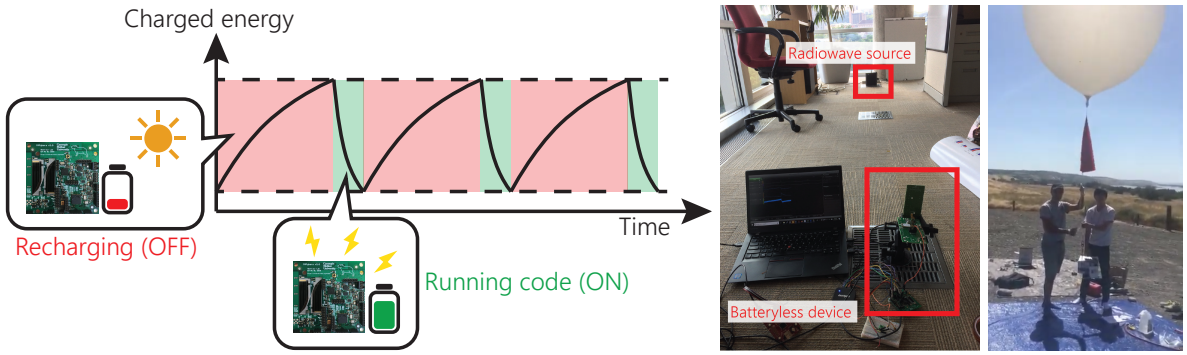
Research Statement

Kiwan Maeng

In the new *Edge Computing* era, sensor devices at the edge perform complex processing of local data instead of merely collecting data and sending it to a centralized server. Edge computing brings many potential benefits over the traditional centralized-server design. Imagine a smart-home device, e.g., Amazon Alexa or Google Home, that can locally process your verbal command without having to send your voice recording to a server. The device would respond faster even on a slow network and would be more secure. Recent advancements in low-power hardware technologies made edge devices computationally powerful enough to realize edge computing. However, an exploding number of devices with each having its unique constraints (e.g., computation, memory, energy, or network) made programming for edge devices ever more complex.

My research broadly focuses on ***optimizing code for low-power edge devices with unique constraints***. My current research concentrates on **batteryless, energy-harvesting devices**, an emerging family of edge devices. Batteryless devices run without a battery or tethered power, exposing unique energy and reliability constraints. Looking forward, I will expand my work to improve the programmability of edge devices with **heterogeneous computation/memory hardware**.

I conduct my research based on three major principles. First, I identify a research problem by *looking at a real, end-to-end deployment scenario*. I often deployed systems I built to gain insights and test out ideas. For example, I have deployed a batteryless kitchen monitoring system (Figure 1b) and multiple next-generation weather balloons that automatically deploy small sensors into the atmosphere (Figure 1c). Second, I study *components across the system stack* to solve problems at the most appropriate level, including the compiler [1–5], hardware [2–4, 6], software runtime library [1–8], and programming model [1–5, 8]. Third, I *leverage statistical data to optimize the system*, both statically and dynamically. For example, I have designed systems using profiled data [2, 3] in compile time to ensure system correctness.



(a) (b) (c)

Figure 1: (a) Execution of a batteryless device experiences frequent failures, where an inactive period of recharging energy (red) is required to run a short computation (green). (b) A batteryless, ML-based kitchen monitoring system built with Samoyed [3] that monitors whether tap water or garbage dispenser is on. (c) A next-generation weather balloon I designed, which drops tiny sensors in the air to collect additional data [6].

Current Research

Batteryless, energy-harvesting devices are a new type of embedded devices that operate without batteries. Instead, these devices use energy scavenged from their surrounding environment, e.g., solar, radio, motion, or thermal gradient. Batteryless devices can be deployed at a large scale to collect and process data without worrying about replacing their batteries periodically. These devices, however, have not been adopted to real-world applications due to their prohibitive programming complexity. Because they usually consume more energy than what they can harvest, program execution is interspersed with a recharging period where the device is powered off (Figure 1a). Such frequent power failures make writing a correct and efficient program prohibitively challenging.

Primarily, my work concentrated on two critical challenges of batteryless devices. First, **running programs correctly and efficiently** was challenging with frequent power failures because volatile states (e.g., data stored in SRAM) of the device are lost on each power failure. Second, **supporting common programming constructs** (e.g., timer-based periodic code execution) was impossible due to frequent power failures. My work [1–5, 8] addressed these challenges, improving programmability, and expanding the usability of batteryless devices.

Supporting Correct and Efficient Computation For Batteryless Devices

Writing even the most basic programs correctly on batteryless devices requires significant programming efforts from an experienced programmer. Batteryless devices lose their volatile states (e.g., register file or SRAM) on each power failure, while their non-volatile states (e.g., Flash, FRAM, or MRAM) persist. Unless appropriately saved (i.e., *checkpointed*) and restored, the loss of volatile states can lead to many correctness issues.

If power failures erase updates to volatile states while not erasing updates to non-volatile states, the system may end up in an *inconsistent memory state*, where volatile and non-volatile states hold updates from different points in time. To avoid such a memory inconsistency, I built Alpaca [1], a new programming model and a runtime system for batteryless devices. Alpaca's central insight is that checkpointing the non-volatile variables with write-after-read (WAR) dependence in addition to volatile states is enough to always restart from a valid memory state [1]. Alpaca's programming model requires the programmer to write a program as a series of code blocks called tasks, which is the granularity of re-execution on a power failure. Alpaca's compiler statically generates checkpointing code to save the volatile progress and non-volatile variables with WAR dependence at each task boundary to keep memory consistent while incurring minimal overhead. Compared to prior work, Alpaca improves the overall performance by 1.2-16.6× while always keeping the memory correct.

If power failures repeatedly erase the volatile states before they get checkpointed, the system may *not make any useful progress*. Such too frequent power failures can happen if there is not much energy for the device to harvest. To ensure progress even on the harshest energy condition, I designed Chinchilla [2], a compiler and a runtime system that converts uninstrumented C code into a program that always makes forward progress. Chinchilla breaks down a program into a sequence of code blocks that is small enough to always run with even the harshest energy condition. Chinchilla's compiler synthesizes a testing harness that repeatedly runs parts of the program on real hardware and measures the used energy. Using the collected statistics from each part of the program, Chinchilla inserts checkpoints to break down the code into chunks that can always run even with minimal incoming energy. Chinchilla dynamically disables unnecessary checkpoints when energy is more abundant. Similarly, I also designed Coala [8], a runtime system that dynamically increases or decreases the checkpointing interval to always progress with additional runtime overhead. Chinchilla and Coala ensure that the program always has enough checkpoints to make forward progress.

Alpaca, Chinchilla, and Coala all make writing a correct program on batteryless devices easy. They serve as a fundamental building block for writing applications on batteryless devices.

Enabling Common Programming Constructs For Batteryless Devices

Enabling correct computation is not sufficient for many real-world applications. Some applications need to run periodically, while others need to run a specific code block without getting interrupted. Still others use private data and need extra security measures to protect such data. Frequent power failures and constrained resources of batteryless devices disallow implementing such common programming patterns, limiting use-cases of batteryless devices.

Some code regions *must be executed at once without interruption, or atomically.* Atomicity cannot easily be guaranteed on batteryless devices because a power failure can happen during a code region that must be executed atomically. For example, consider communication protocols commonly used for controlling sensors (e.g., SPI, I^2C, or UART). The protocols must be executed atomically; a power failure in the middle of the protocol will render the control packets meaningless by making the protocol's timing constraints violated. To enable atomic code execution, I designed Samoyed [3], a system consisting of a programming model, compiler, hardware energy profiler, and a runtime system. Samoyed leverages the fact that a lot of atomic code execution can be broken down into a sequence of smaller atomic code execution. For example, a packet transmission can be decomposed into a series of smaller packet transmissions. From user-annotated code, Samoyed's compiler synthesizes an atomic region that can be re-executed multiple times without side-effects. When synthesizing the atomic region, Samoyed's compiler also exposes a knob that can dynamically control the region's size. With its static energy profiler ensuring that at least the smallest atomic region can always run even with the harshest energy condition, the atomic region's optimal size is selected dynamically through the knob. Samoyed is the first system to realize atomic execution on batteryless devices, allowing many external sensors, actuators, and accelerators that require atomic code execution.

Some code *must be executed periodically*, whose requirements again cannot be met on a batteryless device. If power fails when the device needs to sample data in a periodic data sampling application, the device simply cannot meet the periodicity requirement. To support periodic execution, I implemented CatNap [4], a programming model and a runtime system with a scheduler for scheduling energy recharging. CatNap achieves periodic execution by not only scheduling code execution, but also the necessary recharging for each code execution. CatNap's programming language allows the programmer to specify priorities of events and tasks. CatNap generates a provably correct schedule to always meet the periodic requirements of high-priority events while occasionally sacrificing low-priority tasks. CatNap also trades off the application quality to use

less energy when there is no valid schedule. CatNap enables reliable periodic execution on batteryless devices for the first time, allowing new applications requiring periodic sensing or actuation to be hosted on batteryless devices.

Some applications use user-private data, which *must be kept secure*. Batteryless devices are often deployed in regions publicly accessible, and data stored in off-chip memory can be revealed if an attacker physically probes the data bus or read the non-volatile memory chip directly. Batteryless devices lack hardware support for trusted execution environments (TEE), e.g., Intel SGX, providing no simple way to guard data against such physical attacks. To realize low-overhead security, I built Spitz [5], a compiler and a runtime system that provides security through software-based memory encryption. To prevent attackers from probing the memory bus or reading non-volatile memory chip directly, Spitz uses a compiler and an on-chip scratchpad to encrypt memory without TEE. Spitz always performs computation inside the device's safe on-chip scratchpad and encrypts any data when storing them to an unsafe off-chip memory. Spitz's compiler statically schedules the encryption/decryption and data movement between on- and off-chip memory to minimize overhead. Our ongoing work shows that Spitz's overhead is low, especially when the memory access pattern is regular, as in many emerging ML workloads. Spitz realizes practical memory encryption on batteryless devices for the first time.

Samoyed, CatNap, and Spitz allow the programmer to use programming constructs crucial for real-world applications for the first time. With atomicity, periodicity, and memory security, batteryless devices can host a broader range of applications that were not supported before.

Future Research

Looking forward, I plan to expand my work to tackle programming challenges for other low-power edge devices with unique constraints. There exists an unprecedented diversity of edge devices, each having its own constraints. Batteryless devices have energy constraints, implantable medical devices have size and thermal constraints, chip-scale satellite devices have reliability constraints, and smart-home appliances have security constraints. To make matters worse, devices from different vendors are highly dissimilar. Different devices have different computation hardware and memory structure, providing a spectrum of choices to the programmer governing latency, bandwidth, capacity, security, and non-volatility. Programming to meet the high-level and low-level constraints of each edge device requires significant programming effort.

In the near term, I plan to continue research on batteryless devices. Especially, I will propose a new paradigm of **deployment-aware system design**, a new design paradigm for batteryless devices that considers the deployment environment's energy condition to predict and optimize the system behavior. Looking further, I plan to build a unified programming framework to optimize code for **low-power devices with heterogeneous hardware**, which would significantly reduce the programming effort to write a program tailored to the uniquely-heterogeneous hardware of each device.

Throughout my career, I have been actively participating in an inter-university research collaboration as part of a large-scale funding center, CONIX. I plan to apply similar opportunities for my successful future research, including NSF sources (e.g., NSF CNS CSR, NSF CCF SHF, and NSF CPS) and industry fundings (e.g., SRC, Intel, Microsoft, Google, and ARM).

Near Term: Deployment-aware System Design for Batteryless Devices

The capability of a batteryless device depends on the environment in which it is deployed. When the incoming energy is abundant, the device charges faster and can perform more computation. However, the environmental factor is usually not considered when designing batteryless systems, making systems less efficient and capable.

I plan to propose *environmental profile-guided system design* as a new paradigm in designing batteryless devices. In the current state-of-the-art design paradigm for batteryless devices, the programmer designs the hardware and writes code without a clear understanding of the deployment environment, and passively accepts the resulting performance after deployment. Alternatively, the new paradigm will allow the programmer to design different parts of the system to target application-level performance goals using the profiled energy information from the deployment environment.

The information on the environment can also be used to *plan code execution and recharging*. CatNap [4] took one step towards energy scheduling but only considered the near future because it was relying on the fact that the incoming energy would not change dramatically in a short period. Information about how the incoming energy will change over a larger time scale can allow the system to make better long-term scheduling decisions. For example, a solar-powered system can regulate the rate of code execution during the day to reserve enough energy for the night if the duration and the available energy for the day and night can be precisely predicted.

The deployment-aware design paradigm will fundamentally shift the field of batteryless devices. The new paradigm will eliminate the current batteryless devices' high performance uncertainty by tailoring each system to its deployment environment. Moreover, what was previously impossible will become possible, such as a solar-based system running code at night with the energy provisioned during the day.

Long Term: Unified Programming Framework for Heterogeneous Device Hardware

Low-power devices from different vendors have severely dissimilar computation and memory hardware with each other, making programming complicated. Different devices have different computation hardware, ranging from simple vector processing units to custom architectural accelerators or coprocessors. They provide a programming interface incompatible with each other, work with different input data structure, and offer a spectrum of performance and power characteristics. Also, devices have their unique memory structure, often including on- and off-chip SRAM, DRAM, Flash, MRAM, or FRAM, offering different latency, bandwidth, capacity, non-volatility, and reliability. Leveraging the heterogeneous resources to write an optimized program requires expertise specific to a device, which poorly transfers to another device.

I plan to build a *unified programming model for low-power devices with heterogeneous memory and computation hardware*. Analogous to how a traditional compiler generates optimized binary for each microarchitecture from a hardware-agnostic C code, the idea of a unified programming model is to automatically generate optimized binary leveraging the heterogeneous compute/memory resources of each device from a unified model. Due to the ever-growing diversity of low-power devices, such a unified programming model is essential for programmability and portability. A field that can immediately benefit from such a unified programming model is machine learning (ML). Many low-power processor families have their own ML libraries and hardware with different APIs, different capabilities, and requires different data structures as inputs. If an optimized binary using each library/hardware can be generated automatically, writing an ML program or porting a program to different devices will be much easier. I have experience building efficient fault-tolerant systems for ML training [7], which will help me target ML workloads as a first step.

To cope with the limited compute and memory constraints of low-power edge devices, the unified framework must concentrate on supporting *compressed data format*, including quantized data, variable-width data, and sparse tensors. Although these compressed data formats have been extensively studied in academia for memory-constrained devices, they are rarely used in the real world because of the lack of support from existing hardware and software libraries. I plan to co-design the compression techniques and the hardware/software to make using compressed data formats practical for devices with different compute/memory constraints. Making a compressed data format-friendly system environment for low-power devices will allow the devices to host more extensive programs, e.g., a larger ML model.

As low-power edge devices' computation capability increases, edge devices can process local data on their own, opening up performance, scalability, and security benefits. However, many edge devices are uniquely constrained, making programming complicated without proper system support. I have tackled several programming challenges of batteryless edge devices, moving them one step towards real deployment. Making programming simple on uniquely constrained edge devices will expedite the exploration of different application use-cases and help these devices make a real-world impact.

References

[1] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. In *OOPSLA*, 2017.

[2] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *OSDI*, 2018.

[3] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *PLDI*, 2019.

[4] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *PLDI*, 2020.

[5] Kiwan Maeng and Brandon Lucia. Work in progress.

[6] Kiwan Maeng, Iskender Kushan, Brandon Lucia, and Ashish Kapoor. Enhancing stratospheric weather analyses and forecasts by deploying sensors from a weather balloon. In *NeurIPS Workshop*, 2019.

[7] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. Cpr: Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Under review.*

[8] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *TOSN*, 2020.