*, ser. ICCAD '04, 2004, pp. 775–782.

[56] J. Cong and W. Jiang, "Pattern-based behavior synthesis for fpga resource reduction," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344688

[57] J. Cong and J. Xu, "Simultaneous fu and register binding based on network flow method," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 1057–1062.

[58] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of fpga architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 111–114. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145712

[59] D. Ku and G. D. Micheli, *High Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Norwell, MA, 1992.

[60] Y. Y. Leow, C. y. Ng, and W. f. Wong, "Generating hardware from openmp programs," in *2006 IEEE International Conference on Field Programmable Technology*, Dec 2006, pp. 73–80.

[61] G. Stitt and F. Vahid, "Thread warping: A framework for dynamic synthesis of thread accelerators," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sept 2007, pp. 93–98.

[62] "SDAccel Development Environment," Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2020.

[63] "SDSoC Development Environment," Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html, 2018.

[64] J. Choi, S. D. Brown, and J. H. Anderson, "From pthreads to multicore hardware systems in legup high-level synthesis for fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2867–2880, Oct 2017.