Published: 15 May 2024

RESEARCH-ARTICLE

User-directed Assembly Code Transformations Enabling Efficient Batteryless Arduino Applications

CHRISTOPHER KRAEMER, Georgia Institute of Technology, Atlanta, GA, United States

WILLIAM GELDER, Georgia Institute of Technology, Atlanta, GA, United States

JOSIAH HESTER, Georgia Institute of Technology, Atlanta, GA, United States

https://doi.org/10.1145/3659590
EISSN: 2474-9567

User-directed Assembly Code Transformations Enabling Efficient Batteryless Arduino Applications

CHRISTOPHER KRAEMER, Georgia Institute of Technology, USA
WILLIAM GELDER, Georgia Institute of Technology, USA
JOSIAH HESTER, Georgia Institute of Technology, USA

The time for battery-free computing is now. Lithium mining depletes and pollutes local water supplies and dead batteries in landfills leak toxic metals into the ground[20][12]. Battery-free devices represent a probable future for sustainable ubiquitous computing and we will need many more new devices and programmers to bring that future into reality. Yet, energy harvesting and battery-free devices that frequently fail are challenging to program. The maker movement has organically developed a considerable variety of platforms to prototype and program ubiquitous sensing and computing devices, but only a few have been modified to be usable with energy harvesting and to hide those pesky power failures that are the norm from variable energy availability (platforms like Microsoft's Makecode and AdaFruit's CircuitPython). Many platforms, especially Arduino (the first and most famous maker platform), do not support energy harvesting devices and intermittent computing. To bridge this gap and lay a strong foundation for potential new platforms for maker programming, we build a tool called BOOTHAMMER: a lightweight assembly re-writer for ARM Thumb. BOOTHAMMER analyzes and rewrites the low-level assembly to insert careful checkpoint and restore operations to enable programs to persist through power failures. The approach is easily insertable in existing toolchains and is general-purpose enough to be resilient to future platforms and devices/chipsets. We close the loop with the user by designing a small set of program annotations that any maker coder can use to provide extra information to this low-level tool that will significantly increase checkpoint efficiency and resolution. These optional extensions represent a way to include the user in decision-making about energy harvesting while ensuring the tool supports existing platforms. We conduct an extensive evaluation using various program benchmarks with Arduino as our chosen evaluation platform. We also demonstrate the usability of this approach by evaluating BOOTHAMMER with a user study and show that makers feel very confident in their ability to write intermittent computing programs using this tool. With this new tool, we enable maker hardware and software for sustainable, energy-harvesting-based computing for all.

CCS Concepts: • **Human-centered computing**; • **Hardware → Renewable energy**; • **Computer systems organization → Embedded systems**;

Additional Key Words and Phrases: Energy Harvesting, Intermittent Computing, Battery-free, Block based programming

ACM Reference Format:
Christopher Kraemer, William Gelder, and Josiah Hester. 2024. User-directed Assembly Code Transformations Enabling Efficient Batteryless Arduino Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 8, 2, Article 44 (June 2024), 32 pages. https://doi.org/10.1145/3659590

1 INTRODUCTION

Maker programming platforms and hardware ecosystems like Arduino, CircuitPython, and Makecode, have revolutionized access to physical computing in the last two decades, bringing millions of non-programmers and hobbyists into the fold of programming and technical development. These platforms are formative for many,

Authors' Contact Information: Christopher Kraemer, ckraemer7@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA; William Gelder, wgelder3@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA; Josiah Hester, josiah@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA.

ACM 2474-9567/2024/6-ART44
https://doi.org/10.1145/3659590

providing new applications and usages of computing and sensing, intersecting with engineering disciplines, but also creative and artistic expression. These platforms have at this point been long standby ecosystems for educational curriculum at all levels, from K-12 to college and post-graduate. The ability to make and automate something is a powerful thing.

These platforms and the devices they use are all held back by batteries. In order to truly develop a more sustainable future for computing, we must take firm stand against the ways we as computer scientists, makers, and researchers negatively impact the environment. While it can certainly be said that the use of rechargeable batteries is preferable to fossil fuel use in general, we need to make sure that the positives are not outweighed by the harmful effects caused by the creation and disposal of batteries. The negative impacts of Lithium mining in Chilé, a country with one of the worlds largest natural lithium supplies, include using 65% of the Salar de Atacama region's water. This is an area "where annual rainfall is less than 15 millimetres per year, the activity depletes already scarce water resources that local communities and species depend on."[12] Worse yet, it does not stop there. When a battery has reached the end of its lifespan, they are often dumped into landfils and not recycled. While in a landfill, batteries will leach toxic metals into the ground[20]. The purpose of this paper is not to dissuade battery use entirely. We claim that programming platforms like Arduino, CircuitPython, and MakeCode represent fertile ground for batteryless systems.

We are not the first to recognize that makers and hobbyists are well-suited to develop battery-less devices. Works such as BFree and Battery-Free MakeCode have come to the same realization and developed tools to expand access to this otherwise niche realm of computing[22][25]. Additionally, and as pointed out in Battery-Free MakeCode, different battery-free deployments have been made and established a precedent that battery-free computing is not as limiting as one might assume. There exists battery-free devices above the stratosphere [15] to under the waves [21], from Game Boys [16] to neural networks [26], and from health [34] to mobile phones [38][25]. All of these examples go to show that fully-fledged, battery-free deployments are possible and within the reach of hobbyists and makers if they only had the know-how to program an intermittent device.

Unfortunately, programming an intermitent device is not without its significant challenges. For example, Figure 1 shows how power failures from dynamic energy harvesting cause an Arduino program to never make progress, since after sensing temperature, it will continue to lose power and restart from the top of the `loop()` function. While previous work made significant inroads bridging the gap between novice users and energy harvesting, batteryless devices via power failure resilient Python based [22] and Makecode blocks based programming paradigms [25], these solutions missed two critical points to improve the impact and ubiquity of non-expert, maker focused programming:

(1) **Makers are not a monolith.** While BFree and Battery-free Makecode targeted two very broadly used platforms, the incredible diversity of hardware and programming techniques used still leaves many behind, and the techniques, while applicable to many or most hardware platforms and device ecosystems, are incomplete. For example, they are not technically able to make the single most popular and longest standing platform, Arduino, power failure resilient.
(2) **Makers can handle the truth of intermittency.** The other potentially missing opportunity in past work, is that power failures are *hidden*. Why hide power failures completely from makers? If the programmer / maker could control the rate, or timing of intermittency, or even inform the power failure resilient runtime of the things that matter to them in the code, potentially the entire system could run more efficiently and more in line with programmer expectations and desires.

Contributions

In this paper, we explore and develop a two part approach to these missed opportunities, with the intent of developing a set of tools that can allow for *any* maker platform to be converted and used with energy harvesting,

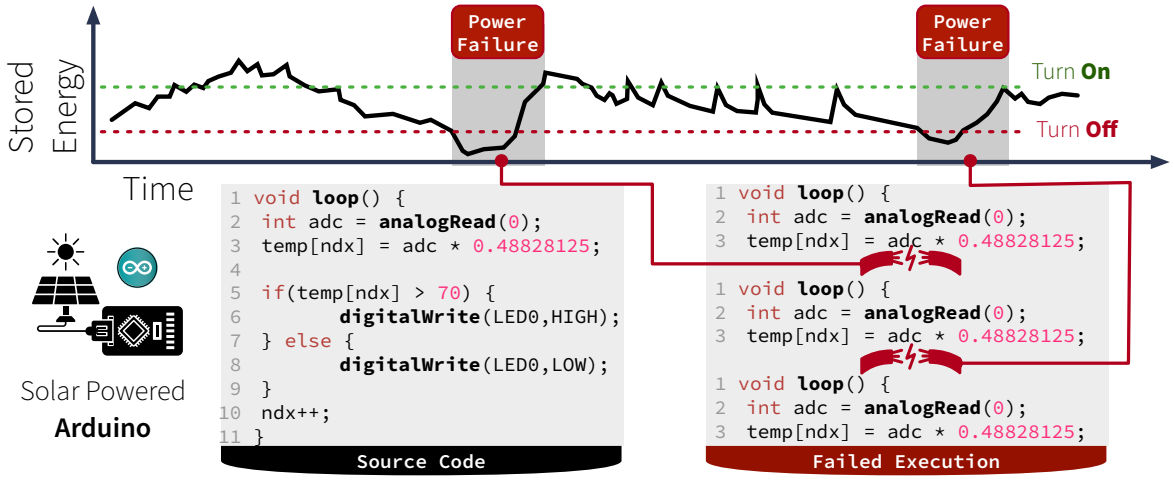


Fig. 1. This illustrates how variable energy affects the execution of a simple temperature-sensing Arduino program. Normally, the program will read a sensor value, convert to Celsius and store it, and then determine if the reading is dangerously high. With energy harvesting via solar power, this Arduino (in this example) will never get through the `loop()` function. Power failures (which can occur at any time) cause it to start from scratch, meaning no progress is made.

battery-free, and sustainable methods. Our goal is to further democratize access to energy harvesting embedded computing devices, by facilitating mechanisms to perpetuate application code through power failures, with user input.

Previous work in expanding access to intermittent computing involved the use of a virtual machine or an online transpiler, such as in BFree[22] and Battery-free MakeCode[25] respectively. Our approach leverages low-level binary rewriting for instrumenting intermittent code. This allows our tool to fit into existing Makefile-based build systems such as the Arduino platform. Additionally, the techniques are generalized enough to be usable (with some engineering effort) in other emerging platforms, as long as the flow from user code to final binary, can be interrupted and transformed by our tool. We name this tool and its components BOOTHAMMER, as a callback to Bootloader type functions on embedded systems (such asd the Microsoft UF2 bootloader, which is ubiquitous across ARM based maker platforms). Bootloaders intercept the operation of a user program, providing a container (of sorts) for a user program to run on, while providing low level access to the hardware itself, to simplify the interface between (for example) the USB system of a host computer, and the microcontrollers internals. BOOTHAMMER works similarly in that it analyzes the generated Assembly of a program, and inserts checkpoint Assembly commands in critical places (for example: at the end of the *loop*() function in an Arduino program), and restore Assembly code at the start of *setup* that restores the memory of the device from before the last power failure, allowing for continued execution at the exact state before failure, instead of redoing all computations and sensing.

Finally, we combine this low-level analysis and transformation approach, with a new insight into user guided intermittent computing. We enable users to annotate points in the code where they would recommend inserting a checkpoint— this could be after a large for loop that captures a number of sensing readings, where the user may know that unless they capture 100 readings, there is no point in checkpointing *in between each sampling operation.* In this way, a user would save a tremendous amount of overhead by letting the checkpointing system know it can skip checkpoints at certain times, reducing the time and memory overhead. These user guided instrumentation

passes allow for the user to finally have some insight and control of how the device handles intermittent power, but while not putting an excessive burden on the user for designing a checkpoint and restore system from scratch.

In summary, our specific contributions include:

(1) We design and develop a tool, BOOTHAMMER, that enables a suite of analysis and transformation for low-level code from the Arduino IDE maker platform, these tools allow for checkpoint / restore mechanisms with very minimal changes to user code and with minimal memory impact and runtime overhead comparable to contemporaries in the field.
(2) We present an exploration and implementation of a new programming paradigm for intermittent computing for makers, where programmer annotations guide transformations and placement of checkpoints, for higher performance and checkpoint resolution.
(3) We conduct a demonstration and evaluation campaign via implementing the tools with Arduino, the first time intermittent computing has been demonstrated with Arduino, the most popular and longest-standing maker ecosystem
(4) We present an evaluation of the performance and correctness of BOOTHAMMER and conduct a user study to evaluate its usability.

We will release all code artifacts as open source via GitHub. We intend that BOOTHAMMER will be used so that everyone should have the opportunity to participate in the nascent trillion-device battery-free Internet-of-Things, and that sustainable practices in computing should eventually lead to positive effects on climate change and reduction of electronic waste.

2 BACKGROUND AND RELATED WORK

This work sits at the intersection of maker programming and development tools, or Integrated Development Environments (IDEs), sustainable computing, and intermittent computing. We detail related work below, in these areas, as well as compare and contrast the two most related works BFree [22], a toolchain for programming energy harvesting devices with Python, and Battery-free Makecode [25], a set of compiler extensions to allow for the online visual blocks based programming tool Makecode [17] to program energy harvesting devices.

Much like these other works, our goal is to expand the ability of makers to access and use more sustainable, long-lived energy harvesting computing devices. With more than one trillion devices likely to be deployed by 2035 [7], powered by rechargeable LiPo batteries, many have raised concerns about the ecological and toxic impacts of e-Waste from this new Internet-of-Things [32, 37]. Batteries degrade every recharge/power cycle, with temperature and other factors hastening the end of life, when disposed of, they are toxic [37], highly flammable, and hard to recycle— with less than 1% of lithium in products was actually recycled as of 2021 [8]. These factors result in rising global e-Waste and hazards concerns. While the maker community and any single research project cannot change this global problem, we believe a paradigm shift in embedded computing focused on sustainability is on the horizon.

In the rest of this section, we briefly review computer systems, architecture, and programming languages approaches to intermittent computing (Section 2.1) maker platforms (Section 2.2), how previous work has closed the gap in making these platforms power failure resilient to enable energy harvesting and battery-free operation (Section 2.3), and finally we address the motivation behind BOOTHAMMER as a binary rewriting tool (Section 2.4) and identify gaps and place our work within this literature (Section 2.5).

2.1 Intermittent Computing for Batteryless Expert Built Systems

When one removes the battery from an embedded device and replaces it with a solar panel (or other harvester), power failures become a significant issue. Since ambient energy is scarce, the device accumulates it in a capacitor.

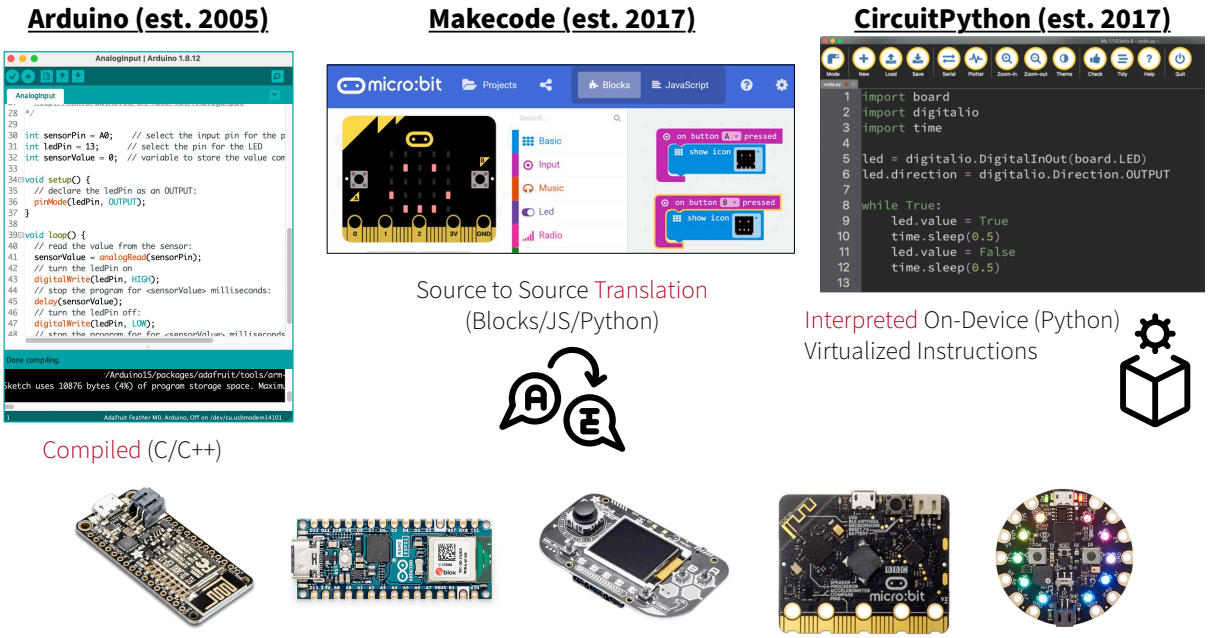


Fig. 2. Three of the more popular Integrated Development Environments (IDEs) for makers. Arduino (the oldest and by far the largest community), Makecode (preferred for education), and CircuitPython (higher powered MCUs allow for interpreted code on-device). All are compatible with a broad array of platforms from Feathers, to PyGamer, Micro:bit and many others. Each IDE offers advantages, yet the most popular, Arduino, is not yet accessible for programming energy harvesting and batteryless devices due to challenges with.

As the capacitor becomes fully charged, the device starts executing the program to finish execution before the capacitor is fully discharged, otherwise, it will lose all of its progress and have to start over.

This fundamental problem of power failures has caused significant lines of research in "Intermittent computing" [18] which refers to the collection of software/hardware/toolchain techniques that enable seamless, consistent, and efficient recovery from power failures due to unpredictable energy harvesting. The most common approach to solve this problem is by implementing a *checkpoint-based programming* model and runtime system [2, 14, 23, 24, 27, 33, 40]. This approach saves checkpoints of program state as the program executes, and restores from the most recent checkpoint after a power failure. Nearly all approaches instrument bare C/C++ at compile-time, often using complex analyses via LLVM and other tools, which are not generally usable by makers and novices. Other approaches monitor supply voltage and save state just-in-time before a power failure [10, 11]. All of these approaches must figure out which points in the code to checkpoint, when to checkpoint, and what to checkpoint, each highly challenging problems that have streams of dedicated work. These approaches require little to no intervention from the programmer— therefore reducing the effort of porting existing code— but possibly, this level of hiding and abstraction of power failures is a mistake, and programmers should understand the notions of intermittent power and energy harvesting, so they can design specifically for them. At a low level, all these approaches suffer from a dependence on highly complex open source toolchains, with many works referenced requiring specific branches or even nightly builds of tools like LLVM, combined in challenging recipes

with other work. This level of expertise to implement these approaches in maker platforms has stymied efforts to migrate them to more novice focused realms.

2.2 General Purpose Maker Ecosystems

The past few decades have seen an explosion in maker ecosystems as cheap microcontrollers and open source hardware/software have become available. Some of the most popular Integrated Development Environments (IDEs) are shown in Figure 2, as well as a few of the popular modern microcontroller platforms that the IDEs target. Arduino is the most well known and oldest ecosystem, including an IDE, tens or hundreds of compatible microcontroller boards, and thousands of libraries to ease the integration of sensors, actuators, and other components. Makecode, CircuitPython, and other ecosystems have emerged since, with varying approaches and population targets, allowing a variety of ways to program and develop with physical computing devices. Each represents a different style in terms of the actual compilation architecture, which is important context for our contribution:

Compiled: Arduino is essentially a thin layer of UI and well designed libraries, and a build tool wrapper around a traditional C/C++ compilation toolchain. In fact, Arduino is the closest to the "bare metal" of the embedded microcontroller, allowing for direct register access, memory management, and inline assembly, with user code being nearly untouched from the IDE to the compiler. This means that performance of Arduino can be fairly high, but compilations may be long, and the C/C++/Java programming style is not regarded as the most accessible programming environment.

Interpreted: CircuitPython and other Pythonic platforms take a different approach, hosting an entire Python bytecode interpreter on board the microcontroller. This interprets and executes the higher level Python language. This is only possible due to speedy 32-bit processors (almost always ARM Cortex-Ms), and large onboard memories now cheaply available. In this scheme, each instruction of the users code is virtualized onto the hardware (i.e., C/C++ operations). This creates a layer of abstraction which brings modern coding abilities and simpler syntax, but at the cost of a high overhead in performance.

Source-to-Source Translation: Finally, Makecode [9], and other web tools like Teknikio/Node Red, offer block-based, event-driven, visual online web applications that allow for programming USB connected devices via WebUSB. These schemes conduct source to source compilation, converting blocks and events or nodes, into C/C++ routines, after parsing/tokenizing the user programs into their abstract syntax tree or some other intermediate representation.

Each of these ecosystems have there own specific hardware, but are intercompatible to varying degrees as most rely on modern ARM Cortex-M0 or Cortex-M4 microcontrollers. Sensor libraries, and specific hardware submodules may vary.

2.3 Accessible Maker Programming for Energy Harvesting Systems

Recent work in Ubiquitous Computing has explored how to leverage these ecosystems for sustainable computing, especially building and programming embedded systems that are powered exclusively by energy harvesting. Batteryless systems fail intermittently due to variable power from energy harvesting, as shown in Figure 1 with an example Arduino code that samples a temperature sensor and turns on an LED if it is hotter than 70 degrees. Energy harvesters like solar panels, will deliver variable power depending on conditions— if a cloud, or shadow goes across the harvester, output drops dramatically, which means stored energy is quickly used up and the device will likely have a power failure soon after. For the source code under execution, this practically means that programs like those shown in Figure 1 may never complete, as the program goes back to the start of `main()` (or the start of `loop()` in Arduino) after each failure.

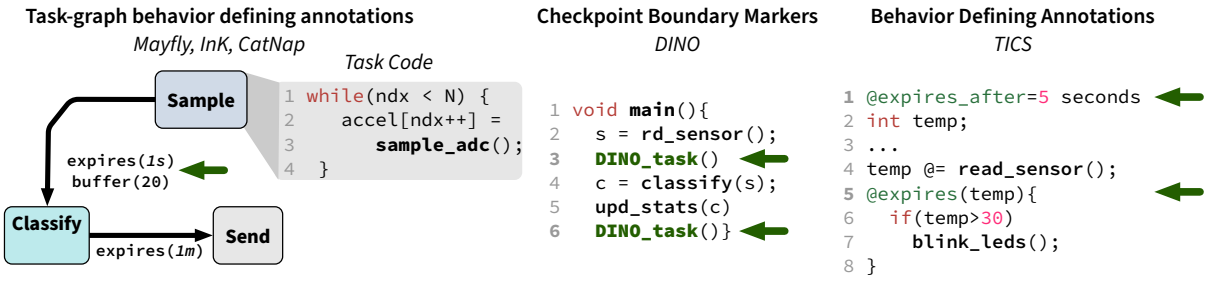


Fig. 3. This figure shows different methods State of the Art intermittent computing systems have integrated user directions into there execution. These works recognize that energy harvesting and batteryless development has different constraints vs. traditional systems, and allow for annotations that specify tasks and timing for simplified reasoning (left), checkpoint boundaries to reduce overheads and target important parts of a program (middle) and timing annotations that direct the program on the longevity of certain data captured by the program (right). These methods are all inaccessible to the maker ecosystems, this work explores bringing user direction into energy harvesting maker ecosystems.

Recent works have made systems to checkpoint state before power failures, so that the device can compute *intermittently*, stringing together execution fragments instead of restarting from scratch every failure. We contrast the two most recent systems below:

The first approach is BFree [22], which provides intermittent computing compatibility for 32-bit devices by porting CircuitPython (a embedded systems variant of Python, supported and built by maker company AdaFruit) to use with energy harvesting. This work went the *interpreted language* route. This approach greatly increased the potential users of intermittent computing and energy harvesting devices, but with some downsides. Interpreted languages are slow: code executes in the runtime and not directly on the device. An interpreter requires significant memory, reducing memory available to user code. Finally, BFree specifically required a user to install a fork of the CircuitPython runtime on the device, and use a custom hardware device not available from online sources(one must make it themselves) to provide non-volatile checkpoint storage. These two requirements decrease the accessibility of the system. In contrast, the second system, Battery-free MakeCode, does not have these issues despite offering a higher level of abstraction due to its source-to-source compilation.

Battery-free Makecode [25] was the first *source-to-source compiler for batteryless systems*. This category of intermittent computing system marries ease-of-programming offered by MakeCode with the capability of performing long-term deployments with battery-less systems while hiding the complexities and challenges of performing computations intermittently. Makecode was unique in making programming accessible online, and therefore Battery-free Makecode made, for the first time, intermittent computing accessible via online web tools, dramatically reducing burden for energy harvesting device programming, even for K-12 students.

Both Battery-free Makecode and BFree had very poor performance compared to state of the art systems, and argued (as we do) that the ease-of-programming and flexibility in use are worth the loss of performance in many contexts, including maker, student, and novice contexts where energy, latency, and performance are not as critical[25][22]. However, it is not clear that these approaches are the best path forward since they all completely mask the effect of intermittency on the device, removing the ability of the programmer to handle and control how a program addresses power failures.

2.4 UF2 Bootloader: Prime Target for Democratizing Access to Intermittent Computing

BOOTHAMMER, as the name implies, is a brute-force approach to a mass democratization of intermittent computing. The UF2 bootloader is used widely across consumer-level and off-the-shelf hardware. Additionally, it works with the Arduino IDE, making it a suitable modification point for expanding intermittent computing to makers, hobbyists, and researchers. At a glance, the UF2 bootloader is the first piece of code to run when the device turns on and allows the user to write/flash the device without the use of a debugger like J-Link, the bootloader has highly privileged access to the device as whole, making it a great target for instrumenting intermittent computing. For example, as the first piece of code to run, less time and energy is wasted checking if there is enough energy to perform some execution in the first place, assuming this approach is using a capacitor bank to buffer stored energy. Additionally, the bootloader has privileged access to things that a user program would not otherwise have access to, including but not limited to its region of RAM/flash, control registers, and the thing BOOTHAMMER is most interested in: the incoming binary. Assembly-level modification of a program is advantageous to the intermittent computing programmer because it provides a level of control over a device(RAM, SP, PC) with very low overhead. This results in less time and energy wasted on performing the checkpoint and more time available for meaningful computation. Unfortunately, the bootloader only "sees" a hex file, so some disassembly of that hex file is required to decipher a user's program.

2.5 Summary of Gaps

Beyond those works mentioned above, another set of work informs BOOTHAMMER in concept, *software systems with manual code adaptation*. These systems for intermittent execution ask the developer to make manual annotations and transformations to their code, either using a special API within the C/C++ program, or providing a map of tasks and requirements on the edges between tasks. This can be quite useful for developers and make for more performant, efficient systems, as these annotations can allow for a programmer to specify a energy conditional, such that a block of code is executed only when energy allows [41], can annotate timing requirements on data, such as an expiration date such that data must be regathered if a power failure is too long [19], and many other functions that are exclusive to intermittent computing. Some works like DINO [27] and TICS [24] allow for manual annotations to either inform checkpoint placement based on programmer knowledge in the former, or taking into account timing requirements for sensor data in the latter. Another work, CatNap [29], combines task graphs and timing annotations with events to allow for adaptive execution. The different types of annotation based ideas are shown in Figure 3. Unfortunately, as pointed out in TICS[24], simple programs with simple bugs were much harder to find and correct when a program was manually annotated or transformed into a task-based system for intermittent computing. Additionally, manual transformation of code typically requires learning a new language and/or some prior education on intermittent computing and energy harvesting[23]. However, manual annotations are a useful direction to allow for the user to have more control over the execution of their program across power failures.

We identify two main gaps from this related work: (1) no low-level compiled approaches to maker platforms, and (2) no ability for maker programmers to control or design for intermittency.

Python in microcontrollers, and MakeCode visual blocks programming represent a fraction of the systems used by makers to build. Arduino and other lower-level compiled ecosystems represent one of the last major places where the battery-free embedded devices have not been formally explored. Providing a way to seamlessly checkpoint programs written with Arduino, or that run on the UF2 bootloader, would make it such that nearly every maker hardware would be compatible with intermittent computing. The challenge is making this happen with existing tools already present in the platform, and performing transformations at a low enough level to be speedy and quick to compile.

Additionally, previous works have shown the importance and promise of batteryless, energy harvesting systems, and have demonstrated expert focused systems that range the gamut from automatic checkpointing (hiding all intermittency) to programmer guided intermittency control. Maker focused platforms (BFree and Battery-free Makecode) have demonstrated the potential of broadening access to batteryless devices by transforming, retrofitting, existing maker ecosystems— however, unlike the former expert focused systems, currently no platform has explored how increasing programmer involvement in intermittency, for a maker/novice instead of an expert, might bring more understandable, performant, and useful programs, especially how this annotation might speed up maker driven code as current systems are unwieldy and slow.

This paper seeks to close the gap for maker ecosystems and batteryless systems (like Arduino) by offering extensible, general purpose low-level transformations for checkpointing. This paper also closes the gap between manual annotations and ease of use for intermittently powered, energy harvesting, batteryless devices built by makers, to enable a way for maker systems to get better performance despite the overhead of checkpointing through power failures.

3 SYSTEM DESIGN

The design idea at the center of BOOTHAMMER is on-chip binary re-writing that automatically provides functionally correct execution across intermittent power failures. Additionally, this tool allows the user to self-identify and place restore and checkpoint locations within their own code to facilitate faster execution and make use of the programmer's unique high-level insights into the operation of their code, which are opaque to a low-level binary rewriting scheme. This strikes the balance of making a tool for completely inexperienced users as well as teaching users about intermittent computing by allowing them to define where in their code they want to checkpoint, and iteratively test various checkpoints for their own application goals, as well as performance. The high level design concept is shown in Figure 4. BOOTHAMMER is made for the Arduino build system and programming ecosystem and is centered around the following design goals:

(1) **Intermittent Execution:** BOOTHAMMER's modifications to the user's code will preserve correctness and allow the program to operate under intermittent power: essentially, a intermittently powered version will behave the same as a continuously powered version.
(2) **Removing the Barrier:** BOOTHAMMER will abstract away the layers of software needed to run a program under intermittent conditions. The user will only need to do the bear minimum to begin using this tool.
(3) **Small Footprint:** BOOTHAMMER will maintain a small enough footprint on the user and user's device to be not only usable but useful. This includes but is not limited to a low memory footprint on the device and fast code modification. The tool should be small enough, fast enough, and lightweight enough to run within the UF2 bootloader that is widely used among Arduino-compatible boards.
(4) **User Guided Checkpoints:** BOOTHAMMER provides no-effort checkpointing for the user as well as source-level markers for the user to choose where to place the restore and checkpoint operations. This provides the user with a starting point to begin writing intermittent programs as well as the ability to teach themselves about where and when to checkpoint.

In the rest of this section, we detail the pieces of the compilation pipeline and how BOOTHAMMER fits into and modifies it as shown in Figure 4.

3.1 Dissassembler-Reassembler Pipeline

Whether a user is flashing a binary, hex, or elf file, the UF2 bootloader is only aware of an incoming hex stream that is saved to flash memory. In this state, no context is available to determine how a program will run and what it will do. In order to uncover, at least partially, how a program runs, BOOTHAMMER performs a disassembly of the incoming hex file in a manner similar to `objdump`. The main challenge of this approach is that no function

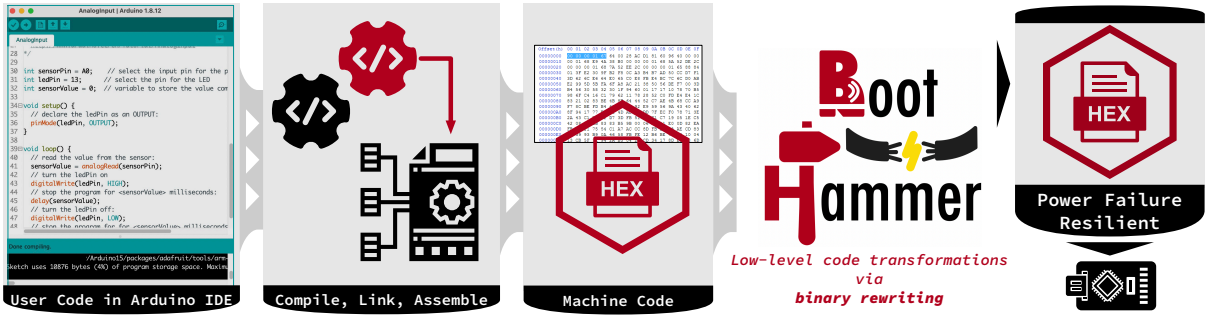


Fig. 4. Shown is the BOOTHAMMER system and where it sits in between the compilation process of Arduino (or other maker IDEs). Arduino hosts an internal GCC compiler with traditional `Makefile`, allowing us to intersept the code before it hits the device, and conduct binary rewriting that is eventually emitted as a power failure resilient ARM binary file that is flashed onto a microcontroller. An FRAM breakout from AdaFruit is used as checkpoint storage.

labels are available in the hex file. Performing a disassembly `objdump` on an elf file reveals sections of assembly code with function labels for each section. No such information is available in the hex file, but this system still needs to know where `main`, `setup`, `loop`, and other necessary functions are. The Arduino IDE guarantees that `setup` and `loop` functions must be present in a user's program, else the compiler will return an undefined reference error and compilation will be terminated[1]. This ensures that all user code is not only reachable from `setup` and `loop`, but that `main` must contain calls to these functions. This establishes a basis for finding `main` in a disassembly. We can build a "signature" for `main` based off of the Arduino-IDE/compiler guarantees about the instructions that must be present in `main`. This means a call to `setup` must be present, a call to `loop` must be present, `setup` must come before `loop`, and there must be a branch back to `loop` so that this function runs over and over again. This knowledge, paired with an extensive `objdump` disassembly of every built-in example program available in the Arduino IDE for the Adafruit Feather M0 Express, reveals that `main` contains the exact same instructions in the exact same order for a given board. Addresses used in an instruction can and will be different between different compilations, but the instruction names themselves are constant for a given signature. However, signatures are not constant between manufactures of Arduino-IDE compatible boards of the same processor(Cortex M0+/ATSAMD21G18A), but they are constant across programs within a manufacturer's board. Put more concisely, Adafruit and Sparkfun have different signatures for `main`.

These signatures for `main` create a predicable way for BOOTHAMMER to recover information and structure from the original program that is not immediately present in a disassembly output. Due to the fact that the Arduino-IDE is ever-changing, we limit the scope of finding a signature for `main` to version 2.2.1 of the Arduino-IDE and for Cortex M0+ boards(ATSAMD21G18A) made by Adafruit, Sparkfun, and Arduino. With all of this said, we do not guarantee that BOOTHAMMER is able to find `main` using this signature matching scheme for all possible Cortex M0+ boards, but we do claim that this scheme is sufficient in most cases. In the unlikely event that `main` is not found, BOOTHAMMER will terminate and not output a modified hex file. With this predictable code output, BOOTHAMMER can easily find `main` based on its signature and uncover the rest of the program from the disassembly. Based on the guarantee that `setup` and `loop` must exist in `main`[1], we can infer that all user code is reachable from `setup` and `loop`, so performing a depth-first search of functions within these functions by following each call will build a tree of functions of the user's code. Once BOOTHAMMER has disassembled the user's program, found `main`, and created a graph of the functions that are reachable from `main`, the tool begins to instrument the code for checkpointing.

The biggest challenge with rewriting a binary/hex file is that linking has already happened and addresses for each instruction have already been assigned. If BOOTHAMMER were to insert an instruction somewhere in the program, nearly all of the addresses of the program will need to be reassigned. Stores, Loads, Branches, and more may now all be offset by the size of the inserted instruction. In order to deal with this, all existing instructions need to be modified in place and the necessary checkpointing code will be added to the end of the program. So long as the addresses within this added section are sane, there is no need for a re-linking step. In order to connect the user's program to the checkpointing code at the end of the program, BOOTHAMMER modifies existing branches. In its most basic form, BOOTHAMMER will instrument a user's code so that a checkpoint occurs once every iteration of the `loop` Arduino function. The branch that would normally point to `loop` is overwritten with a branch to a helper function that calls two other functions: `loop` itself and checkpoint. Once the checkpoint is complete, the helper function returns to `main` and the program runs as normal in an infinite loop. The reassembly step is relatively simple once all of the instrumentation is done. No re-linking is required and BOOTHAMMER saves the hex opcodes of all disassembled and inserted instructions, so the re-assembly step is really an output of another hex file.

3.2 User Instrumentation

An integral part of democratizing access to intermittent computing is giving users the ability to teach themselves about how an intermittent program works. Previous tools like MakeCode-Iceberg[31] provided intermittent computing to the user on their behalf, but did not give the user much control over where and how the program would be checkpointed. BOOTHAMMER seeks to strike a balance between a fully fledged task-based tool like DINO or Alpaca for cutting-edge research in the intermittent computing community and the beginner education focus of MakeCode-Iceberg. With this in mind, BOOTHAMMER provides both an automatic, basic instrumentation of checkpoints and provides the user with a CHECKPOINT macro to define where they want to checkpoint their own code. The automatic approach provides checkpointing once an iteration of the `loop` function found in all Arduino code. This provides a very basic level of checkpointing while the user instrumentation approach can provide checkpointing to whatever granularity the user wants. The CHECKPOINT macro defines a point within a user's code where a checkpoint will occur and, in the event of a power failure, execution will resume from.

4 IMPLEMENTATION

BOOTHAMMER is a binary/hex rewriting tool that modifies an incoming program to work under intermittent conditions. It is intended to work with programs output by the Arduino IDE and in conjunction with an Adafruit FRAM breakout board and a custom FRAM library. The compilation flow is shown in Figure 5. At a glance, BOOTHAMMER disassembles an incoming hex file, identifies functions within the program, instruments restore and checkpoint operations, and reassembles the program into another hex file before it is flashed onto a device. We implemented BOOTHAMMER to work in conjunction with the ARM Cortex M0+ processor and the Adafruit Feather M0 Express board. More details on the experimental setup can be found in section 5. Developing this tool within the framework of the Cortex M0+ means that BOOTHAMMER is able to disassemble all 16-bit Thumb instructions from the ARMv7-M except CBZ, CBNZ, and IT and a handful of 32-bit instructions including BL, DMB, DSB, ISB, MRS, and MSR[5]. The pipeline that transforms an unmodified hex file into one that works under intermittent conditions is described in the subsections below.

4.1 ARM Thumb Disassembler

The disassembler portion of BOOTHAMMER began as a reverse-engineered version of arm-none-eabi-Objdump tool commonly used in embedded systems development. It was built with the ARM reference manual for the Cortex m0+ processor and tested for accuracy against the Objdump of the same hex file. This disassembler within

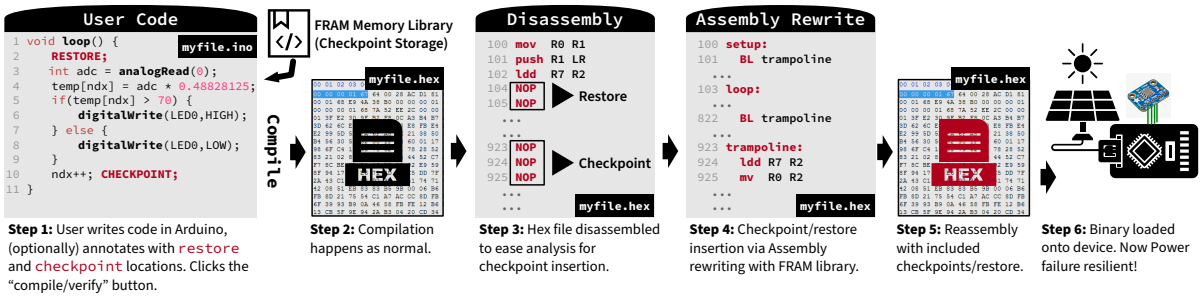


Fig. 5. Shown is our low-level, Assembly / Binary rewriting additions to the Arduino compilation toolchain. Our approach allows users to annotate checkpoint boundaries (similar to DINO [27]), and then inserts checkpoints automatically or by user direction. The tool disassembles the hex file and uses a trampoline to a special library code that handles checkpointing memory to non-volatile FRAM on a dedicated breakout board from AdaFruit. This approach is hardware independent, ensuring that any 32-bit ARM Thumb microcontroller supported by Arduino can be used with our additions.

BOOTHAMMER begins by reading in the hex file output by the Arduino IDE. The hex file is read line by line and is broken down into 16-bit halfword chunks as it reads each line. Unfortunately, the incoming hex file is in little-endian order, so in order for the disassembler to read each halfword, it must first read each two-byte pair and reverse the endianness. Before passing the bytecode to the decoding portion of the disassembler, the halfword bytecode has bits 15:11 checked to see if the current bytecode is actually one-half of a 32-bit instruction. This is because bits 31:27 contain one of the three, 5-bit sequences that identify a 32-bit instruction[3]. If these bits match that of a 32-bit instruction, then another halfword is read in the same manner as above, the two halfwords are stitched together into one 32-bit value and passed to the decoder. Otherwise, the 16-bit bytecode is directly passed to the decoder.

Now that lines of hex have been separated into 32-bit and 16-bit bytecodes, the actual disassembly process can begin. 32-bit and 16-bit instructions are passed to decode32() and decode16(), respectively. These functions work slightly differently due to the difference in bytecode size as well as the difference in the number of instructions supported by the cortex m0+, but both roughly work as a state machine that follows the armv7-m instruction set. Most of these instructions follow the simple pattern of the most significant bits indicating the instruction type and the least significant bits indicating the register, conditions, and/or immediates used in the instruction. For example, the bytecode 0xB5FF represents a PUSH instruction that pushes registers R0-R7 and the PC onto the stack and adjusts the SP accordingly. Breaking this instruction into binary reveals a simple encoding: 0b1011010111111111. Bits 15:11 indicate that this is a push instruction and each bit from 8:0 represents a register. Bit 0 set high means push R0, bit 1 means push R1, and so on and so forth until bit 8 which indicates the LR(the address that will go into the PC during a POP)[4]. This process works very similarly for 32-bit instructions. Looking at a Branch-and-Link(BL) instruction, bit 32:28 denote that it is a BL and the rest roughly make up the PC-relative address offset the represents the destination address[4]. These bytecodes are decoded into an Instruction object containing its address, type, registers used, conditions used, and/or any immediates that make up the instructions. As these instructions are decoded, they are pushed onto a list of other instructions in ascending address order.

As mentioned earlier, BOOTHAMMER began as a simple reverse-engineered version of Objdump for arm microcontrollers. A printDisassembly() function was added for determining the correctness of the disassembler and for debugging purposes. This print feature runs through the list of instructions one at a time and prints its address, bytecode, type, and any registers or other values used in a way that mimics the Objdump disassembler output.

4.2 Custom FRAM Library

BOOTHAMMER uses an external Adafruit FRAM breakout board as a source of non-volatile checkpoint storage. The Feather M0+ communicates with the FRAM board through SPI using our custom FRAM library. We ask users to include this library and initialize the SPI connection with the board in setup() using getDeviceID().

The FRAM library has two major functions that are responsible for saving and restoring from a checkpoint. We use writeRAM() to save the checkpoint to the external FRAM breakout board and readRAM() to read from it. Both of these functions have different modes based on the version on the library a user chooses to include for their specific project. Previous work in intermittent computing such as DINO or Alpaca ask users to organize their code into tasks[27][28]. This has led to common benchmarks being organized into tasks as well and only relying on the stack and registers for computation. In other words, many intermittent computing programs have no dynamic allocation or global variables. This means that our default custom FRAM library only checkpoints the stack and register states. For users that wish to checkpoint with the use of the heap, we provide another version of the library which will save and restore the entire contents of RAM for the cortex m0+. As seen later in section 5, saving the entirety of RAM has a significant performance cost but is available to users who want it nonetheless. We chose the brute-force approach of saving all of RAM to handle dynamic allocation because tracking pointers and objects at the assembly level represents another project in and of itself. For the rest of the paper, when mentioning writeRAM() or readRAM(), we refer to the version that saves only the stack and register state.

The checkpoint-writing function writeRAM() works by saving everything between the SP and the highest address to the FRAM. In order to get the address within the SP register we use an asm volatile statement at the top of the writeRAM() function. This asm statement is a move(MOV) instruction that puts the value of SP into an unsigned integer variable. We convert this address into a pointer. The function then writes the 32-bit memory value that is being pointed to into the FRAM and increments the pointer address. This process continues until the pointer reaches the end of RAM which for the cortex m0+ is 0x20008000. We save the register state by pushing all registers onto the stack before a checkpoint. More details on this is described later in this section. This leaves the SP as the only register that needs to be saved to the FRAM manually. The layout of a checkpoint in FRAM follows a double buffer pattern. The double buffer allows the tool to always read from the correct checkpoint. Address 0x0 is used as an indicator to tell the tool if a checkpoint is present or not, and if so, which section of the double buffer to write to and read from. The checkpoint layout in FRAM has the first buffer begin at address 0x1. This is where we write 4-byte SP value. From address 0x5 onward, the stack is writtent to FRAM. The second buffer follows the exact same layout beginning at address 0x9000. This provides more than enough space for the worst-case scenario of the stack growing extremely large as the total RAM space of the cortex m0+ is 0x20000000 to 0x20008000. After a checkpoint is written, the indicator value is updated to point to that new checkpoint. The FRAM chip itself is able to perform a transaction at the byte level, so we leverage this to perform a transaction for each checkpoint. By updating the indicator value last, we guarantee that when it changes, a new checkpoint has been completely written.

The function readRAM() reads the checkpoint from FRAM and overwrites the RAM on the cortexM0+ to resume where the program last left off. After first reading the indicator value at address 0x0, readRAM() will either return immediately if no checkpoint is present or load the checkpoint into RAM. First, the SP value is read from FRAM and the current SP is overwritten with the checkpointed one. This is extremely important because a checkpointed SP value is likely to be higher than the SP at startup. If a checkpoint is loaded without updating the SP beforehand, it is highly likely that the contents of the stack and/or ram will be corrupted as the checkpoint is loaded into RAM. The readRAM() function itself calls other functions, specifically SPI.transfer(), that adjust the value of the SP. This means that if the SP is not adjusted beforehand, the current stack frame and one or more of the stack frames loaded from the checkpoint will overlap. In the best case scenario, only a few general-purpose

registers are overwritten, but in the worst case a return address is overwritten and a function pops or branches to a bogus address. Either way, the behavior is undefined and will likely cause the board to get stuck in the reset handler. When the SP is updated before the checkpoint is loaded, the current and loaded frames do not overlap and the integrity of the checkpoint is maintained. We load the checkpoint by reversing the write operation in writeRAM(). An unsigned integer pointer is set to point to the address within the loaded SP value. As values are read from FRAM, the address currently pointed at is written with these new values, the pointer is incremented, and the process repeats until the pointer reaches the end of RAM. Lastly, readRAM() branches to the end of writeRAM() once the checkpoint has been fully loaded into RAM. BOOTHAMMER branches to the end of writeRAM() because it's at that point where the checkpoint saved the state of the device. From here, writeRAM() returns to the code that called it and computation resumes from where it left off.

4.3 User Code Analysis

Now that the hex file has been disassembled into a list of Instructions, the work of understanding the basic framework of the program can begin. Among other reasons, such as being geared towards novices and markers, the Arduino IDE was chosen because it outputs its programs in a very predictable way. When using Objdump to disassemble an elf file, sections of assembly are separated by the function that they belong to. However, disassembling a hex file provides no such information. This is where the predictable output is necessary. The main function of a program from the Arduino IDE can easily be identified in a sea of instructions by its unique instruction "signature." This signature is largely based on the presence of four sequential BL instructions and a branch that causes the loop() Arduino function to run in an infinite loop. These four BLs branch to a helper function, setup(), loop(), and yield(). Based on our findings, this instruction signature for main, and Branch-and-Link, order will remain the same for most if not all Adafruit SAMD boards. When disassembling Arduino SAMD boards, we found that this signature changes only slightly and for the most part is the same. Main() is found in a user's program by iterating through the list of instructions until it finds a section of code that matches this signature.

Once main() has been identified, BOOTHAMMER has the key to unlock the rest of the necessary function information of the program. Since setup() and loop() will always be the second and third BLs in the main signature and because instruction decoding calculates the address offset and destination address of each BL, we can follow branches to their destination just as the microcontroller would. As BOOTHAMMER traverses each BL, it builds a function object of Instruction pointers that belong to that function as well as a list of child functions that are called by the current one. BOOTHAMMER uses the function fillFunction() to follow BL instructions to their destination and add pointers to those instructions to its own list until it reaches the end of that function. The end of a function is determined in multiple ways. Some functions end on a branch to the Link Register(LR), and others use the POP instruction, but the true end to most functions occurs at the end of the constant values that the instruction uses. For example, if a function uses 0xDEADBEEF as a value, then it will appear at the end of a function in a disassembly output. The best way to track the end address of the constant list it to read what addresses the Load Register(LDR) instructions use. By keeping a tally of the highest address used in an LDR instruction, BOOTHAMMER is able to add any extra instructions or constants that belong to a function that occurs after a branch or POP instruction. When fillFunction() runs into another BL, it recursively calls itself and builds another function. Once this process is complete, BOOTHAMMER will have built a graph of all functions that are reachable from setup() and loop() with main() as its root node.

It is the responsibility of the function analysis step to find the writeRAM() and readRAM() functions from our custom FRAM library. As mentioned before, these functions are responsible for writing and restoring checkpoints. We identify these functions by placing an identifier value in each function that will be present at disassembly. We chose this approach over signature matching, like how main() was found because the length of these functions is

much, much larger, were regularly rewritten during the development of this tool, and are not as easily identifiable as main(). We use the volatile keyword when declaring an unsigned integer to use as our identifier value. This value is computationally pointless and so the volatile keyword is used to prevent it from being optimized out. WriteRAM() uses 0xDEADBEEF and readRAM() uses 0xBEEFDEAD and they are easily found during function building when the tool is iterating through the disassembled code. The values we use here are not important, they are simply easy to find with manual inspection and are unlikely to show up in a user's program.

4.4 Basic Checkpoint Instrumentation

The basic checkpointing approach works by restoring from setup() and checkpointing once per iteration of loop(). As mentioned previously, at this point BOOTHAMMER has access to the important functions of a user's program, the most important of which is main(), setup(), and loop(), as well as the functions that will read and write a checkpoint. In order to avoid re-linking and redoing the addresses of an entire program, BOOTHAMMER adds new code the end of the hex file and overwrites code within the program without shifting any other addresses. The code added at the end of the program effectively serves as a trampoline between the user's code and the checkpointing code. The branch to loop() within main() is overwritten with a BL to a section of the trampoline that first pushes all registers and the LR onto the stack, second, it branches to loop and it runs as normal. When loop() returns, it calls writeRAM() from our FRAM library. When that function returns, the registers and PC are popped from the stack and the code resumes at the next instruction in main following the overwritten loop branch. It is the pushing and popping of registers to and from the stack that allows this system to save the register and stack state all in one. The trampoline serves as a necessary buffer between the user's code and the writing of a checkpoint. The readRAM() and writeRAM() functions themselves do push some of the registers to the stack, but not all of them. This puts the program at risk of becoming inconsistent with the same code under continuous power. It is true that we could manually add PUSH and POP instructions to the readRAM() and writeRAM() functions, however, this can and will ruin the execution of these functions as the rest of the function assumes a higher SP value. By using the trampoline as a register buffer, we protect the register state from being overwritten during the execution of the FRAM library functions. When execution returns from the trampoline, the main code then continues as normal. It will branch to yield() and then jump back to the overwritten loop() branch in a never-ending cycle.

In order to reestablish connections with peripherals such as serial or SPI, we ask the user to add a restore macro RESTORE to their code after this point in `setup`. The restore macro represents an asm volatile statement of two BKPT instructions with immediate value 0xE. We chose the BKPT(breakpoint) instruction because the immediate value is irrelevant to the instruction[4], so we can put in a unique identifier in this field for BOOTHAMMER to recognize as it transforms as hex file. BOOTHAMMER searches through `setup` to find these two BKPT instructions and overwrites them with a BL instruction. We use two BKPT instructions because a BL is a 4-byte instruction and each BKPT is 2 bytes. By carving out space for a BL to the trampoline before compilation, overwriting this space with a BL does not require any relinking or address readjustment. The section of the trampoline that the restore BL branches to pushes all registers and the LR to the stack and branches to readRAM(). As mentioned in the section about our custom FRAM library, if no checkpoint is present in the FRAM, readRAM() returns to the trampoline, the registers and the PC are popped from the stack, and execution resumes in setup(). When a checkpoint is present, readRAM() loads that checkpoint and branches to the end of writeRAM(). It is BOOTHAMMER that creates this branch and it effectively serves as a goto statement. There are two NOP instructions toward the end of readRAM() that are overwritten with a branch to writeRAM(). BOOTHAMMER knows the address within writeRAM() to branch to because there is a NOP sled/landing pad that is also found during function discovery.

The branch-and-link instructions that BOOTHAMMER makes for the program are constructed following the ARM Thumb reference manual. BL instructions are PC-relative, meaning that the immediate value within the instruction

is an offset to the PC. The destination address is the sum of the PC value and the immediate value where the immediate value can be positive or negative. So, working backward from the destination address we can build a BL instruction to that address. For example, when overwriting the branch to loop() within main(), BOOTHAMMER takes the entry point address of the trampoline and subtracts it from the address of the BL-to-loop instruction in main(). This gives a 32-bit offset that can be encoded into a BL instruction. Other instructions that are necessary for the trampoline are built in a similar way. BOOTHAMMER uses the internal function insertAndDecode() that takes in a hex bytecode, decodes it, and inserts the new instruction at the specified address.

4.5 User Instrumentation

BOOTHAMMER also supports user-guided checkpointing of a program. Just as with the RESTORE macro, there is a CHECKPOINT macro that allows the user to tell BOOTHAMMER where to perform a checkpoint.This macro is an asm volatile statement of with one NOP instruction and two BKPT instructions with an immediate value of 0xF. The importance of the NOP instruction is described in the next paragraph. The tool searches through each function of the program looking for these BKPT instructions and replaces them with a branch-and-link to the last section of the trampoline. This section pushes all registers and the LR onto the stack, calls writeRAM(), and pops all registers and the PC, returning to where the checkpoint was called. The user can add this CHECKPOINT macro anywhere in their code so long as it is after the RESTORE macro.

The reason why the CHECKPOINT macro has an extra NOP instruction is because of a very unique corner case that can break the execution of a program. Some functions do not push registers, most importantly the Link Register(LR), onto the stack and use a Branch-Exchange(BX) instruction to return to the caller function. This is what we describe as the BX LR problem. If the LR is not pushed onto the stack, but within the function a BL occurs to the trampoline, then the value within the LR will be overwritten and lost with no way to recover it. When the BX LR is hit, it is highly likely the BX instruction will branch to a bogus address and break the program. In order to solve this problem with the minimal amount of increased overhead of the program and assembly modification, we add a PUSH(LR) and a POP(PC) to the function that returns using BX LR. The first checkpoint will have its first NOP instruction overwritten with a PUSH(LR) instruction and the BX LR will be overwritten with a POP(PC) instruction. This will push the LR onto the stack where it will be safe from future BL instructions. The POP(PC) instruction pops the saved LR value from the stack and loads it into the PC, recreating the same effect as the BX LR instruction. While it is certainly true that this leads to excess instructions within the code, the cost of a single NOP instruction is a single cycle, meaning that any unnecessary overhead due to this choice is negligible.

4.6 Hex File Generation

Returning the disassembled program back into a hex file is relatively straightforward. Each instruction object contains the original bytecode that it was decoded from, so outputting that instruction into a hex file only requires the formatting steps necessary for a hex file. Each line needs a byte count, address, record type, data section, and checksum. The printHEX() function within BOOTHAMMER builds a hex line of size 0x10, sets the address to the address of the first instruction in the instruction list, sets the record type to 0x00 for data, fills in bytecodes until it reaches the size of the line, and adds the checksum. The checksum for a line/record is calculated by taking the sum of all of the byte values within the line and taking the two's complement of the least significant byte[6]. Lastly, BOOTHAMMER pads NOPs to the end of the last data line and appends end-of-file data lines to the end of the file. The original hex file has now been successfully disassembled, analyzed, modified to work under intermittent conditions, and turned back into a hex file. From here, the hex file is ready to be flashed onto a board.

5 EVALUATION

We begin our evaluation by laying out some questions that we will answer by the end of this section.

(1) **Real World Usage.** We demonstrate the use of BOOTHAMMER in building real-world applications and also present a power consumption analysis (Section 5.2).
(2) **Correctness.** We demonstrate that BOOTHAMMER can produce the correct output/execution under intermittent conditions. We compare the results of unmodified Arduino programs under constant power with the results of a BOOTHAMMER-modified program under intermittent conditions. (Section 5.3).
(3) **Checkpoint/Restore Performance and Overhead.** We compare three checkpoint strategies against each other, state-of-the-art intermittent computing runtimes, and maker-focused batteryless IDEs (BFree, Makecode). (Sections 5.4, 5.5. Lastly, we evaluate BOOTHAMMER on basic metrics such as runtime of the tool itself and code-size increase. (Section 5.6).
(4) **Usability and Applicability.** We report on a small user study to understand how BOOTHAMMER enhances understanding of and programming of batteryless maker-focused applications (Section 5.7).

Before delving into the evaluation, we discuss our methodology.

5.1 Evaluation Methodology

We provide details on our approach to evaluating BOOTHAMMER.

5.1.1 Hardware and Software Infrastructure. We evaluate BOOTHAMMER in all experiments using the most recent version of the Arduino IDE(v2.1.1) and an Adafruit Feather M0 Express connected to an Adafruit SPI FRAM breakout board. For some use cases we use a Sparkfun Red Board. We flashed transformed hex files using a Segger J-Link Debugger.

5.1.2 Benchmarks. Numerical benchmarks are used to evaluate correctness of execution of programs modified by BOOTHAMMER. Similar to past work in intermittent computing, we rely on these benchmarks to ensure we are transforming the program correctly, as any incorrect behavior could cause errors or incorrect outputs that would manifest in the execution. Each of the benchmarks has a true answer (i.e., the 10th Fibonacci sequence is always the same no matter the program), which we can use to validate our execution. We chose a handful of different benchmarks that have been used in contemporary work such as alpaca and Dino[27][28]. The benchmarks are Bitcount(BC), Blowfish(BF), Cuckoo filter(CF), activity recognition(AR), RSA compression, and cold-chain equipment monitoring(CEM). Later, when we compare BOOTHAMMER to BFree and Battery-free MakeCode, we use a slightly different bitcount(BCS), string length(STR), and Fibonacci(FIB) benchmarks.

5.1.3 Checkpoint Strategies: Verbose, Basic, and Strategic. Due to the fact that BOOTHAMMER is a user-guided tool, where and when to checkpoint is up to the user's preference. While BOOTHAMMER has automatic checkpointing for each iteration of the loop() function within Arduino, none of these benchmarks utilize loop() in a continuous fashion and instead use a while(1) loop within setup(), loop(), ends at the end of setup(), or in a called function. This means that we are largely evaluating the user-guided checkpointing side of this tool. This is acceptable because user-guided checkpoints will always have at least one checkpoint within their program, meaning the overhead of automatic checkpointing will be subsumed by user-guided checkpointing.

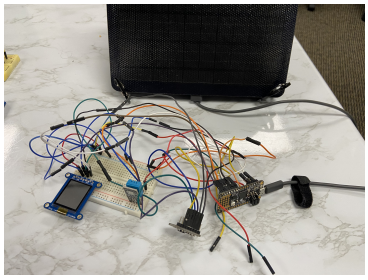
We present and test three different checkpointing categories. The first, Verbose, places a CHECKPOINT macro wherever a task boundary begins and ends in contemporary work such as DINO. This emulates a "task" according to these systems, where a task is an atomic region of code that either completes in entirety or not at all[27]. Verbose checkpointing results in a very high number of checkpoints written to the FRAM, and as seen in Figure 10, incurs a significant time overhead many times that of an unmodified program. This is not to say that this strategy is without any merit. One could imagine an intermittent program, such as an agriculture sensor, where

logging and never losing a single sensor reading would be very important. The next category of checkpointing we deploy is Basic Checkpointing. Here, we keep many of the checkpoints outlined in Verbose but pull out some CHECKPOINT macros from nested loops and remove some redundant checkpoints from functions. As seen in 10, this results in a modest runtime reduction compared to Verbose for many of the benchmarks. We expect Basic to be a common category for makers and hobbyists to use because it is unlikely they will know how to orient their code into tasks but will refrain from minimal checkpointing in the Strategic category. For the last category of checkpointing we tested, Strategic, we employ an aggressive lack of checkpointing. We do this not only to show how far we can push BOOTHAMMER to it's limits, but also to show what is possible when an expert can exploit the structure of a program to keep overhead as small as possible. Here, Strategic checkpointing mostly involves removing many of the checkpoints from Verbose and Basic, and keeping them only when absolutely necessary. From a programming perspective, this takes the shape of placing CHECKPOINT macros in outermost loops. Figure 10 shows that this strategy is the clear winner in terms of runtime overhead. This is expected, of course, as when you CHECKPOINT less often, the program will complete in a shorter amount of time. However, the distinction we are trying to draw here is that while it is possible to lose more progress per power loss using Strategic checkpointing, we more than make up for it by preventing the checkpoint from overburdening a program's runtime.

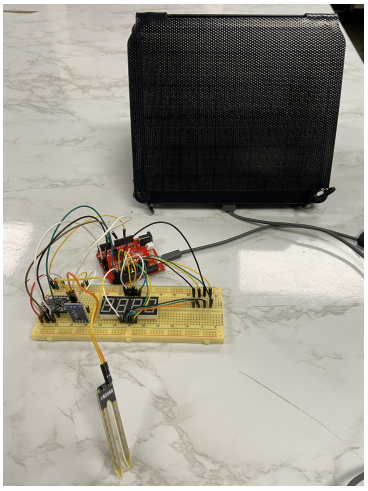
5.2 Use Cases: Real world power consumption and execution

A common use case example for intermittent computing applications relate to smart gardening and agriculture. For example, the Battery-free Makecode paper had a smart terrarium use case where data about the terarrium was displayed on an E-Ink display[25]. Similarly, BFree presented use cases in this realm of environmental sensing by relaying temperature and humidity data over a LoRa Transciever[22]. As seen in Figure 2 we seek to present two hobbyist-level use cases relating to environmental sensing in a similar fashion to BFree and Battery-free MakeCode. Our first use case can be seen in Figure 6a and presents a smart gardening example where an Adafruit Feather M0 Express running BOOTHAMMER modified code is connected to an Adafruit 4MB external FRAM breakout board, a DHT11 temperature and humidity sensor, and an Adafruit Shard Memory Display. The program that has been run through the BOOTHAMMER tool reads the temperature from the sensor, updates the sample count, calculates a running average of all samples taken, and displays this information on the sharp memory display. This use case is powered by a recreational use FlexSolar 15W solar panel. This solar panel is far more than enough to power both of the use cases in Figure 2 but was chosen because of it's durability and built-in USB port for easy connectivity with commercial microcontrollers. The power consumption of this use case can be seen in Figure 7a. This measurement was made using the Otii Arc power analysis tool. Similarly, we see the voltage trace of the first use case in Figure 8a under energy harvesting with the solar panel.

Our second use case also reflects a potential smart-gardening or smart-argiculture, hobbyist-oriented use case where soil moisture is recorded over time. In this case we use a different board to demonstrate BOOTHAMMER's flexibility between board manufactures. As mentioned back in section 4, we build BOOTHAMMER to work with a specific instruction set of ARM Thumb and processor in mind. We are able to use a Sparkfun RedBoard Turbo because it also uses the Cortex M0+ processor. We believe this to be one of BOOTHAMMER's biggest strengths as a tool for intermittent computing as we are not limited to a single device like with Battery-free MakeCode being specific to the BBC Microbit[25]. This use case can be seen in Figure 6b and similarly uses an external FRAM board for non-volatile storage for checkpoints. This program displays the number of soil moisture samples taken on a seven segment display and makes the samples available over serial if a user wishes to retrieve them, up to five hundred. The limit of five hundred samples was chosen arbitrarily and is not a limitation imposed by BOOTHAMMER. We can see that the power traces seen in Figures 7b and 8b are similar to that of the first use case but differ slightly due to the difference in peripherals and board.

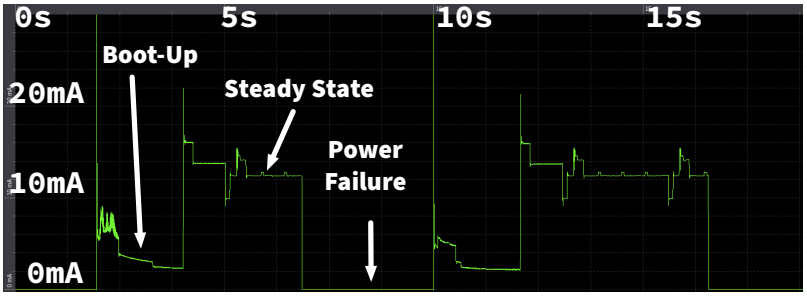


(a) Temperature, Humidity, and Sharp Display Use Case

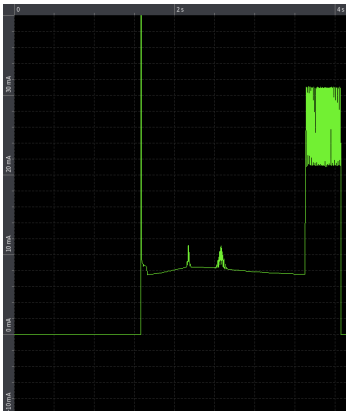


(b) Soil Moisture and Seven Segment Display Use Case

Fig. 6. We present two unique use cases developed using the BOOTHAMMER tool. In 6a we demonstrate a smart gardening example where an Adafruit Feather M0 Express is powered by a solar panel and connected to a DHT11 temperature and humidity sensor. The sample count, most recent temperature reading, and running average of the temperature is displayed on the Adafruit Sharp Memory Display. In 6b we present a similar smart gardening/argiculture example where a Sparkfun Red Board Turbo is connected to a soil moisture sensor and powered by a solar panel. The number of samples are displayed on the seven segment display and an array of up to 500 samples are available is over serial for a user to retrieve at any time. As a note, 500 was chosen arbitrarily and is not imposed by BOOTHAMMER. Both 6a and 6b are connected to an Adafruit SPI FRAM board that serves as non-volatile storage for checkpoints.

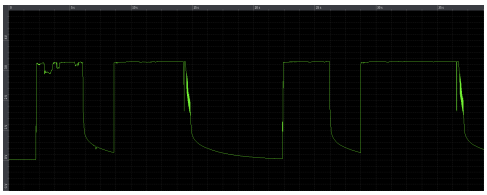


(a) Annotated current (mA) trace of Use Case 6a, with a power failure, bootup sequence, and steady-state sensor execution shown.

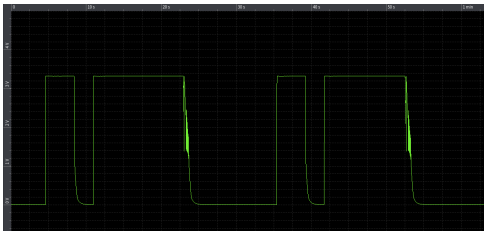


(b) Power trace of Use Case 6b

Fig. 7. Power trace of the use cases from Figure 6



(a) Power trace of Use Case 6a using energy harvesting



(b) Power trace of Use Case 6b using energy harvesting

Fig. 8. Power trace of the use cases from Figure 6 using energy harvesting

5.3 Correctness Stress Test Benchmark

An intermittent computing program is only worth deploying if it can produce the same output under intermittent conditions as it would under constant power. For BOOTHAMMER, we developed a simple correctness stress test specific to this tool's checkpointing scheme. This test calculates the fibonnoci number for 46 one thousand times and stores that value in an array of type long and size one thousand. This array is allocated on the stack and not the heap. We have added a CHECKPOINT macro in this loop so each calculation of fib(46) is checkpointed. In this benchmark we have added two in-program checks to make sure each fibonnaci calculation is correct. Both will stop the program if a calculation of fib(46) is incorrect or if any value in the array is not correct. We ran this program under simulated intermittent conditons by periodically turning power on and off while the program runs. We have verified the output of this program to show that BOOTHAMMER is capable of providing correct results under intermittent conditons. We consider this program a "stress test" because the one thounsand element array of type long, allocated on the stack, will consume at least 4KB of space on the stack and at least 8KB on the FRAM board because of double buffering. For the Cortex M0+, this means we are checkpointing greater than one eighth of the total SRAM space and our tool is able to keep this data structure consistent across power failures. Our ability to keep the stack, and all variables and data on it consistent, makes intuitive sense because of BOOTHAMMER's brute force approach of saving the entire stack for checkpointing.

5.4 BOOTHAMMER vs State-of-the-Art

In order to determine how BOOTHAMMER stacks up against contemporaries in the field of intermittent computing, we run the same benchmarks used in Figure 10 against the state-of-the-art tools such as Dino and Alpaca. We can see that in Figure 9 compares modestly against these other tools, performing better in some cases and worse in others. Again, it is important to mention that this is using Strategic checkpointing and not Verbose or Basic, which would preserve progress in a similar way that these other tools do. However, it is important to mention the differences between these tools and the differences in hardware they are running on. Tools like Dino and Alpaca use the MSP430FR5969 which has non-volatile FRAM built into the device itself[39], whereas most if not all off-the-shelf boards like the Adafruit Feather M0 Express will not. This forces the use of external non-volatile storage that is limited by SPI bus speeds(12MHz). We show the effect of this slowdown by comparing different SPI clock frequencies in Figure 13. So, we push BOOTHAMMER to its limits to show what is possible through using this tool and compare it to the state-of-the-art. As mentioned before, this tool has a very large overhead when running the cuckoo filter(CF) benchmark and this is due to the very short runtime of the unmodified version. In comparing this tool to others in the intermittent computing field, we are not trying to claim this is a replacement for these other systems or that it is inherently better. We do claim, and Figure 9 supports this, that BOOTHAMMER is able to transform code from the Arduino environment so that it not only works under intermittent conditions, but also works within tolerable overheads seen in the rest of the field.

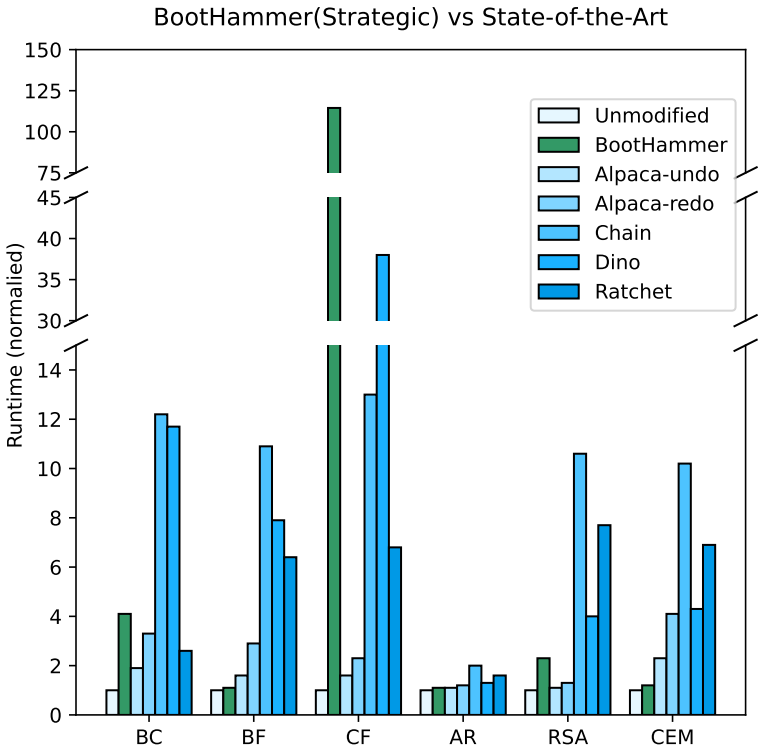


Fig. 9. Here we compare BOOTHAMMER (green) using strategic checkpointing against the state-of-the-art(blue) in the intermittent computing community. BOOTHAMMER performs well and is roughly in line with several of the other task-based systems for intermittent computing. The execption here is CF(Cuckoo Filter) where the normalized runtime was over one hundred times the overhead of the undmodified program. This is likely due to the very short runtime of CF and the outsized effect on runtime due to checkpointing.

5.5 BOOTHAMMER vs BFree and MakeCode-Iceberg

BOOTHAMMER is not the only maker or hobbyist-centered intermittent computing system. Other works such as BFree and MakeCode-Iceberg have attempted to do the same[22][25]. We compare BOOTHAMMER against these tools using the benchmarks that they used, respectively. Figure 11 shows that all checkpointing categories, Verbose, Basic, and Strategic, all perform well and in line with BFree. In the String Length benchmark all three categories perform better than BFree. This is likely due to the high overhead BFree incurs by running CircuitPython and the relatively minor code overhead due to BOOTHAMMER. We say that the code overhead of this tool is small due to Figure 13. Simply increasing the clock frequency of the SPI bus had a dramatic impact on the runtime overhead of the benchmarks used in Figures 10 and 9, especially on benchmarks that incurred a large overhead due to BOOTHAMMER. Even with a large improvement from 4MHz to 12MHz on the SPI Clock, there is still room to improve. The max frequency of the FRAM board is 40MHz and the highest that the ATSAMD21G18A(Cortex

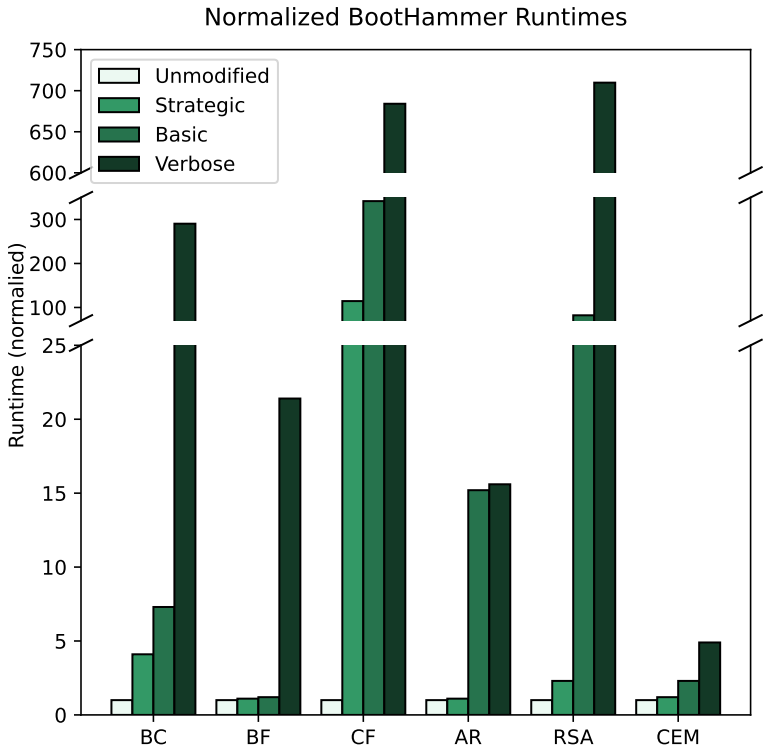


Fig. 10. In this figure we determine the most effective checkpointing strategy between verbose, basic, and strategic. Verbose checkpointing is where a CHECKPOINT macro is placed wherever a task boundary would be placed in contemporary work such as DINO[27]. Basic checkpointing involves keeping most of the CHECKPOINTs from verbose but pulling them out of nested or long loops. The strategic method is an aggressive scheme that tries to minimize checkpointing unless where absolutely necessary

M0+) can handle is 24MHz[35][30], however approaching these frequencies is unwise without fabricating a PCB. Figure 12 also shows that Strategic BOOTHAMMER is able to perform well against MakeCode-Iceberg using its fastest setting. While this tool is slower than MakeCode-Iceberg in the three benchmarks used, the margins are relatively small. This is considerable given that MakeCode-Iceberg is a compiler-based approach for interrmittent computing and requires a non-trival amount of code restructuring in order to work. Additionally, the compiler approach was shown to have very small checkpoint size, reducing the amount of time it takes to save and restore from a checkpoint[25]. The checkpoint size of BOOTHAMMER varies considerably based on the program because the checkpoint size is largely stack-size dependant. However, the checkpoint size will at least be the size of ten 32-bit registers(R0-R7, LR, and SP) and the one-byte indicator. This means that our checkpoint size starts at 41 bytes. This contrasts with the largest MakeCode-Iceberg total checkpoint size of 41 bytes[25].

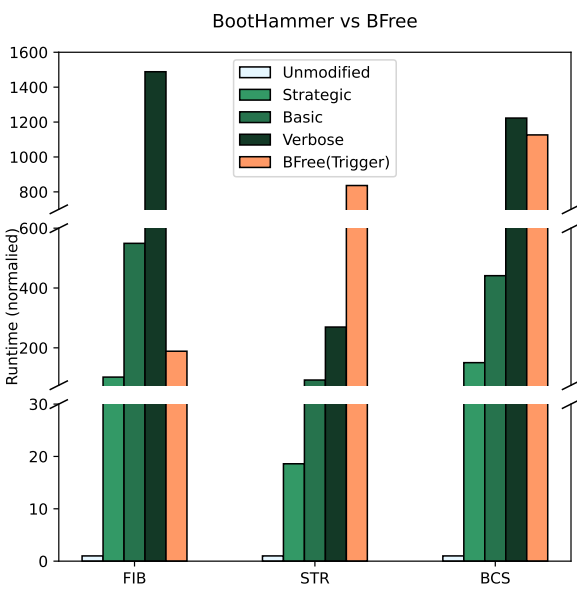


Fig. 11. Comparsion between BOOTHAMMER and BFree(trigger). The above figure compares and contrasts these two systems on three different benchmarks: Fibonnaci(FIB), String Length(STR), and Bitcount(BCS). FIB calculates the 40th fibonnaci number, STR calculates the length of predefined 40 character string, and BCS is a bitcount of 0x7fffffff. We use these benchmarks because the come directly from the BFree project[22]. We include the three different BOOTHAMMER strategies outlined in this paper because most are roughly in line with or better than the performance of BFree.

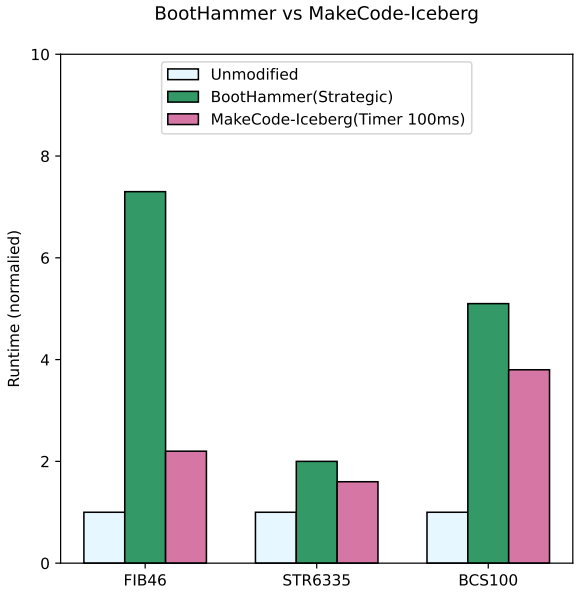


Fig. 12. Comparison of BOOTHAMMER (Strategic) vs MakeCode-Iceberg using the timer-100ms setting. We compare these two systems because they are both geared towards markers, researchers, and those with little experience in intermittent computing. The FIB46 benchmark calculates the 46th fibonnaci number 80 times, the STR6335 calculates the length of a 6335 character string, and BCS100 calculates the bitcount of 0x7fffffff 100 times. We can see that the fastest BOOTHAMMER approach performs comparably to the compiler approach of MakeCode-Iceberg under its fastest approach.

5.6 System Overhead

The last question we need to answer about BOOTHAMMER is what cost it has on the basic metrics of the program being transformed and the system running the tool. Figure 14 shows that the increase in code size between an unmodified benchmark and a transformed benchmark is reasonably small. This highlights one of the main advantages of rewriting at the assembly level. The tool is able to keep necessary changes to the code as small as possible. Additionally, most of the code-size increase seen in Figure 14 is due to the included libraries that are necessary for communicating with the external FRAM board. The RESTORE macro only results in four additional bytes added to the setup() function and the CHECKPOINT macro only adds twelve per use. Also, the trampoline will mostly consist of a handful of PUSH, POP, and BL instructions, meaning the trampoline does not have an outsized effect of the size of the program either. Table 1 shows the time it takes to run BOOTHAMMER on the host machine. We ran BOOTHAMMER on a virtualBox virtual machine running Ubuntu v20.03 on a Windows host

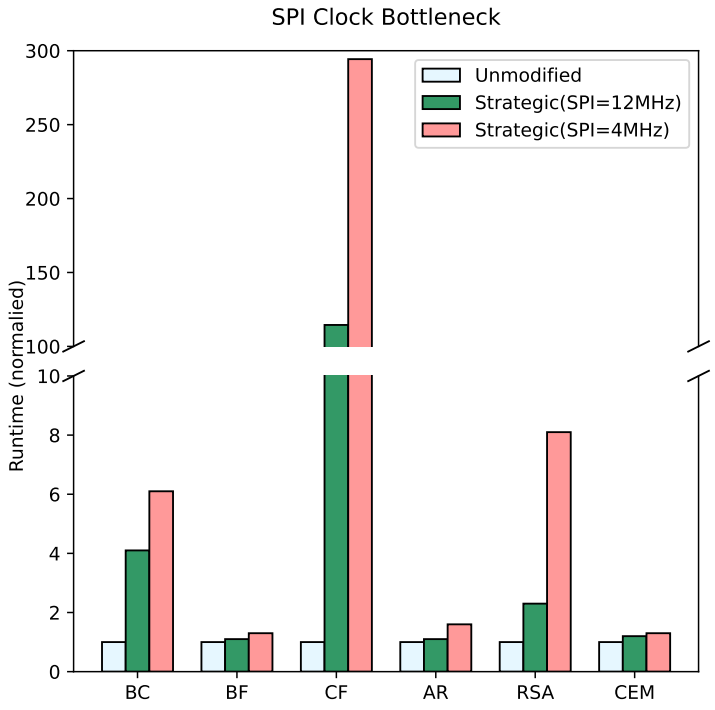


Fig. 13. In this figure we show the results of running the same benchmarks from 9 under different SPI clock frequencies. The difference in normalized runtime of Strategic BOOTHAMMER under 4MHz and 12MHz SPI clock frequencies is large. The 12MHz version vastly outperforms the 4MHz version in benchmarks that have a higher overhead than the others and slighly outperform those with a low overhead. This lends evidence to the idea that the true bottleneck of the BOOTHAMMER system is the SPI bus, not the instrumentation added to the user's code.

machine with an Intel Core i7-9750H CPU and 16GB of RAM. The time is takes to transform these benchmarks is well within reason and places no burden on the host machine or VM.

5.7 User Study

A small-scale user study was conducted with novice and intermediate Arduino programmers to evaluate the usability of BOOTHAMMER and how it may influence their confidence in writing intermittent computing programs. This study was modeled after past work that evaluated battery-free programming via micro:bits using Battery-free Makecode[13, 25, 31]. This study aimed to answer the following research questions:

- **RQ1**: How does BOOTHAMMER affect Arduino programmers' understanding of code execution in relation to power failure?
- **RQ2**: How does BOOTHAMMER affect Arduino programmers' sense of confidence in writing their own intermittent computing programs?
- **RQ3**: What use-cases can Arduino programmers envision for intermittent computing programs?

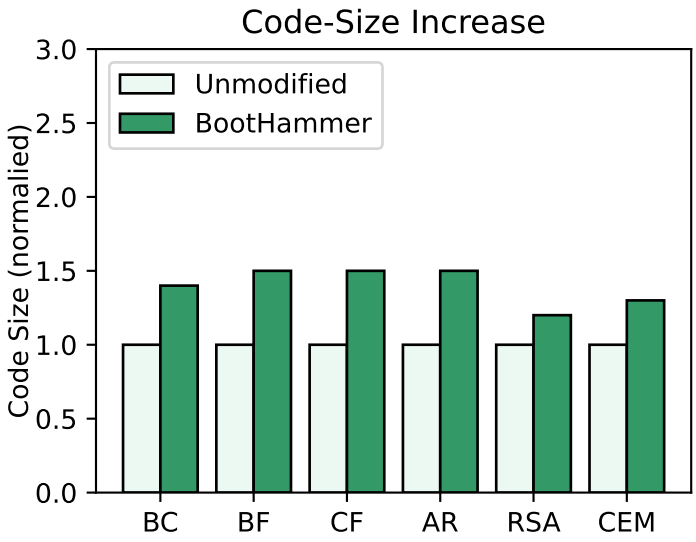


Fig. 14. The code size increase due to BOOTHAMMER's assembly rewriting is modest. Even with the inclusion of extra libraries for communication with the FRAM breakout board and the addition of the assembly trampoline at the bottom of the file, the code-size increase due to this tool is well within reason.

Table 1. In this table we present the average time it takes BOOTHAMMER to transform the benchmarks used in the evaluation section. We can see that while some benchmarks are faster or slower than others, the overall takeaway from this table is that the disassembly, rewriting, and reassembly process places little to no burden on the host machine.

Benchmark	BOOTHAMMER Rewriting Time(s)
BC	0.0094
BF	0.0106
CF	0.0098
AR	0.013
RSA	0.0102
CEM	0.0278

5.7.1 Participants. Inclusion criteria for this user study required that individuals possess varying levels of familiarity with Arduino projects. These levels were defined as **novice** and **intermediate**; participants classified as novice were defined as having only passing experience with Arduino projects, typically motivated by external forces such as introductory prototyping class work, while participants classified as intermediate were defined as having either worked on many past Arduino projects and/or possessed personal motivations to create their own Arduino projects. All participants must have had no prior knowledge of intermittent computing concepts prior to the study.

A total of nine participants, all ranging from undergraduate to graduate programs, were recruited via snowball sampling through word of mouth, email, and instant messaging platforms. Two participants were undergraduate

students studying Computer Science, four were graduate students studying Human-Computer Interaction, two were Ph.D. students in Human Centered Computing, and one Ph.D student in Computer Science. Three participants were categorized as 'intermediate' Arduino programmers. Of these, two were CS undergraduate students while one was an HCI graduate student. The remaining six participants were categorized as 'novice' Arduino programmers. Of these, two were Human-Centered Computing Ph.D. students, one was a CS Ph.D. HCI student, and the remaining three were HCI graduate students. Due to the qualitative nature of this study, we determined the sample size to be adequate in reaching data saturation.

5.7.2 Study Design and Execution. Each participant was invited to a prepared laboratory space featuring a table with a laptop connected to the experimental Arduino prototype and an industrial floor lamp positioned to face a usb-compatible solar panel. Three scenario programs were written for the study and were displayed on the laptop for each participant to interpret. The experimental Arduino prototype consisted of an Adafruit Feather M0 Express, and a breadboard outfitted with an Adafruit 168x144 Monochrome Sharp Memory Display, and the DHT11 Temperature and Humidity sensor. This setup is the same as the Use Case from subsection 5.2 and as seen in Figure 6a. A Segger J-Link was used to flash programs to the Feather M0 Express. With both written and verbal consent, audio was recorded via an audio recording mobile application and deleted upon upload to a secure cloud location. Power to the experimental Arduino prototype was supplied via the solar panel and floor lamp. During execution, researchers coordinated to guide each participant throughout each session. One researcher facilitated the session while the other provided technical support by manipulating the experimental Arduino prototype, operating the floor lamp, and loading the scenario programs.

The study procedure consisted of three phases: a background interview, participant interpretation of three scenario programs/prototype demonstrations, and three likert scale questions. The background interview consisted of questions that collected student demographics, e.g. '3rd year undergraduate CS major', relevant experience with Arduino-based projects, e.g. 'tutored students for projects', descriptions of past Arduino projects, e.g. 'smart home garden', and how power was supplied to said projects.

The first scenario had each participant interpret a counting program that did not make use of `BOOTHAMMER`. The program incremented a count over time that was shown on the display installed on the breadboard. Each participant was encouraged to voice their understanding of the program as they interpreted in real time. Next, a researcher would power on the floor lamp and flash the program to the Arduino prototype. The participant was asked to describe the displayed output, taking special note of the current count. The lamp was then powered off, thereby shutting down the prototype. Each participant was asked to note the last count that was shown on the display before power loss. Power was then restored and each participant was asked to describe the count shown on the display again. Since this first scenario did not make use of `BOOTHAMMER`, the displayed count always began from zero upon startup.

The second scenario had each participant interpret a version of the same program, but modified to use the RESTORE and CHECKPOINT macros defined by `BOOTHAMMER`. Each participant was asked to voice their understanding of the modified program which was also displayed alongside the original. Similar to the first scenario, power was supplied to the prototype and each participant was asked to describe the output count. Power was then shut down and each participant was asked to note the last count displayed. Power was then restored and each participant was asked to describe the display upon startup. Since this scenario leveraged `BOOTHAMMER`, the displayed count always resumed from the last count shown prior to power loss.

The final scenario entailed a researcher-described application of intermittent computing that consisted of a smart garden example that leverages solar panels to power temperature and humidity sensors. Each participant was asked to interpret the scenario program that made use of `BOOTHAMMER` macros, which calculated and displayed the average ambient temperature and humidity. Power was supplied to the prototype and the program was executed while each participant was asked to describe what was shown on the display. Each participant was then

asked if they could envision other applications of intermittent computing programs that may be relevant to their own interests.

To conclude each session, three likert questions were asked in the form of statements that each participant could either agree with or disagree with on a scale of 1 through 5, where 1 corresponded to "strongly disagree" and 5 corresponded to "strongly agree". The three statements were: (1) "I'm interested in writing my own intermittent computing program", (2) "I feel confident in my ability to write an intermittent computing program using BOOTHAMMER ", and (3) "I can imagine different use-cases for intermittent computing programs". Each participant was encouraged to elaborate on each of their given scores.

5.7.3 Results. Data analysis entailed the transcription of participant recordings and subsequent thematic analysis via Grounded Theory methodology [36]. Average values for likert statements were calculated for each participant response. Results are framed to address our three research questions (RQ).

RQ1: How does BOOTHAMMER affect Arduino programmers' understanding of code execution in relation to power failure? All participants demonstrated an understanding that power failure in the traditional computing scenario resulted in the loss of stored memory and execution, while in the intermittent computing scenario, stored memory was persisted and execution resumed. Elaborations for likert statement (3) "I can imagine different use-cases for intermittent computing programs" provided a variety of descriptions that all demonstrated an understanding that intermittent computing programs save/maintain their state upon power failure in generalized terms. These descriptions, coupled with clarifying commentary made throughout the second phase of study sessions demonstrated that all participants grasped the nature of power failure and its effect on code execution.

Participants having more extensive programming experience expressed curiosity in the lower level hardware and software details of how the tool worked and asked many questions related to the mechanics of the prototype. Participant inquiries covered topics such as the debugging experience of intermittent computing, the use of C++ macros, and how state is preserved upon power failure at the hardware level. Upon first exposure to the RESTORE and CHECKPOINT macros, three participants intuited their functionality accurately while the remaining six understood upon further interaction with the prototype scenarios.

RQ2: How does BOOTHAMMER affect Arduino programmers' sense of confidence in writing their own intermittent computing programs? All but one participant felt confident or strongly confident in their ability to leverage BOOTHAMMER to write their own intermittent computing programs. For likert statement (2) "I feel confident in my ability to write an intermittent computing program using BOOTHAMMER ", the average score was 4.6 and the lowest was neither agreeing or disagreeing that BOOTHAMMER made them feel more confident in their ability to write an intermittent computing program. Seven out of the nine participants offered comments on how simple and intuitive the use of the CHECKPOINT and RESTORE macros were. One novice participant expressed how intuitive the use of macros were despite not knowing their low-level functionality, stating how "... it looks very clean. Just two words that are easy to understand, and I'm just guessing its functionalities". One intermediate participant expressed that documentation was not even needed. Three participants (two intermediate and one novice) described their understanding of the RESTORE and CHECKPOINT macros to a surprising level of technical accuracy, going so far as to use the term "macro" in their interpretations.

On the point of macro syntax, however, two participants commented that they would feel more confident with supplemental documentation such as code examples and answers to frequently asked questions.

Participant interest in intermittent computing was moderate. The average score for likert statement (1) "I'm interested in writing my own intermittent computing program" was 3.5. Participant reasons for low interest in intermittent computing included the lack of motivation or need for its capabilities, barriers to retrieving collected data, and the preference for plug-and-play implementation. Participant reasons for high interest included potential applications in smart home devices capable of maintaining settings/configuration in the face of power failure.

Table 2. User study participants' use case descriptions.

Use Case	Participant Description
Longitudinal e-cigarette study	"... useful for long term usage pattern detection… to try to see how people are quitting smoking."
Persistent health device status	"... health related devices… remember their last status."
Elephant enrichment data collection in zoos	"... when they place their trunk in the hole, it could play a different tone… we need (the device) to be wired all the time… maybe we could get away with doing solar."
Persistent smart home device status	"In South Africa, we have bad power problems… (smart devices) would forget all this data… Knowing I could build projects that can actually just remember states would be so useful."
Lowering barriers for remote sensing	" I think this also lowers the barrier of entry for doing longitudinal testing in remote locations... for months even."
Saving game state	"let's say this controller, it could actually keep track of the save when the power goes out"
Musicians recording audio remotely	"There's also a lot of musicians that record samples and stuff out in weird places'.
Interactive environments	"you could set up a smart space where it can sense what you're doing or your emotional states and generate music throughout your house...if you could run those on this type of stuff, then you could save whatever data you're collecting."

RQ3: What use-cases can Arduino programmers envision for intermittent computing programs? While the average score to likert question (3) "I can imagine different use-cases for intermittent computing programs" was 3.7, participants were able to provide a range of use case descriptions from those that aligned with their areas of interest, sensors deployed in remote settings, smart home contexts, as well as more creative or unique applications such as saving state for video games, interactive environments, and even data collection for elephant enrichment programs within zoos. Full descriptions of all offered use-cases can be found in table 2.

While the results of this user study indicate that BOOTHAMMER can be an effective tool for introducing the maker community to the field of intermittent computing, we acknowledge two limitations that may have impacted our findings. First, the small sample size consisting of students limits the generalizability of our insights, specifically in regards to how non-academics and hobbyists may view the tool/intermittent computing concepts and possible applications. While the tool enables the creation of intermittent programs, some participants found it difficult to immediately apply theory to possible applications due to lack of domain familiarity and knowledge with intermittent computing.

6 DISCUSSION AND FUTURE WORK

6.1 Limitations

BOOTHAMMER is not without it's drawbacks. The cost of operating at such a low level is that serious code restructuring is out of the question. The ability to in-line a function or unroll a loop could be very useful in transforming a program to work better under intermittent conditions because it would make the execution more of a straightforward timeline. While this is not impossible to do at the assembly level, it not only makes much more sense to do at the compiler level because of extensive support for these modifications in tools like LLVM, but

also that this level of modification would require painful attention to address and register changes. Another major drawback of this approach is that global variables and dynamic allocation is not supported by the default FRAM library and that using the other library incurs a serious performance overhead. One of the major motivations for exploring assembly rewriting for intermittent computing was the anticipated performance benefits that assembly code would provide. While this was true to some extent, not nearly as much as we anticipated at the beginning of this project. Lastly, we had hoped to include BOOTHAMMER within the UF2 bootloader as an on-device code rewriting tool. In it's current state, BOOTHAMMER exists externally. A user must run BOOTHAMMER on a host machine and then flash the transformed hex file onto the target device. The major reasons for this decision include limited space available to the bootloader, significantly increase debugging difficulty, and ease of use. It makes more sense for BOOTHAMMER to slip into an existing build pipeline during compilation than expertly fit into the limited space afforded to the UF2 bootloader.

6.2 Beyond Arduino: UF2 Bootloader and other systems

One of the original goals laid out in section 3 was that BOOTHAMMER would live inside the UF2 bootloader and rewrite programs as they were flashed onto the device. While significant effort was undertaken to explore and modify the UF2 bootloader to make this possible, the engineering effort required to fit this tool within the very tight space requirements were outside the scope of this project and would work against making a properly function tool. In other words, it made more sense to write a strong binary rewriter than a poorly functioning bootloader. Still, that idea has crafted BOOTHAMMER into a lightweight and effective tool for rewriting a hex file and instrumenting checkpoint and restore operations. For the Cortex M0+ and other SAMD21 chips, the bootloader exists between address 0x0 and 0x2000. This amount of space is very limiting when you consider that the bootloader already takes up most of this address space. A quick exploration into the UF2 bootloader for SAMDX1 chips reveals that many features are turned off in order to fit within this relatively small address space. However, it is possible to extend the space allotted to the bootloader. This unfortunately adds some of its own challenges because the board libraries within the Arduino IDE are set to have the user code begin at address 0x2000(for the SAMD21 boards). Even though we did not end up putting BOOTHAMMER within the UF2 bootloader, the results of this paper show that the true important result is the instrumented user program and not a modified bootloader. Additionally, it may be more fruitful in the long run if this tool is added as an option in the compilation pipeline of the Arduino IDE. Users would be able to turn BOOTHAMMER on and off just as they could chose between Small, Fast, and Faster optimization settings. Regardless of the future of BOOTHAMMER, we have made a swift tool that lays a strong foundation for assembly rewriting for intermittent computing and hope to see this idea explored further as the field changes and shifts.

6.3 Learning from Intermittent Computing

There has now been an exploration into the lower end of the systems stack to democratize access to intermittent computing. When we look at this in the light of tools like MakeCode-Iceberg/Battery-free MakeCode that operate at a much higher end of the stack, we can much more broadly see not only the advantages and disadvantages of both, but more importantly how these tools might one day fit together into an intermittent computing toolkit. We could leverage the compiler to restructure code, identify dynamic allocation, changes in pointers, and global variables, and inform the lower level system where specific changes in branches and registers should occur or be saved. This entire toolkit could then be formatted to bridge the gap from those first learning to code to higher end researchers that wish to deploy intermittent systems in the wild. This intermittent computing toolkit, or even IDE, could have the tools to inform users how different regions of code as well as specific instructions affect their energy budget. This would ideally create a feedback loop that slowly builds a generation of energy-conscious programmers just the Arduino IDE did for maker and hobbyist programmers.

6.4 Community Tools and Hardware Support

BOOTHAMMER currently supports all ARM instructions that run on the Cortex M0+. While this covers all thumb instructions, it only handles a few of the 32-bit instructions and does not cover all possible ARM chips. However, the framework of this tool exists in such a way that future developers could build and add different instructions into it. For us, it does not make sense to support all possible instructions for each and every instruction set out of the gate. We hope that future developers see this as a call to action should BOOTHAMMER grow to help makers, hobbyists, and researchers begin writing their first intermittent computing programs. As mentioned back in section 5, this tool would heavily benefit from a PCB with an FRAM on board in order to reduce checkpoint overhead and remove the SPI bus as the current performance bottleneck.

7 CONCLUSION

This paper has presented BOOTHAMMER, a suite of tools and integrations for low-level (Assembly based) transformation of embedded systems programs to make them power failure proof— such that execution of the program can continue from where the program was exited due to a power failure. BOOTHAMMER is meant to fill a gap in older (but extremely popular) maker ecosystems like Arduino, which compile user code directly to executable binaries on hardware platforms. Additionally, BOOTHAMMER supports programmer guided checkpoint location annotations inside the Arduino IDE, reminiscent of trends in intermittent computing [27], so that programmers can highlight important or connected bits of code, self-organizing the program into chunks, that allow for much more efficient execution of the program. As BOOTHAMMER is open source and ready to be adapted and extended, we hope this suite of tools enables a broad spectrum of hardware ecosystems to be made useful for energy harvesting, and batteryless devices, via extension and engineering by the open source maker community. By finally allowing for intermittently powered Arduino development, we unlock a giant community of makers to pursue and expand conceptions of sustainable computing and a lower ecological impact from the Internet-of-Things.

ACKNOWLEDGMENTS

This research is based upon work supported by the National Science Foundation under award number CNS-2145584. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 2024. https://docs.arduino.cc/learn/programming/functions/
[2] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*.
[3] ARM. [n.d.]. ARM Architecture Reference Manual Thumb-2 Supplement. https://developer.arm.com/documentation/ddi0308/d/The-Thumb-Instruction-Set/Instruction-set-encoding?lang=en
[4] ARM. [n.d.]. ARM Architecture Reference Manual Thumb-2 Supplement. https://developer.arm.com/documentation/ddi0308/d/Thumb-Instructions/Alphabetical-list-of-Thumb-instructions
[5] ARM. [n.d.]. Cortex-M0+ Devices Generic User Guide. https://developer.arm.com/documentation/dui0662/b/The-Cortex-M0--Instruction-Set/Instruction-set-summary
[6] ARM. [n.d.]. GENERAL: Intel HEX File Format. https://developer.arm.com/documentation/ka003292/latest/
[7] ARM. 2021. White Paper: The economics of a trillion connected devices. https://community.arm.com/iot/b/internet-of-things/posts/white-paper-the-route-to-a-trillion-devices. [Online; accessed 02-March-2021].
[8] Hyuntae Bae and Youngsik Kim. 2021. Technologies of lithium recycling from waste lithium ion batteries: a review. *Materials advances* 2, 10 (2021), 3234–3250.
[9] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. Association for Computing Machinery, 105–116.

[10] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[11] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* (2015).

[12] Sophie Bauer. 2023. Explainer: The opportunities and challenges of the lithium industry. https://dialogochino.net/en/extractive-industries/38662-explainer-the-opportunities-and-challenges-of-the-lithium-industry/

[13] BBC. 2017. BBC micro:bit celebrates huge impact in first year, with 90% of students saying it helped show that anyone can code. https://www.bbc.co.uk/mediacentre/latestnews/2017/microbit-first-year. [Online; accessed 02-March-2021].

[14] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 209–220.

[15] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. 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*. 767–781.

[16] Jasper De Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–34.

[17] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *ACM SIGPLAN Notices* 53, 6 (2018), 19–30.

[18] Josiah Hester and Jacob Sorber. 2019. Batteries not included. *XRDS: Crossroads, The ACM Magazine for Students* 26, 1 (2019), 23–27.

[19] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–13.

[20] Mitch Jacoby. 2019. It's time to get serious about recycling lithium-ion batteries. https://cen.acs.org/materials/energy-storage/time-serious-recycling-lithium/97/i28

[21] Junsu Jang and Fadel Adib. 2019. Underwater backscatter networking. In *Proceedings of the ACM Special Interest Group on Data Communication*. 187–199.

[22] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020. BFree: Enabling Battery-free Sensor Prototyping with Python. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 4 (2020), 1–39.

[23] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. 2022. WARio: efficient code generation for intermittent computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 777–791.

[24] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 85–99.

[25] Christopher Kraemer, Amy Guo, Saad Ahmed, and Josiah Hester. 2022. Battery-free makecode: Accessible programming for intermittent computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 1 (2022), 1–35.

[26] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Intermittent learning: On-device machine learning on intermittently powered system. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 4 (2019), 1–30.

[27] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices* 50, 6 (2015), 575–585.

[28] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* (2017).

[29] Kiwan Maeng and Brandon Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1005–1021. https://doi.org/10.1145/3385412.3385998

[30] Microchip. 2021. Low-Power, 32-bit Cortex-M0+ MCU with Advanced Analog and PWM 25.6.2.3 Clock Generation – Baud-Rate Generation.

[31] Microsoft. 2020. MakeCode: Hands on computing education. https://www.microsoft.com/en-us/makecode. [Online; accessed 02-March-2021].

[32] Nature. 2021. Lithium-ion batteries need to be greener and more ethical. https://www.nature.com/articles/d41586-021-01735-z. [Online; accessed 16-November-2021].

[33] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. 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*. 159–170.

[34] Saul Rodriguez, Stig Ollmar, Muhammad Waqar, and Ana Rusu. 2015. A batteryless sensor ASIC for implantable bio-impedance applications. *IEEE transactions on biomedical circuits and systems* 10, 3 (2015), 533–544.

[35] FUJITSU SEMICONDUCTOR. [n.d.]. 4 M (512 K x 8) Bit SPI MB85RS4MT.
[36] Anselm Strauss and Juliet Corbin. 1994. Grounded theory methodology: An overview. (1994).
[37] Jie Sun, Jigang Li, Tian Zhou, Kai Yang, Shouping Wei, Na Tang, Nannan Dang, Hong Li, Xinping Qiu, and Liquan Chen. 2016. Toxicity, a serious concern of thermal runaway from commercial Li-ion battery. *Nano Energy* 27 (2016), 313–319.
[38] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R Smith. 2017. Battery-free cellphone. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 2 (2017), 1–20.
[39] TI. [n.d.]. MSP430FR5969. https://www.ti.com/product/MSP430FR5969
[40] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*.
[41] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 41–53.