Research Statement | Milijana Surbatovich, Fall 2022

My research vision is to make "extreme edge" computing devices ***correct, secure, and trustworthy***. To accomplish this, I create formal models, analysis tools, and runtime systems that ***span different abstraction layers from systems to programming languages***. My strategy has been to first use ***rigorous, precise reasoning*** to define aspects of correct execution on these tiny, embedded devices and then make my formal correctness criteria practical by designing both ***enforcement mechanisms,*** e.g., runtime systems and compiler transformation tools, and ***language abstractions***, e.g., type systems and programmer annotations, that together give a programmer assurance their application will execute as intended.

Extreme edge computing is an exciting and important research domain because it has great potential for positive societal impact, but incorrect and insecure computation will cause real harm. The extreme edge computing model moves sophisticated sensing and data processing into the deployment environment, rather than relying on a centralized server. This model keeps sensitive data local and lessens cloud communication, promising increased security and enabling remote and large-scale deployments, to the benefit of application domains like medical wearables, remote wildlife monitoring, tiny satellites, or sensor-augmented ("smart") infrastructure. Batteryless *energy harvesting devices* (EHDs) are key to realizing this promise since they have long lifetimes with few maintenance requirements, making these remote and large-scale sensor deployments practical and sustainable. EHDs collect all their energy from the environment and thus do not require much intervention (e.g., to replace a battery) or produce battery waste. However, harvested energy is typically too weak to power devices continuously, causing frequent, arbitrary power failures that introduce new challenges in programmability and correctness. Given the envisioned application domains, an EHD must execute software without bugs that render the device inoperable or that leak sensitive information. Otherwise, a medical application might fail when a patient is depending on it, or an attack on agricultural infrastructure could ruin a crop harvest. While researchers have developed *intermittent systems* to support programming these devices, they rely on ad-hoc correctness notions that preclude proving the correctness and security properties necessary to trust that an EHD will not cause harm.

My research lays the groundwork for designing ***formally correct intermittent systems*** that meet specified correctness guarantees, ***improving reliability and programmability***. In my PhD work, I have developed languages and systems for guaranteeing memory consistent and timely sequential intermittent execution. In the future, I will expand on this foundation to provide formalisms, systems, and language abstractions for ***secure*** and ***concurrent*** intermittent execution. In the longer term, I will leverage my skills in applied formal methods and cross-abstraction system building to develop ***models and tools for hardware/software co-design*** on extreme edge devices.

Current Research

An EHD can turn on to operate only when sufficient energy is available in its environment. Once it exhausts its harvested energy, the device turns off at some arbitrary point until it can recharge, clearing volatile execution state like the program counter. To make progress, the intermittent system checkpoints state before power failure and restores it after the device reboots, to resume execution from the saved point. Determining what state to save and when is a complex question, and an incorrect decision introduces *memory consistency* and *timing bugs* that would not occur on continuously powered executions. In my work, I have shown that existing ad-hoc correctness notions of intermittent execution are insufficient, leading to unaddressed bugs [1]. I then developed the first formal model of intermittent execution and used it to define correctness conditions for memory consistency [2,3] and timing properties [4] on intermittent systems. Once I had formal correctness definitions, I designed both *enforcement mechanisms* and *abstractions for programmers to specify their desired properties*, in practical implementations for modern programming languages like Rust. My work thus provides a necessary foundation for building intermittent systems that can meet correctness guarantees.

Defining and Enforcing Memory Consistency Properties

To make progress, many intermittent systems provide "atomic region" constructs. The system saves state at the start of the region; if power fails before the region completes, the system will restart execution from the beginning of the region, potentially causing re-execution. Checkpointing insufficient state causes these re-executions to exhibit

memory consistency bugs that ultimately compute wrong results, e.g., by reading results of a failed execution. Since checkpointing all state wastes precious energy and memory, past systems attempt to identify what state is necessary to save, but without defining the properties they aim to provide.

My research defines correct intermittent execution w.r.t. memory consistency, identifying specific memory state correctness criteria an intermittent system must guarantee. First, I showed that ad-hoc correctness reasoning is insufficient; prior work did not correctly handle input operations across power failure. The crux of the issue is that EHDs often run sensor-driven applications, where the sensor inputs are non-deterministic and can cause re-executions to see different values each time. I characterized a class of bugs caused by this non-determinism and developed a suite of *static and dynamic bug detections tools*, called Ibis [1], giving the programmer a way to reason about subtle input dependencies. The key contribution of Ibis is that it identified an entire class of memory bugs missed by most prior systems, confirming the need for more rigorous correctness reasoning about intermittent systems. To enable such reasoning, I then created the first *formal model of intermittent execution*, which included reasoning about non-deterministic inputs, and *defined a correctness theorem*—an intermittent execution is correct if and only if it corresponds to (i.e., refines) some continuous execution [2]. Proving whether an intermittent execution satisfies this theorem was challenging because even a correct intermittent execution frequently has differences in memory state and input values due to arbitrary re-execution. A key piece of this project was defining invariants on the memory state describing which of these differences were acceptable and which would cause the system to introduce bugs. These invariants allowed us to concretely identify what memory state needs to be recovered by the system and then implement a runtime system that must uphold the correctness theorem. The key contribution of this work is that it provides an intuitive yet formal correctness theorem for intermittent execution for the first time and defines the memory invariants a system must satisfy to uphold it, building a formal foundation for future intermittent system designs.

Leveraging Types to Give Programmers Control

A typical state-of-the-art intermittent system identifies variables to be checkpointed via compiler passes that require little to no programmer involvement. My previous work formulated correctness criteria as low-level memory invariants, which integrated well with these existing systems. This design choice gave programmers a correct system for little to no extra effort, for most applications. I observed, however, that some programs *need* to observe the effects of failed executions to work as intended, e.g., systems code that counts reboots, or a sensor processing application that should react to any seen input. Thus, I re-cast the correctness theorem in terms of *non-interference*—data from partial, failed executions is restricted and should not affect unrestricted portions of the result. To give the programmer control, I leveraged this insight to design a type system, Curricle, for specifying and checking properties of intermittent execution and implemented it on top of the Rust compiler [3]. Curricle uses *information flow reasoning* and *type qualifier inference* based on the access patterns of a variable to rule out programs that violate non-interference. This novel, type-level method of reasoning about intermittence gives the programmer the ability to specify relaxed correctness conditions while having assurance that the program will execute as specified intermittently if it type-checks. Leveraging Rust's linear type system allows more precise analysis than in past works, which target C, simplifying the recovery system implementation, as well as allowing future intermittent systems programmers to use memory safe Rust instead of C.

Correctness Properties Beyond Memory Consistency

Formalizing correct intermittent execution and developing tools and abstractions to express and ensure basic memory correctness creates a stable foundation for reasoning about other aspects of intermittent execution that are necessary for full assurance that an application executes as desired. For instance, arbitrary time is lost during power failure while the device recharges, which could make sensor readings stale or inconsistent. I created formal definitions for timing properties via correspondence to the timing of continuous executions. I then made my formal definitions practical by creating the compiler toolchain Ocelot [4]. While past systems addressing timing issues required additional hardware timers and concrete deadlines and runtime mitigations from the programmer, Ocelot takes in simple annotations from the programmer indicating which data have time constraints and *generates correct-by construction programs*, leveraging the existing capability of atomic re-execution to enforce timing constraints, as the final execution of an atomic region must correspond to the timing of continuously powered execution.

Future Directions

The overall mission of my research is to provide the formal models, enforcement mechanisms, and language abstractions necessary for designing emerging computing platforms to meet well-defined correctness guarantees from the ground up, allowing programmers to unlock the full potential of their system without compromising trustworthiness. I will first expand on the foundation I have already created to ***formalize correctness properties of multi-tenant or multi-device intermittent systems***, a challenging but interesting area as concurrency/parallelism complicates memory consistency and requires formalizing additional correctness properties, e.g., resource consumption and ***security***. In the longer term, I will leverage my skills in developing formal models side-by-side with languages and low-level system implementations to reason about ***full stack correctness*** for extreme edge devices, supporting the hw/sw co-design necessary to create systems that are both efficient and formally correct.

Formally Correct Multi-tenant Intermittent Systems

Real-world deployments of EHDs should support multi-tenant systems, e.g., an agricultural sensor running both moisture detection and an image processing application to identify livestock, potentially as part of a distributed swarm of sensors. Unlike a single application, which can run "bare-metal" with minimal system interference, multi-tenancy needs more robust support to guarantee fair scheduling and isolation between applications. Designing a formally correct multi-tenant intermittent system requires definitions, enforcement mechanisms, and abstractions for (1) *concurrent intermittent execution*, (2) *principled resource consumption*, and (3) *security policies and threat modelling*.

Modelling concurrent intermittent execution. No existing intermittent system has a principled approach to concurrency, using either vague semantics or unnecessarily conservative restrictions in communication between concurrent routines, since they target C. Moreover, no system supports parallelism. I plan to extend my formal model with concurrency constructs and leverage its Rust-based implementation to precisely reason about less restrictive communication, with some initial work already completed.

Principled resource consumption. A multi-tenant intermittent system will schedule applications based on a notion of fair use of energy and potentially other device resources. A task that uses resources poorly, e.g., requires more energy than can fit in one buffer charge, may break scheduling guarantees. I am interested in exploring the power of type systems to guide programmers in writing programs with principled resource consumption for EHDs. For example, in Culpeo [5], my co-authors and I showed that operations with high current load, especially near the end of a task, can cause drops in voltage that shut down the device even when energy remains, due to circuit-level effects of the power system. Capturing this insight in a language abstraction would help programmers write applications with more disciplined voltage usage, e.g., high current operations only at the start of a task, without directly reasoning about power system idiosyncrasies. A type-checked program has easier resource constraints to satisfy, making the program simpler to schedule and more efficient.

Security policies and threat modelling. Compared to a traditional embedded device, an attacker has more power as they can influence when the device turns off, triggering when memory state clears and when delicate system restoration code runs. I want to explore whether these increased capabilities can be exploited to infer application behaviour and parameters, or whether they can leak sensitive application information to an outside attacker or a malicious multi-tenant application. Apart from attacks, an intermittently powered device may simply not behave as the end user expected, causing unintended harm. While I have some prior experience in studying unintended privacy violations in smart home devices [6, 7], I am eager to collaborate with Security/Privacy and HCI researchers to understand how users interact with their device and mitigate harms caused by actual EHD deployments.

Towards a Framework for Full-Stack Correctness Reasoning

Extreme edge devices must be low power and energy efficient, requiring programmers to blur the hardware/software interface to use their limited resources effectively. Unfortunately, blurring the interface also makes reasoning about correctness more difficult, as it blurs what properties each stack layer must provide. A longer-term goal of my research agenda is to develop formalisms and tool support for reasoning about the computing stack of an edge device end-to-end, to support hardware/software co-design. I am well-qualified to pursue this challenge because my research strategy of applying formal method techniques to develop languages

and systems demonstrates how formal models expose (a) ***correctness criteria*** that the hardware and software must satisfy together and (b) ***design insights*** that lead to simpler implementations.

Effective co-design on an edge device requires determining what aspects of the hardware and deployment environment should be exposed to the programmer/compiler (hw → sw), and what information the programmer should give to the compiler (or lower) to maximize resource usage (sw → hw). As a step in each direction, I am eager to explore (1) the effect of the deployment environment on correct EHD scheduling and (2) language abstractions for low power architectures.

The deployment environment impacts correct execution of applications because meeting timing constraints depends on the frequency and duration of power failures, which in turn depends on the energy availability of the environment. For example, a workload on a satellite may have enough energy to meet its timing goals when in sunlight, but not when the satellite is in eclipse. Thus, a framework for determining scheduling guarantees of multi-tenant systems must be parameterized by characteristics of the deployment environment as well as the device, e.g., the energy buffer size. I will identify what real-world aspects of the deployment environment must be parameterized and develop a formal schedulability analysis for deployment-aware "soft real-time" guarantees. This result will allow programmers to check under what environmental conditions and on which devices an application will meet its timing goals, guiding the programmer to adjust their application or choose a different device, if the constraints cannot be met.

Ultra-low-power devices should ideally use energy efficient heterogeneous or reconfigurable architectures, such as ASICs or FPGAs, which may use custom compilers and memory models. Co-designing programming languages can make these fiddly devices easier to program and can simplify hardware design. For example, language abstractions can guide the compiler to manage a scratchpad and main memory, reducing programmer burden and the complexity of compiler analyses, as long as the compiler is trusted to maintain consistency. Similarly, a DSL that simplifies control flow or aliasing can make a program easier to map onto a reconfigurable architecture, improving efficiency. In turn, simplifying the design of OS and ISA layers makes the more distant goal of formal verification of edge devices more feasible by simplifying the most complex layers of the computing stack.

The results of this research agenda will make real-world deployments of extreme edge devices more trustworthy, unlocking the benefits of ubiquitous smart sensing without causing unintended harm. I am excited to enable provably correct and secure computation in an emerging domain.

[1] **Milijana Surbatovich**, Limin Jia, Brandon Lucia. *I/O Dependent Idempotence Bugs in Intermittent Systems*. In OOPSLA 2019.

[2] **Milijana Surbatovich**, Limin Jia, and Brandon Lucia. *Towards a Formal Foundation of Intermittent Computing.* In OOPSLA 2020.

[3] **Milijana Surbatovich**, Naomi Spargo, Limin Jia, and Brandon Lucia. *A Type System for Safe Intermittent Computing*. In submission, 2023.

[4] **Milijana Surbatovich**, Limin Jia, and Brandon Lucia. *Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems.* In PLDI 2021.

[5] Emily Ruppel, **Milijana Surbatovich**, Harsh Desai, Kiwan Maeng, and Brandon Lucia. *An Architectural Charge Management Interface for Energy-Harvesting Systems.* In MICRO 2022.

[6] Camille Cobb, **Milijana Surbatovich**, Anna Kawakami, Mahmood Sharif, Lujo Bauer, Anupam Das, and Limin Jia. 2020. *How Risky Are Real Users' IFTTT Applets?* In SOUPS 2020.

[7] **Milijana Surbatovich**, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. *Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes.* In WWW 2017. `