Understanding Energy Efficiency in IoT App Executions

Shulin Zhao*, Prasanna Venkatesh Rengasamy*, Haibo Zhang*, Sandeepa Bhuyan*,
Nachiappan Chidambaram Nachiappan§, Anand Sivasubramaniam*, Mahmut T. Kandemir*, Chita R. Das*
Dept. of Computer Science & Engineering, The Pennsylvania State University
*Email: *{suz53, pur128, haibo, sxb392, anand, mtk2, das}@cse.psu.edu §nachi@alumni.psu.edu*

***Abstract*—Billions of Internet-of-Things (IoT) devices such as sensors, actuators, computing units, etc., are connected to form IoT platforms. However, it is observed that such hardware platforms today spend a significant proportion of their energy in communication between the CPU and the sensors (which are controlled by micro-controller unit (MCU)). Motivated by this observation, two simple, yet effective, optimizations are proposed to minimize the energy consumption. The first optimization, called *Batching*, interrupts the CPU after collecting multiple sensor data points at the MCU (instead of only 1), and thus, minimizes the interrupt overheads. The second optimization, called Computation Offloading to MCU (*COM*), offloads app-specific computations to the MCU to minimize data transfer overheads, and makes use of the relatively low energy footprint, and low-compute capabilities of the MCU in place of the CPU in the hub. However, questions such as why these two schemes are needed, where the energy benefit comes from, which IoT apps are suitable for these optimizations, etc., remain unclear.**

To better understand the *Batching* and *COM* approaches towards energy efficiency in IoT app executions, we characterize ten representative workloads on a Raspberry Pi and ESP8266 MCU platform, and evaluate the energy savings using these two optimizations and illustrate that for light-weight workloads (where *COM* is applicable), *Batching* and *COM* reduce the energy consumption by 52% and 85%, respectively when compared to the baseline. And for heavy-weight apps (where *COM* is not possible due to limited capacity of MCU), by offloading the light-weight apps and batching for the heavy-weight, *Batching* + *COM* (*BCOM*) benefits 10% energy savings compared to the baseline.

***Keywords*-Internet of Things (IoT), Energy Efficiency, Batching, Offloading, Micro-controller Unit (MCU)**

I. Introduction

The number of active connected consumer electronics devices has already exceeded the total human population today [1], and is expected to grow to ≈ 75 billion by the year 2025 [2]. These devices collectively form the Internet-of-Things (IoT) space, and perform a wide range of sensing such as understanding some user-level behavior (e.g., the number of steps walked, weather prediction, earthquake detection, emergency warning, etc.), and communicate the sensed data to a network based end user interface (e.g., an app in a mobile phone, cloud host, etc.). To extract high-level user behavior, the IoT devices employ low-level sensor events, and perform domain-specific (or user-level behavior/event-specific) computations on them. These devices are projected to grow at a rate beyond 8 billion devices every year, and they have become the main source of user interactions. Therefore, it is imperative for system architects to understand these devices/systems better, and answer the following key questions: 1. *how efficient are these devices/systems?*; 2. *where are the main sources of inefficiencies in current setups?*; and if there are inefficiencies, 3. *what is the best way to address such inefficiencies?* Previous efforts [3]–[18] have optimized energy consumption from an architecture or application perspective. However, the reasons of the energy inefficiencies at the full-stack system level remain unclear. In this paper, we address the aforementioned three questions by using commercially available off-the-shelf IoT platforms as a baseline for *modeling, characterizing and identifying simple system level solutions in the IoT space*.

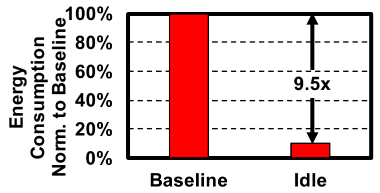


Figure 1: Energy consumption of an idle IoT hub; and when 10 apps are running as baseline.

A typical IoT platform today consists of the following components: (i) a set of sensors such as accelerometers, light sensors, etc., (ii) high-performance CPU cores, (iii) an auxiliary Micro-Controller Unit (MCU) to read the raw sensor values, and (iv) network interfaces to communicate the user-level events to end users. With the exception of the sensors, all other components are put together to form a generic commodity IoT platform (also referred to as IoT hub) such as the NXP SABRE board [19], Raspberry Pi [20], Intel Joule [21], Intel Edison [22] boards, etc. These IoT hubs provide a multitude of IO ports (e.g., GPIO, USB, SPI, I2C, etc.) and interfaces for low-level sensors.

To study how much effect an IoT app/workload has on the energy consumption of the IoT hub, we use a particular IoT hub (Raspberry Pi and ESP8266 MCU) and show the energy consumption (in Figure 1) when: (a) the IoT hub is idle with CPU cores, and MCU are in sleep mode; and (b) the baseline energy consumption average of 10 IoT different apps from various domains such as Smart Home, Health Care, Smart City, etc., as normalized to the idle hub. As can be observed from Figure 1, when running these sensor-driven IoT apps (e.g., step-counter app uses accelerometer sensor values as inputs), $9.5\times$ more energy is consumed than when the hub is idle. This indicates a chronic inefficiency, when the workload is executing and so, there is a clear motivation for optimizing these inefficiencies in workload execution for saving potentially $9.5\times$ energy consumption.

To understand the reasons behind this inefficiency, we first analyze the energy consumption of 4 sub-tasks namely: (i) the low-level sensor data reading at the MCU, (ii) the MCU interrupting the CPU, (iii) the data transfer overhead from the MCU to the CPU, and (iv) the subsequent high-level user behavior computation. We observe that data transfer between the CPU and the MCU occurs frequently at each interrupt, and the CPU wastes considerable energy in handling per-sensor interrupts from the MCU. To optimize for this data transfer cost bottleneck, a recent work – *BEAM* [4] has proposed to reuse the same low-level sensor data (that is once made available at the CPU), to be read by multiple concurrently executing apps. Doing so can potentially amortize for the sensor data read and the subsequent data transfer costs for all the shared sensors across concurrently executing apps. This scheme does not benefit single application executions (as there are no other applications to share the sensor reads with). Consequently, we tested its effectiveness using 14 IoT execution scenarios consisting of two to four concurrently executing applications (with up to 4 overlapping sensors for *BEAM* optimizations) on a Raspberry Pi and ESP8266 MCU hub. The results collected from real hardware show that *BEAM* provides 29% energy saving on average, still leaving a potential 71% room for further optimizations.

In this paper, we make the following key contributions:

Identifying the bottlenecks through applications characterization: We first perform a fine-grained characterization of a diverse set of 10 IoT apps involving 10 different sensors to analyze the power and energy footprint of the four sub-tasks described above. From this analysis, we find that, application execution spends most of its energy on transferring the sensor data from the MCU to the CPU (81%), and the interrupt handling and subsequent user-level computation at the CPU ($\approx$ 15%). To optimize these two aspects, we study two key optimizations at the system level.

Identifying the bottlenecks through applications characterization: We first perform a fine-grained characterization of a diverse set of 10 IoT apps involving 10 different sensors to analyze the power and energy footprint of the four sub-tasks described above. From this analysis, we find that, application execution spends most of its energy on transferring the sensor data from the MCU to the CPU (81%), and the interrupt handling and subsequent user-level computation at the CPU ($\approx$ 15%). To optimize these two aspects, we study two key optimizations at the system level.

Reducing data transfer energy using *Batching*: We characterize the data transfer cost for various workloads and show that the physical medium for data transfer is actually efficient and does not consume much energy. However, the CPU stalling for all the sensor data required for performing the user level computation to be sent back from the MCU expends the most energy. Motivated by this observation, *Batching* makes the MCU to store intermediate sensor data reads in its (limited capacity) buffers and batches all the interrupts (1000s of them for one user level computation instance) to one interrupt for every user-level computation. This has a direct consequence of relieving the CPU from waiting for the intermediate sensed data, and gets all the required data in one shot. By doing so, the CPU can now transition to sleep mode and hence save $2.6\times$ energy.

Reducing the CPU computation energy: To further close the gap, we explore whether the user-level computations are "offload-able" to the low-power MCU and completely get the CPU to sleep for the whole computation – and transition to active mode only when necessary. By doing so, we only need to transfer the high-level user behavior that gets computed as a result of sensor data reads at the MCU to the CPU. This offload helps the CPU sleep much longer – while the computation is done by a much more power-efficient MCU, translating to a system wide energy benefits. We classify those apps that fit this **C**omputation **O**ffload to **M**CU (*COM*) paradigm (no loss in performance) as light-weight and in such apps, we see a boost in energy savings of an average of $6.7\times$ compared with baseline.

In heavy-weight scenarios where the app-specific compute requirements are not fit into the MCU's capabilities, we still observe a 10% energy saving from *Batching* and *COM*.

Although we do not claim the novelty, these two optimizations briefly described above form the core of energy efficiency optimizations in IoT platforms. When employing them both (when applicable), we end up reducing the energy consumption from the *Baseline* case by 68%.

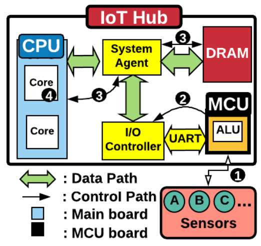
II. Background and Motivation

In this section, we first provide an overview of a typical IoT system and show how sensors connect to an IoT hub. Then, we analyze a popular IoT app, step-counter, to explain the hardware - software interactions in an IoT setup.

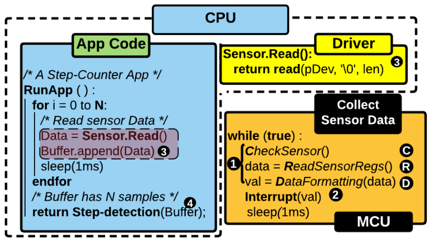
A. IoT Hub

An IoT hub is the main control unit that processes various signals received from one or more sensors. An IoT Hub (e.g., Raspberry Pi [20], Intel Edison [22], Intel Joule [21]), as illustrated in Figure 2a, includes cores, System Agent (also called North Bridge), DRAM, I/O controller, interfaces such as Ethernet, USB, HDMI, Camera, Peripheral I/O (PIO, e.g., I2C, UART, SPI) on the Main board, and an external MCU board connected through the PIO bus interface. Note that, PIO buses are also built-in the MCU board.

How do the sensors connect to an IoT hub? There are two ways of connecting sensors to an IoT Hub, either directly to the PIO buses on the Main board, or to the PIO buses on the MCU board.



(a) A typical IoT hub with MCU.

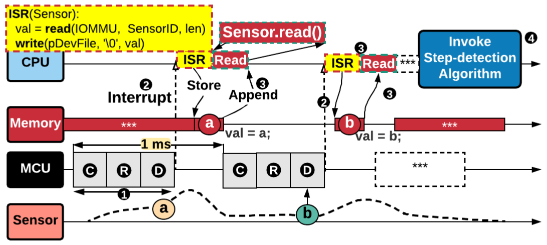


```
/* A Step-Counter App */
RunApp ( ) :
  for i = 0 to N:
    /* Read sensor Data */
    Data = Sensor.Read()
    Buffer.append(Data) ❸
    sleep(1ms)
  endfor
  /* Buffer has N samples */ ❹
  return Step-detection(Buffer);
```

```
Sensor.Read():
  return read(pDev, '\0', len) ❸
```

```
while (true) :
  CheckSensor()                  Ⓒ
❶ data = ReadSensorRegs()        Ⓡ
  val = DataFormatting(data)     Ⓓ
  Interrupt(val) ❷
  sleep(1ms)
```

(b) Step-counter app code.



```
ISR(Sensor):
  val = read(IOMMU, SensorID, len)
  write(pDevFile, '\0', val)
```

(c) Timeline of the Sensor.Read() with MCU.

Figure 2: Overview of an application running on an IoT hub.

Connecting Sensors to the Main board: When plugged in, the sensor-specific Linux driver on the Main board probes and initiates the polling function for it. A polling function typically uses the CPU to *poll* an I/O device and waits (as a blocking call) until the device responds back. Note that the Main board has an independent Programmable Interrupt Controller [23] which can receive interrupts. However, most sensors such as accelerometer, sound, motion, etc., are unsophisticated and hence do not have any logic/support for the interrupt mode [24]–[30]. Thus, the CPU needs to block and wait until the device responds back.

Connecting Sensors to the MCU board: For allowing the CPU to carry on with useful work and not get blocked by sensor-polling activities, sensors are connected to the PIO buses on the MCU board, instead of the Main board, as shown in Figure 2a. After the MCU collects readings from the sensors attached to the MCU board, ❶ it sends interrupts to the Main board through the I/O controller and puts the sensor values on the PIO bus; ❷ CPU receives the interrupts sent from the MCU board; ❸ CPU handles the interrupts by loading the sensor values from the PIO bus; and ❹ CPU stores the sensor values in the DRAM on the Main board. This process is expected to be quite efficient, as all the sensors can communicate with the MCU board in polling mode while the CPU is from the blocking calls. As a result, the CPU now only receives an interrupt when the MCU finishes reading a sensor value.

B. An Example IoT App Execution: Step-Counter

We pick a representative simple application, step-counter, to illustrate how the software and hardware interact in an IoT hub. As the name suggests, step-counter app is used to count a person's steps during a given period of time using an accelerometer sensing at a predefined sampling rate. It is widely used in SmartHome [31] and HealthCare [32] areas. At a high level, the app code is shown in Figure 2b. The body of the main loop in this app performs 3 actions: collecting data samples from a sensor, buffering the samples in memory and sleeping for $1ms$ (as the sampling rate is predefined to $1kHz$ for this app). This loop repeats for N iterations determined by the application developer. After the loop execution, a step detection algorithm [33] is triggered to take the set of N samples as its input to compute the number of steps detected.

On the hardware side, the four events work one after another, as depicted in Figure 2a. To further understand the chronological executions of these four events, we show the timeline of the *Sensor.Read()* in Figure 2c:

❶ **Sensor Data Collection in MCU:** As shown in Figure 2c, a step-counter application has to perform three tasks to simply read a sensor data sample – Ⓒ Checking Sensor, Ⓡ Reading Sensor Register, and Ⓓ Data Formatting.

Task I: Checking Sensor Availability. When the MCU starts to read a new sensor value, the first task is to check the sensor's availability. This process includes several checks, e.g., verifying the ready bit of the sensor, checking if the current sensor is working under multiple condition thresholds, and validating the electronic conditions. Some of these checks may result in an error, leading the MCU to stop reading and throw an error message. If no error occurs in this step, then the MCU proceeds to the next step - reading the sensor.

Task II: Reading Sensor Data Register. To read the sensor value, the MCU sends a read command to its own I/O controller built in the MCU board, along with some added information, e.g., sensorID, address of sensor register, etc. The I/O controller identifies the corresponding sensor, and then polls the raw value from the sensor's data register, and finally forwards it back to the MCU.

Task III: Transform Raw-Data To Information. The collected raw data from the I/O controller is decoded by the sensor-specific driver running on the MCU to output meaningful values. For example, the accelerometer sensor [24] outputs the raw voltage value (e.g. $1235\ mV$) which is formatted to the correct acceleration value (e.g. $1235 \times 10^{-4}\ m/s^2$).

❷ **MCU Interrupts CPU:** After the previous three tasks are completed on the MCU board, the MCU puts the new sensor value on the PIO bus and interrupts the CPU to notify it that a new sensor value is ready to be picked up.

❸ **Interrupt Processing on CPU:** As shown in Figure 2c, after the CPU receives the interrupt from the MCU board, it handles this interrupt in two parts: *Interrupt Processing* and *Data Transfer*. The *Interrupt Processing* part includes checking the priority of this interrupt, acknowledging that this interrupt comes from the MCU board, and context switching. Then, the *Data Transfer* part picks up the sensor value from the PIO bus and stores it in a DRAM buffer on the

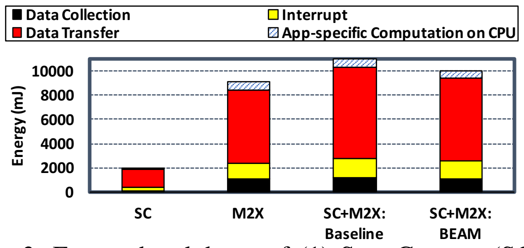


Figure 3: Energy breakdown of (1) Step-Counter (SC); (2) AT&T-M2X (M2X); (3) *Baseline*: SC + M2X; and (4) *BEAM* applied on top of (3).

Main board. Then, whenever an app calls the *Sensor.Read()* function, the sensor value already stored in this memory buffer is copied to the user buffer allocated by the app.

❹ **Compute on CPU based on the sensor readings:** After the above four routines are repeated for N samples, as shown in Figure 2b (*for* loop), all N sensor readings are populated in the `Buffer`. The step-counter app subsequently invokes the step-detection algorithm [33] to report the number of steps detected from these N sensor samples.

The above four events – sensor data collection, MCU raising an interrupt, data transfer, app-specific computation, form the crux of any IoT app. In the next section, we examine the inefficiencies by analyzing 4 common execution scenarios.

C. *Inefficiencies: Interrupts and Data Movement*

As stated in Section I, the energy consumption when running an app on the IoT hub is $9.5\times$ more than idle (Figure 1). In this section, we investigate the reasons for such energy inefficiency observed in state-of-the-art IoT platforms.

In Figure 3, we examine four common execution scenarios (executed on real hardware, methodology and specifications shown in Section III and Table II). In two of these scenarios, only one app (SC or M2X) executes on the IoT platform, and in the other two both the apps execute concurrently (one with no optimizations, viz., *Baseline* platform and the other with *BEAM* [4] optimization enabled. Due to their varying computational and sensing loads, these scenarios have varying effects on their energy consumption as shown on the y-axis. For example, SC and M2X apps consume 1902 mJ and 9071 mJ of the energy when running independently. When running concurrently (SC+M2X), the total energy consumption is 10973 mJ. With *BEAM* optimizations applied, energy-saving improves by only 9%. *BEAM* amortizes for the sensor reading, interrupt, data transfer costs by reading and transferring the sensor data just once when multiple concurrent applications at the CPU requires data from common sensors. We emphasize that *BEAM* works only when two or more apps are sharing sensor readings. If there is only one app running on the IoT hub, or if there are two or more apps running concurrently without sharing any sensor readings, *BEAM* would not bring any benefits. We evaluate these scenarios in more detail in Section IV.

SC and M2X apps share one common sensor – the accelerometer. Note, M2X reads four more sensors (barometer,

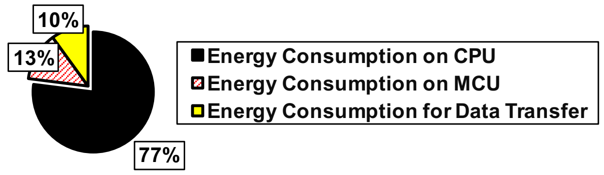


Figure 4: Energy breakdown for data transfers in baseline design. 90% energy is consumed by CPU and MCU waiting for the data transfer, while only 10% is consumed by the physical data transfer. Thus the software design for the data transfer routine is inefficient.

temperature, air quality, light sensors) that are not needed by the SC app. This implies that *BEAM* can potentially save on the data transfer cost for only one out of five sensors readings during the execution of M2X if SC has already read the accelerometer's sensor data (or vice-versa when M2X has read the data first and SC needs it afterwards).

In Figure 3, we further break down the energy consumption into the four routines mentioned in Section II. In all these four scenarios, irrespective of how compute/sensing intensive the scenario is, the app executions expend $70-80\%$ of energy in data transfers, $10-12\%$ in interrupts, and $< 5\%$ in data collection and app-specific computation.

Summary: The above study emphasizes that an app execution needs to *optimize data transfers and reduce the CPU interrupt costs to translate into maximum energy savings.*

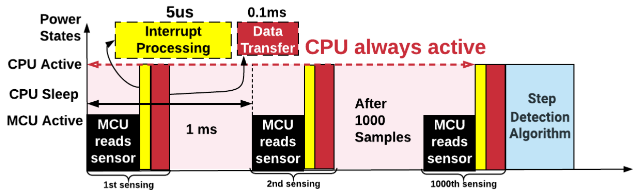
III. Methodology

In this section, we study two simple, yet efficient techniques proposed to address the above inefficiencies existing in current designs that make processing the sensor data expensive. The first scheme, called *Batching*, accumulates N sensor data points at the MCU before sending them to the CPU, thereby reducing the number of interrupts. The second scheme, called Computational Offloading to MCU (*COM*), offloads the computation from the CPU, thereby allowing the CPU to go to sleep mode when idle. Finally, we combine both the techniques (*Batching* + *COM*, *BCOM*) to optimize the energy efficiency further.

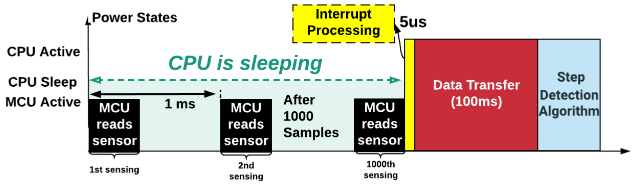
A. *Batching*

As discovered in Section II-C, data transfer routine consumes the most energy. This motivates us to further ask these questions: *What causes this energy inefficiency in data transfer? Hardware design or software design?* To answer these questions, during data transfer in the step-counter app, we again breakdown the energy of CPU and MCU, shown in Figure 4. One can observe that only 10% energy is consumed when the physical data transfers; 90% energy is consumed by CPU (77%) and MCU (13%) due to the software stack design. Because the energy consumption on CPU is dominant, we further study the power states of CPU to better understand the software stack inefficiencies.

In CPU cores, there are two main power states: active mode and sleep mode. Each of these power states has



(a) Power states of MCU and CPU in *Baseline*.



(b) Power states of MCU and CPU in *Batching*.

Figure 5: Power states changes over-time in (a) *Baseline* and (b) *Batching* for Step-Counter. In *Baseline*, the CPU is in active mode all the time; in *Batching*, the CPU can sleep for a long period.

different power consumption. The power consumption in sleep mode is 3.3× less than the active mode (1.5 $Watts$ vs. 5 $Watts$). To take advantage of the power efficiency of the sleep mode, it is the "default" power state of a CPU as long as there is no active jobs waiting. However, waking up a CPU from sleep mode to active mode is costly – it needs around 1.6 ms for the CPU to transition between these two modes [34], [35], and the transition power is as high as 2.5 $Watts$ on average, this translates to 2.5$Watts$ × 1.6ms = 4 $mJoules$ energy overhead. Therefore, only if a CPU is able to sleep for longer than 1.14 ms ($= \frac{4mJoules}{5Watts-1.5Watt}$), it can actually provide energy savings. Otherwise, going to the sleep mode causes more energy consumption.

For example, the step-counter app needs 1000 samples in 1 second to calculate a precise value for detecting steps. After an MCU collects one sample from the accelerometer per 1ms, the MCU interrupts the CPU to transfer this data. As shown in Figure 5a, reading an accelerometer sensor consumes $1W \times 0.3ms = 0.3mJ$ energy. After that, the MCU interrupts the CPU to transfer this sensor read (12 $bytes$). Then, the interrupt handler loads these 12 $bytes$ data from the I/O controller and stores them in memory, consuming around 0.1 ms. After these steps are repeated for 1000 times, finally the CPU processes the app-specific computation, and reports the number of steps detected in these 1000 sensor readings. Note that, in this scenario, due to frequent interrupts, the CPU is in the active mode all the time.

From the application's perspective, it is acceptable to batch 1000 samples in memory, as long as the output is computed in 1 second, (*QoS* or Quality of Service). Thus, a batching scheme is proposed, as shown in Figure 5b. This scheme takes advantage of the unused memory capacity on the MCU board and batches as much sensor data as possible without interrupting the CPU. During that time, the CPU is allowed to sleep for a longer time than baseline. By batching 1000 accelerometer sensor readings in the MCU board, the CPU is allowed to sleep during the 999.3 ms sensing time. After the MCU has collected all 1000 sensor samplings, it interrupts the CPU and transfers all data in bulk.

To better explain why *Batching* brings benefits, we breakdown the energy consumption of the step-counter app in the *Baseline* and *Batching* schemes, as shown in Figure 7, and provide the two key observations.

1. The CPU has better opportunity to sleep for longer. As shown in the first bar, the interrupt routine consumes 16% energy, and the app-specific computing routine consumes 77% energy in the *Baseline* design. By batching 1000 samples, the CPU can go into the sleep mode, where the power consumption is only 1.5 $Watts$. Taking advantage of the lower power consumption in the sleep mode, the energy consumption of the data transfer routine is reduced to 65% translating to ≈ 50% saving in total energy, as shown in the second bar of Figure 7.

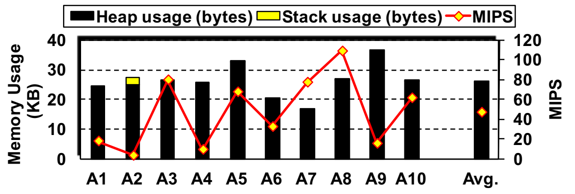


Figure 6: Memory usage and number of instruction executed.

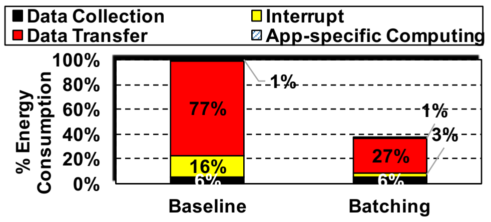


Figure 7: Energy breakdown of the Step-Counter in *Baseline* and *Batching*. Note that, in *Batching*, 1000 sensing samples are transferred in a bulk. Due to *Batching*, CPU can sleep for 93% of the time, translating to 63% energy savings.

2. The number of interrupts from the MCU to the CPU is reduced. As shown in Figure 5b, the number of interrupts from the MCU to the CPU is reduced from 1000 to only 1 in the step-counter app, consequently reducing the CPU interrupt handling energy. The *Batching* scheme reduces 79.68% interrupt energy, translating to a 13% reduction in total energy consumption, as shown in Figure 7.

Discussion: In the *Batching* scheme, both the interrupt and data transfer energy consumption are reduced. Further, as we will show in Section IV, reducing interrupts by *Batching* achieves 52% energy savings on an average, compared to the *Baseline*. However, this scheme does not completely mitigate the energy consumption incurred by the data transfer routine (which contributes 81% energy consumption to the total energy). The reason behind this can be explained as follows: even with the *Batching* scheme, the CPU is still active for

(1) transferring sensor data from the MCU board to the Main board ($100ms$) and (2) processing app-specific computations ($2.21ms$). The data transfer routine exists only because apps take advantage of relatively large compute capabilities of the CPU and memory to process the app-specific computation. The MCU also is equipped with compute capabilities, albeit lesser than the compute and memory capabilities of the CPU. If the whole app computation fits in the MCU's compute capabilities without any *QoS* violations, all of the data transfer cost and interrupt cost can be avoided. To explore whether such offload to MCU is possible or not, we next systematically characterize the capabilities and drawbacks of moving the app executions to the MCU in Section III-B.

B. Computation Offloading to MCU (COM)

A Computation Offloading to MCU (*COM*) scheme is proposed to address the data transfer cost by letting the computation take place entirely on the MCU. To achieve that, we examine the following design issues.

1) What are the requirements of IoT workloads?
2) If computation could be offloaded to MCU, how is the performance affected?
3) How can computation be offloaded to MCU?
4) How much can we benefit by offloading to the MCU?

We next address these four design issues one by one.

1) What are the requirements of IoT workloads?: To answer this question, we characterize the memory and CPU requirements of 10 popular IoT workloads (described in Table II) in Figure 6. In one running instance of each workload, we dump the heap and stack traces to discover the memory usage shown in the left y axis, and million instructions per second (MIPS) rates to illustrate the required CPU efforts, as shown in right y axis. The average memory usage in these ten workloads is 26.2 KB, including 25.8 kB of heap usage and 0.4 KB of stack usage. The average MIPS executed for these workloads is 47.45. Of these apps, earthquake requires the minimum memory usage (16.8 KB) and JPEG incurs the most memory usage (36.3 KB). On the compute side, Heartrate requires heavy compute demand (108.80 MIPS) and Step-counter requires very little computation throughput (3.94 MIPS).

Takeaway: The main CPU core in the IoT hub can potentially execute at the rate of $24,000$ MIPS. Out of these, the maximum MIPS required by any IoT app is only 0.5% (108 MIPS for earthquake app). This further encourages us to move towards the low-compute capacity, more energy-efficient MCU for all the computations.

2) If offloaded, how would the performance be affected?: Note that, after offloading, the MCU takes over the app-specific computations and only sends the end-to-end output results (e.g., the number of steps detected) to the CPU. The CPU is freed from handling the interrupts and processing app-specific computations, and either goes to sleep mode

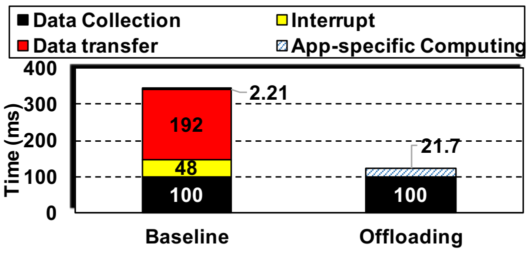


Figure 8: Timing breakdown for step-counter app in *Baseline* and *COM* schemes.

when idle, or processes much heavier tasks such as interacting with users through User Interfaces (UI), or training an AI model based on sensing history [36].

Since the MCU is slower than the CPU, we next explore the following question: *Does offloading speed up or slow down apps executions?* To better understand how the performance is affected after offloading, let us again study the timing breakdown in the step-counter app, as shown in Figure 8, for the four routines: data collection, interrupt, data transfer, and app-specific computation. Note that, after offloading, the step-counter app only has two routines, namely, data collection and app-specific computation; both interrupt routine and data transfer routine are eliminated. Therefore, there is a trade-off:

1) On MCU: App-specific computing takes more time, because MCU is slower than CPU.
2) On CPU: Both interrupt and data transfer routines timing overheads are introduced.

Summary: We can conclude that *if slowdown from MCU is less than the overhead caused by interrupts and data transfer, then the* COM *scheme could achieve better performance than the* Baseline*;* otherwise, the *COM* scheme is worse. Consider the step-counter example again: the computing routine in *COM* consumes 21.7 ms, while in *Baseline* consumes only 2.21 ms. However, because both the interrupt (48 ms) and data transfer (192 ms) overheads are eliminated, *COM* still has better performance than baseline: $(21.7 - 2.21) \; < (48 + 192)$.

3) How to offload?: Usually the CPU and MCU run different operating systems (OS) and use different ISAs and different compilers. For example, Raspberry Pi 3B (used as the main board in this work) has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, runs Raspbian OS [37] as default, implements an ARM V8 ISA. The ESP8266 MCU (used as an MCU board in this work) has an L106 32-bit RISC microprocessor core, runs RTOS as default, and implements a Xtensa LX ISA [38]. The ESP8266 MCU has its own compiler ABIs (Espressif).

To offload IoT apps from the Main board to the MCU board, developers have to port all codes to using the specific ISA and compiler required by the MCU. For example, the code for sleeping at the CPU for 1 second is $sleep(1)$, while on MCU, the same code is $delay(1000)$. After porting, the binaries need to be built using specific tool-chains deployed

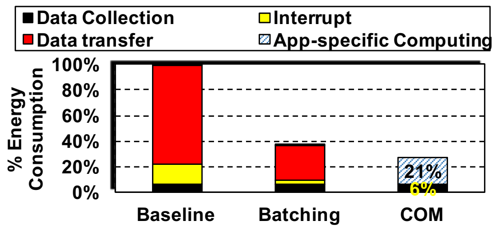


Figure 9: Energy breakdown of the step-counter app when using (a) *Baseline*; (b) *Batching* scheme; (c) *COM* scheme.

to the MCU board. After deployment, there usually are several ways to execute the offloaded image: (1) Binding all apps to a binary and executing it sequentially [39], and (2) Sharing time slacks and context switching [40], etc.

4) What are the benefits of offloading to MCU?: By offloading to the MCU, the CPU is freed to either handle other heavy tasks, or go into sleep when idle. Here, we assume that, during processing of IoT workloads on the MCU, the CPU is in the sleep mode, which consumes only around 30% power compared to the active mode.

As shown in Figure 9, we again break down the energy consumption of the step-counter app in the *Baseline*, the *Batching*, and the *COM* schemes. One can now observe that: **App-specific computing routine consumes more energy:** As discussed above, MCU is slower than CPU (For example, ESP8266 [38] is around $19\times$ slower than Raspberry Pi 3B [20]). Therefore, the app-specific tasks running on MCU consumes 21.7 ms, compared to 2.21 ms in *Baseline* and *Batching* scheme as shown in Figure 8. During this time of the app-specific computing, note that CPU is sleeping and also consumes energy. Because of this, the app-specific computing routine in the *COM* scheme consumes 21% energy, more than 1% in baseline and *Batching* scheme.

Overall, when compared to the *Baseline*, 73% energy is saved by the *COM* scheme (63% better than *Batching*).

IV. Experimental Evaluation

We use commercially available IoT hardware platforms to analyze the energy efficiency and performance implications of the discussed schemes. In this section, we first describe the hardware platforms, various sensors and workloads used in this study, and then present our experimental results.

A. The IoT Platform for Our Experiments

Recall that the IoT hub architecture described in Figure 2a consists of one Main board, one MCU board connected to the Main board through I/O controller, and a set of sensors attached to the MCU board which sends data to the Main board. For the Main board, we used a Raspberry Pi 3B, and for the MCU board, we used ESP8266 which is connected to the Raspberry Pi via a miniUSB UART cable. The Raspberry Pi integrates a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB LPDDR2 DRAM at 900 MHz, and several accelerators. The ESP8266 board consists of a L106 32-bit RISC microprocessor core based on the Tensilica Xtensa Diamond Standard 106Micro running at 80 MHz, 80 KB user-data RAM, and PIO buses.

Table I: Specifications of sensors studied in this work. Only `S10` is *MCU-unfriendly*, rest are *MCU-friendly*.

No.	**Sensor Name**	**Input Bus type**	**Read Time (ms)**	**Power (mW)**			**Output Data [Type,Size]**	**Sampling Rate(Hz)**	
				Min.	**Typical**	**Max.**		**Max.**	**QoS**
S1	Barometer [43]	SPI	37.5	2.12	19.47	28.93	Double,8B	157	10
S2	Temperature [44]	I2C	18.75	1	13.5	20	Double,8B	120	10
S3	Fingerprint [45]	TTL Serial	850	432	600	900	Signature,512B	-	-
S4	Accelerometer [24]	Analog	0.5	0.63	1.3	1.75	Int*3,12B	1M	1k
S5	Air Quality [46]	I2C	0.96	1.2	30	46	Int,4B	400	200
S6	Pulse [47]	Analog	0.1	9.9	15	22	Int,4B	1M	1k
S7	Light [25]	I2C	0.1	16.8	21	25.2	Double,8B	400k	1k
S8	Sound [26]	Analog	0.1	16	40	96	Int, 4B	1M	1k
S9	Distance [48]	Analog	0.2	120	150	175	Double,8B	5k	1k
S10	Low-Res. Img [49]	TTL Serial	183.64	30	125	140	RGB,24kB	-	-
	>1k High-Res. Img [50]	**Camera Serial**	**500**	**382**	**425**	**700**	**RGB,619kB**	**-**	**-**

B. Measurement and Tracing Tools

Since there are no open-sourced IoT simulation platform available, we relied on real hardware. Hence, for various characterizations, we instrumented the Raspbian Linux Kernel in the IoT hub with `oprof` library [41] to analyze memory, interrupts, and MIPS characteristics of IoT apps. For power measurements, we connected the power delivery socket of the IoT hub to Monsoon Power Monitor [42], and dumped fine-grain power statistics (every 100 ns).

C. IoT Sensor Specifications

The salient specification of the ten sensors used in this work are summarized in Table I. Note that, each sensor is different from others in terms of its sampling rate, timing parameters, bus type, power consumption, or output data. The maximum sampling rate of a sensor is limited by various parameters. Different apps require different sampling rates. We refer to this application-specific feature as *QoS sampling rate* in this work. For example, the QoS sampling rate of the accelerometer (used by the step-counter app) is $1kHz$. Sensor data are passed on to apps after the sensor's driver processes them (as described in Section II). If a sensor's driver routines can be handled by an MCU, that sensor is said to be *MCU-friendly*; otherwise, it is considered *MCU-unfriendly*. For example, a high resolution image sensor (in Table I) needs substantial compute power and memory size and thus, it is MCU-unfriendly due to the MCU's inherent limited computing and memory capacity. On the other hand, a low resolution image sensor is MCU-friendly.

D. IoT Workloads

We used 11 popular workloads (10 light-weight apps that can be offloaded and 1 heavy-weight app used for studying *Batching+COM = BCOM* technique's benefits), each exercising 1-5 of the above sensors during its execution. The chosen workloads belong to diverse app domains such

Table II: Salient features of the workloads used in this study. A1 to A10 are light-weight and can be offloaded to MCU (*COM* optimization). A11 is heavy-weight, and requires the main board's CPU for computation.

No.	Benchmark	Category	Sensor Used	Sensor Data (KB)	# Interrupts	User-level Tasks
A1	CoAP Server	Building Automation	S7, S8	11.72	2000	Constrained Application Protocol
A2	Step counter	Health Care	S4	11.72	1000	Step-detection Algorithm
A3	arduinoJSON	Protocol Library	S1, S2	0.16	20	JSON Formatting
A4	M2X	Cloud Communication	S1, S2, S4, S5, S7	20.47	2220	Cloud Interfacing with AT&T
A5	Blynk	Smartphone Interactions	S1, S2, S4, S5, S10	36.91	1221	Platform interacting with Smartphones
A6	Dropbox Manager	Web Control	S8, S9	11.72	2000	File Sync, Upload, etc.
A7	Earthquake Detection	Smart City	S4	11.72	1000	Earthquake Predicting Algorithm
A8	Heartbeat Irregularity Detection	Health Care	S6	3.91	1000	ECG Feature-extraction
A9	JPEG Decoder	Security	S10	23.81	1	Inverse Discrete Cosine Transform (IDCT)
A10	Fingerprint Register	Security	S3	0.5	1	Fingerprint Enroll, Identify, etc
A11	Speech-To-Text	Smart City	S8	5.86	1000	Voice-to-text conversion

as HealthCare, Building Automation, IoT Protocol servers, SmartCity, etc., as described in Table II.

All these workloads need inputs from various sensors, with 0.16 KB to 37 KB of data moved before executing the app-specific computation, and they receive between 1 and 2000 interrupts from the MCU to the CPU to get the data into the CPU buffers. At a high level, the app-specific tasks performed on the data can be grouped in two categories:
①**IoT Protocol Servers:** CoAP Server [51], arduinoJSON [52], M2X [53], Blynk [54] and Dropbox Manager [55] workloads use the MCU to read various sensors. The data are then processed via the workload-specific protocols to get wrapped into various objects and subsequently transferred to a smartphone client or the cloud using WiFi, Ethernet, etc. For example, CoAP [56], and JSON [57] are open standards for IoT and general object transfers in the web, while Dropbox [55] and M2X [53] are vendor-specific standards.
②**Stand-alone IoT workloads:** Apart from these generic protocol-based apps, we also use stand-alone apps from three different domains, with two apps from each domain.
i) Health Care Domain: Step counter [33]: counts the number of steps walked/run by a user wearing the device. Heartbeat irregularity detection [58]: analyzes the fluctuations in a person's heart beat based on the heart rate sensor.
ii) Security Domain: JPEG decoder [59]: performs IDCT algorithm [60] on the raw camera frames. Fingerprint register [61]: uses the fingerprint sensor to check whether a given fingerprint matches a set of registered fingerprints.
iii) Smart City Domain [62]: Earth quake detection [63]: uses accelerometer to sense and detect if there is an earthquake. Speech-To-Text [64]: uses sound input on sphinxbase machine learning models [65] to produce output text.

We study the *Batching* and *COM* schemes on these apps.

E. Experimental Results

We characterized both the compute and memory requirements of the first 10 apps (A1 to A10) in Table II.

1) Single-app Scenarios: For each of the 10 IoT workloads, we compared three schemes, namely, *Baseline*, *Batching*, and *COM*, and report the energy consumption results in Figure 10. All the results presented in this graph are *normalized* w.r.t. the *Baseline* scheme (explained below).

Baseline: Here, the data transfer routine consumes around 81% of energy on average, and the interrupt routine consumes 10% energy on average. These two routines are the top two most expensive routines across all workloads. Note that this observation clearly confirms the discussion in Section II-C. As discussed by prior works [8], [11], [14], [15], [66], data transfers are among the costliest operations in such energy-constrained systems. Next, data collection routine consumes 6% energy on average, because of the low power consumption of the MCU. The app-specific computing routine consumes even lesser than that. Across all the workloads studied, these two routines (app specific computation and sensor data collection) are consistently the two least energy consuming routines. For example, in apps such as step-counter (A2), these routines account for around 9% of the total energy consumed. In most apps, the data collection routine typically consumes more energy than the app-specific computing routine, because the former relatively consumes more time to read sensors by MCU than compute by CPU. However, in a few apps, such as earthquake detection (A7), the app-specific computing routine includes analyzing the accelerometer data and confirming whether an actual earthquake happened by accessing public earthquake APIs in real-time to verify if tremors were detected by the sensors. Such tasks are relatively heavier and contribute to the energy consumption. We next show the benefits obtained from *Batching* and *COM* schemes.

Batching: We observe that, by batching sensor readings in MCU, the average total energy consumption is reduced by 52%. These savings come primarily from two parts: *allowing CPU to sleep longer:* During sensor readings in MCU (e.g., $1000ms$ shown in Figure 5b), CPU goes to the sleep mode, which saves, on an average, 50% data transfer energy, translating to 43% total energy savings. *Number of interrupts is reduced:* The overheads of interrupt handling are reduced mainly because the number of interrupts is reduced by 95%, which translates to 2% total energy. Combining these two savings, the *Batching* scheme provides about 52% energy savings, compared with the *Baseline*.

COM: By offloading computations to the MCU, the app-specific compute routine consumes on average 9% energy,

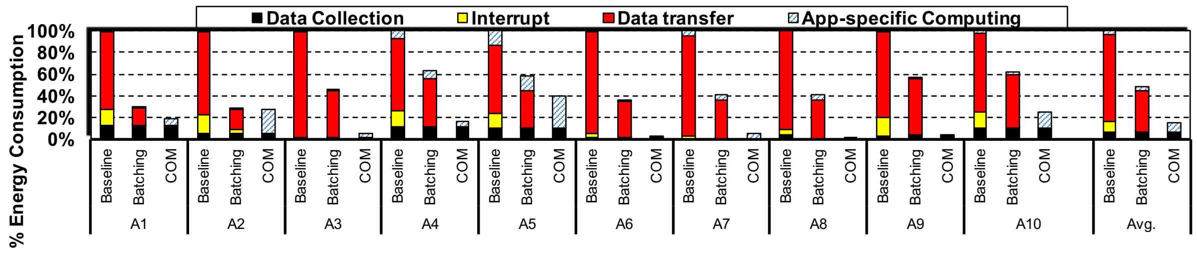


Figure 10: Normalized energy breakdown with three schemes shown across 10 workloads. The lower the better.

3% larger than *Baseline*, because the MCU is slower than CPU and during the longer computing time, CPU is in sleep mode and still consumes energy. However, by offloading to MCU, the energy consumption of both the interrupt and data transfer routines are eliminated. These achieved savings are larger than the overheads introduced by slower MCU processing. Consequently, on an average, the *COM* scheme reduces 85% of the energy consumed by the *Baseline*.

2) Multiple-apps Scenarios: To compare *Batching* and *COM* against another proposal (*BEAM* [4]), we focused on multiple app scenarios, as *BEAM* only works when multiple applications share one or more sensors. Among the ten workloads (A1-A10) discussed above, we study all 14 combinations of workloads which share sensor(s) with another. We breakdown the energy consumption of all combinations as shown on the x-axis of Figure 11. Overall, one can observe that, BEAM gives around 29% energy saving on average compared to the *Baseline*, while *COM* provides an average of 70% savings across all workloads. In *BEAM*, the A2+A7 (step-counter app and earthquake app) scenario provides most of the energy savings (48.20%), because both A2 and A7 use the *same* accelerometer sensor with the *same* sampling rate. Therefore, by sharing the sensor between them as such, this scenario reduces half of the energy consumption caused by interrupts and data movements. However, the A5+A7 (Blynk app and earthquake app) scenario experiences only 8.46% energy saving when *BEAM* is used. This is primarily because Blynk collects data from 5 different sensors and only one sensor (accelerometer) is shared with the earthquake app. Clearly, with such a low volume of sensor data getting shared between these workloads, *BEAM* cannot save much energy.

3) Heavy-weight app involved scenarios: All 10 workloads discussed above are light-weight and can be offloaded to the MCU. To study scenarios involving heavy-weight app(s) which cannot be offloaded, we further introduce one additional app: speech-to-text [64] (A11). This app takes the audio signals collected by one sound sensor and references an offline PocketSphinx [67] model to convert the audio to text. To convert 1 second audio recording, A11 requires 4683 $MIPS$ CPU throughput and a 1.43 GB memory footprint; therefore, it cannot be offloaded to the MCU. Figure 12(a) shows the energy breakdown when A11 runs alone on the CPU for the *Baseline* and *Batching* schemes. From this figure, one can observe that the app-specific routine dominates the energy consumption (78%) in the *Baseline*. On the other hand, via *Batching*, 5% energy savings are achieved, much less than 52% in Figure 10. The reason is the data transfer and interrupt routines only consume 8% energy; therefore, even though *Batching* brings 62.5% savings in these two routines, that only translates to 5% of total energy.

Figure 12(b) focuses now on multiple (including both heavy-weight and light-weight) apps execution scenarios, and the energy breakdown when A11 and A6 run concurrently under the *Baseline*, *BEAM*, *Batching*, and *BCOM* schemes. *BEAM* only provides 2% energy savings, while *Batching* gains 7%, shown in the third bar. Note that, with the *BCOM* scheme, A6 is offloaded to MCU and A11 runs on CPU. By doing this, *BCOM* provides 9% energy savings.

To further show the scenario where more light-weight apps as well as A11 run concurrently, Figure 12(c) breaks downs the total energy consumption, when A11, A6 and A1 run concurrently. In this scenario, *BEAM* gains only 2% savings, while *Batching* achieves around 8%. On the other hand, when employing the *BCOM* scheme, both A1 and A6 are offloaded to MCU, and this provides 10% energy saving. **Takeaways:** The *COM* is suitable for light-weight apps; *Batching* is suitable for heavy-weight apps; and when both (light-weight + heavy-weight) run concurrently, these two schemes are orthogonal and complement one another.

F. Sensitivity Studies

Performance Speedup: To show the performance speedup by the *COM* scheme, Figure 13 normalizes the performance of ten workloads (A1-A10) w.r.t. that of *Baseline*. On average, *COM* provides 1.88× speedup compared to *Baseline*. In fact, 8 out of the 10 workloads tested benefits from the *COM* scheme, and only 2 workloads, arduinoJSON (A3, 0.9× slower) and heartbeat irregularity detection (A8, 0.8× slower) experience slowdown. The reasons for these slowdowns are: A3 only collects a small amount of sensor data (0.16 KB), and then processes the JSON formatting, which involves string-to-double conversion, memory access, etc. These tasks can be handled by the Main board within 0.45 ms, while requiring 7 ms on the MCU board; and in the case of the A8 workload, the extra time needed due to slower MCU is greater than the latency incurred by interrupts and data transfer. As a result, *COM* has slightly worse performance degradation than *Baseline* when executing A3

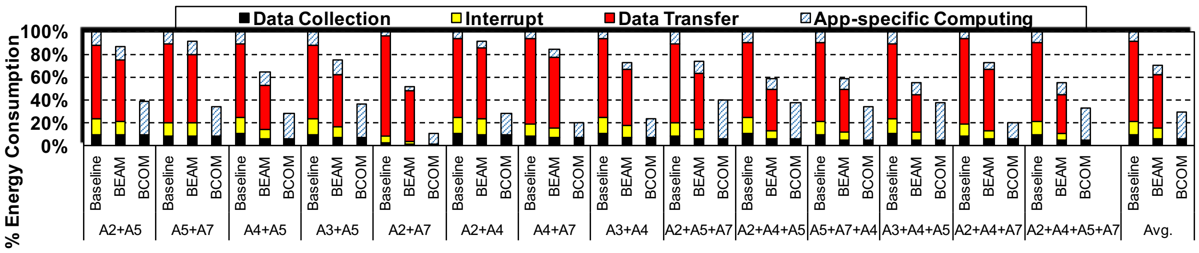


Figure 11: Normalized energy breakdown with three schemes when multiple apps run concurrently. The lower the better.

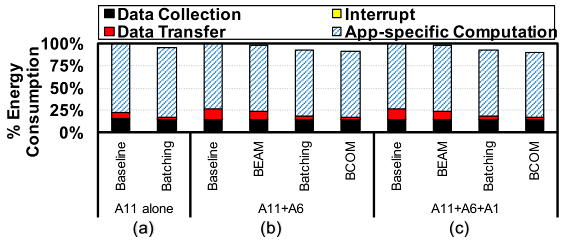


Figure 12: Energy breakdown when one heavy-weight app (speech-to-text, A11) is involved.

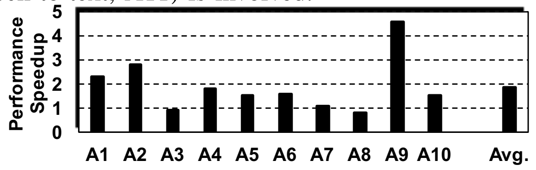


Figure 13: Performance speedup normalized to baseline.

and A8. However, *COM* provides averaged 2.14× speedups for other apps, compared to the *Baseline*.

Future work: Note that this work does not consider any architectural changes. Software-based optimizations (*Batching* and *COM*) studied in this paper work well for light-weight workloads, however, they fail to achieve large savings in heavy-weight workloads, that are too compute and memory intensive for the MCUs to handle. The energy consumption of data transfer is high, mainly because there is no DMA or shared-memory hardware support and both CPU and MCU have to be involved during the transfers. As our future work, we plan to explore hardware optimizations to address the energy inefficiencies in heavy-weight workloads.

V. Related Work

We summarize the related works into three areas:

Energy Optimizations in Embedded Systems: Various optimizations in the embedded domain have proposed individual component specific optimizations [4], [66], [68]–[83]. For example, [68] optimizes IO energy by exploiting data bus under-utilization; [69] optimizes the peak power for CPUs; [70] identifies time slack between DRAM and CPU computations to save energy. All these techniques concentrate only on a few distinct components in the IoT hub such as CPU side, sensing side, etc., and as we show in this paper, the entire pipeline from sensing to computations requires optimizations to save energy.

Data Transfer Optimizations in High-end Heterogeneous Systems: Data movement optimizations have been well studied in high-end heterogeneous systems [84]–[100]. [101] designs a set of APIs to offload a data intensive task to a SSD processor. [102] proposes a data encoding technique based on online data clustering to save energy. [103] proposes spatial in-memory big data analytic system to offer efficient in-memory spatial queries and analytic. Our work targets low power systems with light-weight computations that can be offloaded to a small MCU itself, and help the CPU to aggressively transition to the sleep mode and save energy.

Computational Offloading: Computational offloading have been well studied in GPU/mobile/IoT systems [104]–[111]. [112] presents cross-architecture computation offloading framework for native apps. [113] reduces energy for sensing a GPS location on mobile phones by offloading GPS processing to the cloud. All these techniques do not address how to partition compute to CPU and MCU. *COM* leverages the compute resources at MCU to effectively reduce the CPU energy consumption and save the overall execution energy in both single and concurrent app execution scenarios.

VI. Conclusions

Energy efficiency is the most important parameter for IoT platforms, which have and will dominate the digital transformation landscape in foreseeable future. Two sources of energy inefficiencies in a typical IoT system are the frequent CPU interrupts and data transfers between the MCU and CPU. To address these inefficiencies, two simple, yet effective techniques, *Batching* and *COM*, are studied in this work. *Batching* only interrupts the CPU after collecting N sensor data points at the MCU to minimize the interrupt overhead; and *COM* offloads the app-specific computations to the MCU to minimize the data transfer overhead. Our experimental results on a prototype design show that for light-weight workloads, *Batching* provides on an average 52% energy saving, and *COM* provides 85% for single-app scenarios. Working together, *BCOM* provides 10% and 70% energy savings for multiple-apps scenarios, with and without heavy-weight apps involved, respectively.

Acknowledgment

This work has been supported in part by NSF grants 1439021, 1629915, 1526750, 1629129, 1763681, 1317560, 1822923, 1626251 and 1439057, and a DARPA/SRC JUMP grant. We thank J. Sampson, A. Pattnaik, P. Thinakaran, J. Gunasekaran and A. Sarma for their feedback on this paper.

References

[1] Liam Tung, “IoT Devices will Outnumber the World’s Population this Year for the First Time.” ”https://zd.net/2OOqLGj”, 2018.

[2] Statista, “IoT Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions).” ”https://bit.ly/2dRtPP0”, 2018.

[3] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou, “E-Android: A New Energy Profiling Tool for Smartphones,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 492–502.

[4] C. Shen, R. P. Singh, A. Phanishayee, A. Kansal, and R. Mahajan, “Beam: Ending Monolithic Applications for Connected Devices,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 143–157.

[5] S. Pan, C. Ruiz, J. Han, A. Bannis, P. Tague, H. Y. Noh, and P. Zhang, “UniverSense: IoT Device Pairing Through Heterogeneous Sensing Signals,” ser. HotMobile, 2018, pp. 55–60.

[6] N. Klingensmith and S. Banerjee, “Hermes: A Real Time Hypervisor for Mobile and IoT Systems,” ser. HotMobile, 2018, pp. 101–106.

[7] Z. Tian, K. Wright, and X. Zhou, “Lighting Up the Internet of Things with DarkVLC,” ser. HotMobile, 2016, pp. 33–38.

[8] J. Li, A. Liu, G. Shen, L. Li, C. Sun, and F. Zhao, “Retro-VLC: Enabling Battery-free Duplex Visible Light Communication for Mobile and IoT Applications,” ser. HotMobile, 2015, pp. 21–26.

[9] S. Luo, C. Zhuo, and H. Gan, “Noise-aware DVFS Transition Sequence Optimization for Battery-powered IoT Devices,” ser. Proceedings of the Design and Automation Conference (DAC), 2018, pp. 27:1–27:6.

[10] M. S. Golanbari and M. B. Tahoori, “Runtime Adjustment of IoT System-on-chips for Minimum Energy Operation,” ser. Proceedings of the Design and Automation Conference (DAC), 2018, pp. 145:1–145:6.

[11] S. Sen, “Invited - Context-aware Energy-efficient Communication for IoT Sensor Nodes,” ser. Proceedings of the Design and Automation Conference (DAC), 2016, pp. 67:1–67:6.

[12] M. Sanduleanu and I. A. M. Elfadel, “Invited - Ultra Low Power Integrated Transceivers for Near-field IoT,” ser. Proceedings of the Design and Automation Conference (DAC), 2016, pp. 143:1–143:6.

[13] S. Gangopadhyay, S. B. Nasir, and A. Raychowdhury, “Integrated Power Management in IoT Devices Under Wide Dynamic Ranges of Operation,” ser. Proceedings of the Design and Automation Conference (DAC), 2015, pp. 149:1–149:6.

[14] X. Xu, Y. Shen, J. Yang, C. Xu, G. Shen, G. Chen, and Y. Ni, “PassiveVLC: Enabling Practical Visible Light Backscatter Communication for Battery-free IoT Applications,” ser. MobiCom, 2017, pp. 180–192.

[15] Y. Li, Z. Chi, X. Liu, and T. Zhu, “Chiron: Concurrent High Throughput Communication for IoT Devices,” ser. MobiSys, 2018, pp. 204–216.

[16] J. Hester and J. Sorber, “Flicker: Rapid Prototyping for the Batteryless Internet-of-Things,” ser. SenSys ’17, 2017, pp. 19:1–19:13.

[17] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, “Bespoke Processors for Applications with Ultra-low Area and Power Constraints,” ser. Proceedings of the International Symposium on Computer Architecture (ISCA), 2017, pp. 41–54.

[18] Y. Chen, S. Lu, C. Fu, D. Blaauw, R. Dreslinski, Jr., T. Mudge, and H.-S. Kim, “A Programmable Galois Field Processor for the Internet of Things,” ser. Proceedings of the International Symposium on Computer Architecture (ISCA), 2017, pp. 55–68.

[19] NXP, “MCIMX7SABRE: SABRE Board for Smart Devices Based on the i.MX 7Dual Applications Processors.” ”https://bit.ly/2STZ3sn”, 2018.

[20] Raspberry Pi, “Rasperry Pi 3 Model B,” ”https://bit.ly/1WTq1N4”, 2018.

[21] Intel, “Intel Joule Module Expansion Board Hardware Guide.” ”https://intel.ly/2LNWyoS”, 2018.

[22] Intel, “Intel Edison Compute Module Hardware Guide.” ”https://intel.ly/2uTPkJr”, 2018.

[23] Raspberry Pi, “BCM2835 Interrupt Controller ,” ”https://bit.ly/2IlEQes”, 2018.

[24] Sparkfun, “ADXL335, A Small, Low Power, 3-Axis Accelerometer.” ”https://bit.ly/23Qjc4j”, 2018.

[25] ROHM Electronic Components, “Digital 16bit Serial Output Type Ambient Light Sensor IC,” ”https://bit.ly/2LQIRW6”, 2018.

[26] Li, Jiankai, “Grove Sound Sensor,” ”https://bit.ly/2A6jgHA”, 2018.

[27] M. Mikusz, S. Houben, N. Davies, K. Moessner, and M. Langheinrich, “Raising awareness of IoT sensor deployments,” in *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, 2018, pp. 1–8.

[28] R. Piedrahita, Y. Xiang, N. Masson, J. Ortega, A. Collier, Y. Jiang, K. Li, R. P. Dick, Q. Lv, M. Hannigan, and L. Shang, “The Next Generation of Low-cost Personal Air Quality Sensors for Quantitative Exposure Monitoring,” *Atmospheric Measurement Techniques*, pp. 3325–3336, 2014.

[29] B. Xu, D. Kim, D. Li, J. Lee, H. Jiang, and A. O. Tokuta, “Fortifying barrier-coverage of wireless sensor network with mobile sensor nodes,” in *Wireless Algorithms, Systems, and Applications*, Z. Cai, C. Wang, S. Cheng, H. Wang, and H. Gao, Eds. Springer International Publishing, 2014, pp. 368–377.

[30] B. Xu, Y. Zhu, D. Kim, D. Li, H. Jiang, and A. O. Tokuta, “Strengthening Barrier-coverage of Static Sensor Network with Mobile Sensor Nodes,” *Wirel. Netw.*, pp. 1–10, Jan. 2016.

[31] J. Frst, K. Chen, M. Aljarrah, and P. Bonnet, “Leveraging Physical Locality to Integrate Smart Appliances in Non-Residential Buildings with Ultrasound and Bluetooth Low Energy,” in *IoTDI*, 2016.

[32] C. Doukas and I. Maglogiannis, “Bringing IoT and Cloud Computing towards Pervasive Healthcare,” in *UbiComp*, 2012, pp. 922–926.

[33] Bobra, Neraj, “Embedded Pedometer,” ”https://bit.ly/2LuBgQb”, 2018.

[34] R. Zahir, M. Ewert, and H. Seshadri, “The Medfield Smartphone: Intel Architecture in a Handheld Form Factor,” *IEEE Micro*, pp. 38–46, 2013.

[35] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, “Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017, pp. 517–531.

[36] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, “DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing,” in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 351–360.

[37] Raspberry Pi, “Raspbian,” ”https://bit.ly/2zsBPRS”, 2018.

[38] Xtensalx, “Xtensalx Overview Handbook,” ”https://bit.ly/2KCOx4G”, 2018.

[39] expresslogic, “Real-time Operation System,” ”https://bit.ly/2vMh6Yb”, 2018.

[40] ARM, “ARM Cortex-M7,” ”https://rtos.com/”, 2018.

[41] Linux, “Oprofile,” ”https://bit.ly/2Mr2HrB”, 2018.

[42] Monsoon Solutions Inc., “High Voltage Power Monitor.” ”https://bit.ly/2LWkJpj”, 2018.

[43] Bosch Sensortec, “BMP280 Digital Pressure Sensor,” ”https://bit.ly/2NKsOdc”, 2018.

[44] Bosch Sensortec, “BMP180 Digital Pressure Sensor,” ”https://bit.ly/2sj53AK”, 2018.

[45] Adafruit Industries, “Adafruit Optical Fingerprint Sensor,” ”https://bit.ly/2mJwFf5”, 2018.

[46] Sparkfun, “Ultra-Low Power Digital Gas Sensor for Monitoring Indoor Air Quality,” ”https://bit.ly/2qQKqKu”, 2018.

[47] Jenkin, Michael and Roumani, Hamzeh, “Pulse Sensor,” ”https://bit.ly/2v9gQlF”, 2018.

[48] Parallax Inc., “PING Ultrasonic Distance Sensor,” ”https://bit.ly/2AdGFH1”, 2018.

[49] Arducam, “ArduCAM Mini Released,” ”https://bit.ly/2M6xCwk”, 2018.

[50] SONY, “Diagonal 7.20 mm Approx. 8.51M-Effective Pixel Color CMOS Image Sensor,” ”https://bit.ly/2AXnFx0”, 2018.

[51] 1248.io, “MicroCoAP,” ”https://bit.ly/2NN20ZA”, 2018.

[52] Blanchon, Benot, “ArduinoJson,” ”https://bit.ly/2lDet6m”, 2018.

[53] ATT M2X, “Arduino M2X API Client,” ”https://bit.ly/2v94VEm”, 2018.

[54] Blynk Inc., “Blynk,” ”https://bit.ly/2mNkgqw”, 2018.

[55] lucasromeiro, “Dropbox Manager,” ”https://bit.ly/2v7DSJM”, 2018.

[56] CoAP, “Constrained Application Protocol,” ”https://bit.ly/2OPnl6d”, 2018.

[57] JSON, “Introducing JSON,” ”https://bit.ly/2kh3jU4”, 2018.

[58] tutRPi, “Raspberry Pi Heartbeat Sensor,” ”https://bit.ly/2AduvOk”, 2018.

[59] Bodmer, “Arduino JPEGDecoder Library,” ”https://bit.ly/2NKsZoS”, 2018.

[60] wikipedia, “Discrete Cosine Transform ,” ”https://bit.ly/2hOMhLG”, 2018.

[61] D. Shah and V. Haradi, “IoT Based Biometrics Implementation on Raspberry Pi,” pp. 328 – 336, 2016.

[62] E. Ngai, F. Dressler, V. Leung, and M. Li, “Guest Editorial Special Section on Internet-of-Things for Smart Cities and Urban Informatics,” *IEEE Transactions on Industrial Informatics*, pp. 748–750, 2017.

[63] Intel, “Earthquake Detector in Javascript,” ”https://bit.ly/2AduQ3y”, 2018.

[64] CMUSphinx, “CMU Sphinx common libraries,” ”https://bit.ly/2OQj8z9”, 2018.

[65] CMUSphinx, “Open Source Speech Recongnition Toolkit,” ”https://bit.ly/2KBoJG7”, 2018.

[66] Y. Chen, S. Lu, H. Kim, D. Blaauw, R. G. Dreslinski, and T. Mudge, “A low power software-defined-radio baseband processor for the Internet of Things,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 40–51.

[67] cmusphinx, “PocketSphinx 5prealpha,” ”https://bit.ly/2mKcUnJ”, 2018.

[68] Y. Song and E. Ipek, “More is Less: Improving the Energy Efficiency of Data Movement via Opportunistic Use of Sparse Codes,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.

[69] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Determining Application-specific Peak Power and Energy Requirements for Ultra-low Power Processors," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[70] H. Cherupalli, R. Kumar, and J. Sartori, "Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[71] J. Ajay, C. Song, A. S. Rathore, C. Zhou, and W. Xu, "3DGates: An Instruction-Level Energy Analysis and Optimization of 3D Printers," ser. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017, pp. 419–433.

[72] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. Feng, "Measuring and Modeling On-chip Interconnect Power on Real Hardware," in *IISWC*, 2016, pp. 1–11.

[73] M. Gorlatova, J. Sarik, G. Grebla, M. Cong, I. Kymissis, and G. Zussman, "Movers and Shakers: Kinetic Energy Harvesting for the Internet of Things," ser. SIGMETRICS, 2014, pp. 407–419.

[74] F. Lai, M. Radi, O. Chipara, and W. G. Griswold, "Workload Shaping Energy Optimizations with Predictable Performance for Mobile Sensing," in *IoTDI*, 2018, pp. 177–188.

[75] B. Campbell, Y. Kuo, and P. Dutta, "From Energy Audits to Monitoring Megawatt Loads: A Flexible and Deployable Power Metering System," in *IoTDI*, 2018, pp. 189–200.

[76] D. Ma, G. Lan, W. Xu, M. Hassan, and W. Hu, "SEHS: Simultaneous Energy Harvesting and Sensing Using Piezoelectric Energy Harvester," in *IoTDI*, 2018, pp. 201–212.

[77] F. Renna, J. Doyle, V. Giotsas, and Y. Andreopoulos, "Query Processing for the Internet-of-Things: Coupling of Device Energy Consumption and Cloud Infrastructure Billing," in *IoTDI*, 2016, pp. 83–94.

[78] T. Abdelzaher, M. T. A. Amin, A. Bar-Noy, W. Dron, R. Govindan, R. Hobbs, S. Hu, J. Kim, J. Lee, K. Marcus, S. Yao, and Y. Zhao, "Decision-Driven Execution: A Distributed Resource Management Paradigm for the Age of IoT," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1825–1835.

[79] H. Chen, W. Lou, Z. Wang, and F. Xia, "On Achieving Asynchronous Energy-Efficient Neighbor Discovery for Mobile Sensor Networks," *IEEE Transactions on Emerging Topics in Computing*, pp. 553–565, 2018.

[80] P. V. Rengasamy, H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "CritICs Critiquing Criticality in Mobile Apps," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018, pp. 867–880.

[81] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Characterizing Diverse Handheld Apps for Customized Hardware Acceleration," in *IISWC*, 2017, pp. 187–196.

[82] H. Zhang, P. V. Rengasamy, N. C. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "FLOSS: FLOw Sensitive Scheduling on Mobile Platforms," in *Proceedings of the Design and Automation Conference (DAC)*, 2018, pp. 173:1–173:6.

[83] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP Chains on Handheld Platforms," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 655–667.

[84] S. George, M. J. Liao, H. Jiang, J. B. Kotra, M. T. Kandemir, J. Sampson, and V. Narayanan, "MDACache: Caching for Multi-Dimensional-Access Memories," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018, pp. 841–854.

[85] T. Zhang, X. Zhang, F. Liu, H. Leng, Q. Yu, and G. Liang, "eTrain: Making Wasted Energy Useful by Utilizing Heartbeats for Mobile Data Transmissions," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 113–122.

[86] X. Zhang and G. Cao, "Efficient Data Forwarding in Mobile Social Networks with Diverse Connectivity Characteristics," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 31–40.

[87] D. C. Schmidt, J. White, and C. D. Gill, "Elastic Infrastructure to Support Computing Clouds for Large-Scale Cyber-Physical Systems," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2014, pp. 56–63.

[88] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "C-States: Fine-grained GPU Datapath Power Management," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 17–30.

[89] G. Li, J. He, S. Peng, W. Jia, C. Wang, J. Niu, and S. Yu, "Energy Efficient Data Collection in Large-scale Internet of Things via Computation Offloading," *IEEE Internet of Things Journal*, 2018.

[90] X. Tang, A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. Das, "Quantifying Data Locality in Dynamic Parallelism in GPUs," *Proc. ACM Meas. Anal. Comput. Syst.*, pp. 39:1–39:24, 2018.

[91] S. Qayyum, M. Shahriar, M. Kumar, and S. K. Das, "PCV: Predicting contact volume for reliable and efficient data transfers in opportunistic networks," in *38th Annual IEEE Conference on Local Computer Networks*, 2013, pp. 801–809.

[92] W. Du, J. C. Liando, H. Zhang, and M. Li, "When Pipelines Meet Fountain: Fast Data Dissemination in Wireless Sensor Networks," ser. SenSys '15, 2015, pp. 365–378.

[93] H. Huang, S. Guo, W. Liang, and K. Wang, "Online Green Data Gathering from Geo-Distributed IoT Networks via LEO Satellites," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.

[94] J. Misic and V. B. Misic, "Lightweight Data Streaming from IoT Devices," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.

[95] T. Chan, Y. Ren, Y. Tseng, and J. Chen, "eHint: An Efficient Protocol for Uploading Small-Size IoT Data," in *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, 2017, pp. 1–6.

[96] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters," in *International Conference on Distributed Computing Systems*, 2017, pp. 977–987.

[97] P. Thinakaran, J. Raj, B. Sharma, M. T. Kandemir, and C. R. Das, "The Curious Case of Container Orchestration and Scheduling in GPU-based Datacenters," in *SoCC*, 2018, pp. 524–524.

[98] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Exploiting Staleness for Approximating Loads on CMPs," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2015, pp. 343–354.

[99] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018, pp. 533–545.

[100] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU Memory System for Multi-Application Execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 223–234.

[101] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading Communication with Computing Near Storage," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[102] S. Wang and E. Ipek, "Reducing Data Movement Energy via Online Data Clustering and Encoding," *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[103] N. Agrawal, "Simba: Building Data-Centric Applications for Mobile Devices." USENIX Association, 2015.

[104] J. Wang, J. Tang, D. Yang, E. Wang, and G. Xue, "Quality-Aware and Fine-Grained Incentive Mechanisms for Mobile Crowdsensing," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 354–363.

[105] Z. Lv, N. Wang, J. Wu, and M. Qiu, "IoTDeM: An IoT Big Data-oriented MapReduce Performance Prediction Extended Model in Multiple Edge Clouds," *J. Parallel Distrib. Comput.*, pp. 316–327, 2018.

[106] T. Abdelzaher, N. Ayanian, T. Basar, S. Diggavi, J. Diesner, D. Ganesan, R. Govindan, S. Jha, T. Lepoint, B. Marlin, K. Nahrstedt, D. Nicol, R. Rajkumar, S. Russell, S. Seshia, F. Sha, P. Shenoy, M. Srivastava, G. Sukhatme, A. Swami, P. Tabuada, D. Towsley, N. Vaidya, and V. Veeravalli, "Will Distributed Computing Revolutionize Peace? The Emergence of Battlefield IoT," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1129–1138.

[107] Q. Li, L. Zhao, J. Gao, H. Liang, L. Zhao, and X. Tang, "SMDP-Based Coordinated Virtual Machine Allocations in Cloud-Fog Computing Systems," *IEEE Internet of Things Journal*, pp. 1977–1988, 2018.

[108] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, "DROPLET: Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 1–8.

[109] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 649–660.

[110] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-memory Capabilities," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 31–44.

[111] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic Computing in GPU Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

[112] G. Lee, H. Park, S. Heo, K. Chang, H. Lee, and H. Kim, "Architecture-aware Automatic Computation Offload for Native Applications," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.

[113] J. Liu, B. Priyantha, T. Hart, H. S. Ramos, A. A. F. Loureiro, and Q. Wang, "Energy Efficient GPS Sensing with Cloud Offloading," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys, 2012.