Is This General Purpose Embedded Operating System Really General Purpose? Extending the Tock Operating System for RISC-V and Intermittent Computing

---

A

Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Master of Science

by

Samyukta Venkat

August 2020

APPROVAL SHEET

This

Thesis

is submitted in partial fulfillment of the requirements
for the degree of

Master of Science

Author: Samyukta Venkat

This Thesis has been read and approved by the examing committee:

Advisor: Bradford Campbell

Advisor:

Committee Member: Benton Calhoun

Committee Member: Joanne Bechta Dugan

Committee Member:

Committee Member:

Committee Member:

Committee Member:

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

August 2020

ACKNOWLEDGMENT

First and foremost, I would like to thank my advisor Brad Campbell. Professor Campbell has always provided sound guidance and has patiently answered my countless questions. He has shown me that it's okay to not know everything as long as you're willing to learn and I've grown so much in my time working with him. Next, I would like to acknowledge the other members of my committee - Joanne Bechta Dugan and Benton Calhoun - who have provided me with valuable feedback on my work. I would also like to thank Professor Dugan for pushing me to do this work in times when I wasn't sure that I could. I would like to recognize Branden Ghena who has also fielded many of my questions and has been an invaluable resource in times of great need.

I would like to acknowledge my labmates Nurani Saoda, Nabeel Nasir, and Alexander Sarris who have all supported me throughout these projects. They've discussed my work with me and helped me solve technical challenges. I'm grateful to have worked with them and the rest of the wonderful people in my lab. Outside of the lab, I am incredibly lucky to have found close friendships with Nabeel and Ishika Paul. Our long and often late night conversations have undoubtedly kept my spirits high throughout this process (even when we felt like we goofed off for too long in the moment!). I would like to thank my family who has shown their support throughout grad school but particularly in the last few months while I've been quarantining with them. Lastly, but certainly not least, I would like to thank Scott Kirkpatrick for his emotional support throughout this process.

IS THIS GENERAL PURPOSE EMBEDDED OPERATING SYSTEM REALLY GENERAL PURPOSE? EXTENDING THE TOCK OPERATING SYSTEM FOR RISC-V AND INTERMITTENT COMPUTING

Abstract

by Samyukta Venkat, M.S.
University of Virginia
August 2020

:

Embedded computing is moving in new directions with cyber-physical systems and IoT applications. Devices are becoming more sophisticated while also being subject to large design changes to lower their resource consumption. In this work, we are exploring two trends in hardware as they relate to an operating system — the layer that is supposed to help utilize but mask the hardware changes. We extend Tock, a general-purpose embedded operating system, for a new processor architecture and a new energy source to understand how the design of a resource constrained OS impacts its generality. In this work, we carefully redesign the kernel-userspace interface to make it architecture independent, and evaluate our design by adding RISC-V support. Further, we design a new scheduler for Tock which allows the system to run on harvested, intermittent energy and make progress despite power failures. We identify and implement several design extensions for the kernel that are required to support this operation, and measure the time, memory, and developer overhead. The information gained from these two extensions of Tock exposes bottlenecks and identifies design choices that keep Tock from being portable and general-purpose across various use cases.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

To anyone in my life that nonchalantly assumed I could write a thesis.

Chapter One

Introduction

1.1 Direction of Embedded Computing

Embedded computing is moving in new and exciting directions. In recent years the usage of embedded devices has exploded due to increased desire for various cyber-physical systems and IoT applications [1]. Where embedded computers used to mainly exist in systems such as vehicles, appliances, and medical/industrial equipment, now they are being deployed in huge numbers in buildings, out in the environment, and being worn on bodies. We are in the midst of the next computing revolution and this thesis explores different areas of increased interest for realising this ubiquitous computing vision.

There are two sides to this progression of embedded computing. On the one hand, IoT applications and devices are becoming much more sophisticated. A desire for responsive or real-time operation often means moving computation to edge devices. With the larger variety of applications being introduced, there is an emphasis being put on security, reliability, and robustness of these systems [2]. There is also a focus on usability and programmability given the vision to develop and deploy billions or trillions of devices [3]. Largely, the sophistication expected from these devices happens in software which means that good software support is critical. There are two key aspects to good software support. The underlying system should be robust so that buggy or malicious code cannot break the whole system. Secondly, software support should enable scalability — it's hard to write monolithic C programs every time a new hardware feature comes out. Therefore, software

support must keep up with changing needs to enable complex applications and programmability across many use cases.

On the other hand, to reach maintainability and sustainability goals, devices are changing. They are becoming smaller, switching out batteries in favor of energy harvesting circuitry, and they are being built around new hardware architectures. Whereas wireless devices traditionally use batteries with a lifetime of several months to a few years, and require periodic maintenance to replace batteries, emerging devices can scavenge their own energy. Further, to increase performance, new chips include specialized accelerators to optimize complex functions such as neural network classification or cryptographic operations. These trends increase the hardware diversity and complexity that software developers must consider.

We must figure out how to navigate these conflicting pulls from wanting more out of the devices while needing them to have a smaller footprint. We explore the operating system layer because it is supposed to assist in taking advantage of hardware changes while also masking the extent of the complexity from the developer. Currently there are two trends in hardware development that we are interested in exploring — the RISC-V ISA and intermittent computing. We will look at these areas as they relate to a general-purpose embedded operating system to see how well the operating system can support them.

The operating system under consideration is Tock. Tock is a secure embedded operating system that provides multiprogrammability and process isolation with a kernel written in Rust [4]. Tock offers robust fault isolation which prevents buggy code from corrupting the whole system. Additionally, Tock provides the ability to load and update processes at runtime with a small footprint (just the process binary). Tock has a fairly large and active community surrounding it. These features make it a good candidate for exploration. However, prior to the work in this thesis, Tock exclusively had support for ARM Cortex-M cores and could only run on continuously powered systems. The goal of this work was to document and assess the process of extending Tock, first, in the face of a new architecture and second, without a steady power supply. We can use this information to assess how general-purpose Tock's design really is and its adaptability to these device changes.

1.2 RISC-V

The first trend in embedded computing we would like to extend Tock's support for is RISC-V, an open-source, royalty-free standard for an instruction set architecture (ISA). RISC-V offers a lot of flexibility in implementing the ISA and a low barrier to entry through being royalty-free which makes it a good candidate for innovation in processor design. The flexibility aspect makes it scalable from resource-constrained computing to much larger scale computers. There is also a requirement for RISC-V cores to be compliant with the RISC-V base ISA. Compliance is an important aspect for when RISC-V chips start to proliferate because it makes portability of software easier. Microcontrollers today have many different architectures and designs which makes it difficult to run a software system across a large number of devices. Many companies are interested in using RISC-V-based chips which shows that this architecture could have wide deployment in the future [5]. If that is the case, supporting RISC-V with Tock is a necessary and important step. The contribution of the RISC-V section of this work was getting basic Tock support for RISC-V which was the starting point for a significant period of development in Tock that came after.

1.3 Energy Harvesting & Intermittent Computing

Another trend for ubiquitous computing is energy harvesting and batteryless devices. A wide variety of applications in health, transportation, energy, and other sectors could benefit from a consistent stream of data from various, often hard-to-reach places [6, 7]. Not only does this style of computing increase convenience and give us a wealth of knowledge about many spaces, but it will also be a key player in creating a sustainable future. However, from the need to have so many of these devices arises the need for them to be batteryless and capable of harvesting energy from their surroundings [8]. Developing software for certain energy harvesting devices can be challenging given the nature of the power source. Devices can have frequent power outages which makes it difficult for useful work to get done and programming around the outages is not an easy task [9]. Given the number of devices needed for this vision of ubiquitous computing, enabling developers to write useful software

is important. This work will focus on making progress specifically for transiently or intermittently powered devices.

Energy harvesting sources often do not provide steady power like batteries or wall-power can. Energy sources for intermittent devices can range from solar cells to piezoelectric harvesting (motion). These sources are often variable - they can provide times of high energy (e.g. in direct sunlight) or they can provide little to no energy (e.g. indoor/dim lighting/dark) [7]. This variability can cause devices to drop below their operating thresholds several times a second [10]. The uncertainty of energy harvesting sources has created the need for software systems that take the onus off of the application developer to figure out how to make useful progress.

There are two basic models for making forward progress in face of frequent volatile memory resets. The first is checkpointing which generally inserts trigger points in a program at compile time when the system will save the contents of volatile memory to nonvolatile memory [11, 12]. After a power failure, execution will resume using the execution context saved by the last checkpoint. Trigger points can be used to check energy levels to decide whether to save state or not. The second method to make progress is a task-based approach where applications are broken down into smaller energy-atomic tasks. Tasks are then encoded into a task-graph to declare the flow of tasks. Device restarts begin execution at task boundaries (i.e. if an outage occurs during task A, the device will resume from the beginning of task A). A task-based approach requires less state to be stored when compared with checkpointing as the device only needs to know where in the task-graph it is and what data was being passed between tasks [10]. Prior work in this area uses one of these two models as the base for a runtime environment.

Both checkpointing and task-based approaches have certain advantages and disadvantages. Saving state in a task-based method often has less overhead than using checkpointing because less information is required to know which task to begin with. However, there is higher developer overhead from needing to break programs into tasks that can complete in a power cycle [13]. Task-based systems usually come with constructs that need to be learned by the developer. Checkpointing can also require frequent measurements of how much energy is left which adds additional overhead. Both approaches are being developed in the literature without one being a clearly superior approach.

Generally, existing systems are special purpose and are highly tailored to run on a narrow selection of hardware (i.e. the WISP RFID platform) [14]. The difficulty in using a prior work platform is the requirement to conform to the hardware and software stack that's implicitly specified by the system build. Furthermore, there's usually tight coupling between the applications and the underlying software system. This means that switching between systems can be challenging and recompiling software means recompiling the whole system.

In this section of the thesis, we will introduce the use of a general-purpose operating system for intermittent computing and its benefits. Given that, over time, most computing paradigms have shifted towards being more general purpose, we think that intermittent devices may do the same. Using an operating system provides more structured abstractions and separation between the applications and the kernel. Having these abstractions makes portability to new platforms easier. Additionally, having an operating system allows greater freedom in choosing the language and libraries used. Application code doesn't need to be compiled with the kernel, rather, they can be isolated binaries. We envision a future for intermittent device software that provides flexibility and configurability based on developer needs. One can imagine a system that can use checkpointing for porting legacy code or a task-based approach for novel application development and that has the ability to customize the scheduler. Additionally, an operating system can make jobs such as device updates more easily achievable by having the kernel handle replacing process binaries. We discuss the need for an operating system in further detail in Section 2.4. This work not only seeks to push on the boundary of intermittent software systems in a direction we feel is inevitable, but also explores the capability of Tock in adapting to this type of workload.

Tock was designed for persistently powered devices, so we added features in order for meaningful progress to happen through power failures. For this initial proof-of-concept, we augmented Tock with a task-based approach. We chose a task-based approach as tasks map well to the process abstraction that already exists in Tock. The key features added to Tock include the task-graph abstraction, a new scheduling algorithm to accommodate a task-based workflow, a new mechanism for inter-process communication, and the ability to save necessary state to nonvolatile memory. The addition of these features to Tock allows us to begin to test it's feasibility for use as a system for

intermittent workloads. We tested our system with a lighting control application which included sensing, computation, and transmit components. To evaluate the system, we look at time, memory, and developer overhead. The contributions of this section of the work are 1) proposing our vision for the direction of intermittent computing 2) integrating support for intermittency into Tock, an otherwise general-purpose operating system, and 3) assessing the constraints of Tock in being extended for this new type of workload.

Chapter Two

Background & Related Work

2.1 Tock Operating System

Tock is an open-source secure operating system for embedded platforms. The Tock kernel is written in Rust which is a type-safe and memory-safe language that rivals the speed of C [15]. Tock provides a process abstraction similar to the abstractions provided by other general purpose operating systems. Tock leverages the memory protection unit (MPU) that has been commonly found in embedded chips in recent years to provide hardware-isolation between those processes. Tock supports concurrently running processes and uses a preemptive round-robin scheduler to switch between them. Additionally, the Tock ecosystem includes Tockloader, a command-line utility to manage the applications on-board a device [16]. Tock has contributions from almost 100 developers and a core team of eight developers.

One of Tock's main features is sandboxed processes which maintains separation between each process and the kernel. In this environment, updating an application on a device will be easier because the binary that needs updating is only the application binary. With prior work systems that are compiled with applications, updating an application requires an update of the entire firmware. In the future, it would be more feasible to have something like an IoT app store from where processes are downloaded and added to a device. Additionally, Tock provides capsules which are platform-agnostic peripheral drivers. Capsules and the process abstraction can make the application development process easier. The various design properties of Tock and the active community surrounding it make it a good candidate for this work.

2.2 RISC-V Background

RISC-V is an open-source standard for an instruction set architecture (ISA). Open-source means that no fees need to be paid to use the ISA in an implementation, unlike an architecture such as ARM. This can reduce the barrier to entry for certain entities like small companies or academics that want to design processors [17]. RISC-V offers a lot of flexibility in how to implement the ISA. However, just because RISC-V is open-source does not mean implementations must be as well. Entities are free to have their implementations be proprietary. The work done with RISC-V is supported by RISC-V International to ensure implementations are compliant and implement the ISA specification properly. If there is compliance, then ideally there will be better software compatibility across different RISC-V cores [18].

Flexibility in implementing RISC-V comes from being able to change aspects of the design such as using various ready-made or custom ISA extensions, the usage of privilege modes to fit the use case (e.g. user, machine, hypervisor modes), or the use of memory management. This is supposed to allow RISC-V to have good scaling properties from rack-scale computing down to tightly resource-constrained computing. Having an open-source architecture has other benefits [19]. It means that significant parts of a chip design that aren't custom have tools for verification openly available [20]. It also makes it faster to create and iterate rapidly on processor designs. Given that Moore's Law is slowing down, it is thought that lower-power designs must come at the chip design level [21]. There are many companies interested in using RISC-V and some are already using it in their processors [5, 22]. There is a lot of excitement around RISC-V and its support is growing. This momentum indicated that it was important for Tock to support boards with RISC-V processors and run applications that Tock can run on ARM-based platforms. Without that support, Tock would be potentially missing a large part of the future embedded device market.

2.3 Intermittent Computing Background

Similarly to continuously powered embedded devices, the applications, system software, and hardware used as the building blocks for intermittent devices are becoming more flexible and complex.

Energy Source	Power Density	Continuous?
RF	0.0002 - $1\mu W/cm^2$	Yes
Solar	100mW/cm^2	No
Piezoelectric Vibration	$200\mu W/cm^3$	No
Thermal	$60\mu W/cm^2$	Yes

Table 2.1 Energy Sources for Harvesting & Their Characteristics [23]

This section will look at approaches to making progress in the face of intermittent power and examine how these approaches have changed over time. We will also briefly look at how hardware is trying to push intermittent system development to be more general purpose.

2.3.1 Making Forward Progress

Achieving meaningful progress on an intermittent device is challenging because of the variability of the energy source. Common sources in use include ambient light energy, kinetic energy, thermal energy, and radio frequency (RF) energy. These sources all have different characteristics and, thus, apply to different use cases. Table 2.1 gives a brief overview of these energy sources and how they differ. Making forward progress is often difficult because these sources may not provide steady power above a load's operating threshold the way a continuous source like wall-power or a battery does. This can cause a device to shut off multiple times a second and each time that happens, the contents in volatile memory are lost. Making forward progress requires storing information in nonvolatile storage which can be recalled to inform the subsequent program executions of what has been completed.

2.3.2 Task-Based vs. Checkpointing

There are two main approaches to making forward progress in intermittent software systems. The first is checkpointing. Checkpointing approaches generally insert points in the program execution where the contents of volatile memory are saved in nonvolatile memory before a power failure. There are various approaches to inserting checkpoints including using a program's control flow

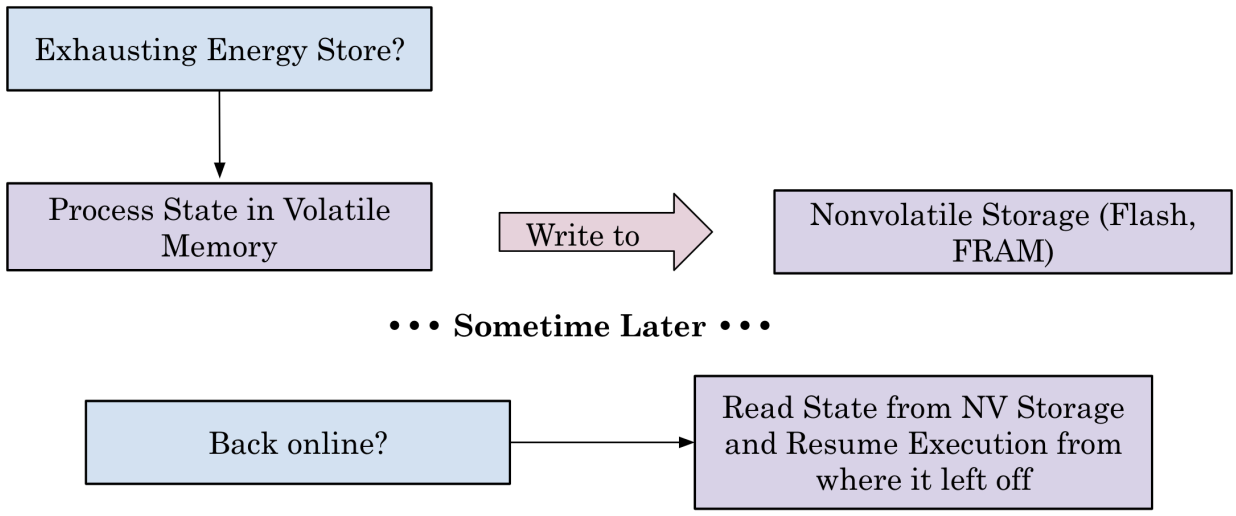

Figure 2.1 Example checkpointing approach. Checkpoints can be placed throughout code with various strategies (ex. at loop edges or around code blocks that are guaranteed to execute using less energy than the on-board storage [11, 24])

graph (*HarvOS*) [12], overprovisioning checkpoints and selectively disabling them (*Chinchilla*) [25], or identifying idempotent sections of code (*Ratchet*) [26]. Checkpoints can also be trigger points for checking the energy level to see if the contents of volatile memory actually do need to be stored [11]. Checkpoints can be time-based with a certain frequency, manually inserted, hardware-assisted with an interrupt for energy levels, or inserted through static code analysis.

Task-based approaches on the other hand ask developers to break down applications into multiple energy-atomic tasks. The tasks are then usually encoded in a graph to enforce the ordering of the tasks. The system saves state between tasks. The information that needs to be stored in the event of a power failure is where in the graph the execution is and what data is being passed between tasks. This approach allows forward progress by only needing to rollback to the beginning of a task if a power failure occurs in the middle of the task. There are several approaches to task-based systems such as using privatization-buffers for memory consistency (*Alpaca*) [27], using task-threads and other abstractions for an event driver kernel (*InK*) [28], or identifying time-critical code sections and deferring execution of the rest of the code when necessary (*CatNap*) [24].

Both of these approaches each have their advantages and disadvantages. There is greater developer overhead with task-based systems. The burden is placed on the developer to break down

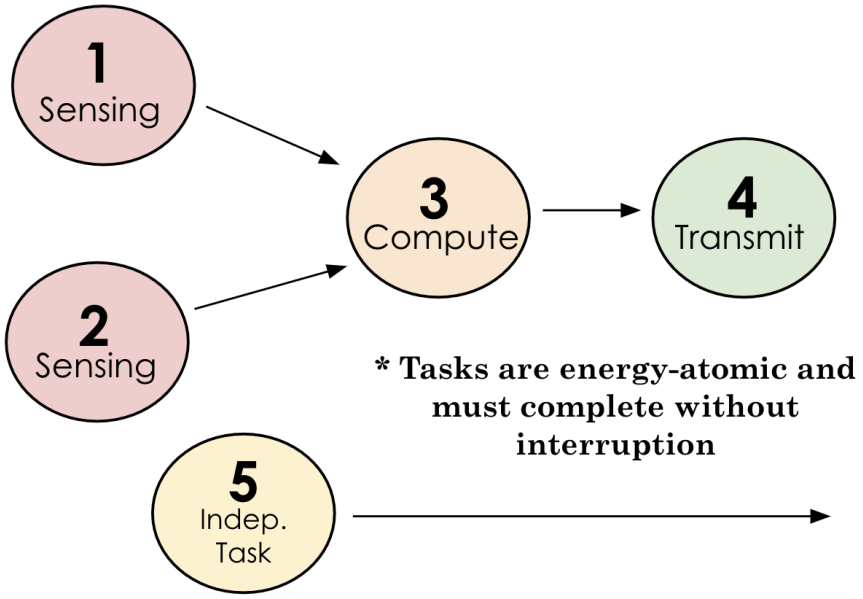

Figure 2.2 Example task-based approach. Information needed to resume execution is graph location and data passed between tasks.

monolithic applications into appropriate chunks that can feasibly execute in a power cycle. Porting legacy code from the last 20 years of sensor networking isn't easy with a task-based approach. On the other hand, the writes to nonvolatile memory for checkpointing are larger and usually happen more often than in task-based systems which makes the latter have faster performance. [29]. State of the art checkpointing could be closing that gap, however, it usually relies on having nonvolatile main-memory [13].

2.3.3 Energy Harvesting Platforms Are Becoming General Purpose

Early energy harvesting platforms were generally specialized sensor systems [30, 7]. Even the first techniques for masking intermittency were used on dedicated sensors [11]. However, like computing platforms more generally, energy harvesting needs are becoming more and more general purpose.

Evolution of Intermittent Hardware

There has been a shift in the last few years to develop hardware that focuses on flexibility which is important for two reasons. Firstly, application developers shouldn't have to settle for hardware

Name	Year	Type	Hardware Used	Energy Source
Mementos [11]	2011	Checkpointing	Simulated WISP4.1	Simulated RF
Hibernus [31]	2014	Checkpointing	TI MSP430FR5739 Eval Board	Simulated Frequencies (RF)
QuickRecall [32]	2014	Checkpointing	TI MSPEXP430FR5739 Board	Simulated (Sqaure Wave)
DINO [33]	2015	Checkpointing	WISP + TI Proj. Boards (All MSP430FR5969 MCUs)	RF + Mechanical via Drill
Ratchet [26]	2016	Checkpointing	ARMv6-M Simulator	Simulated
Chain [34]	2016	Task-based	WISP5	RFID Reader (20cm distance)
Mayfly [10]	2017	Task-based	WISP	RFID Reader (25cm distance)
HarvOS [12]	2017	Checkpointing	ST Nucleo L1254RE	N/A
Chinchilla [25]	2018	Checkpointing	WISP5	RFID Reader (20cm distance)
InK [28]	2018	Task-based	TI MSP-EXPFR5969 Eval Board	Ekho (Solar & RF)
Alpaca [27]	2019	Task-based	WISP5	RFID Reader (20cm distance)
Coati [35]	2019	Task-based	Capybara	Sub-10mA Bench Supply
Samoyed [36]	2019	Checkpointing	Capybara	RFID Reader (75cm distance)
TICS [13]	2020	Checkpointing	WISP 4.1	RF
CatNap [24]	2020	Task-based	TI MSP430FR5994 MCU with Commodity Hardware	RFID Reader
Coala [29]	2020	Task-based	WISP5.1 + MSP-EXP430FR Launchpad with Solar Cell	RF + Solar

Table 2.2 A summary of related work in software for intermittent systems. A large number of systems are implemented on a platform with some version of a TI FRAM chip and use RF as the energy source.

that doesn't meet specifications. Secondly, to meet specifications developers should not need to build platforms from the ground up as it is quite difficult and time-consuming to do so. Flexibility in hardware platforms can allow developers to customize the hardware to their use case without building a platform from scratch. As Table 2.2 illustrates, over the last decade much of the research in intermittent software has used the *WISP* platform or a device that uses an MSP430 MCU with FRAM (same MCU as WISP) [37]. Using a microcontroller with FRAM makes sense as the write speeds and power consumption of access are much lower than flash. Additionally, a large fraction of recent work uses RF energy as the target source. The usage of similar hardware provides a unified baseline for the numerous different systems being tested. Also, catering to RFID-scale devices is useful since it is the most constrained energy source. However, the focus on RF has slightly stifled the development of platforms around other viable sources of energy including solar and kinetic energy. Furthermore, often when using RF energy, the test device is placed 15-20cm from the energy source. This doesn't seem to be feasible for a host of use cases that the research community wants intermittent devices to be used for.

To address the lack of diverse hardware, platforms such as *Flicker* and *Capybara* have been developed [38, 39]. Capybara is a co-designed software/hardware platform that dynamically adapts to the energy needs of the application at runtime with a reconfigurable energy storage mechanism. This prevents situations where the energy stores are under provisioned for needed tasks or, conversely, situations where a device spends too much time charging. Flicker is another platform that promotes flexibility through its modular design that enables rapid prototyping. Developers can configure the Flicker platform to use different energy harvesting methods or sets of peripherals. Both of these platforms highlight a push in the direction of general purpose platforms for intermittent computing.

Evolution of Intermittent Software

Several systems have been proposed over the years with increasing sophistication that all use some novel variation on the task-based or checkpointing model. *Mementos*, an early software system proposed in 2011 used checkpointing as the method for making forward progress. Mementos placed checkpoints during compile time in the code depending on which instrumentation mode was chosen (e.g. at each loop latch or after each function call) [11]. A more recent checkpointing system called *TICS* proposed in 2020 is an improvement on the previous "naive" checkpointing systems that save all of volatile memory. TICS not only accommodates timer-driven, hardware assisted, and manual checkpointing, but also takes into consideration time sensitivity and memory consistency [13]. Over time, systems are able to accommodate more developer choices and simultaneously have the software system increase its responsibility to counter drawbacks of previous systems.

Concerning task-based models, there has also been a similar increase in the complexity over the years. The first task-based model, *Chain*, was proposed in 2016. Chain is a C language extension and runtime library that promised progress at task level granularity. This work used channels to provide atomicity and consistency [34]. Looking at a model from more recent work, *Coala* is a system that saves state on a sub-task scale and uses energy-aware adaptive task scheduling [29]. For both checkpointing and task-based approaches, things are getting increasingly more complicated. Iterating on both of these approaches is obviously needed to arrive at a model that is efficient and

performant. However, both models have their merits and developers should have an avenue to use both.

2.4 Why an Operating System?

This thesis begs the question - should an intermittent device run an operating system? It is a non-trivial energy and time overhead. First we'll look at low-level programming goals. The body of prior work in intermittent system software illustrates the need for support in developing applications for these devices. Writing logic for each new device to make forward progress when power outages are a hindrance is quite difficult to do. Furthermore, the target applications for these devices are complex. Devices are eventually supposed to sense, run complex computations, transmit data, perform encryption, etc. With these applications comes scheduling needs which can be complex to handle. Re-writing the software to do these operations every time is reinventing the wheel - and if it's done poorly, it can be difficult to scale or add functionality. Writing code without any support requires directly interfacing with hardware peripherals and various hardware features. Usually, hardware abstractions in software can provide interfaces that utilize hardware features well and can cut down on development time. Lastly, good software (and hardware) support allows an application developer to focus on developing applications.

Next, we can look at some higher level goals for intermittent devices. These devices are supposed to be networked and in operation for decades. This need for longevity will certainly call for security and likely functionality updates. Doing these updates can be greatly aided by having software support that verifies the security and reliability of the update [40, 41]. Furthermore, a system like Tock can reduce the size of an update that a device must receive from a full system binary compiled with the applications, to a binary of only the updated application.

Likely, an operating system is not appropriate for all intermittent devices. For example, if a device is going to be single purpose with specialized hardware interfaces or constrained to the point of not being able to have any additional overhead, any operating system overhead is probably unacceptable. However, even seemingly constrained and special-purpose devices often have requirements that suggest a more general purpose software layer is needed. For example, prior intermittent soft-

ware work tested applications like equipment monitoring and voice-activated activity recognition (applications that will eventually also require a security component) [28]. There are devices that will not be able to accommodate a full operating system, yet, it looks like many could benefit from one. Therefore, looking into how to potentially make a system like Tock work on these devices is worthwhile.

2.5 Related Operating Systems

As the focus of our work is extending an embedded operating system for various device alterations, we will discuss some other embedded operating systems that are generally used for persistently powered systems. We will look at FreeRTOS [42], RIOT OS [43], Contiki [44], and TinyOS [45]. FreeRTOS is a real-time operating system for microcontrollers that has widespread use and support for over 40 MCU architectures. RIOT OS is an open-source real-time operating system for IoT devices that provides some POSIX features and supports applications in C/C++. RIOT OS also offers additional components like a peripheral API. Contiki is a lightweight operating system for low-power microcontrollers that provides the ability to dynamically load programs. Finally, TinyOS is an application-specific operating system for heavily constrained devices.

We will now discuss some of the key differences between Tock and the previously mentioned related embedded operating systems. With the exception of Contiki, the other systems don't have any notion of separated processes. This means that generally the applications and system are tightly coupled. Contiki has loadable process modules, however, they aren't quite sandboxed like Tock as Contiki does not leverage the memory protection unit (MPU) for fault isolation. Support for using the MPU for protection against applications writing to forbidden memory is not supported by most of the mentioned prior work systems. FreeRTOS has some support for it so applications cannot alter kernel memory, but the FreeRTOS syscall interface trusts any pointers that the application passes to perform kernel writes on its behalf [4].

Chapter Three

Tock for RISC-V

In this section, we will describe the process of porting Tock to the RISC-V architecture. The scope of this work involved significant changes at the user kernel boundary. The basic support for Tock on RISC-V was a springboard for much of the work that has since been done on Tock. It also spurred discussions for altering some of the fundamental design choices that were made prior to this effort.

3.1 Challenges

There were challenges in adapting Tock for the RISC-V architecture as the architecture and its hardware implementations are in their infancy. Additionally, Tock had only been developed while using ARM Cortex-M based processors which brought about other issues.

3.1.1 ARM-Centric Design Choices

We found that there were some design choices in Tock that were thought to be portable and general-purpose but realistically catered to the ARM processors that had been used while developing Tock. Tock's design is to have clear separations between kernel code, architecture-specific code, chip-specific code, and platform-specific code. The kernel code should be generalizable to any architecture, chip, or platform. However, we found that in practice, this wasn't the case and that constructs specific to the ARM architecture had bled into the kernel design.

For example, the `User-Kernel Boundary Trait` that must be implemented by each architecture to handle context switches had function signatures that only applied to the hardware features pro-

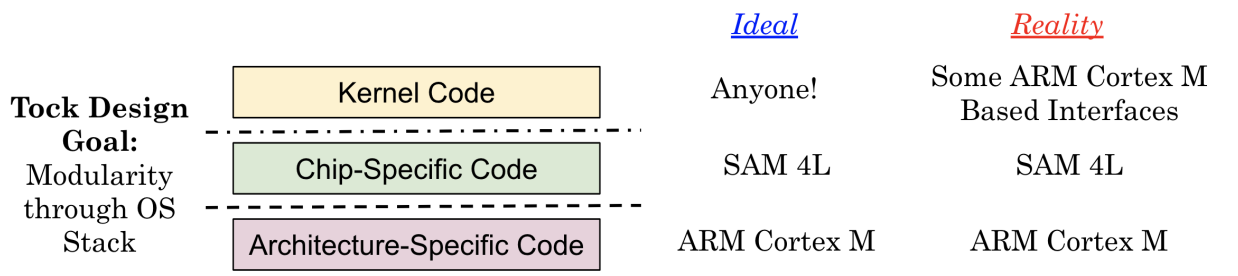

Figure 3.1 ARM Based Platforms Influence in Tock

vided by ARM cores. ARM has an SVC (SuperVisor Call) instruction that is made by unprivileged code which switches the processor mode to privileged to complete an operation needed by the unprivileged code [46]. These SVC calls use a stack frame specific to ARM, however, the `User-Kernel Boundary Trait` had functions such as `pop_stack_frame()` and `push_stack_frame()` which could not be implemented by the RISC-V architecture. In general, ARM tends to offer more hardware support for operations like context-switching that RISC-V does not [47]. Adapting Tock's interfaces around this was not a trivial thing to do.

3.1.2 An Early-Stage Architecture

Porting Tock to RISC-V was likely not as simple as porting to an established instruction set architecture (ISA) would be. There are various implementations of the RISC-V ISA for embedded platforms. They disagree on best practices for details such as which privilege modes to support, what type of interrupt controller to use, and the setup of the vector table [19]. There is an ISA specification but given the flexibility of the ISA, it is still an evolving specification. RISC-V also takes input from various parties that try to use it. So, in a sense, it looks specified from the outside and there are platforms that are built around it but some of the principles surrounding RISC-V are still changing.

In practice, this had implications on the hardware that we had available to us for this porting effort. One of the first boards used was the HiFive1 (revision A) board which only provided machine mode (one privilege mode) [48]. We ended up using was the Arty-E21 (FPGA core) which offered user-mode and machine-mode [49]. The HiFive board used a platform-level interrupt controller

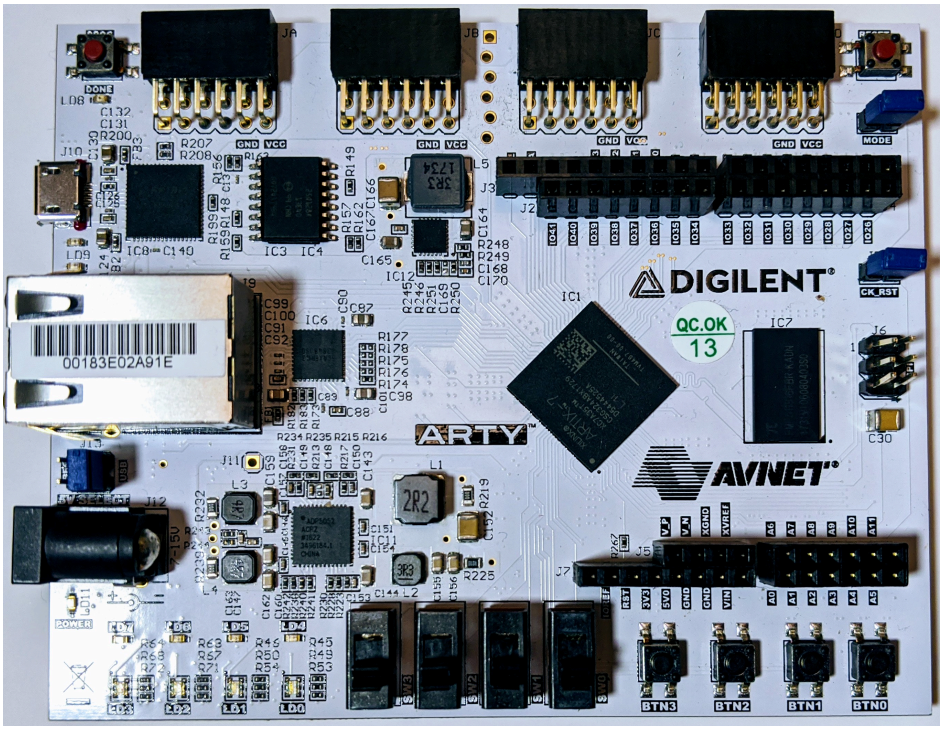

Figure 3.2 Arty-E21 FPGA Hardware Platform

(PLIC) and a core-local interrupter (CLINT). The Arty-E21, on the other hand, used a core-local interrupt controller or (CLIC). This lack of standardization makes porting between multiple RISC-V platforms a challenge. The OS needs to be able to support any and all of these platforms and should have an interface that can accommodate that. That way, it can provide the needed hardware and ISA abstractions to that a developer would find useful. The other technical challenge in working with this hardware was the occasionally sparse documentation and bugs in the FPGA implementation of the processor core.

3.2 Design & Implementation

The major effort in porting Tock to RISC-V was getting the interface between the applications (userspace) and the kernel working. This section will document that process along with the hardware platform used.

3.2.1 Hardware Platform

The hardware platform used for the initial effort to port Tock to RISC-V was the Artix-7 FPGA development board with the SiFive E21 core (Arty-E21). This board was chosen as it had the ability to switch between privilege modes (user and machine mode) which made more practical for use with Tock. The core and kernel were programmed onto the chip using OpenOCD.

3.2.2 Core-Local Interrupt Controller

The first task was to get an interface in place for the interrupt controller which, in this case, was a core-local interrupt controller (CLIC). After figuring out the interrupt mappings and the memory mappings of the interrupt registers, we were able to do useful operations like print debug statements to the console over UART. This section was tricky to get right as the originally used iteration of the core wasn't correctly registering interrupts and the documentation was somewhat lacking.

3.2.3 Context Switching & Exception Handling

After there was some notion of handling interrupts in this core, we looked at the context switching and exception handling between the application space and the kernel space. It was imperative to correctly switch between user-mode and machine-mode and then actually be able to service the context switch reason correctly. There are four context switching and exception handling scenarios and Table 3.1 documents the steps for each scenario. The `mscratch` register is a temporary holding space for anything that machine mode wants to store that cannot be altered by code running at a lower privilege [50]. Here, it is used to store the kernel stack pointer before switching to user mode. `mstatus` holds operating information such as interrupt enabling and previous privilege modes after an exception. The last register to note is `mcause` which had the information on what caused the trap handler to be called. There were also changes made on the userland side (libtock-c is the repository for C apps and the support needed to run them on Tock), which correctly passed parameters and implemented calling the syscalls. Figure 3.4 shows a diagrammatic representation of "1. Before Switching to App" from Table 3.1.

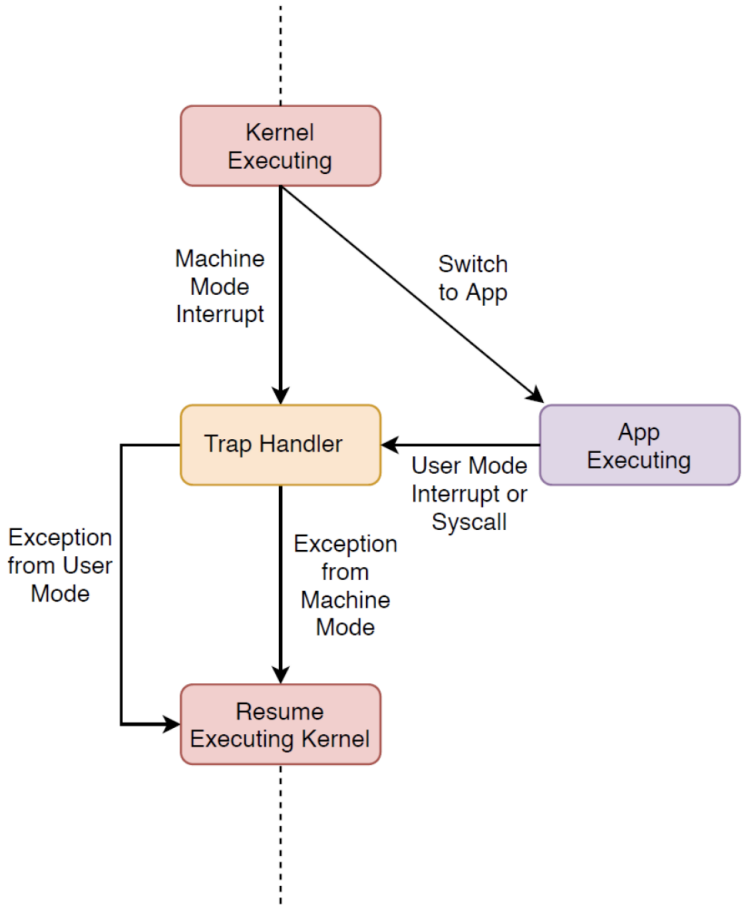

Figure 3.3 Context Switching Flow Diagram

1. Before Switching to App

- Save kernel registers, pointer to app stored state, and kernel return address on the stack
- Save kernel stack pointer to mscratch register
- Restore app registers from stored state
- Modify mstatus register to switch to user mode for app execution

2. Exception from Machine Mode

- Save registers
- Jump to board-specific trap handler code
- Restore registers

3. Exception from User Mode

- Save app registers, mcause, app stack pointer and mepc to stored state struct
- If it's an interrupt, jump to board specific trap handler code
- Modify mstatus to stay in machine mode

4. Returning to Kernel from App

- Check mcause to see what the exception was and handle appropriately (interrupt, syscall, fault)

Table 3.1 Steps for Context Switching & Exception Handling

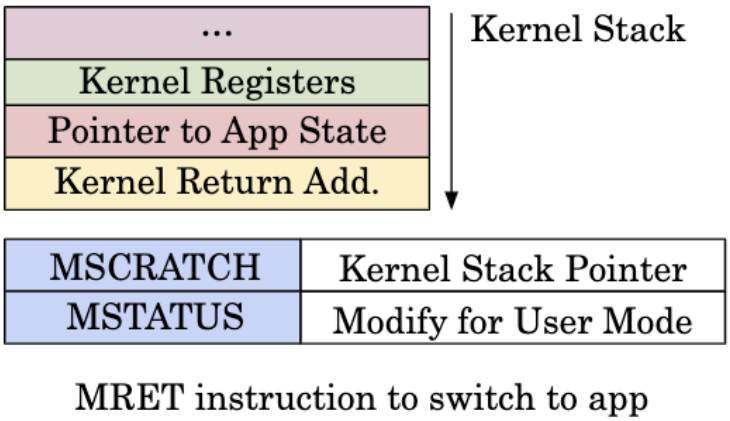

Figure 3.4 Steps for Kernel to Context Switch to App Execution (refer to first section in Table 3.1)

Old UKB Trait	Revised UKB Trait
switch_to_process()	switch_to_process()
process_detail_fmt()	process_detail_fmt()
fault_fmt()	fault_fmt()
get_syscall()	set_syscall_return_value()
pop_syscall_stack_frame()	set_process_function()
push_function_call()	

Table 3.2 Changes in the UKB Trait After Initial Porting of Tock to RISC-V

3.2.4 Altering the User Kernel Boundary (UKB) Trait

Altering the `User-Kernel Boundary (UKB) Trait` was slightly challenging, because as mentioned before, it was more tailored to the ARM architecture than was previously realized. The `UKB Trait` is an interface defined by the kernel to manage switching from kernel to userspace. The `UKB Trait` is meant to be architecture-agnostic so that any architecture can implement the trait without needing to hack around it. As mentioned in Section 3.1.1, the SVC instruction that aids in making system calls (sends information from the applications to the kernel) has a special stack frame setup by the hardware on ARM processors. The context switch support on RISC-V is not as rich in the hardware. Thus, functions like `pop_syscall_stack_frame()` were removed. The interface prior to this work and the resulting interface after this work was completed is shown in Table 3.2. Table 3.3

Old UKB Trait - All functions are rust unsafe functions

```
/// Context switch to a specific process.
switch_to_process(&self, stack_pointer: *const usize, state: &mut Self::StoredState)
    → (*mut usize, ContextSwitchReason);

/// Display architecture specific data for a process identified by its stack pointer.
process_detail_fmt(&self, stack_pointer: *const usize, state: &Self::StoredState, writer: &mut Write);

/// Display any general information about the fault.
fault_fmt(&self, writer: &mut Write);

/// Get the syscall that the process called with the appropriate arguments.
get_syscall(&self, stack_pointer: *const usize)
    → Option<Syscall>;

/// Remove the last stack frame from the process and return the new stack pointer location.
pop_syscall_stack_frame(&self, stack_pointer: *const usize, state: &mut Self::StoredState)
    → *mut usize;

/// Add a stack frame with the new function call.
push_function_call(&self, stack_pointer: *const usize, remaining_stack_memory: usize,
callback:process::FunctionCall, state: &Self::StoredState)
    → Result<*mut usize, *mut usize>;
```

Table 3.3 Old UKB Trait Details

New UKB Trait - All functions are rust unsafe functions

```
/// Context switch to a specific process.
switch_to_process(&self, stack_pointer: *const usize, state: &mut Self::StoredState)
    → (*mut usize, ContextSwitchReason);

/// Display architecture specific data for a process identified by its stack pointer.
process_detail_fmt(&self, stack_pointer: *const usize, state: &Self::StoredState, writer: &mut Write);

/// Display any general information about the fault.
fault_fmt(&self, writer: &mut Write);

/// Set the return value the process should see when it begins executing again after the syscall.
set_syscall_return_value(&self, stack_pointer: *const usize, state: &mut Self::StoredState, return_value: isize);

/// Set the function that the process should execute when it is resumed.
set_process_function(&self, stack_pointer: *const usize, remaining_stack_memory: usize,
state: &mut Self::StoredState, callback: process::FunctionCall, first_function : bool)
    → Result<*mut usize, *mut usize>;
```

Table 3.4 New UKB Trait Details

and Table 3.4 contains additional details on the old and new UKB traits and their functionality and function signatures.

The functions in the revised `UKB Trait` are as follows. `switch_to_process()` switches from kernel execution to running a process. This function is also returned to when switching back from application to kernel space and it returns the `ContextSwitchReason`. `process_detail_fmt()` displays data about a process that is architecture specific. `fault_fmt()` displays information about a fault that has occurred. `set_syscall_return_value()` sets the return value to be given to the process from the kernel as a result of the syscall invoked by the process. Lastly, `set_process_function()` sets which function the application should start at when it is next executing. Section 3.3.2 will cover the further revised interface after this work was completed.

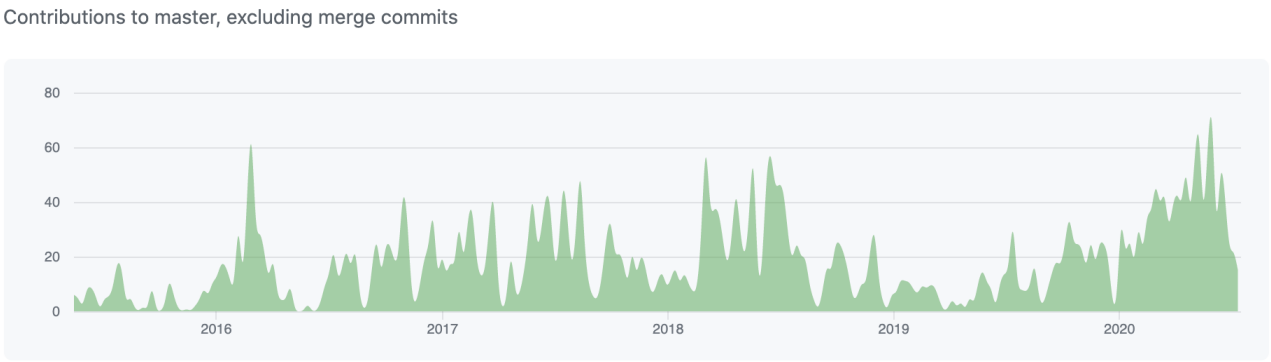

Figure 3.5 Large Increase in Development Effort Since Initial RISC-V Work was Completed in June 2019. Measured by # of Commits to Tock Master Branch.

3.3 Implications for Tock Development

Porting Tock to RISC-V has sparked a lot of subsequent improvements, discussion, and questions around not just Tock on RISC-V, but Tock's design in general. We would like to discuss the aftermath of our work here as we find its core contribution to be the momentum it generated after it's completion. Porting Tock to RISC-V has also exposed some advantages of Tock's design. The changes made to Tock after the completion of this work (mentioned later in this section) with the addition of an altered syscall interface will be used in a Tock 2.0 release.

3.3.1 Contributions & Future Directions

Porting Tock to RISC-V has had a measurable impact on the development of Tock in subsequent months. Getting RISC-V support had been a goal for the Tock community for sometime before this work was completed. In June 2019 basic Tock support for RISC-V was presented at the Secure Internet of Things Project (SITP) retreat and the code was merged into the main Tock code. Figure 3.5 shows how Tock development has picked up pace since the completion of this work. This increase in development rate reflects excitement around further growing Tock's support for RISC-V platforms. It also reflects lots of work being done to start correcting design flaws that were found in Tock as a result of our porting effort. These changes will be discussed in the next sections. Our work took the plunge that had been afoot and has led to meaningful change in the Tock ecosystem.

Many individuals and companies are interested in RISC-V which made having Tock support for RISC-V a pressing issue. Companies like Google and Western Digital are part of a long list of member of RISC-V international [5]. Google released OpenSK, an open-source implementation for a security key that is run on Tock [51]. In recent months the OpenTitan Tock Working Group with members from various companies, universities, and projects has been formed with goals relating to RISC-V and improving Tock [52]. Tock having RISC-V support was needed and helped accelerate Tock in a direction of interest.

Tock's design choice of a narrow syscall interface was found to be advantageous. The syscall interface which allows apps to send and request data to the kernel is quite small. There are only 5 system calls (6 with the intermittency work presented later in this thesis). When compared to other operating systems, this is a relatively narrow interface. However, this has proved to be quite useful when porting between platforms. Having a small syscall interface makes it easier to think about what is required by the architecture and hardware to support it. The syscall interface is still being iterated on but the property of having few necessary calls has proven to be a useful design decision.

3.3.2 Context Switch Interface

Further changes have been made to the `User-Kernel Boundary Trait` since the completion of the work that has been described thus far. There have been continuing conversations about creating an interface in the kernel that is architecture-agnostic. The changes that have since been made include adding `initialize_process()`, removing `fault_fmt()`, and changing `process_detail_fmt()` to `print_context()`. `initialize_process()` is invoked by the kernel before a process can begin executing to do any architecture specific process setup. `fault_fmt()` was replaced during a refactor that moved some portion of printing state into the chip-specific code. Having it in the kernel requires a process on-board to invoke the function which is not always the case. The same refactor also added `print_context()`. These changes have been made over the course of several months and illustrate that the `User-Kernel Boundary Trait` continues to evolves.

3.3.3 RISC-V Timer Subsystem

The RISC-V timer subsystem is quite different than what is offered in ARM Cortex-M cores. ARM timers count up, fire an interrupt if equal to a value set in a compare register, can be reset, stopped, and started. The timers are usually 24-bit or 32-bit. The RISC-V timer is a 64-bit timer that only counts up and an interrupt is fired when the value in the compare register is less than or equal to the timer value. These design differences have started a conversation in Tock around how to handle the RISC-V timer, particularly in a 32-bit system.

The interface in the kernel to manage process timeslices is one area that has seen change in relation to this problem. This interface must be implemented by each architecture. The kernel's interface for this was called `systick` and was developed for the SysTick timer on ARM processors. The SysTick timer is a feature which fires an interrupt at a set interval to assist the operating system in context switching. This enabled Tock's scheduler to preempt processes. This is not a feature offered on RISC-V chips. The interface was recently changed to `SchedulerTimer` which can be implemented by RISC-V. Now, SysTick specific code is in the architecture-specific part of the code for Cortex-M based processors.

Current RISC-V chips do not have the support to implement this timer interface in a way that it acts like the the SysTick timer. To address this, a `Scheduler Trait` is being worked on to make the scheduler cooperative for the currently supported RISC-V chips. If the timer hardware exists to implement the `SchedulerTimer Trait`, then now it can do so without having to navigate the ARM-specific terminology and functions.

Chapter Four

Tock for Intermittent Computing

4.1 Vision for Intermittent Software

We believe that innovations in intermittent computing along with the previously discussed increase in complexity of these systems make the need for a general-purpose batteryless device inevitable. In order to produce the wide range of batteryless devices that are needed, writing software should not be painstaking and device flexibility is paramount. Here we will discuss what our future vision for Tock and operating systems in general on intermittent devices looks like.

4.1.1 Flexibility in Software System Features

The advantage of running an operating system like Tock is the strict division between the applications and the kernel. Devices are easier to multiprogram as applications don't need to be compiled with the underlying system. The process binaries are isolated which in theory provides a "write once, run anywhere" sort of framework. Additionally, there is flexibility with using external libraries without needing to consider the entire toolchain. Another benefit of this approach is that a developer isn't tied down to one language. Additionally, having a process abstraction provides the ability to change applications without having to alter and re-upload the operating system and helps isolate faulty processes from the rest of the system.

We envision a system where programmers can use default settings provided in an intermittent operating system, or be able to change various parameters to suit their use case better. Given that there are trade-offs between using a task-based approach and a checkpointing approach, a

developer should be able to choose between them. Generally, checkpointing has lower programmer overhead (particularly when porting legacy code) but state of the art checkpointing systems don't have the same performance as state of the art task-based systems. In a system that supports both of these approaches, developers can choose, for example, to port a legacy long-running program using checkpoints or write a new application using a task-based method.

There are other customizations that could be useful as well. A developer should be able to choose a scheduling algorithm to suit their use case. For example, some may want a priority-based scheduler and some cases may be better off with a simple round-robin scheme. Another aspect of customization could be the aggressiveness of state-saving. State could be saved at all task-boundaries or just when the system notes that it is low on energy. Lastly, porting an operating system to a new platform should not be difficult. A large portion of recent work all runs on the same family of MCUs and are all custom systems. Having an operating system with structured abstractions will make porting to new hardware much easier.

4.1.2 Updates, New Features, & Cooperative Operation

If an energy harvesting node is deployed somewhere hard to reach and intended to operate for decades, it is reasonable to expect that the node's functionality should be able to change over time. A node should be able to perform software updates to improve its operations and patch security problems. An operating system will make updates easier. If the board can receive the needed binary, the kernel has the ability to update that application or put a new version in flash.

We commonly think of embedded devices and energy harvesting devices as single-purpose when their firmware is extremely specific to that purpose. This is distinctly different from other general purpose platforms. Instead, the device could have several applications in flash that make use of the various peripherals onboard. A subset of these apps can be chosen to run thus determining the node's functionality. Maybe during the day a node reports sensor readings and at night switches to a different set of sensors or a long running computation. There can also be a notion of changing what a node needs to do over longer periods of time. Similarly, there can be cooperation between nodes where a node can offload computation to another node if one node is in much better harvesting

conditions than another. This adaptability is critical as energy harvesting devices fulfill their main motivation and scale up significantly in number. If this aspect of developing intermittent systems is not taken into consideration, the billions of devices we intend to be operating for decades will not be future-proof.

4.1.3 Applications

There are many interesting applications that could be more easily enabled with the use of a full operating system. We will discuss some potential applications in this section. These applications can leverage the separation of processes from the operating system in order to make the devices easier to program.

Indoor Occupancy Detection

Occupancy detection is a key part of creating smart buildings by providing occupancy data for a variety of applications including some which help reduce energy consumption [53]. Camera based occupancy detection can be a hazard when it comes to privacy concerns and a low power setup is important for scalability. Techniques are being developed to do RF-based occupancy detection. This requires multiple distinct nodes to send packets to a central node which can use the characteristics of the packets to determine occupancy [54]. There is interest in making this application run locally and on resource constrained devices [55].

Image Recognition to Retrofit Meters & Gauges

There are many gauges and meters deployed in the world that are not "smart" and often require someone to manually go and check them. This can be highly inconvenient and can also can make quick response to issues infeasible. People want to digitize and make these gauges and meters wireless and that need is being addressed by products like EnergyCam [56]. EnergyCam's website claims that the Munich airport alone has 10,000 analog meters. If there's a need to retrofit meters and gauges around the world, it becomes clear these devices will need to become batteryless. However, EnergyCam must be battery or wall-powered. Similarly to EnergyCam, an intermittent device

could take a very low resolution picture and do some image analysis in order to send some useful information from these meters and gauges.

How does an OS help?

Both of these applications would be easier to develop with an operating system. They both have machine learning components that may require some adaptation or updating over time. One could imagine that these devices come in a form where the sensing code could be left alone but only the model or analysis portion needs to be changed. This is more difficult if the whole system needs to be compiled together. There is a need for many such applications on intermittent devices and they would benefit from the use of an operating system.

4.2 Extending Tock for Intermittency

Tock was designed for continuously powered systems so it lacked the mechanisms needed to make meaningful progress through power failures. Given a task-based approach, the preemptive round-robin scheduling algorithm does not work. The scheduler has to appropriately schedule tasks in the correct order and only when the tasks they are dependent upon complete. This section will lay out the design details of the features added to Tock.

4.2.1 Design Details

The design choice for augmenting Tock was to use a task-based approach. The process abstraction in Tock mapped well to tasks and so this approach was a more natural first extension. Figure 4.1 displays the system overview and indicates the operation of Tock prior to this work and then after the features for intermittency were added. The design decisions were guided by the various needs for computing through power failures in a task-based manner while also trying to adhere to Tock's existing design priciples when possible.

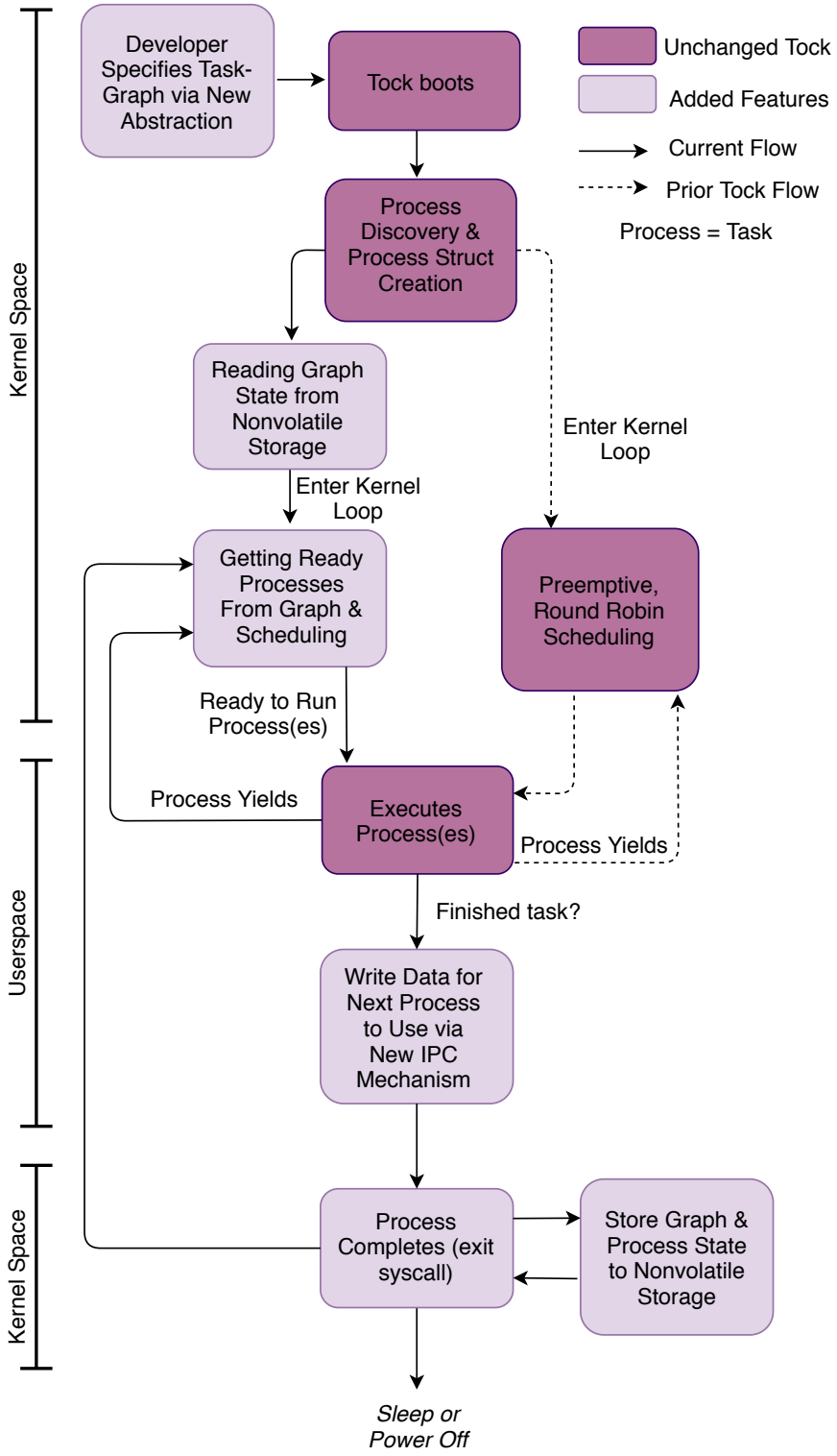

Figure 4.1 System Diagram. The darker boxes are Tock constructs that were largely left unchanged or were part of the system before. Lighter boxes are the extension for power failure immunity.

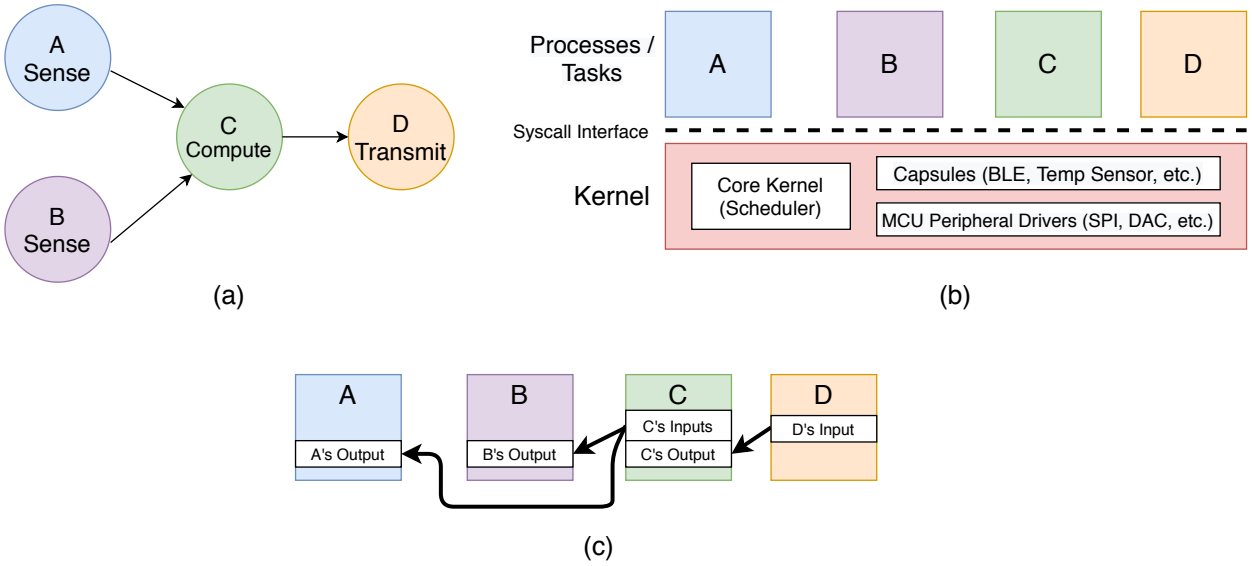

Figure 4.2 Tock system overview with a task-based approach. (a) is an example task graph which first senses, then performs a computation on that data, then transmits the results of the computation. (b) shows tasks loaded onto the Tock kernel. (c) is the shared memory model where a process's input from a previous task is a pointer to an output buffer in the memory space of that previous task.

Graphical Representation of Tasks

The graphical representation of tasks is similar to that of previous work. In a task-based approach, a programmer must break down their application(s) into discrete parts that are energy-atomic. This means if a power failure occurs before completing a task, when the device restarts, it will rollback to the beginning of that task. The nodes of a graph represent each task and the edges provide information about the ordering of tasks. Given that the Tock process abstraction was used for tasks, each task on the device is a process. Thus, for the purposes of this work for intermittent support, the terms task and process will refer to an energy atomic section of code. Figure 4.2 illustrates how a task graph maps to Tock processes and how they pass data between each other.

Graphs are represented using a 2-D matrix that the programmer must give to the kernel. Each entry in the array will represent a graph edge. For example `[[0, 1], [1, 2]]` would be the ordering `proc 0` → `proc 1` → `proc 2`. If an edge looks like `[3, 3]` that means that `proc 3` doesn't depend on anything else and can run at any time. This format supports multiple graphs. If multiple processes are free to run, Tock will run up to two processes concurrently.

Augmenting the Syscall Interface

Prior to this work, Tock did not have a way for a process to indicate to the kernel that it had completed. In order for a process to be able to signal that it is done executing, an `exit` system call was added. The full set of system calls now includes `yield`, `subscribe`, `command`, `allow`, `memop`, and `exit`. `yield` stops a process from being scheduled until a callback reschedules it. `subscribe` sets a callback function to execute after a specific event has occurred. `command` sends instructions to a specific driver. `allow` specifies a portion of memory to be shared between the process and the kernel. `memop` allows the process to do various things with the process memory. Finally, `exit` provides a way for the process to indicate that it has run to completion. With the addition of this syscall, the kernel can know when to move on to the next task in the task graph.

This change to the syscall interface also required an additional process state as the scheduler uses the state to determine whether or not to schedule a task. The states include `Running`, `Yielded`, `Fault`, and now, `Ended`. A process will be put in the `Ended` state if a process successfully calls the `exit` syscall.

Task Scheduling

The scheduler was altered with a function to parse through the graph and determine which processes are ready to run. It checks through the provided dependencies and will mark all of the tasks that are not waiting for the completion of another process. This function is run when there are no ready tasks left to run. The ready-to-run processes are executed in order of the assigned process IDs that are used to create the task-graph. The fundamental change to the scheduler was the list of processes that it has to choose from which enables it to adhere to the task-graph.

Inter-Process Communication

Generally the tasks in the task graph need to be able to pass information between each other. Tock had an existing mechanism for inter-process communication (IPC), but a new IPC implementation was developed for this system. Processes in Tock have a struct containing all of the information that the kernel needs to know about them. This struct has been augmented to support input and output

buffers between processes. Each process has a pointer to it's output buffer and one or multiple pointers for it's input buffer(s). This structure allows multiple processes to share data with a single process and vice-versa. The output buffers for each process are within their own memory space so that a process could write to its own memory.

When a board starts up, all the processes on a board need to be loaded into main memory from flash. During that setup, each node of the task graph is used to setup the appropriate input/output pointers. In a task's code, memop syscalls are made to request the pointers for the apps to write to or read from the appropriate buffers. For a call to request an input buffer, only the memop type specifier is required. For requesting the location(s) of the input buffer(s), a process must also provide the specifier for the process(es) it is reading from.

Saving State to Nonvolatile Memory

The kernel is responsible for writing enough state to nonvolatile memory so the device can restart at an appropriate place in the task graph after recovering from a power failure. The upside of this design is that writing to nonvolatile memory is consolidated to one kernel function which makes it easier to maintain memory consistency compared to when processes can also write to nonvolatile memory. Before a power failure is expected, the kernel writes information about what point in the task graph has been reached and any information that needs to pass between processes. All the information is stored in a struct that is written to nonvolatile memory. When Tock boots, it reads in the struct from nonvolatile storage (the struct is initialized if it's the first time booting), computes the ready to run processes, and begins executing them. All of this is done using existing capsules and drivers in Tock to interface with nonvolatile storage.

One of the implications of this design decision was having to put asynchronous calls in an otherwise synchronous kernel. When Tock boots, it must make a blocking nonvolatile storage read call and while saving state it must make a write call. At that point the kernel waits for interrupts or callbacks or sleeps, but cannot schedule any other processes to run. Generally, these calls are made by processes or capsules and are not blocking as the kernel can schedule other processes to

run. While this changes the synchronous paradigm of the kernel, it is unavoidable as the task graph must be established before processes can be scheduled.

Working with Augmented Tock

The current development flow for a programmer using Tock would be as follows. 1) Write the tasks/processes and flash them onto the device. 2) Look up the respective application identifiers on the board. This can be easily done using Tockloader. Tockloader is a command line tool to install Tock and applications onto the hardware. Tockloader will flash the applications onto the board in size order from largest to smallest. 3) Create a graph using those identifiers that will be given to the kernel. 4) Update any apps that need an input pointer with the correct parameter to pass to the `memop` call (see Section 4.3.3). The second step exists at this time as there is still some open discussion on the correct identifiers to use for processes. Once this has been settled, the second step can be eliminated. Section 5.1.4 also discusses how to eliminate the fourth step.

4.3 Implementation

This section will cover the details of the constructs used for augmenting Tock for intermittency. This includes the specifics of how the kernel stores state and how information is passed between processes. There are times when existing design choices in Tock placed constraints on this work but also instances of Tock enabling much needed generality.

4.3.1 Graph Information Struct for Saving State

A struct detailing various information about the processes was used to keep state about the task execution. This struct called `graph_info` had four fields in order to preserve state through power failures. These fields can be seen in Listing 4.1. The first is the initialization indicator. This is a 32-bit "magic" value that lets the kernel know if the struct has been initialized in memory or not. The first time the device boots, the graph information struct has not been initialized yet. The kernel detects this and initializes a new struct. On subsequent executions the struct is read out of nonvolatile memory.

The next three fields in the struct are arrays that hold process state data. The first is the array of ready to run processes. This is an array of bytes, one for each process. This array holds information on which processes have been marked as ready to run on the current execution or prior executions. The second array is the array of ended processes. This one is similar to the ready processes but instead indicates which processes have run to completion. These two arrays together contain the information that the kernel needs to determines which processes can be executed. The final field is the the data that needs to be passed between processes stored in an array that corresponds with the output buffer's process ID. When all of the processes have completed running (not including long running independent tasks), the graph is reset and the program execution begins again from the start of the task graph.

Listing 4.1 . Graph Information Stored in Nonvolatile Storage

```
struct GraphInfo{
    init_indicator: u32 ,
    ready_procs_arr: [u8; numProcs],
    ended_procs_arr: [u8; numProcs],
    ipc_data: [bufferSize; numProcs]
}
```

4.3.2 Interacting with Nonvolatile Memory in the Kernel

The kernel is responsible for making the calls to read and write the `graph_info` struct from nonvolatile storage. Both reads and writes to nonvolatile storage block applications from being scheduled. For this implementation, the nonvolatile storage type is flash. Reads to flash are relatively fast whereas writes are quite slow. An upside of the decision to keep all of the IPC data in the `graph_info` struct was the limited number of writes to flash. Tock has a unified interface for any type of nonvolatile storage, so, though the chosen platform used flash, it could easily be swapped with another storage type like FRAM without changing the kernel calls. Figure 4.1 depicts multiple different paths after a process exits which dictates how aggressively state is saved. In this implementation, any process that is ready to run will do so, the kernel will write the state to memory,

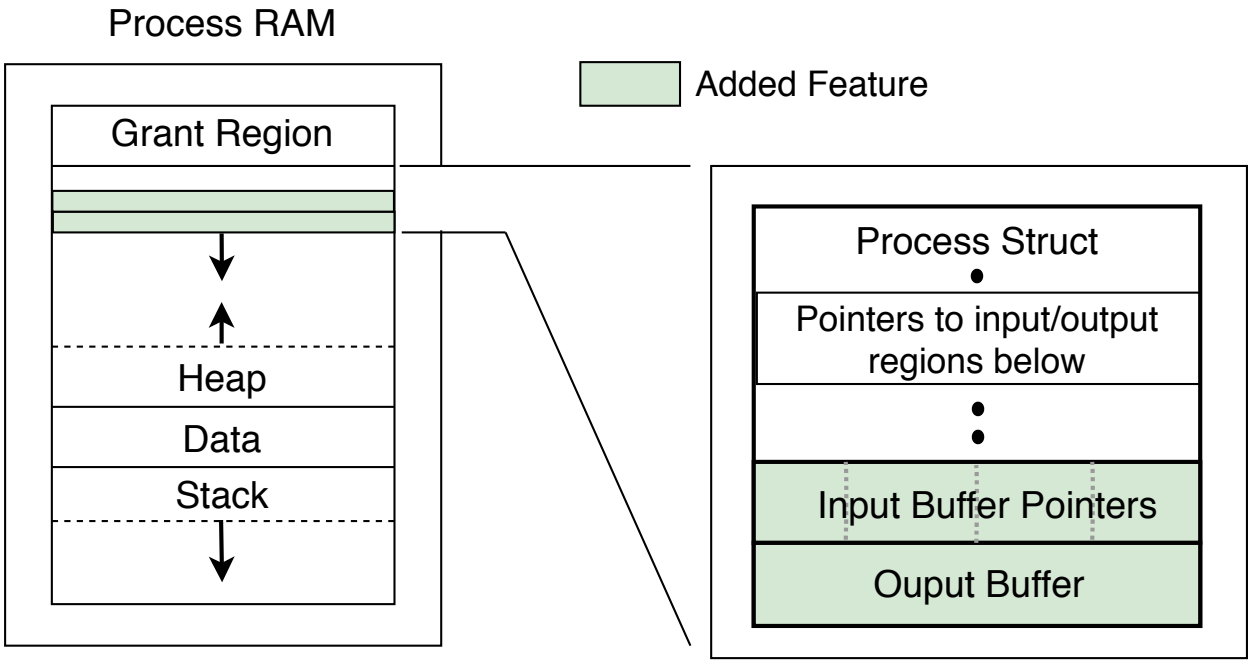

Figure 4.3 Mechanism for shared memory. This is the process space in memeory for a single process.

and then the chip will sleep and essentially wait to be rebooted. This made it easy to manually simulate power outages.

4.3.3 Altering The Process Struct for IPC

As mentioned in the design section, the inter-process communication mechanism was designed to handle multiple processes needing to pass on data to one process and one process being able to pass its input to multiple other processes. This requires allocating a buffer for processes to write their outputs to, and providing pointers to processes for where they should read their inputs from. Since the number of buffers required scales with the number of processes, we allocate buffers in the grant region of processes. Tock uses grant regions for capsules (type-safe drivers for sensors, MCU peripherals, etc.) to be able to dynamically respond to process requests. Grant regions are in process memory but are only accessible to the kernel which is how capsule resources are carefully monitored.

Because each process is assumed to be energy-atomic, and in one activation cycle only one process might execute, we initially allocated the buffers in the grant region of the process that would use the buffer's contents as inputs. That is, processes would write their outputs to memory shared from another process. This way, when a process starts executing, its input buffer is easily accessible. Allocating buffers this way works for simple graphs of processes, but does not scale to more complicated graphs. In particular, multiple processes might be writing their output to the same process. So instead, we allocate each process's output buffer in its out memory space, and provide input pointers so a process can read the outputs of other processes. When processes are loaded into memory, a starting grant region is allocated followed by a process struct. The input and output buffers were added after this process struct as shown in Figure 4.3.

The number of processes that can be loaded into memory is limited. On the platform used for this implementation, that number is generally four. This number is flexible depending on the size of the processes. It was used to dictate how many pointers are in the input pointer array. The pointers in the input pointer array are set during process loading based on the process ID. If `process 1` depends on input from `process 0`, then the 0th pointer in `process 1`'s input pointer array will point to the output buffer of `process 0`. When an application requests an input pointer, it must also supply the appropriate process ID in the `memop` call. A process will not need to do this for the output buffer pointer as there is only one.

4.3.4 Hardware Platform

This system was implemented on the Imix platform [57]. Imix was developed for use with Tock as a catch-all platform for developing Tock applications. The Imix uses a SAM4L Cortex-M4 MCU and has 512 kB of flash storage and 64kB of SRAM. This platform has sensors for light, temperature, humidity, and an accelerometer. Additionally for BLE, the Imix has a Nordic NRF51822 chip. This platform was chosen for a few reasons. Firstly, This platform is quite stable for use with Tock which is useful for this iteration of the work as we wanted to focus on adding intermittency to Tock without the overhead of porting it to a new platform. Secondly, Imix is a good general use platform that allowed for a variety of applications to be developed.

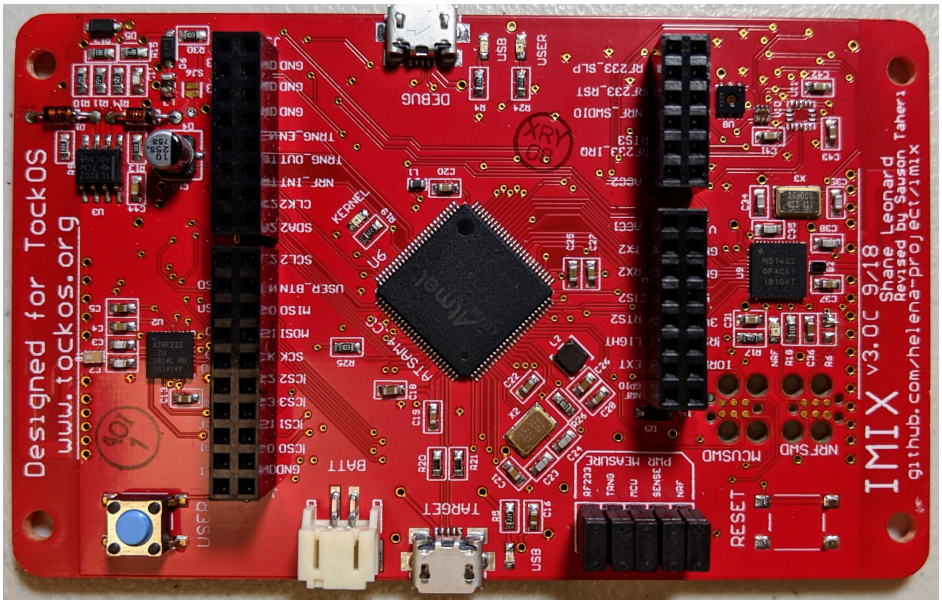

Figure 4.4 Imix Hardware Platform

4.4 Evaluation

We evaluate Tock for energy harvesting platforms by implementing an application that is broken down into energy-atomic tasks. We look at the various overheads of Tock and discuss each one. We also implement a set of applications that can be selectively chosen at each boot to change the functionality of the device.

4.4.1 Experimental Setup

For our experiments, the augmented version of Tock was run on an Imix board. The Imix platform used a Cortex-M4 microcontroller running at 48MHz. The board had the goal of `sensing` → `computing` → `transmitting`. The time overhead was measured using a Techtronix TBS2000 Series oscilloscope. A GPIO pin was selectively toggled at runtime to assess the time taken by different parts of the code.

4.4.2 Lighting Control Application

The application that was tested was a task-based implementation of a lighting control program. This application was broken down into three tasks. The first was sensing, then computing, and finally transmitting. First, the ambient light was sensed, then a computation was done on the sample to get a value by which the lighting should be adjusted. Lastly, that value was advertised over BLE. The kernel was set up so that it ran a process and a physical button press caused a reset to happen. The metrics used to assess Tock were the time overhead, the memory overhead, and the developer overhead.

4.4.3 Results & Discussion

We will discuss the results of implementing features for power failure immune computing on Tock. This section will look at the memory and time overhead of the added features compared to the base Tock implementation. The developer overhead will be commented on in relation to other task-based systems.

Memory Overhead

The memory overheads can be seen in Table 4.1. Flash with Tockloader padding refers to how the Tockloader utility flashes applications onto the board. Tockloader loads processes into flash by rounding up to a power of two (i.e. if a program is over 4096 bytes, it will be padded to take up 8192 bytes). Tock has a much larger overhead in both nonvolatile memory and RAM than prior work. Prior work systems are normally in the range of several KB in non-volatile storage and under 2 KB in memory. In part, this is to be expected as it is a general-purpose operating system and provides abstractions that require a larger memory footprint. Additionally other systems don't have dedicated kernel or app space. Memory is reported in relation to applications that are under test.

It's also important to look at how the changes made to Tock affect the memory overhead. The kernel grew by a few KB which had a pretty significant impact on the RAM usage. There are some places that can be optimized. When the graph structure is read in and out of nonvolatile storage there is a lot of care taken that information is converted from byte streams to structs in a safe way.

	Flash	Flash w/ Tockloader Padding	Memory
Kernel Unchanged	154.8KB	155.1KB	24.6KB
Kernel for Intermittency	157.9KB	158.2KB	32.8KB
Lighting App Sense	4.3KB	8.2KB	8.2KB
Lighting App Compute	7.8KB	8.2KB	8.2KB
Lighting App Transmit	30.7KB	32.8KB	16.4KB

Table 4.1 Memory Overhead

There is a lot of cloning/copying and some of this could likely be done more optimally. There were four functions added to the core kernel code which is not trivial. The impact of these on the memory footprint could certainly be reduced, however, they do represent a significant augmentation to the functionality of the kernel.

It is pertinent to discuss whether it is feasible to run something like this on an intermittent device. There is a lot of additional process overhead from mapping the task abstraction to Tock processes. If there are libraries in common, then they get compiled with each process separately. Something to note about the application sizes is that all of them were issuing `printf` commands, without which they would likely have had smaller memory footprints. Having the task abstraction for certain applications makes sense. It's easier to "plug-and-play" and it's often useful to write modular code. However, Tock may also lend itself well to checkpointing so that additional process overhead is limited. It isn't really feasible to run this particular implementation of Tock on the platforms seen in much of the related work. However, with some optimizations and the addition of FRAM, it may be feasible to run on some slightly less constrained intermittent devices in the near future.

Time Overhead

The startup time and the time to write to flash were calculated for this implementation of Tock. We took these measurements by toggling a GPIO pin in the Tock code. The startup time includes initializing and configuring all of the on-board peripherals/capsules, process discovery & creation, reading the graph state from flash, getting the processes that are ready to run, and finally, scheduling them. The startup time along with the itemized list of each contributing operation is shown in Table 4.2. For these measurements the Tock bootloader, which allows interfacing with the device

Tock Function (Startup Cost)	≈ Time (μs)	Percentage
Total Tock Startup Time	8500	
Peripheral Initialization	7600	89.4%
Process Discovery/Creation	390	4.6%
Reading State from Flash	110	1.3%
Getting Ready Processes & Scheduling	400	4.7%
Tock Function (Addt. Overhead)	≈ Time (μs)	Percentage
Writing to Flash	10800	N/A

Table 4.2 Time Overhead

over USB, was removed. With the exception of writing to flash, a lot of the time costs of Tock are in line with recent work. However, it's not really a valid comparison given the difference in the platforms used. For example, the Imix SAM4L's clock speed was set at 48MHz which is at least six times faster than the MCUs in related systems.

Instead, we can use the time overheads to see how the new parts of Tock add to the time overhead and what parts of Tock are the bottlenecks. The features added to Tock contribute approximately 6% to the startup time which shows that the new functionality doesn't hurt the time overhead very much. The biggest part of the startup comes from initializing and configuring all of the on-board peripherals. This cost could be partially lessened by having a configuration script for Tock which only initializes and configures peripherals that are necessary for the selected applications. The other obvious part to look at is writing to flash which takes longer than the startup time to complete. Most likely, the best way to mitigate this is to use FRAM as nonvolatile storage as it has quicker write times than flash. There was also 200 ms of time from when the MCU was powered to when Tock started. This could be attributed to the chip doing operations such as clock and register setup, but further investigation is needed.

Process discovery in Tock is different than systems in prior work. Tock looks through flash at runtime for the Tock Binary Format (TBF) headers, uses that to see where processes are, and then loads them into memory. This is a design choice for existing Tock because it is a more general purpose design and in a continuously powered system, this overhead is only incurred once. However, this is not a fundamental problem. Tock can be further altered by directly giving the kernel appropriate function pointers at compile-time.

Developer Overhead

In terms of developer overhead this system has reasonable usability with a few notable hindrances. The language constructs to learn include how to create a task graph, the `memop` calls for the input/output pointers, and using the `exit` function. This is comparable with other task-based systems. Usually other systems also have a construct for creating tasks. However, Tock does not need an additional construct for that since tasks map to processes. Some recent work has more language constructs to learn, however those provide features like timing constraints which this implementation of Tock does not offer. Besides those three constructs (two that go in the actual application code), applications can be written like normal C-code. The other upside of this platform is having the process isolation and Tockloader utility (prior to deployment) so that applications and the kernel can be compiled and updated independent of each other. All that said, there is a little bit of difficulty in development arising from the need to know the task's IDs. This is discussed further in Section 5.1.4.

4.4.4 Toggling Node Functionality

We implemented a second application to test the idea of using two different task graphs to toggle the operating of the node. The idea of multiple task graphs in a task based system is not new. However, this is exploring the notion of having a platform ship with apps and being able to toggle it's functionality while deployed. The setup for this was having two different sensing goals for the device which can be seen in Figure 4.5. There were four simple tasks deployed on board. The tasks were 1) sense temperature, 2) print temperature to console, 3) sense light, and 4) print light reading to console. The subset of tasks was chosen based on the input from a GPIO pin. Either the board would sense and print the temperature ($1 \rightarrow 2$), or it would sense and print the ambient light ($3 \rightarrow 4$). Another Imix board was used to send the signal via a push-button for which set of processes to run. In this implementation, the device needed to reboot before a new task graph could be assigned to the kernel. Given the potential frequency of rebooting, this isn't really a problem. This application illustrates how a device could be shipped with many apps and multiple task graphs. A node's operation could easily be changed by updating which task-graph is selected for use.

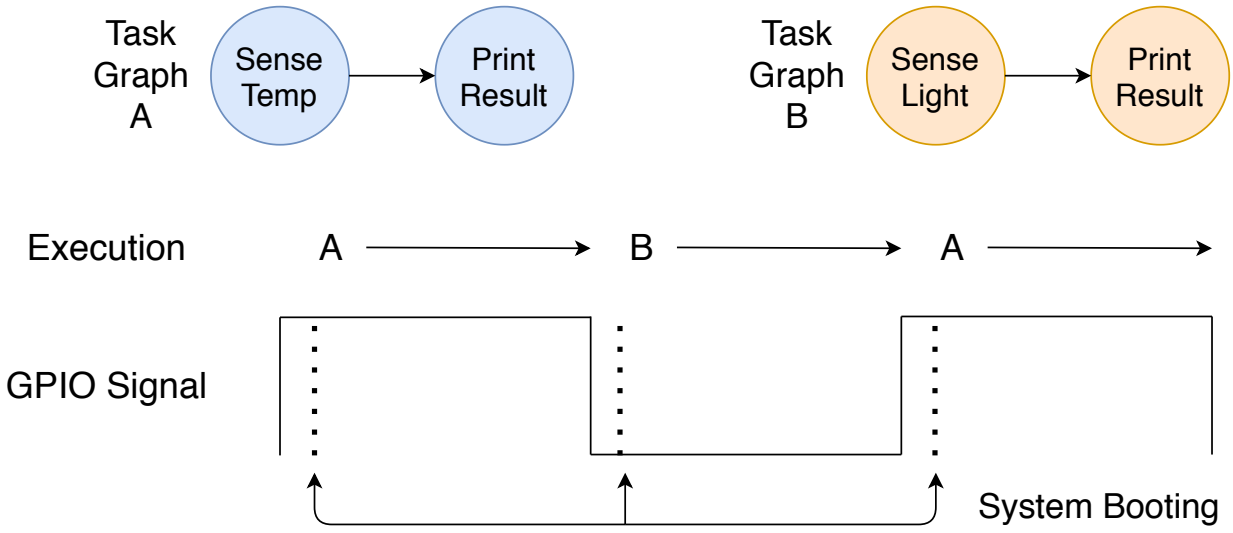

Figure 4.5 Toggling the functionality of a device depending on the input from a GPIO pin. Switches between temperature sensing and light sensing. This illustrates the idea of toggling the functionality of a node after deployment.

Chapter Five

Limitations & Future Work

5.1 Tock for Intermittency System Changes

In this section we will address the shortcomings of our evaluated system and address the next steps to improving the performance of Tock for intermittent computing.

5.1.1 Loading Processes into Memory

In this implementation, Tock's process discovery and creation model was left relatively untouched. As mentioned previously, Tockloader loads processes into flash by rounding up the address to a power of two. Then, based on a configured parameter for the number of processes to load and the space in RAM, Tock loads all of the processes it discovers and has room for from flash at boot. This presents an issue for a task-based system where the tasks are energy-atomic and often a large computation could have many nodes in its task graph. All of the individual tasks have to be loaded into memory as there is not a robust mechanism to pick a subset of tasks to run in an energy cycle. An additional challenge is that inputs in Tock are pointers to a previous task's space in RAM. We explored only loading in processes that need to be loaded in (i.e. processes that will run in the current execution + processes that they were dependent on), however a new method for process discovery that enables this feature wasn't fully implemented.

5.1.2 Tooling for Graph Processing

The current implementation of Tock relies on the developer to ensure that the graph given to the kernel matches with the processes on board. That is, the task graph must contain a subset of the process IDs of the apps loaded into flash. In prior work such as Mayfly, there is a mechanism in place for doing task graph validation which would be useful in our system. Additionally, right now the developer has to obtain the process IDs of the relevant tasks in order to construct the task graph. This is not particularly difficult but can become cumbersome during development. A graph would be better made up of task names or some identifier that is converted to the correct representation for the kernel.

5.1.3 Inter-Process Buffers

As discussed in Section 5.1.1, a process needs its predecessor process in RAM to get a valid input pointer to the previous process's space in memory. This is inconvenient and there are a number of different ways to change process discovery and the shared memory model. Process state and outputs could be saved with each process in nonvolatile storage as opposed to in the single `graph info` struct. Rather than restoring state by putting the output buffers exactly where they were, the kernel could perform a hand-off of the inputs. In terms of loading processes, Tock looks through flash for the correct headers in order. Initial thoughts on improving process discovery is to add some notion of the addresses for each application and an ID translating to an index for each one. This way, the kernel can directly find specific processes in order to load them. Additionally, the output buffers were small and could only pass four bytes. We did that as a proof of concept but increasing the buffer size should be relatively simple.

5.1.4 Universal Task ID & Decoupling Task-Graphs From Applications

There are two key things that would improve the developer experience and application functionality with Tock. The first is a having a unified name construct that can be used by developers to refer to other tasks. It should be possible while writing apps and requesting the input buffer pointer to just

use the name of the previous task. It would cut down on the number of steps to test each iteration of code development.

The other issue is that the task-graph identifiers are referenced in the applications which should not be the case (i.e. through the input buffer pointers). This mostly boils down to an implementation issue due to lacking some functionality on the kernel side. One way to remove this coupling is to have a more robust interface exposed by the kernel to request pointers to input buffers. It would be up to the kernel to be more hands-on with extracting data flow needs from the task graph. The application could request the number of input pointers it has and then make the appropriate number of requests for each input pointer. This way a print app, for example, would not have to know what its predecessor process is. This type of interface would also address the application side issues of not having a universal task ID.

5.1.5 Checkpointing

An important step towards enabling the future vision laid out in prior sections is adding the ability for Tock to checkpoint. This could be using any of the state of the art checkpointing technologies such as checkpoints assigned at compile time then altered at runtime. However, Tock could also manage the checkpoints with the ideas described in Section 5.2.

5.1.6 Memory Protection Unit

The memory protection unit was disabled for this work to allow processes to write to their output in the grant region within their own process memory. Other processes then have read access to this one area. For future iterations the memory protection unit will be enabled to ensure that processes cannot do anything malicious while accessing these areas.

5.2 Self-Assessment for Improved Performance

A larger scale project for Tock on energy harvesting devices could be a self-assessment mode for automomatic checkpointing and adaptive scheduling. Self-assessment would require the device to have some ability to measure the energy it takes to run certain tasks. The following could be

an example of this. iCount is a design that enables energy metering on battery-powered devices [58]. It uses the observation that the switching frequency of the PFM voltage regulator is linearly related to the load current. Adding a wire from the regulator to the microcontroller gives the platform the ability to use this information. Given that PFM regulators can be part of energy harvesting circuitry, it's possible to use a design like this with an application running on Tock that can provide some notion of the energy consumption of a process [7]. With this information, the device can make smarter decisions about task scheduling. Additionally, if Tock had the mechanism for checkpointing, it could use this information to insert checkpoints into execution where deemed necessary. This could be a feature that is useful for development or that can be run when the node is deployed to measure the metrics in a real-world environment especially with changing conditions or code updates.

5.3 New Domains for Tock

This section will explore some other thoughts related to Tock and possible future directions. One area of interest is security for the IoT. The design of Tock and usage of Rust can prevent malicious code from crashing the system. However, we would like to discuss the greater security of a deployment and the security and verification for data transactions. The inspiration for this came from numerous articles on blockchain being promising for IoT security. Centralized deployment models can be, for example, highly subject to DDoS attacks [59]. There are two angles from which Tock could be useful. Firstly, the need for security is additional evidence for needing multiprogramming on these devices necessitating Tock. If security protocols are wanted on a large portion of IoT devices and they are implemented at the application layer, it is likely that one could want the security features to be modular. Of course, this would run alongside whatever the device's operational code is to achieve it's needed function. Secondly, Tock could actually provide some security primitives. There is some precedent for offering security in an OS as seen with TLS (Transport Layer Security) components in certain real-time operating systems [60]. There is also work being done in lightweight blockchain for IoT [61]. Tock could have an API to assist with security related work.

This widespread interest in security is further indication that Tock will be useful for widespread IoT adoption.

Another consideration for Tock could be running on slightly more resource intensive devices which is contrary to it's original intention to run on quite constrained devices. There could be some use cases where one would like to deploy a gateway or device that isn't necessarily as memory or power constrained where running Tock may be useful. It may be that the device isn't as constrained but one may want to avoid the overhead of running something like Embedded Linux as it can be expensive [62]. A nice property of having more platforms adopt the same software system is the unified environment and portability of apps. It isn't clear how well Tock would scale up but it's another dimension in which to explore Tock's performance.

Chapter Six

Conclusion

In this work we have laid out a trajectory for embedded computing and two important upcoming areas that need focus. We look at these areas though Tock which provides a multiprogrammable secure environment to run applications — a property which can enable a grander vision of IoT. The software systems that support development on these platforms are important because they enable programmability and the needed abstractions for these radical device changes. The two areas we explored with Tock were RISC-V and intermittent computing. RISC-V is positioned to be a leading architecture for chips in the future and starting Tock's development for RISC-V has greatly accelerated progress for Tock.

On the energy harvesting side, it is clear to us that intermittent systems are becoming more general-purpose and, if not now, will be in need of an operating system soon. We cannot expect to deploy billions or trillions of sensors with software that is tightly coupled with the hardware and expect that they will operate optimally for a lifetime. We set up our vision in this work and used Tock as an avenue to start adapting to this change that is inevitable. As all computing paradigms before now, intermittent devices that have the capability will require an operating system to fulfill their potential. Though this particular iteration may not be ready to run on an extremely constrained device, we felt it important to take the plunge and see what we can do with a full operating system. This work should be continued in a manner that makes an operating system work for intermittent computing by making the best of the latest research. However, it needs to do this in tandem with maintaining the principled abstractions provided by an operating system.

Finally, to answer the question that was posed as the title of this thesis - is this general purpose operating system really general purpose? It seems that Tock has a basic design that supports general purpose computing but needs further iterations on how to best support the most prevalent architectures and platforms.

REFERENCES

[1] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—a survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802–1831, 2017.

[2] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, "The role of edge computing in internet of things," *IEEE communications magazine*, vol. 56, no. 11, pp. 110–115, 2018.

[3] A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: Software challenges in the iot era," *IEEE Software*, vol. 34, no. 1, pp. 72–80, 2017.

[4] (2020). Tock embedded operating system, [Online]. Available: https://www.tockos.org/.

[5] (2020). Members at a glance, [Online]. Available: https://riscv.org/members-at-a-glance/.

[6] J. Paulo and P. D. Gaspar, "Review and future trend of energy harvesting methods for portable medical devices," in *Proceedings of the world congress on engineering*, WCE, vol. 2, 2010, pp. 168–196.

[7] S. Sudevalayam and P. Kulkarni, "Energy harvesting sensor nodes: Survey and implications," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 3, pp. 443–461, 2010.

[8] W. S. Wang, T. O'Donnell, L. Ribetto, B. O'Flynn, M. Hayes, and C. O'Mathuna, "Energy harvesting embedded wireless sensor system for building environment applications," in *2009 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology*, IEEE, 2009, pp. 36–41.

[9] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–6.

[10] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–13.

[11] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 159–170.

[12] N. A. Bhatti and L. Mottola, "Harvos: Efficient code instrumentation for transiently-powered embedded sensing," in *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, IEEE, 2017, pp. 209–220.

[13] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 85–99.

[14] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an rfid-based battery-free programmable sensing platform," *IEEE transactions on instrumentation and measurement*, vol. 57, no. 11, pp. 2608–2615, 2008.

[15] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 234–251.

[16] (2020). Tockloader, [Online]. Available: https://github.com/tock/tockloader.

[17] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[18] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of risc-v isa compliance," in *Design, Automation and Test in Europe*, 2020.

[19] A. Bradbury. (2019). The future of operating systems on risc-v, [Online]. Available: https://www.youtube.com/watch?v=emnN9p4vhzk.

[20] K. McDermott. (2020). Getting started with risc-v verification, [Online]. Available: https://riscv.org/2020/05/getting-started-with-risc-v-verification/.

[21] D. Patterson, "50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, IEEE, 2018, pp. 27–31.

[22] (2020). The beauty of risc-v is really the flexibility and the openness, [Online]. Available: https://www.youtube.com/watch?v=kBCm_sImzs4.

[23] S. Kim, R. Vyas, J. Bito, K. Niotaki, A. Collado, A. Georgiadis, and M. M. Tentzeris, "Ambient rf energy-harvesting technologies for self-sustainable standalone wireless sensor platforms," *Proceedings of the IEEE*, vol. 102, no. 11, pp. 1649–1666, 2014.

[24] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1005–1021.

[25] ——, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 129–144.

[26] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 17–32.

[27] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *arXiv preprint arXiv:1909.06951*, 2019.

[28] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, 2018, pp. 41–53.

[29] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Dynamic task-based intermittent execution for energy-harvesting devices," *ACM Transactions on Sensor Networks (TOSN)*, vol. 16, no. 1, pp. 1–24, 2020.

[30] S. DeBruin, B. Campbell, and P. Dutta, "Monjolo: An energy-harvesting energy meter architecture," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, 2013, pp. 1–14.

[31] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, 2014.

[32] H. Jayakumar, A. Raha, and V. Raghunathan, "Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, IEEE, 2014, pp. 330–335.

[33] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.

[34] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 514–530.

[35] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1085–1100.

[36] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1101–1116.

[37] (2020). Wisp5, [Online]. Available: http://wisp5.wispsensor.net/.

[38] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet-of-things," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–13.

[39] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 767–781.

[40] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71 907–71 920, 2019.

[41] (). Ota firmware updates, [Online]. Available: https://docs.particle.io/tutorials/device-cloud/ota-updates/.

[42] (2020). Freertos, [Online]. Available: https://www.freertos.org/.

[43] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[44] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *29th annual IEEE international conference on local computer networks*, IEEE, 2004, pp. 455–462.

[45] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*, Springer, 2005, pp. 115–148.

[46] (2019). Svc, [Online]. Available: http://www.keil.com/support/man/docs/armasm/armasm_dom1361289909139.htm.

[47] A. Heinrich. (2016). Context switch on the arm cortex-m0, [Online]. Available: https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/.

[48] (). Hifive1, [Online]. Available: https://www.sifive.com/boards/hifive1.

[49] (). E21, [Online]. Available: https://www.sifive.com/cores/e21.

[50] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume ii: Privileged architecture version 1.9," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.

[51] E. Bursztein. (). Say hello to opensk: A fully open-source security key implementation, [Online]. Available: https://security.googleblog.com/2020/01/say-hello-to-opensk-fully-open-source.html.

[52] (). Opentitan tock working group (ot), [Online]. Available: https://github.com/tock/tock/tree/master/doc/wg/opentitan.

[53] J. Ahmad, H. Larijani, R. Emmanuel, M. Mannion, and A. Javed, "Occupancy detection in non-residential buildings–a survey and novel privacy preserved occupancy monitoring solution," *Applied Computing and Informatics*, 2018.

[54] J. Wang, H. Jiang, J. Xiong, K. Jamieson, X. Chen, D. Fang, and B. Xie, "Lifs: Low human-effort, device-free localization with fine-grained subcarrier information," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016, pp. 243–256.

[55] M. F. R. M. Billah and B. Campbell, "Unobtrusive occupancy detection with fastgrnn on resource-constrained ble devices," in *Proceedings of the 1st ACM International Workshop on Device-Free Human Sensing*, ser. DFHS'19, New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–5, ISBN: 9781450370073. [Online]. Available: https://doi.org/10.1145/3360773.3360874.

[56] (2020). Q-loud energycam, [Online]. Available: https://www.q-loud.de/energycam-automatic-mechanical-meter-reading.

[57] (2020). Imix, [Online]. Available: https://github.com/helena-project/imix.

[58] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy metering for free: Augmenting switching regulators for real-time monitoring," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, IEEE, 2008, pp. 283–294.

[59] U. Javaid, A. K. Siang, M. N. Aman, and B. Sikdar, "Mitigating iot device based ddos attacks using blockchain," in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, 2018, pp. 71–76.

[60] (). List of open source real-time operating systems, [Online]. Available: https://www.osrtos.com/.

[61] A. Dorri, S. S. Kanhere, and R. Jurdak, "Towards an optimized blockchain for iot," in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, IEEE, 2017, pp. 173–178.

[62] (). Migrating vxworks vxworks applications applications to linux to linux, [Online]. Available: http://www.mvista.com/download/Migrating-VxWorks-apps-to-embedded-Linux-slides-with-notes.pdf.