Battery-free MakeCode: Accessible Programming for Intermittent Computing

CHRISTOPHER KRAEMER, Northwestern University
AMY GUO, Northwestern University
SAAD AHMED, Northwestern University
JOSIAH HESTER, Northwestern University

Hands-on computing has emerged as an exciting and accessible way to learn about computing and engineering in the physical world for students and makers of all ages. Current end-to-end approaches like Microsoft MakeCode require tethered or battery-powered devices like a micro:bit, limiting usefulness and applicability, as well as abdicating responsibility for teaching sustainable practices. Unfortunately, energy harvesting computing devices are usually only programmable by experts and require significant supporting toolchains and knowledge across multiple engineering and computing disciplines to work effectively. This paper bridges the gap between sustainable computing efforts, the maker movement, and novice-focused programming environments with MakeCode-Iceberg, a set of compiler extensions to Microsoft's open-source MakeCode project. The extensions automatically and invisibly transform user code in any language supported (Blocks, JavaScript, Python)into a version that can safely and correctly execute across intermittent power failures caused by unreliable energy harvesting. Determining where, when, and what to save in a checkpoint on limited energy, time, and hardware budget is challenging. We leverage the unique intermediate representation of the MakeCode source-to-source compiler to design and deploy various checkpointing techniques. Our approach allows us to provide, for the first time, a fully web-based and toolchain-free environment to program intermittent computing devices, making battery-free operation accessible to all. We demonstrate new use cases with multiple energy harvesters, peripherals, and application domains: including a Smart Terrarium, Step Counter, and Combination Lock. MakeCode-Iceberg provides sustainable hands-on computing opportunities to a broad audience of makers and learners, democratizing access to energy harvesting and battery-free embedded systems.

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, Amy Guo, Saad Ahmed, and Josiah Hester. 2022. Battery-free MakeCode: Accessible Programming for Intermittent Computing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 1, Article 18 (March 2022), 35 pages. https://doi.org/10.1145/3517236

1 INTRODUCTION

Physical-computing platforms like Arduino, CircuitPython, and MakeCode are integral in teaching computational thinking to novice programmers, e.g., elementary or middle school students [10, 16, 30]. Coupled with devices

Authors' addresses: Christopher Kraemer, kraemer@u.northwestern.edu, Northwestern University, Evanston, Illinois; Amy Guo, AmyGuo2023@u.northwestern.edu, Northwestern University, Evanston, Illinois; Saad Ahmed, saad.ahmed@northwestern.edu, Northwestern University, Evanston, Illinois; Josiah Hester, josiah@northwestern.edu, Northwestern University, Evanston, Illinois.

2474-9567/2022/3-ART18 $15.00
https://doi.org/10.1145/3517236

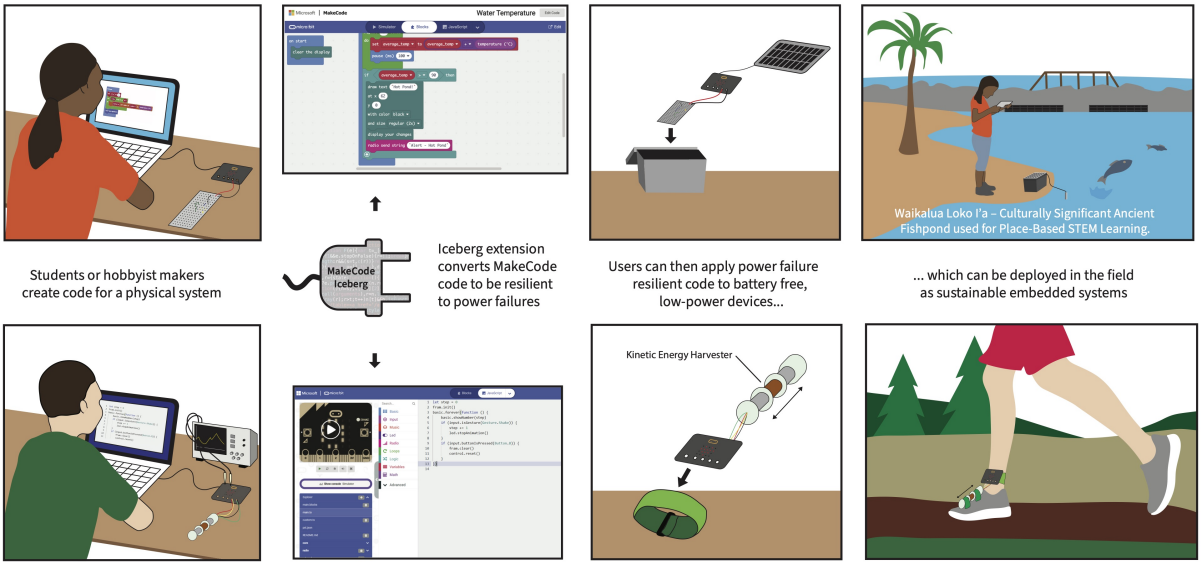


Fig. 1. The two broad motivating use cases of MakeCode-Iceberg: *makers*, and *physical computing education*. On Top, a K-8 student in a classroom can use existing Makecode online infrastructure to program an environmental monitoring application in Blocks. With the Iceberg extension, students do not need to modify their programs in any way, yet can use energy harvesters instead of batteries (i.e., a solar panel). On Bottom, a maker can build and prototype interesting energy harvesting devices rapidly in JavaScript or Python, without the need for a compiler or complex development environment. In both cases, the programmer can program the device from their web browser, connect energy harvesters, and deploy these devices in the wild or on-body perpetually.

like the micro:bit, Arduino Uno, and AdaFruit feather, these all-in-one systems provide a gentle yet exciting introduction to computing in the physical world [4, 6]. Beyond pedagogy, these systems have enabled the rise of the "maker" movement, where non-expert programmers use these platforms to build any physical computing idea they can imagine.

Unfortunately, these devices are typically powered by batteries or tethered to the wall, which requires human intervention for maintenance when deployed in the wild or limits them to the length of the power cable from the power outlet. These restrictions limit students' learning opportunities and decrease makers' reach and creativity for ideas. Worse, the dependence on batteries only further embeds the ongoing sustainability crisis into curricula and practice. Billions of battery-powered (even rechargeable) devices result in billions of dead batteries, creating a negative environmental impact after disposal.

The high maintenance and environmental cost of batteries lead to concerns about the wireless sensor networks domain [20, 32, 33]. This sustainability issue has led to battery-free embedded devices powered by ambient energy sources (vibrations, radio frequency transmissions, and light). These battery-free devices are likely to form the future of physical-computing devices and the Internet Of Things [21, 31, 53] due to being maintenance-free and enhancing long-term deployment. Recent battery-less device demonstrations include phones [58], satellites in space [15], implantables [56], devices conducting machine learning [35], handheld gaming consoles [17], and even underwater sensing [28]. Energy harvesting opens up new application ideas for makers and new educational opportunities for students, both thinking about sustainability: a solar-powered environmental monitoring system

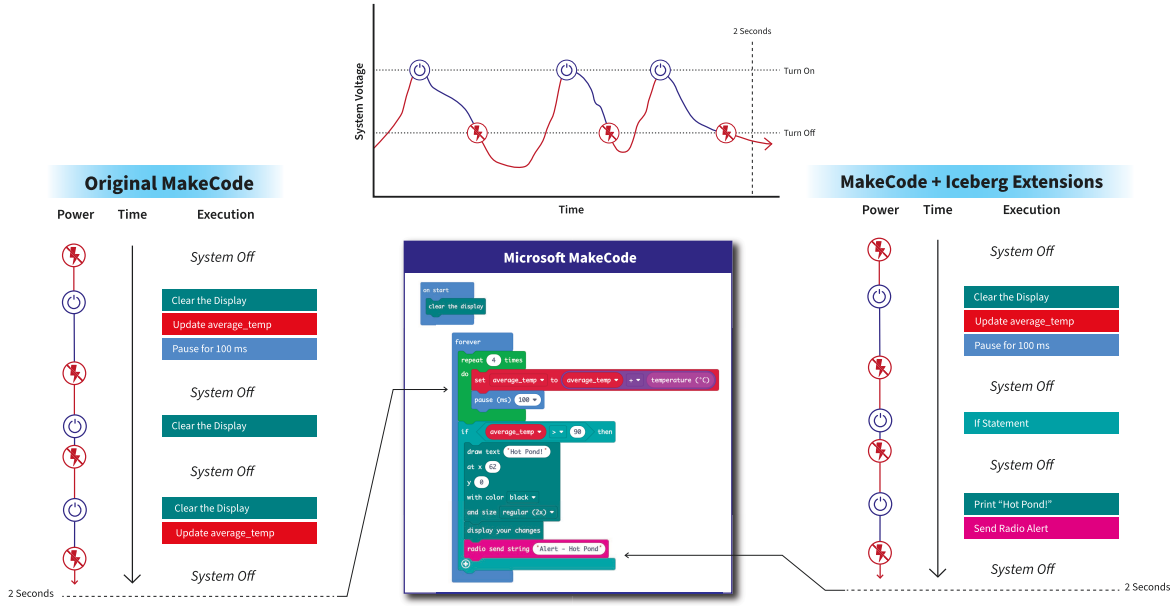


Fig. 2. Figure shows an illustration of how variable energy affects execution of MakeCode applications. Power failure can occur at any time instance and at any line of code.

for the school greenhouse or a motion-powered step counter for use during physical education class. We describe two motivating use cases for our two targeted populations, students and makers, in Figure 1.

Energy harvesting is unpredictable, so programs running on battery-free devices may fail at any point [11, 36]. The energy harvested is rarely enough to keep the device continuously powered. This energy unpredictability results in inconsistent CPU, memory, and peripheral states [3] and bugs [14] that are not present in the regular, continuous execution of the same program. Programming energy harvesting devices is especially hard, as the programmer must consider how to safely, quickly, and correctly checkpoint state before a power failure and then restore that state once energy has returned. Figure 2 shows why an application programmed in a block-based language like Microsoft's MakeCode would continually restart from scratch because of power failures, never accomplishing anything.

The field of *intermittent computing* has arisen to address this problem: developing systems, architecture, programming models, and tools for masking or handling these power failures gracefully, allowing programs to execute across power failures seamlessly, with reduced user burden. However, except for the BFree system [33] (which provides a power failure resilient Python runtime), all existing solutions are generally targeted at expert or near expert users who program in C/C++. Worse, developers must use specialized hardware platforms, complex toolchains, and software that require in-depth knowledge of tools like LLVM, Make, and GCC that are far from friendly to the non-expert and novice. Lack of access to powerful enough systems to run complex toolchains compounds these barriers to entry into intermittent computing: low-tier Chromebooks used in schools may not run sophisticated LLVM transformations required to compile a failure resilient C/C++ program.

This paper bridges the accessibility gap for intermittent computing by developing source code transformation extensions for the popular Microsoft MakeCode [47] online IDE and compiler. MakeCode has lowered the barrier to entry to embedded systems for students and makers by providing a web-based multi-language (drag-and-drop

Blocks, JavaScript, Python) development environment for embedded systems. Its exciting demo applications, compatibility with a wide range of physical-computing devices, and in-browser compilation (no installation needed) make it a popular choice by experts, students, and makers alike. We conduct a design space exploration of intermittent computing to understand the best combination of state-of-the-art intermittent computing techniques to enable this new functionality.

1.1 Motivating Use Cases

Figure 1 shows our two motivating use cases, (1) enabling young students, often Indigenous, motivated by sustainability, conservation, and resource management, to explore sustainable computational solutions for these areas via energy harvesting powered physical computing devices. We work with teachers at one such school, a public and bi-lingual school in the US serving the Indigenous population. We envision students deploying water monitoring sensors and using this data to learn about the water cycle and habitat and introduce interventions to heal or protect it. Our second use case focuses on makers (for example, building a motion-powered step counter) similar to the BFree system [33] published at UbiComp'2021, which enabled makers to use Python to program energy harvesting battery-free devices. MakeCode-Iceberg has a similar motivation but has multiple advantages: it does not require the creation or fabrication of a custom board to work (unlike BFree), does not require installing anything, and supports multiple languages, not just Python.

1.2 Contributions

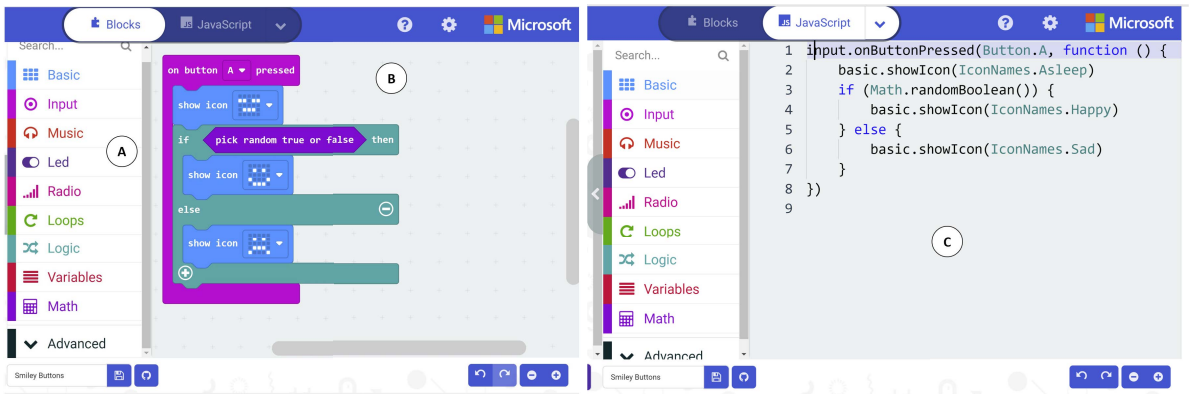
Figure 2 shows the fundamental problem of power failures in MakeCode programs (Block based or otherwise). We build a set of extensions to the core source-to-source compiler of MakeCode, MakeCode-Iceberg, that *democratizes access to participation in the battery-free energy-harvesting future of ubiquitous computing*. Our contributions lie in applying intermittent computing techniques to a novel space, an online source-to-source compilation framework. In the design space exploration, we target the best techniques for efficient online compilation of power failure resilient MakeCode programs in any language supported (Blocks, JavaScript, and Python). Our code transformations occur within the compiler's middle-end and modify the Intermediate Representation(IR), making them hardware and language independent. We make the following specific contributions:

(1) We develop an IR and middle-end approach to allow for multi-language development (Javascript, Python, Blocks) in an online environment generating embedded firmware that is hardware independent and resilient to power failures caused by energy harvesting. This IR approach allows any MakeCode supported device to become energy harvesting.
(2) We instantiate our extensions in the popular micro:bit version of Microsoft MakeCode, without requiring any external custom hardware or modifications to the core development workflow.
(3) We demonstrate the extensions' usefulness via four different use cases with nine peripherals and two energy harvesters, running on unmodified micro:bits with a breadboard and standard components purchasable from hobbyist maker stores (SparkFun and Adafruit).
(4) We release MakeCode-Iceberg as an open source extension of Microsoft MakeCode[1].

2 PROGRAMMING BATTERY-FREE PHYSICAL-COMPUTING DEVICES

Physical-computing devices form the core of modern Internet-of-Things platforms due to their pervasive and ubiquitous nature. They seamlessly integrate with the physical world, and more than one trillion devices will likely be deployed by 2035 [5]; currently, rechargeable batteries are likely to power them all [49]. Batteries are worrying as they have limited power cycles before they need to be replaced, resulting in trillions of dead batteries that will only add to worldwide e-waste concerns. To address this, a paradigm shift in embedded computing

[1]Source code, documentation, and demo of MakeCode-Iceberg available at https://github.com/ka-moamoa/makecode-ic.



(a) Block-based (b) Text-based

Fig. 3. The MakeCode Web Application

has begun: moving from batteries to energy harvesters (solar panels, etc). The question remains: can anyone participate in the development of these devices? Or will this only be in the purview of expert computer scientists? This section reviews novice programming environments and systems and the main challenges keeping them from supporting batteryless devices.

2.1 MakeCode: Accessible Computing for Novices

Microsoft MakeCode [47] is a platform that offers a visual drag-and-drop interface to program physical devices. It uses a block-based programming model which has shown promise for getting students excited about programming and increases learning gains [61].

2.1.1 Visual & Block-based Programming: Inspired by Blockly [51] and Scratch [50], MakeCode uses a block-based programming language, exposing instructions using a block palette where users can drag and drop blocks to build their programs. Each block is conceptually organized and color-coded in the palette, as shown in sections A and B of Figure 3a. Each block represents a programming construct, e.g., loops and variable declaration. One can easily switch between block-based and text-based programming, providing an easy progression from visual block-based to text-based programming, as shown in pane C of Figure 3b.

2.1.2 Web-based Platform: MakeCode is a web-based IDE, requiring no installation or any prior knowledge for a user to get started [48]. A user can plug in a micro:bit (or other supported device) to any computer, and link it with the web browser application. Binaries are produced from user code, that can run on any supported platform (like the micro:bit). MakeCode also provides a simulator and a debugger to help novices fix errors in their applications. Users can make extensions to existing blocks that allow them to build proof-of-concepts for almost any idea before deploying the hardware. Because of this accessibility, MakeCode has seen broad uptake among classrooms.

2.1.3 Micro:bit: With over five million users, the micro:bit (see Figure 4) is the most extensively used platform for teaching computational and digital thinking skills [46]. It includes a temperature sensor, accelerometer, touch sensor, LEDs, BlueTooth Low Energy, and more, which can help novices develop exciting applications such as games, digital watches, and musical instruments. Its low price ($20 USD) and easy to deploy accessories have made it a go-to novice platform capable of running complex applications.

Table 1. Comparison between existing state-of-the-art programming models and MakeCode-Iceberg. C/C++ variants of intermittent computing programming models are not novice friendly. On the other hand, BFree [33] relies on custom hardware and a specialized runtime to offer hobbyist a platform to run Python under intermittent supply and is slow as Python is interpreted on-device. MakeCode-Iceberg has no custom hardware requirement, and requires no custom runtime, it can work with any platform supported by MakeCode. [✗: Does Not Satisfy Criteria. ✓: Partially Satisfies Criteria. ✓: Satisfies Criteria.]

	C/C++ w/ Checkpoints	**C/C++ w/ Tasks**	**Python [33] (Interpreted)**	**JS/Python/Blocks MakeCode Transpiler**
Well known language(s)	✓	✓	✓	✓
Ease of programming	✗	✗	✓	✓
No rewriting required	✗	✓	✓	✓
Novice friendly	✗	✗	✓	✓
Hardware independent	-	-	✗	✓
Fast and efficient	✓	✓	✗	✓

2.2 State-of-the-Art Programming Techniques for Batteryless Systems

Intermittent computing [21] refers to the collection of techniques that allow for seamless, consistent, and efficient recovery from power failures due to unpredictable energy harvesting. Table 1 shows three different approaches (along with our approach on the right) for programming intermittently powered devices.

The *checkpoint-based programming* model [2, 12, 34, 55] saves checkpoints of program state as the program executes, and restores from the most recent checkpoint after a power failure. Most approaches instrument bare C at compile-time[2, 12, 55],and make decisions on when to checkpoint at runtime [59]. Other approaches monitor supply voltage and save state just-in-time before a power failure [8, 9]. All of these approaches require little to no intervention from the programmer, reducing the effort of porting existing code. However, C/C++ is not known for being novice-friendly and requires programmers to have in-depth familiarity with computer organization concepts like memory, addressing, and pointers.

The *task-based programming* model [7, 13, 23, 37, 39–41, 64] wraps C/C++ code into atomic, independent "tasks" that are connected together into a task graph to form a program. For example, sensing ten accelerometer readings might be one task, with the output sent to another task that averages the readings. After executing each task, the data is committed to non-volatile memory, where any subsequent tasks can read it. With this model, checkpointing approaches are simplified, as only the memory between tasks is saved. While task based methods are often speedier, special attention is required by the programmer while defining the size of each task. The application can get stuck and not terminate at any point if the energy required to execute a task becomes greater than the energy buffered in the capacitor [14]. This challenge requires the programmer to know about the underlying hardware to execute the application efficiently, which is impossible for novices.

The *interpreted language* route, relies on a C/C++ runtime to interpret and execute a higher level language, such as Python. BFree [33] provides the first interpreted language that can run on intermittent devices, by porting CircuitPython. 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 accessibility of the system. In contrast, MakeCode does not have these issues despite offering a higher level of abstraction due to its source-to-source compilation.

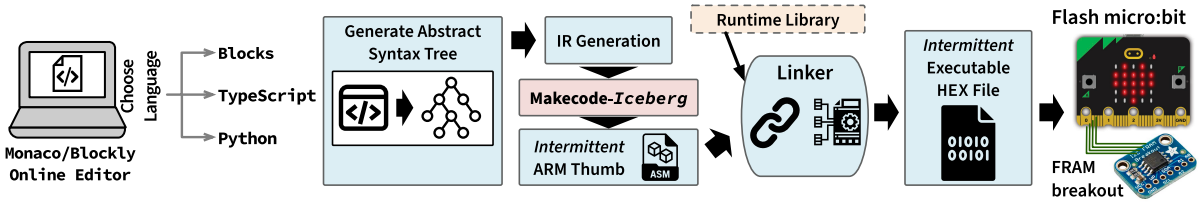


Fig. 4. Shown is the Makecode system [18] and components, along with the location of the Intermediate Representation (IR) level additions that our system targets. Makecode uses source to source compilation to support multiple high level languages that are eventually emitted as ARM assembly that is compiled online and flashed onto a microcontroller that has an existing Makecode bootloader (such as the micro:bit). An FRAM breakout from AdaFruit is used as checkpoint storage.

The fourth, right most category, the primary contribution of this work, contributes the first *source-to-source compiler for batteryless systems*, as shown in Table 1. This category marrys 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.

3 MAKECODE-ICEBERG: AN OVERVIEW

MakeCode-Iceberg is a set of extensions to the MakeCode source-to-source compiler, in the middle-end, that allows novices to program intermitently powered embedded computing systems in an online, user friendly environment. It supports any programming language supported by original MakeCode, including Blocks, Python, and JavaScript. MakeCode-Iceberg has the following design goals.

(1) **Energy Harvesting.** Leave batteries behind and survive exclusively on harvested energy.
(2) **Accessible.** Make it as easy as possible to get started with intermittent computing. MakeCode allows significant access because it is online first; no toolchains or compilers must be installed. We must embody this concept and make sure any additions we make will not require users to build their own hardware or change their code beyond importing an extension.
(3) **Low Added User Burden.** Developing programs with MakeCode-Iceberg will be the same as with the unmodified MakeCode; users will not have to learn new APIs, understand the intricacies of energy harvesting, or dive deep into checkpointing and memory consistency.
(4) **Failure Proof and Correct Forward Progress.** Despite frequent power failures due to dynamic energy input, programs will make progress. This progress will be correct, as in, just the same as if continuously powered. This guarantee will enable long device operation in the field and reliable devices.
(5) **Hardware Independence.** With millions of users, MakeCode is compatible with a comprehensive range of Maker platforms available off-the-shelf. Any system support needed to enable intermittent computation must be independent of the underlying hardware to function across all platforms properly.

Figure 4 shows different MakeCode components and how they interact to generate the executable file for the platform. The accessibility of our approach is: the user purchases a solar panel, breadboard, micro:bit, FRAM breakout, and some jumper cables from Adafruit, plugs their micro:bit into a library or school computer, accesses MakeCode-Iceberg online, and programs their application. The entry fee to battery-free sensing, and intermittent computing, is $50 or less. Before we discuss our modifications to MakeCode's system, the following is a brief overview of how the existing MakeCode system works.

3.1 MakeCode Internal Compiler Operation

Microsoft's MakeCode comes with an online editor, the front end, which allows users to program using Blocks, JavaScript, or Python. The front end translates this user code into an abstract syntax tree(AST) and then into an Intermediate Representation(IR). The IR within MakeCode serves many different functions within the editor. It first serves as a common language to translate between each of the user-level programming languages. For example, if a user created a program in Blocks but wanted to switch to programming in JavaScript, the Block code will be translated down into the IR and then up to JavaScript. The IR is also used to output code to the JavaScript simulator within the editor. The IR is made up of a list of procedures. A procedure is essentially a function at the IR level and contains a list of statements. Each statement is an expression defined by the corresponding language's grammar in which the program was written. If the user wanted to download code to their Micro:bit, the IR is given directly to the compiler's backend, where it would be translated into ARM thumb assembly.

3.2 Key Insight: Modify MakeCode Middle-End to Support Automatic Checkpointing

The Intermediate Representation, or IR, is an in-between for higher-level programming languages and assembly code, it is generated from the front-end compilation pass. The back-end is the part of the compiler which generates the machine or assembly code. By adding instrumentations in the middle-end of the compiler, operating only on the IR, we remain hardware agnostic and do not impose any requirements on software. An alternative to this approach is to modify the runtime on the embedded device itself, similar to BFree. However, each new target device would need a different runtime, that calls hardware specific actions. Our approach is to remain at an abstraction level above the hardware. So long as a compiler back-end exists to transform the IR into a given assembly language, the IR is device agnostic, and each output assembly will see modifications to the IR. Utilizing the IR is a supply chain approach to intermittent computing. Instead of targeting each specific output device and having to tailor-make the intermittent computing instrumentation, targeting the common point of reference means that a single change is seen in many different devices.

Modifying the IR allows syntactic "tricks" for checkpointing. While modifying the assembly language gives the user access to registers and other valuable features for preserving a program's state, the higher level of abstraction in the IR can restore these options "in the background." There is no need to save the entire contents of a device's RAM when the code can be modified to build it back automatically. Say we are trying to checkpoint at different iterations of a loop. The low-level approach needs to save the entire stack, the registers, and at least the memory region necessary for the current function. The IR approach only needs to save the variable state and a path back to that position in the loop. By following that path set by the instrumentation, the program will build back the necessary state saved by the low-level approach. This approach will significantly reduce the state needed to save to non-volatile storage. This claim does not say that having access to the stack and RAM is useless for checkpointing. Low-level instrumentations may have an easier time checkpointing the state of peripherals than the IR approach. *The IR intermittent computing instrumentation exchanges very low-level hardware granularity for a high level of insight into the user's program and code that is hardware independent.*

3.3 MakeCode-Iceberg: Compiler Extensions for Energy Harvesting Targets

The point in-between the completion of IR generation and translation to ARM thumb is the exact point where MakeCode-Iceberg is placed in the system, as shown in Figure 5.

Figure 4 shows the existing MakeCode compilation process [18]. A critical design decision for MakeCode-Iceberg is to keep the intricacies of intermittent computing oblivious from the end-user to ensure ease of use and wide-scale adoption. For this purpose, we target the middle-end of MakeCode's compilation process while keeping the existing block APIs and definitions of MakeCode unmodified. The Intermediate Representation(IR) of the code is at the right level of abstraction i.e. neither tied to the hardware nor the programming language.

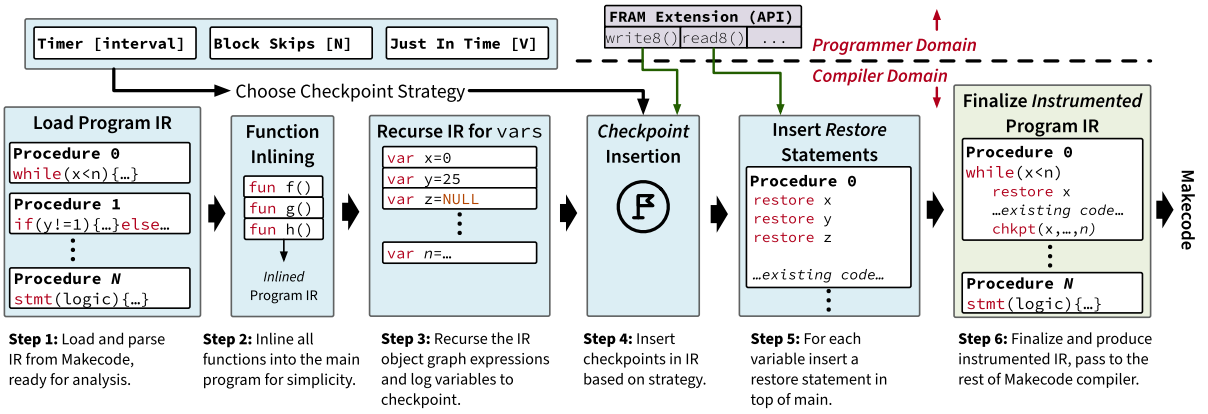


Fig. 5. Shown is our Intermediate Representation (IR) level addition to the Makecode system, which inserts checkpoint/restore functions for intermittent operation at the IR level, leveraging a programmer level extension which provides an API for read/write to FRAM non-volatile memory on a dedicated breakout board from AdaFruit. This approach is hardware independent, ensuring that any microcontroller supported by Makecode, can be used with our additions.

IR modification ensures that we achieve hardware independence (Goal 5) without modifying the programming workflow of MakeCode (Goal 3). We also transform the existing IR of the program to insert checkpoint and restore routines to ensure fault tolerance (Goal 4). This design also makes our system invisible to the user, i.e., the user does not need to specify where to place checkpoint and restore routines in the program to ensure correct program execution. Our modifications automatically configure all settings related to checkpointing and restoring (Goal 3). Afterward, the transformed IR is assembled in a binary file, ready to be deployed on the device, thus *requiring no user intervention at any programming stage*, or any extra hardware or runtime systems (Goal 2). There are several challenges attached to working at the IR level.

- **No Stack Frame Access:** In order to correctly resume application execution, conventional checkpointing routines need to access the system stack to copy arguments and the callee functions onto the non-volatile memory. This information is hardware dependent and varies from platform to platform.
- **Function Argument Tracking:** In order to keep track of the changes in the global state, there is a need to track arguments being passed to each function. There is a possibility where a reference to a global variable is being passed to a function and its value gets modified using a local variable.
- **Peripheral State:** Peripherals often need to be reinitialized after a power failure. In the case of an external LED screen, the current display should be preserved in a checkpoint. Unfortunately, the IR does not have access to low-level hardware components like the internal memory of a peripheral.

We address these challenges by function in-lining and mapping arguments to predefined, custom variables. These custom variables take the place of the arguments in the body of each in-lined function. At the top of the body of the in-lined function, each argument value in the function call is assigned to one of these custom variables. These custom variables are global, meaning they are added to the list of variables that are checkpointed. This change allows MakeCode-Iceberg to checkpoint within the body of an in-lined function. In terms of checkpointing peripherals, MakeCode-Iceberg does not automatically restore their state. Instead, users can use variables that are automatically checkpointed to preserve the state of a peripheral device. While this approach does not work for all possible peripherals, many of the onboard components of the micro:bit do not require a complex reinitialization and is adequate for the demonstrations in Section 5.

4 DESIGN AND IMPLEMENTATION

MakeCode-Iceberg is a compiler modification to an online coding platform. As seen in Figure 4, MakeCode-Iceberg is situated in the middle of the *user code* to *micro:bit hex/binary* pipeline. Figure 5 illustrates the pipeline of MakeCode-Iceberg itself, highlighting the critical aspects of how it works. We built an extension for AdaFruit FRAM breakout boards that provides an interface to save and restore data to the non-volatile FRAM. This extension is compatible with regular MakeCode, loadable in the online web application by simply inserting the GitHub repository URL. Different settings are available within the FRAM extension to adjust how the compiler works for debugging and fine-tuned control. These settings turn on and off intermittency and checkpoint optimizations and settings for those optimizations. The compiler transformations of MakeCode-Iceberg assume the user has loaded the FRAM extension, as the transformations auto-generate code that saves checkpoints and restores them to and from the FRAM breakout.

The general process of the transformations is the following: First, functions are inlined to ensure the program has a neat, sequential timeline that it can follow. Next, the program is analyzed to determine what state needs to be saved and the proper instructions to accomplish that are built (calling the FRAM extension APIs). Lastly, Iceberg performs a final pass on the program to insert the checkpoint and restore operations.

4.1 Compiler Pass Overview

We modify the existing IR generated by MakeCode, using multiple compiler passes, to retain a program's state despite intermittent power failures. One pass handles all issues related to function inlining and another handles checkpointing. Together, the compiler accomplishes the following major tasks:

(1) **Function Inlining:** Nested function calls make checkpointing much more difficult as it is difficult to track the stack pointer and contents in the middle-end as functions are called. For example, restoring a checkpoint that occurred in a nested function call would require saving the stack or path back to the checkpoint. Unfortunately, the MakeCode middle-end does not have access to a stack abstraction, and "jumping" back to the checkpoint requires knowledge of every possible path to that checkpoint. MakeCode-Iceberg instead performs a pass on the `main` procedure and inlines all functions the user has written, excluding any calls to external libraries or extensions. Function in-lining transforms a potentially complex control flow graph for checkpoint/restore into a simple, linear timeline.
(2) **State Analysis:** MakeCode-Iceberg manages the state of the program via preserving the state of the global variables of the program across power failures. This is possible since MakeCode elevates (nearly) all variables to global [18] in its transformations. Each checkpoint has an assigned ID, which allows for restoration by ID on boot. All global variables that are modified after their initialization are added to a list of variables to checkpoint.
(3) **Instruction Building:** All variables added to the checkpoint list will need to be saved to non-volatile storage. MakeCode-Iceberg handles this by composing calls to the FRAM driver for each variable in the checkpoint list. These calls are built to make a variable's position in FRAM static, meaning its address never changes. This design allows a read and a write to FRAM to refer to the same variable via addressing.
(4) **Checkpoint Insertion:** Checkpoints contain the entire checkpoint list and may be wrapped in an optimization for faster execution. Checkpoints are inserted at the end of a basic block that modifies the program's state. A restore operation is inserted at the beginning of the program to load a checkpoint if one is present.

An example code being transformed is shown in Figure 6. We refer to the figure throughout.

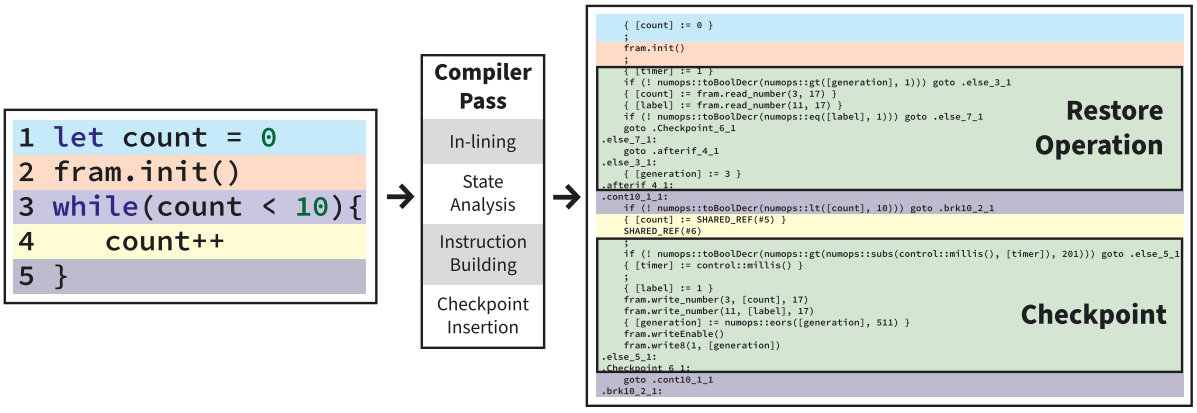


Fig. 6. Code to IR representation of how MakeCode-Iceberg transforms a program. First the user's program is translated in the IR. The system then adds checkpoints if a block contains variables that need to be checkpointed.

4.2 Compilation Step #1: Function Inlining

A function-less program is much easier to checkpoint than one with nested functions that complicate the stack and call graph. The function that performs the inlining is called `flatten`, which begins by iterating through the body of the primary procedure, also known as the "on start" block in MakeCode, and continues until it finds a procedure call. A procedure is the IR version of a function, and they occur either on their own or part of another line of code. In both cases, `flatten` begins by finding the procedure the user's code refers to and then adding it to a list. A unique sequence number identifies all procedures in this list. Any procedure call will contain that identifying sequence number within the IR instruction's metadata that describes the instruction. `flatten` will ignore any sequence numbers corresponding to extensions or custom files procedures. We reduce code bloat by only transforming the user's program while capturing the relevant state. Once `flatten` retrieves a sequence number from a procedure call, the called procedure is found in the list of sequence numbers, and the contents of that procedure are copied into main.

Objects within Typescript, the JavaScript subset language used to build MakeCode, is pass by reference. Passing the object itself means deep copies of the IR instructions must be made when inserted into main. If not, any changes made to one instruction will be reflected in all of the others. This challenge is particularly problematic because individual instructions need to be modified to replace the uses of argument variables with our custom global variables. Without this replacement, the argument would be undefined, and the function's state would not be captured. Deep copies are made by building a new object with identical data to the original. The inline function is inserted just before the original call to that function. This position was chosen just in case a return value must be calculated beforehand.

A widespread setup for embedded devices is to have `onStart` and `Forever` functions, as seen in Arduino, and MicroPython. The code in the `onStart` occurs at the start of the program and the `Forever` loops continuously once the `onStart` ends. MakeCode does the same, and MakeCode-Iceberg provides support for this. The `onStart` function is the main procedure within the MakeCode IR and calls a runtime function to set up the code in `Forever` to indefinitely loop. While MakeCode-Iceberg does not yet support MakeCode's basic form of parallelism through fibers(non-preemptive threads), and the `Forever` loop does fit into this category of parallelism, it can be faithfully recreated with a while(true) loop in the main procedure. When inlining a `Foreverloop`, a `ForeverTop` label is first inserted to denote the function's beginning and replaces the original call. Then the code from the body of

the original `Forever` loop is inserted into main. A goto instruction to the `ForeverTop` label is then inserted at the bottom. This causes the code to loop indefinitely, and this code block is compatible with the checkpointing transformations.

Once an identical copy of the function has been inserted into main, fixing the arguments and the return value begins. Take, for example, a simple function that takes in two arguments, `a` and `b`, and returns their sum. The original call contains the actual values of `a` and `b`. The new instructions are made and inserted at the top of the inlined function, whether a variable or a number. These new instructions assign the input values of the function call to MakeCode-Iceberg's custom variables. These values are then used to replace all uses of `a` and `b`, respectively. If the original call was used to get a value, then the return value of that inline function takes the place of the function call.

As a post-step to `flatten`, labels copied over are modified to reflect their new positions in `main`. For example, if an in-lined function contains an `if/else` statement and is called twice, two copies of the same label will be present in main. In this case, the `if` statement would have multiple labels of the same name to jump to, leading to problematic or undefined behavior. This problem is fixed by looping through `main`, identifying which labels correspond to which blocks of code, and adjusting label names accordingly so that no two labels correspond to the same jump instruction. Jump instructions are used to build loops and `if/else` statements. All loop labels in MakeCode come with unique IDs, and these are used to identify which labels to change. `if/else` blocks are easily identified by `else` and `afterif` prefixes to their labels. A mapping is made between the old and new names of the label. All `if` statements in the MakeCode IR contain the branch condition and a jump to the corresponding `else` label. The `else` label is read from this instruction, a new label is made and takes its place, and then MakeCode-Iceberg iterates through `main` and replaces the old `else` label when it finds it. This same process works for the `afterif` labels, which allow MakeCode to skip over an `else/if` or `else` statement when the `if` condition is hit.

4.3 Compilation Step #2: State Analysis

In the analysis step, the compiler performs a pass to determine which variables need to be checkpointed and which do not. The pass determines this by looping through the list of global variables and seeing which ones are modified at least twice (i.e., non-constants). The first modification is the initialization, and anything after that indicates some state change to the program. The MCU state retention only pays attention to global variables because MakeCode already promotes most local variables to global ones. By promoting most variables to global, the critical state of the program can be reasonably captured by checkpointing the global variables that have a modification later in the program. MakeCode-Iceberg currently only does this for numeric variables, since strings are not useful for most purposes on embedded platforms. However, a simple hashing mechanism or convert-to-array procedure could enable this.

4.4 Compilation Step #3: Instruction Building

Once the list of variables that need to be checkpointed is made, the compiler moves into step three of MCU state retention: Setup. The setup step involves building all of the necessary IR instructions used in the transformation step. Setup's job is to build the appropriate FRAM write and read calls (as shown in step 4 of Figure 5). Each variable has its own slice of the FRAM, so values do not get mixed when the program restores its state. The read and write calls interact with the custom FRAM driver added as a custom extension to the MakeCode-Iceberg project. A write IR instruction is a call to the function within the driver called `writeNumber()`. This `writeNumber()` function takes a reference to the variable, the address the compiler has given it, and the total length of the FRAM address space used to create the double buffer system described in the next section. The compiler determines each address by taking the order in which the variables occur in the list of variables to checkpoint and multiplying it

by four to give it enough bytes in the FRAM. We give each variable four bytes because MakeCode's "number" type is 32 bits. For example, the generation is at address zero in the FRAM, meaning the first variable is at address one, and the second is at address five. The write function needs to know the entire length of the FRAM checkpoint space to write to both buffers in the double buffer system. The driver itself handles the alternation between buffers by adding an offset to the passed-in address to write to buffer2. The read IR function call works in the same way except that it calls `readNumber()` and only needs to give an address and total checkpoint space length.

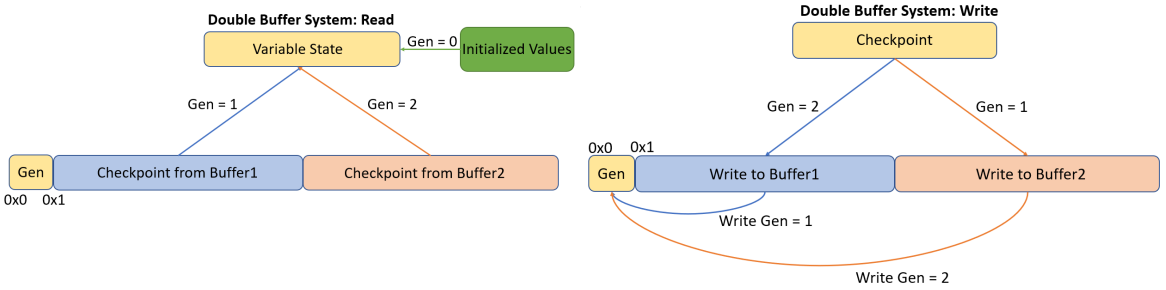
The FRAM read and write calls make up the bulk of the built instructions in the setup step, but a few more are necessary to make the whole system work. The if statements used to restore a checkpoint and to provide the optimizations are also made in this step. Additionally, the instruction to change the generation is also made. This instruction uses bitwise exclusive-or to alternate between the values 1 and 2. A special call to `write8()` in the FRAM library is specifically made for the generation. As the name would imply, `write8()` writes a single byte to the FRAM and takes a value and an address. The value is the generation variable, and the address is always zero.

4.5 Compilation Step #4: Checkpoint Insertion

At this point in MakeCode-Iceberg's execution, all user functions have been in-lined into `main` and most of the checkpoint instructions have been built. The transformation step begins by iterating through `main`, IR instruction by IR instruction, and notes whenever it finds a variable that belongs to the checkpoint list. When the pass hits the end of a basic block, it will insert a checkpoint at that spot if and only if that block contained a variable of interest. Once the checkpoint has been inserted, it will also insert a label that identifies that specific checkpoint in the program. For example, the first checkpoint in a program will be identified by label "Checkpoint1," the second by Checkpoint2," and so on. These labels allow the restore operation to jump back to the last checkpoint. The labels themselves are enumerated by the compiler as a number, meaning the integer value of one corresponds to Checkpoint1. A unique custom variable called "label" stores this value and is added to the checkpoint list. This way, it is saved to the FRAM as if it were any other variable in the program. See the restore operation section below to see how the label variable is used to jump back to a checkpoint.

4.5.1 Checkpoint Operation. The SPI FRAM breakout board from AdaFruit has a transaction system, but only for single-byte values. If power failed while the FRAM was writing the contents of the checkpoint buffer, then it is possible that only part of the buffer would be updated, resulting in a corrupted checkpoint. To always ensure a reliable checkpoint to restore from, we double buffer checkpoints. The diagram for how this system works can be seen in Figures 7a and 7b. This double buffer system requires two times the length of the buffer plus one in bytes. This extra byte is for a generation that indicates which buffer to read from and which buffer to write. While only two bits are necessary for this generation as it will only contain a 0, 1, or 2, we can only write at the granularity of a byte for this specific FRAM breakout board.

Before running a new program, the FRAM will need to be cleared to zero to remove any other program's checkpoints. At startup, the program will read the value of the generation from the FRAM. If the generation is 0, then no checkpoint is present, and the program will initialize each variable according to its definition in the program. If the generation is 1, then the contents of `buffer1` are read as the program state. Likewise, when the generation is 2, `buffer2` is used. It works in the opposite direction when writing to the FRAM. This switching preserves the most recent checkpoint and overwrites the oldest. When the generation is 1, the program writes to `buffer2` to preserve the contents of `buffer1`. When the generation is two, the program writes to buffer1 to preserve the contents of `buffer2`. The generation is then flipped after completely writing the checkpoint to either buffer. This change indicates to the read function where the newest checkpoint can be found and acts as a "commit" to the checkpoint. If the program fails before finishing a write, then the corrupted buffer is never read from because the generation remains unchanged. We can never have an inconsistent state for the generation because of the one-byte transaction system of the FRAM. This atomicity means the generation will remain the



(a) Read operation for the double buffer system

(b) Write operation for the double buffer system

Fig. 7. Double buffering of checkpoint

same or completely change, never in-between. The program will continue to write to the corrupted area of FRAM until the generation is written, marking that it now holds a correct checkpoint.

4.5.2 Restore Operation. As mentioned above, if the value of the generation is 1 or 2, then a checkpoint is present. As a part of FRAM initialization, the value of address zero will be read into the generation variable. Each variable in the checkpoint list is assigned to its saved value when a checkpoint is present. As mentioned in the transformation section above, a unique custom variable is used to track which checkpoint to jump back to. This variable, label, has its state restored like any other variable in the checkpoint. For each checkpoint label in the program, there is an if statement within the restore operation that contains a jump back to that label. This form of checkpoint restoration was another motivation for function in-lining, as it made jumping back to the last checkpoint much simpler. When label is equal to one, the program will jump back to checkpoint1, label equal to two will jump back to checkpoint2, and so forth. When no checkpoint is present, the program never enters the restore block and continues as the user initially programmed it. However, the generation is initialized to one when no checkpoint is present. This preserves the double buffer system.

4.6 Checkpoint Triggering Mechanism

Checkpoints in MakeCode-Iceberg are inserted at the end of a basic block and the restore is inserted at the top of the program. A basic block is a region of code that executes sequentially. The FRAM driver is configured so that the read and write instructions load and save a checkpoint all at once, respectively. This configuration means that no matter the checkpoint size, only one call to set the FRAM to write or read mode is needed. This instruction is followed by the write or read for each variable in the checkpoint list. A checkpoint in a MakeCode-Iceberg program may exist following the Bruteforce, just-in-time, block-skipping, or timer-based methods.

4.6.1 Bruteforce. The Bruteforce method of checkpointing involves leaving the inserted checkpointing code as it is. This method means that a checkpoint will occur whenever a basic block modifies a variable in the list to checkpoint. Bruteforce makes the occurrence of a checkpoint very often, and the other methods below wrap this code in an if statement. Bruteforce is the naive approach to checkpointing at the IR level. The approaches discussed below activate checkpoints based on some logic, thereby increasing performance by not wasting time on superfluous checkpoints. This logic is automatically generated by MakeCode-Iceberg.

4.6.2 Just-in-Time. The just-in-time approach triggers a checkpoint whenever a power failure is likely to happen. This approach requires a capacitor to act as an energy buffer for the energy harvesting mechanism. The checkpointing procedure, the updates to the buffer and the write to FRAM, are wrapped in an `if` statement

to skip unnecessary checkpoints. The condition for the if statement is whether the voltage of the capacitor is below a certain threshold determined by the user within MakeCode-Iceberg. The voltage is measured using an `analogRead()` of a pin on the micro:bit connected to the capacitor. If the energy harvester cannot provide power, the capacitor will begin to drain as it powers the micro:bit, and the program's execution continues. As the voltage drops in the capacitor, the if statement of a checkpoint will return true, and the device's state will be saved to FRAM before all power is lost.

4.6.3 Block-Skipping. The block-skipping approach activates a checkpoint when the program has encountered a checkpoint after a set number of times. For example, if block-skipping was used and the interval was set to five, a checkpoint would occur once every five times the program encountered a checkpoint during execution. This method utilizes a counter variable to keep track of how many checkpoints have been skipped. Like the other optimizations, an `if` statement is used to skip the checkpoint. The condition for this method is if the counter variable is equal to the interval. When the condition is false, the checkpoint is skipped, and the counter is incremented by one. The checkpoint is made and the counter is set to zero when the condition is true.

4.6.4 Timer-Based. The timer-based approach triggers a checkpoint after a user-defined amount of time. This approach utilizes the built-in `control.millis()` function to return the amount of time since the device started running in milliseconds. Just like the Just-in-Time approach, an `if` statement wraps the checkpointing code. If the time since the last checkpoint is greater than the user-defined value, the condition is true and a checkpoint is made. Otherwise, the checkpoint is skipped, and execution continues as usual. When a checkpoint occurs, the time since the last checkpoint is set to the current value of control.millis().

5 DEMONSTRATIONS AND USE CASES

We present multiple demonstration applications built from scratch or adapted from existing MakeCode tutorials as part of our evaluation. This section engages in proof by demonstration, showing a range of battery-free, energy harvesting, untethered applications that MakeCode-Iceberg enables for novice programmers, students, and makers of all skill levels. Of the applications we have built or evaluated (as *any* Makecode program can be instrumented with MakeCode-Iceberg), we discuss four notable examples: i) Smart Terrarium, ii) Combination Lock, iii) Step-counter, iv) Health Wearable. We present these demonstrations for several reasons:

(1) We want to show that our additions are seamless; we can do just about anything MakeCode can do, as long as you have energy.
(2) These use cases are meant to test a variety of different programming constructs and approaches, and multiple types of peripherals, inputs, and actuators, from complex devices like E-Ink displays and OLEDs, simple temperature and soil moisture sensors, input devices like buttons, and output devices like audio buzzers and a radio.
(3) We want to show that you can build exciting things with commonly available energy harvesters and components in every maker or school kit.
(4) We want to show how some of these applications are new and interesting and enabled only by MakeCode: i.e., long term usage (no battery), mobile, or just sustainable.

We explored other applications and tutorials that are available on our website[2].

The key point is that all of the use cases presented below can run unmodified on either MakeCode or MakeCode-Iceberg, as our approach is hidden from the user in the middle-end of the compilation of a program for MakeCode. However, when using MakeCode-Iceberg, the program automatically and seamlessly recovers from power failures caused by energy harvesting intermittency. For the "Smart Terrarium" use case, we design an experimental plan

[2]We host an instance of MakeCode-Iceberg along with source code and documentation here: https://github.com/ka-moamoa/makecode-ic

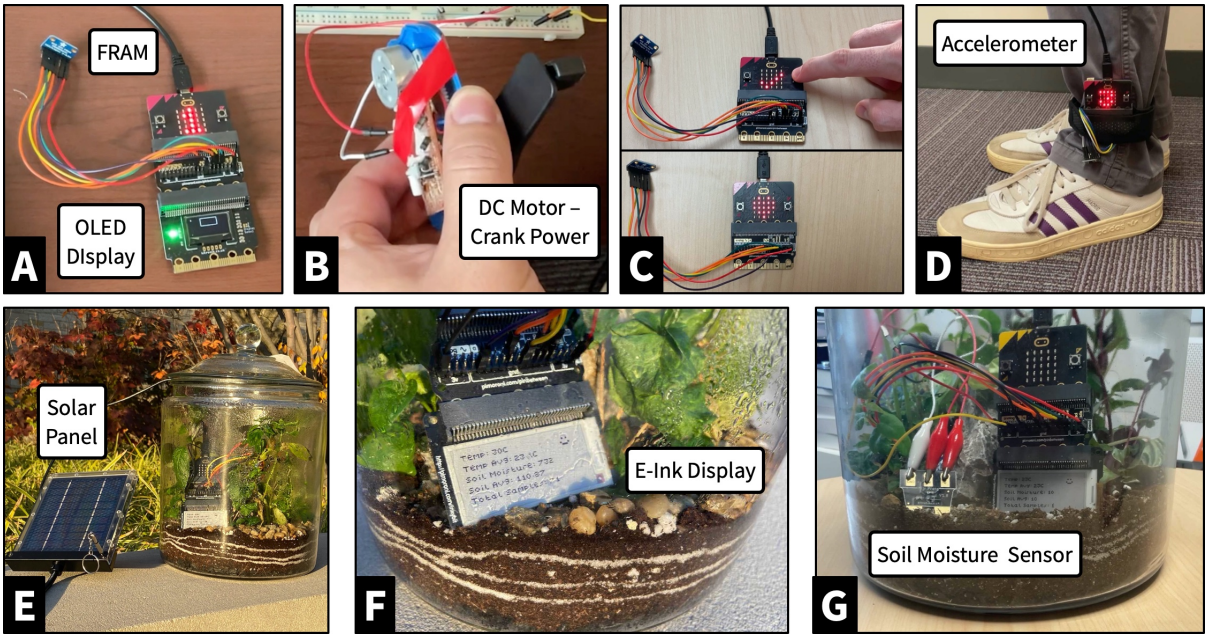


Fig. 8. Assembled hardware to demonstrate four hobbyist-grade applications written in unmodified Blocks, JavaScript, or Python for the Makecode platform with MakeCode-Iceberg extensions using running on intermittent power from energy harvesting: **(A)** Counter with OLED display, powered by **(B)** a hand crank energy generator scavenged from a $8 crank to power flashlight. **(C)** A Combination Lock where the user inputs a button sequence, a buzzer sounds on each correct entry, and the full entry gets a heart on the LED matrix. **(D)** A Step Counter using the built-in accelerometer. **(E-G)** A Smart Terrarium, an enclosed water cycle and ecosystem, monitored via soil moisture and temperature sensors with status displayed on E-Ink.

Table 2. Breakdown of the sensors, interfaces, actuators and output devices, and energy harvesters for all demonstrations we discuss. The table shows the breadth of components MakeCode-Iceberg supports with zero modification by users.

Demonstration	Sensors	Interface	Actuators/Output	Energy Harvester
Smart Terrarium	Soil, Temperature	E-Ink Display	None	Solar
Combination Lock	None	Buttons, LED Matrix	Buzzer	Solar
Step Counter	Accelerometer	LED Matrix	None	Solar
Crank Counter	None	OLED Display	Radio	DC Motor

to demonstrate this resilience to power failures. We discuss the design, implementation, and behavior of the remaining use cases powered by energy harvesting.

Figure 8 shows all the use cases and the sequence of actions demonstrating recovery from power failures. Table 2 provides a listing of peripherals (actuators, sensors, input devices), associated libraries, and energy harvesters for each demonstration. The main takeaway from this section is that MakeCode-Iceberg enables usage of any peripherals and energy harvester for MakeCode programs written in any of JavaScript, Python, and Blocks.

5.1 Smart Terrarium

A terrarium is a sealed glass container that contains soil and plants, which creates an internal environment for plant growth, as the transparent walls allow for both heat and light to enter the terrarium. The sealed container and exposure to heat and light through the transparent walls create a small-scale water cycle. Moisture from both the soil and plants evaporates in the higher temperatures inside the terrarium. Water vapor condenses on the walls of the container and eventually falls back to the plants and soil below. Because of this sealed water cycle and ecosystem, some sealed terrariums have lived for sixty years or more. Terrariums are often built for decorative or educational purposes. They are often artistic expressions as well as fascinating scientific explorations.

Combining MakeCode-Iceberg and Terrariums make sense because (1) it strengthens the connection between environmental sensing and battery-free embedded computing, (2) it intersects computing, environmental sciences, and art, and (3) it allows for long term deployment inside a terrarium, which could be sealed for years, with no possibility of battery replacement, making energy harvesting ideal. MakeCode-Iceberg enables seamless and continuous data collection despite power failures.

5.1.1 Program and Experimental Setup. The setup is shown in Figure 8E,F,G. As in all our use cases, a micro:bit is attached to a Pinbetween (which breaks out all pins on the bit), connected to an E-Ink display (an InkyBit from Pimoroni), a soil moisture sensor, and a temperature sensor are also connected. The electronic paper (e-paper) or E-Ink display is ideal for intermittent computing applications as they retain display state even if there is no power. However, these displays take significant energy and time (in some cases up to 15 seconds for the display chosen, but three seconds on average) to update. Using a library that allows partial updates would be more efficient. However, in keeping with the design goals, we do not modify libraries to show the full capability of MakeCode-Iceberg in using unmodified user code and third-party libraries. A 6V 1Watt solar panel, the size of a postcard, is used to power the entire setup. The FRAM breakout board is also connected via the SPI port for storing checkpoints. The program continuously captures moisture and temperature readings, once every ten seconds, keeps a running average and count of samples, and then displays this all on the E-Ink display. The first picture shows the terrarium outside being powered fully by outdoor light. The second picture is indoor and powered by a work light.

5.2 Combination Lock

Another reasonably common tutorial or programming exercise for novices using MakeCode is to create a combination lock where the LED matrix display is used to keep track of the progress of the combination. We made a version of this application, where the buttons (A, B, and the Logo on the top center of the micro:bit) are used to enter a sequence. This demonstration is shown in Figure 8C. The buzzer is used to designate if the input is correct or incorrect on each button press. An Array object holds the sequence, and a number of if-else statements check the status of the sequence. The combination lock has to keep state across each entry and then remember if the lock has been unlocked or not. Proper execution requires the instrumentation pass of MakeCode-Iceberg to maintain these variables across multiple attempts or attempts that happen while the device is off. This demonstration also shows that MakeCode-Iceberg does not change or diminish compatibility with any onboard peripheral, such as the buzzer, and is compatible with input devices like the buttons. In this demonstration, we powered the device with a solar panel (the same one used by the Smart Terrarium). We tested the execution and state retention of MakeCode-Iceberg by turning on/off the work light and checking that the combination lock application was at the same stage of the sequence when power returned. While we used the solar panel, the hand crank or other harvester could also power the device since MakeCode-Iceberg does not depend on any particular harvester and is oblivious to the energy input.

5.3 Step Counter

We implemented a step counter (also called a Pedometer) demonstration, similar to another of the first simple tutorials that novice MakeCode users are greeted with on the front page of the online application. This use case is shown in Figure 8D. The pedometer, in this case, detects a significant motion by sampling the accelerometer and calculating the force magnitude. The application then logs that step to a variable in MakeCode and keeps a running count of the steps. The total number of steps is displayed on the LED matrix. Of course, if the power goes out, this variable is lost for regular MakeCode, but with MakeCode-Iceberg, this variable is stored in Non-volatile FRAM and automatically restored via our instrumentation when power is returned. For the experiment, we also used a solar panel to power the device and then attached the device to one of the author's legs and walked around to count steps in and out of the solar panel, experiencing a few power failures but recovering.

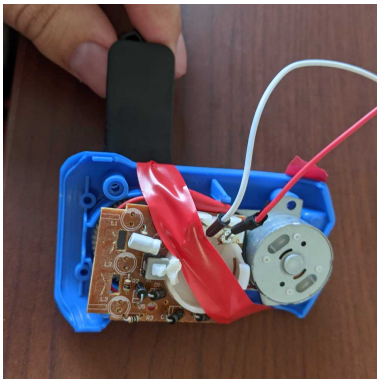
5.4 Crank Counter

We tested out the integration and usage of other peripherals in a simple counting application. We focused more on investigating the usage of an external OLED Display from Kitronik (and associated library), the micro:bit onboard radio (2.4GHz), and low-level wireless communication. This demonstration is shown in Figure 8A and B. This demonstration was also powered by a different energy harvester, using a crank-powered flashlight commonly found on eBay, Amazon, and AliExpress and cheaply manufactured. Essentially this is just a geared DC motor connected to a crank. When the axle of an electric motor is turned, a back-EMF is generated, resulting in an electrical potential across its terminals. The output voltage is proportional to the rate of turn. Cheap, crank-powered flashlights are all based on this mechanism. Some works have used this modality to power electronics and gather the rate of change [60]. We purchased one of these devices, removed the battery and enclosure, cut the flashlight LEDs off with wire cutters, and soldered two wires onto the output of the rectifier circuit exposed on the. Multiple DIY Instructables online show how this can be done, and it took us less than ten minutes to set up. Figure 8B shows the motor and circuit extracted from the flashlight.

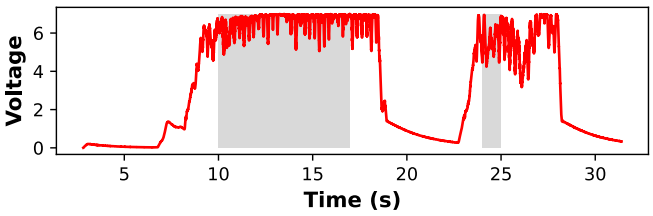
The demonstration program counts and updates a variable every 500ms of on-time, which serves as a rudimentary counter for the amount of time the crank has been turned, which could be used for any number of applications. We print that value to the OLED screen (along with drawing a rectangle on the screen), update the LED matrix with the value, and send that number over the radio to another micro:bit. When the crank stops (and energy stops), the counter value and program state will have been automatically backed up to FRAM. When the crank starts again, that value is retrieved from FRAM, and the program is started from where it left off, all enabled by our MakeCode-Iceberg extensions.

5.5 Other Energy Harvesters

Nothing fundamental prevents a user from using any energy harvester since both MakeCode and MakeCode-Iceberg are oblivious to the type of energy harvester and power input. Our extensions are entirely in the middle-end of the compilation toolchain and are independent of hardware (in fact, this happens afterward, based on the IR we generate and transform). The critical point about harvesters is that some, for example, RF-based ones, may not generate enough energy to boot up the micro:bit and specifically get through the lengthy USB checks of the pre-installed bootloader. This startup sequence can sometimes take 1-2 seconds. While other work like BFree [33] rewrote portions of the boot-up sequence, our approach does not have access to any hardware or bootloader code (which is an advantage, as our approach can run on any hardware/MCU target supported by MakeCode). Additionally, the harvester must be able to get past 5V to overcome the startup voltage requirement on the USB regulator (we recommend reusing the USB plug for the energy harvester input, as that reduces a chance of overvolting the micro:bit by plugging directly into the post-regulated 3.0V power rail).

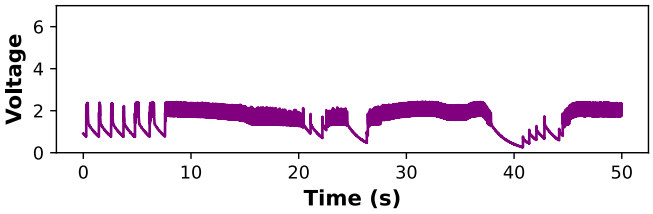


(a) Crank Harvester



(b) Crank Voltage

(c) Powercast Harvester



(d) Powercast Voltage

Fig. 9. Voltage traces at the micro:bit regulator input when the crank (DC Motor) or Powercast (wireless transmitter) harvesters were transduced. The shaded areas show that when the device was on, the Powercast could never achieve a high enough voltage even when put 1cm from the antenna.

We attempted to run our demonstration use cases on three different energy harvesters to make this point concrete. As discussed previously, solar panels and the retrofitted DC motor (crank) worked flawlessly and provided enough power and a high enough voltage to work with all demonstrations we presented. We also attempted to use a PowerCast wireless energy harvesting 5W transmitter and harvester [52]. Figure 9 shows traces of both the crank and Powercast. The Powercast does not provide enough power or voltage, even at extreme proximity (1cm), to overcome the high bootloader energy requirement of the micro:bit. Figure 9d shows that as the Powercast 5W transmitter approaches the harvesting antenna connected to the micro:bit, the voltage does increase; however, it never exceeds 3V. While disappointing, wireless power has long been less power-rich than other sources. If the micro:bit bootloader were improved to reduce the USB search time, the Powercast would be more feasible in theory.

5.6 Cost

Our additions do not put a high-cost burden on the programmer, keeping in line with our goal of accessibility. A programmer could begin making a sustainable physical computing device for less than $45, using off-the-shelf parts from Adafruit, Sparkfun, and Amazon/eBay/AliExpress (we express in USD, but in some other countries, these parts are more accessible and cheaper). This price includes $20 for the micro:bit, $3-5 for a Pinbetween or similar pin breakout, $10 for the FRAM breakout board from Adafruit (which has expanded their offerings recently to larger FRAM boards), and $10 for a 6V 5W solar panel, or $8 for a crank-powered flashlight. We

believe this price point is suitable to be accessible for many schools (especially when compared to the cost of robotics-type curricula) and individual makers.

The solar panel we used was purchased from Amazon for $20 (but a similar model can be found on Aliexpress for less than $2), and the crank-powered flashlight we took apart to get the energy harvester is easily purchase-able on Amazon/eBay/AliExpress for less than $10.0. Similarly, other shake-to-power flashlights have been used for maker electronics. The Powercast (and most wireless power solutions) is upwards of $500 for a transmitter. A Powercast is not a hobbyist or maker accessible energy harvester modality, costing upwards of $500 for the power transmitter. Possibly, alternative methods for wireless power, like NFC from a phone, or cheap transmitter, could be used, along with a coil. In general, we find it encouraging that novices and makers can use MakeCode-Iceberg with very low cost energy harvesters.

6 EVALUATION

The main questions of our evaluation of MakeCode-Iceberg are:

- What is the impact on performance of varied checkpoint strategies, and can we determine a clear winning strategy? (Section 6.2)
- What is the impact on execution time of our transformations? (Section 6.3)
- Do our transformations change the operation or output of a program? (Section 6.4)
- What is the overhead of running our transformations inside the online MakeCode application? (Section 6.5)
- What is the impact on binary size and memory of the transformations? (Section 6.6)

6.1 Experimental Setup

Figure 10 shows our setup for evaluation. For all experiments, we use a BBC Micro:bit V2 on a T-Type Micro:Bit GPIO expansion board connected to an Adafruit SPI Non-Volatile FRAM breakout(64Kbit) as the device under test(DUT). Constant Power execution times were recorded from the internal control.micros() function and transmitted over serial. For the resiliency experiments of Section 6.4, the DUT was connected to power via a 3.3v pin on a Teensy 3.6 and the battery or USB connector of the micro:bit. The Teensy was used to provide constant power or simulate periodic energy losses. When the DUT completes the execution of a benchmark, it writes an output pin connected to the analog input of the Teensy to high and turns an LED on the front panel. When the Teensy receives this signal, it records the time in microseconds to measure performance. Due to the uniqueness of the JIT checkpoint trigger, its resiliency tests involved an energy buffer of capacitors in parallel totaling 8700µF.

We use the following three benchmark programs, widely used in the intermittent computing domain [39], for evaluating the performance of each checkpointing approach discussed in Section 4.6.

(1) Bit Counter counts the number of ones in the number 2146500607 in a loop 100 times
(2) String Length counts the number of characters in a 6335 character string
(3) Fibonacci calculates fib(46) 80 times in a loop

In addition, we also evaluate our system on Smart Terrarium, a real-world application. The checkpoint sizes for each application are shown in Table 3

6.2 Comparing Checkpoint Strategies Performance

We evaluate different design-points for MakeCode-Iceberg's checkpointing system before embarking on the best configuration that helps us conserve the maximum amount of energy, preserve state, and keep time overheads due to checkpointing low.

6.2.1 What to checkpoint? The best strategy for writing onto non-volatile memory is driven by the time required to save the state. The more time it takes to save the state, the more energy it consumes, and the less the

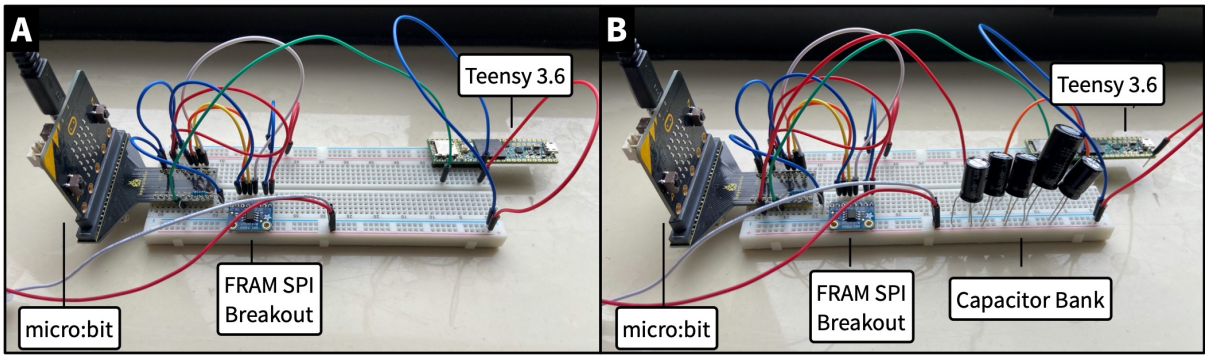


Fig. 10. Evaluation Setup. (A) Shows the experimental setup to evaluate MakeCode-Iceberg, using a Teensy 3.6 to control the experiment. **(B)** Shows the setup to evaluate the Just-in-Time checkpointing approach with a capacitor bank which is read by the analog pin on the micro:bit.

Table 3. MakeCode-Iceberg checkpoint content and its respective size in Bytes.

Checkpoint Content	**Size** (Bytes)
Fibonacci	41
String Length	17
Bitcount	33

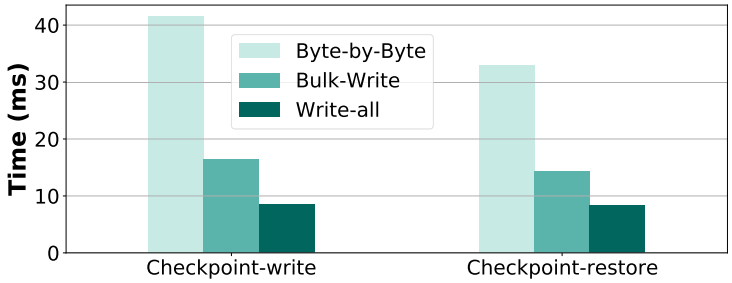


Fig. 11. Figure shows a comparison of read and write time for different approaches for what to checkpoint.

system runs normal device operations. Figure 11 shows a quantitative comparison of different approaches for writing the checkpoint onto non-volatile memory.

We can observe that the cost of writing a single byte onto the non-volatile memory is 5× more than writing the entire data onto non-volatile memory (note that these numbers do not include the execution time required for tracking changes in the state; necessary step to ensure which byte to update). Dividing the memory into chunks of equal sizes (similar to memory pages in mainstream computing) helps, but the increase is still much more than writing the bytes all at once. Furthermore, all checkpointed variables are global. This fact implies that these variables will be adjacent to each other in memory. Therefore, the best approach for writing onto non-volatile memory is to write all variables that need to be checkpointed at once in order to save energy.

Table 4. Size of hex file (bytes) for benchmark applications using different instrumentation strategies

Benchmark	Unmodified	Instruction-level	Block-level
`fib`	1317900	1353228 (+2.6%)	1321996 (+0.3%)
`strlen`	1364492	1380364 (+1.1%)	1365516 (+0.07%)
`bitcount`	1316876	1342476 (+1.9%)	1318924 (0.15%)

6.2.2 Where to checkpoint? Our goal is to adopt an approach that incurs the least amount of execution overhead and wasted computations, thus sparing more time for application execution. Table 4 shows the code size increase for adding checkpoints after each instruction and at the end of a state-modifying basic block.

Theoretically, instruction-level instrumentation gives us the best performance as it saves execution of each instruction executed during an energy cycle. However, we can see from column 3 of Table 4 that the amount of code increase is non-negligible. In the case of Smart Terrarium, the code size was too large to be downloaded on the system. Additionally, the number of additional instructions run on the device adds to the execution time, thus degrading the system's performance. On the other hand, block-level instrumentation only adds a checkpoint call at each basic block that contains instructions that modify the checkpoint state of the program. This change results in a significant reduction in the number of additional instructions added to the program. Block-level also significantly reduces the execution time for the application, especially if a checkpoint call is inserted inside a loop.

6.2.3 When to checkpoint? Figure 12 shows normalized execution time for each approach for when to checkpoint? as discussed in the previous section. The bruteforce approach refers to the one which checkpoints at the end of each basic block even when the device could run further on the same charge. We can see from Figure 12 that the timer-based approach performs much better than all other approaches. Basic-block skipping can only skip a fixed number of blocks before a checkpoint, whereas the timer approach can execute a higher number of blocks with the same energy budget.

An interesting result is the JIT's execution time; the time required by JIT is more than the timer-based approach. This difference is primarily because of the time required to access the hardware to get the current voltage of the capacitor bank. It is much more than what a hardware-assisted approach would have saved by not instrumenting the code.

6.2.4 MakeCode-Iceberg Configuration. Based on our microbenchmark results, we pre-configure MakeCode-Iceberg with the timer approach set to 100ms and write those checkpoints using write-all. Unless otherwise specified, all use cases and demos use the same configurations for checkpointing.

In this way, the users do not have to dive into the details of how to checkpoint their code, and these technicalities remain oblivious to the user, thus meeting our design goals.

6.3 Comparing Execution Time: ARM Thumb with and without Transformations

After evaluating MakeCode-Iceberg on micro-benchmarks, we now evaluate our MakeCode-Iceberg we now evaluate our MakeCode-Iceberg on an actual application, the Smart Terrarium demo, in addition to the benchmark applications in order to measure its performance and correctness when compared with the unmodified application. Our experimental setup for Smart Terrarium is the same as the one explained in Section 6.1

This experiment compares execution times for all checkpointing approaches against the unmodified MakeCode program. The results for each checkpointing strategy can be observed in Figure 13. We dissect the results for each checkpointing approach in the following subsections.

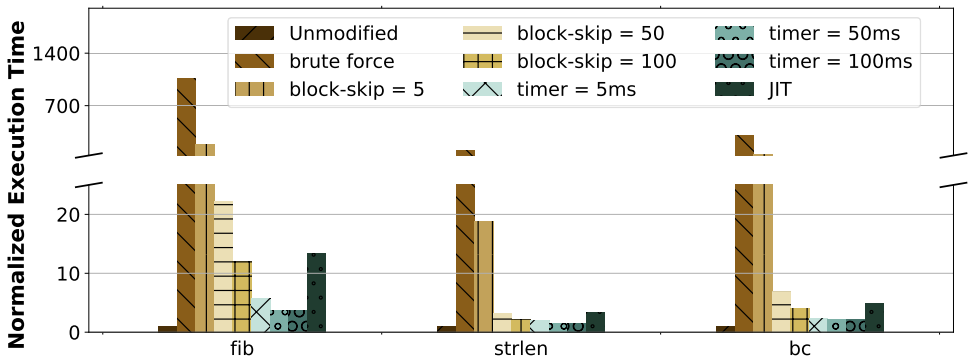


Fig. 12. Comparison between execution time for different methods of *when to checkpoint?*. We can see that the timer-based approach performs best as it can execute a maximum number of blocks in the given time and is not limited to the number of blocks to execute in an energy cycle.

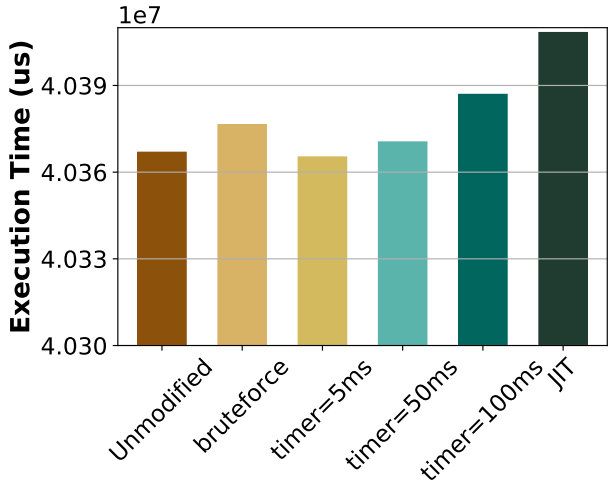


Fig. 13. Execution time for the Smart Terrarium

6.3.1 Timer-based Checkpointing. As seen in the figure, the timer approach dramatically reduces the time to complete the program compared to the bruteforce approach. Here, checkpoints occur roughly every 5, 50, and 100ms compared to almost constant checkpointing in bruteforce. The 5, 50, and 100ms intervals were chosen to give an often(5ms) and not as often(100ms) checkpointing strategy with a transition value(50ms) to show a combination of the two. The often checkpoint of 5ms was chosen to show two things. The first would be that checkpointing slightly less often brute force would likely have a dramatic effect on the program's speed, and by looking at the figure, it certainly does. The 100ms was intended to push the limits of this strategy and encounter the possible tradeoff of lost progress in the event of a power failure in return for faster completion. Overall, a shorter interval retains more progress than a longer one in exchange for a slowdown in performance.

6.3.2 JIT-based Checkpointing. The JIT-based checkpointing method has the advantage of never needing to write a checkpoint until necessary. This advantage allows the program to run mostly uninterrupted by the

Table 5. The table shows whether each benchmark could give a correct output and be resilient against power failures occurring at 2, 4, and 6-second intervals. We can see that all benchmarks retained the state across power failure except Smart Terrarium when the frequency of power interruption was 2 seconds. The micro:bit cannot refresh the e-ink display quickly enough to display the numbers in time before the subsequent power failure occurs.

Benchmark	Unmodified	Timer-100			
		Constant Power	2 sec	4 sec	6sec
`fib`	✗	✓	✓	✓	✓
`strlen`	✗	✓	✓	✓	✓
`bitcount`	✗	✓	✓	✓	✓
`Smart-Terrarium`	✗	✓	✗	✓	✓

checkpointing code. However, as shown in Figure 13, JIT is not as fast as the timer-based approach. The slowdown is likely due to the high number of calls to the `analogRead()` function. The `analogRead()` is used to get the voltage of the capacitor bank. In the future, an event or interrupt-driven JIT model may be a better approach and result in faster executing times. This change would likely make JIT the best approach. The only downside is that extra hardware is necessary for this method, but adding capacitors to create an energy buffer is relatively trivial.

6.3.3 Summary. MakeCode-Iceberg must incur minimal overhead to ensure responsiveness of application and user engagement. This task can occur only when the system makes maximum progress on the given charge and uses significant energy for performing valuable computations.

Figure 13 also shows a side-by-side comparison of all checkpointing approaches for the Smart Terrarium use case. We can see that JIT requires more time to complete execution than any static placement technique. This time difference is primarily because of the additional hardware attached to the system to measure capacitor voltage. As JIT relies on capacitor voltage to trigger the checkpoint, the system has to continuously pool the capacitor's voltage to check whether the device needs to trigger the checkpoint or not. These voltage reads are expensive both in terms of time and energy.

6.4 Benchmarking for Correctness and Resilience

Table 5 shows the evaluation of benchmark applications under intermittent energy supply with power failures every 2, 4, and 6 seconds respectively, for the timer-based approach. We compared our performance against the unmodified version, which was unable to complete. It can be observed that all applications were able to finish their execution even under frequent power failures correctly. All benchmark applications and the use case were able to complete their execution with the correct output using MakeCode-Iceberg. However, However, the Smart Terrarium use case failed to complete execution when the power failures occurred every 2 seconds. This failure is not because MakeCode-Iceberg could not restore the system state under frequent power failures. It is primarily because the microbit can not update the e-ink display (it requires 3 seconds on average). This problem can be resolved by using libraries that enable partial updates.

6.5 Compiler Overhead

Table 6 shows the increase in the compile-time due MakeCode-Iceberg's changes. Total time is the time it takes to generate a hex file from the user's code. CompileBinary is the name of the function where MakeCode-Iceberg operates. The machine running the instance of MakeCode has an Intel i7-9750H CPU at 2.60GHz, 16GB of RAM, and uses 64-bit windows 10 Pro version 20H2.

Table 6. Compilation time (μs) increase for different applications

Benchmark	Unmodified		MakeCode-Iceberg		Percentage increase	
	Total	Compiled Binary	Total	Compiled Binary	Total	Compiled Binary
`fib`	566.5	564.2	618.1	617.7	9.11%	9.48%
`strlen`	526.7	525.7	712.8	711.8	35.33%	35.40%
`bitcount`	598.8	598.2	626	625.5	4.54%	4.56%
`Smart Terrarium`	748.8	747.8	767.8	767.1	2.54%	2.58%

Table 7. Bytes of the HEX files for each benchmark application

Benchmark	Unmodified	Brute Force	Block	Timer	JIT
`fib`	1317900	1320972 (0.23%)	1321996 (0.31%)	1321996 (0.31%)	1321996 (0.31%)
`strlen`	1364492	1365516 (0.08%)	1365516 (0.08%)	1365516 (0.08%)	1365516 (0.08%)
`bitcount`	1317388	1320460 (0.23%)	1320460 (0.23%)	1320972 (0.27%)	1320460 (0.23%)

We can observe that the increase in the compilation time is not significant. MakeCode is already a highly optimized online tool meant to run on Chromebooks and similarly low-end laptops. We only slightly increase the overhead and time for compilation below a few milliseconds, which is not noticeable by the user. The mean increase in compilation time is 12.9% for total time and 13% for the compiled binary case. We also note that the RAM impact on the laptop we tested on is so low as to disappear into the noise from the Chrome web browser running.

6.6 Memory Usage

One of the greatest strengths of checkpointing at the level of the IR is that it dramatically reduces its size. This advantage can best be seen in table 3 and limits the checkpoint state to only the most necessary pieces. The state is saved for each variable in the program, and each is four bytes. Each variable has two positions in the FRAM because of the double buffering system, and the generation keeps track of which set to use. For example, the Bitcount benchmark checkpoints the state of three 4-byte user variables plus the label variable. This setup means each side of the double buffer will have 16 bytes of state each. Adding the one-byte generation at address 0 brings the total to 33 bytes. While this system is not necessarily the most scalable checkpointing mechanism, it is doubtful that a MakeCode-Iceberg program would have the nearly 1000 variables needed to fill the FRAM board.

7 USER STUDY AND STEM TEACHER INTERVIEW

To complement the evaluation results of MakeCode-Iceberg in Section 6, we conducted a user study and separate interview with a STEM teacher to evaluate the following research questions:

- **RQ1**: Would MakeCode-Iceberg make it easier for novices and non-experts to *design and deploy* creative and new intermittent-programming applications?
- **RQ2**: Would MakeCode-Iceberg enable new pedagogy and curricula for K-12 students experiencing programming for the first time.

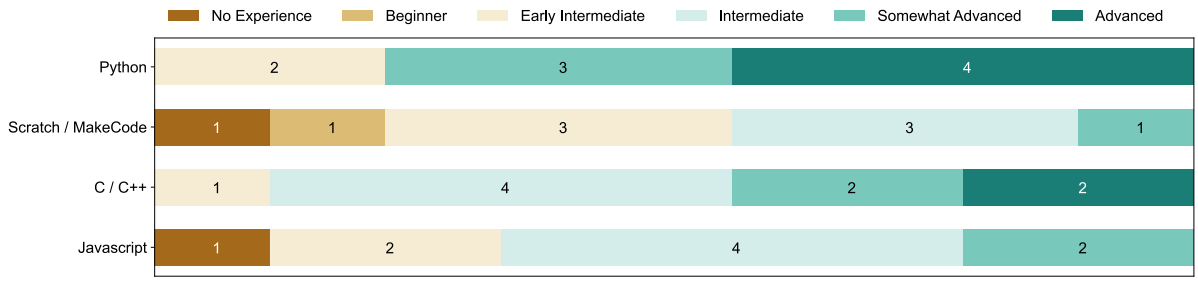


Fig. 14. Skill level or experience of participants with various programming languages.

7.1 User Experience Study

To answer **RQ1**, we conducted a study with a small set of students and asked them to experiment with MakeCode-Iceberg to create a real battery-free micro:bit application and provide feedback on MakeCode-Iceberg's use.

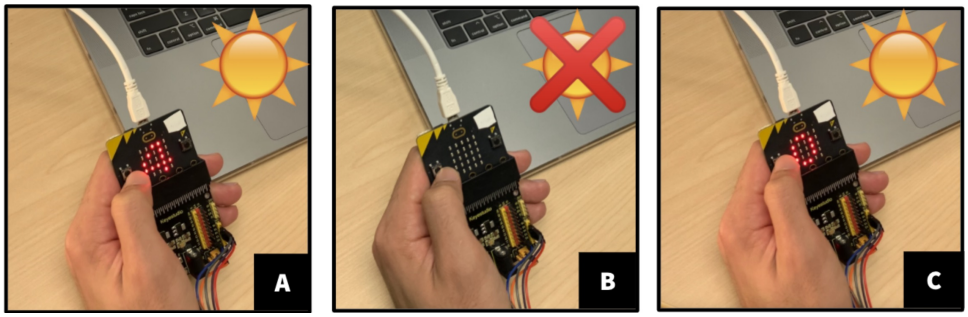
7.1.1 Participants. We conducted the user experience study on 9 participants, all with various levels of experience with physical computing. Of the 9 participants, 6 participants were drawn from a wearables class, composed of juniors, seniors, and grad students, found via email announcements to a class taught by an author of the paper. The remaining 3 participants were computer science undergraduate students found via word-of-mouth. All 9 participants have various levels of programming experience (shown in Figure 14).

Our participants ranged in class standing from undergraduate sophomore to graduate Ph.D. students, with 4 to 12 years of formal computing education and 3 to 12 years of programming experience. Of the 9 participants, 7 had prior experience programming with embedded systems, ranging from 5 months to 8 years. Prior embedded systems applications created by participants included smart wearables, devices that detect handwashing, Arduino hobby projects, deep learning-based gesture recognition systems, and class projects. Only 3 participants had previously used MakeCode to build and program embedded devices.

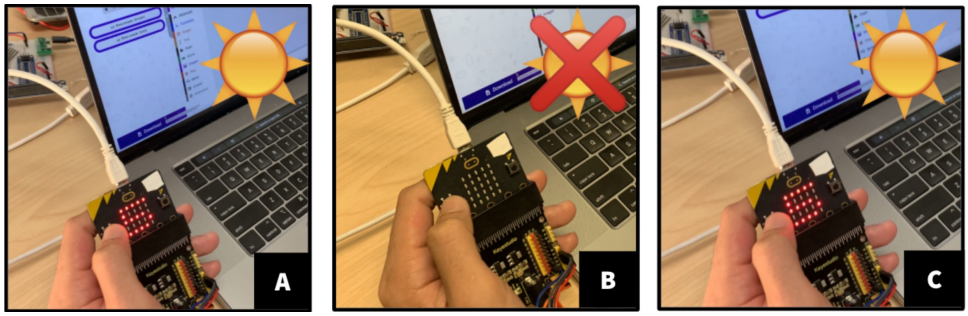
7.1.2 Design and Execution of the Study. Each participant was invited to a room prepared explicitly for the MakeCode-Iceberg study. The participant was asked to sit in front of a MacBook laptop on a regular office table. This laptop was connected via a USB port to a micro:bit. A web browser was opened on the monitor that contained a short description of the experiment. After completing the entry survey and reading an introduction to MakeCode-Iceberg and background information about intermittent and battery-free computing, the participant was given the opportunity to ask questions before beginning the programming challenges.

The first programming task was to create a program in regular MakeCode with a state variable that updated on user input, a simple counter. The participants were given a simple example program that displayed a number on the micro:bit LED screen, which increased by one when the user pressed button A (shown in Figure 15a). Participants were encouraged to create a unique program or replicate the example program. After completing their program, the code was downloaded to a micro:bit connected to the laptop via USB and tested while still connected to continuous power. The micro:bit was then disconnected from the USB and reconnected to a solar panel. Participants were asked to observe their program's performance while the experimenter intermittently turned the solar lamp on and off approximately every 5 seconds.

After observing the performance of their regular MakeCode program, they copied the same code into MakeCode-Iceberg, downloaded the new version of the code to the micro:bit, and observed the program's performance. At the same time, the experimenter intermittently powered the solar lamp approximately every 5 seconds (shown in Figure 15b).



(a) Regular MakeCode



(b) MakeCode-Iceberg

Fig. 15. Programming challenge sequence for the study. The participant is shown how MakeCode-Iceberg recovers the state after a power failure, right where the program left off, via a simple counting program. A work light turns on and off (controlled by the study coordinator) which activates a solar panel energy harvester powering the participants device.

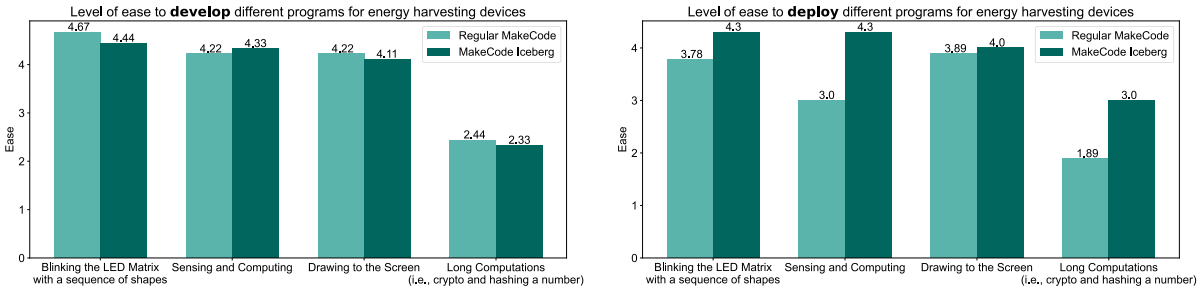
For the final programming challenge, the participant was asked to simulate a use case where MakeCode-Iceberg was used to track and record conditions in a terrarium. The participant was tasked to create a program in MakeCode-Iceberg that sampled either a temperature or light sensor every 2 seconds, then plot the total number of readings taken, the current sensor reading, and the average of the sensor readings taken on a Kitronik 128x64 OLED display. The program was then observed while the micro:bit ran. The micro:bit was connected to a solar panel intermittently powered by a solar lamp that turned on and off approximately every 5 seconds.

After completing the experiment, the participant was asked to answer an exit survey with questions related to the experience with MakeCode-Iceberg on the same laptop.

7.1.3 Results. At the end of the individual user experience sessions with MakeCode-Iceberg, participants responded to a series of 5-point Likert-type scales rating the difficulty of developing and deploying different programs for energy harvesting devices using either regular MakeCode or MakeCode-Iceberg. After transforming the Likert scale responses into numerical values (1=Very Difficult, 5=Very Easy) and calculating the mean values across the 9 participants, we analyzed the perceived difficulty level.

While the development of these programs would require similar levels of difficulty in regular MakeCode and MakeCode-Iceberg (see Figure 16a), the deployment of these different programs was perceived to be easier in MakeCode-Iceberg (see Figure 16b).

Additionally, participants responded to a series of 5-point Likert scale questions regarding the usability of MakeCode-Iceberg. The responses to these questions were generally positive, with a few exceptions. In Figure 17,



(a) Ease of use in Development

(b) Ease of use in Deployment

Fig. 16. Participants response to ease of use questions for various applications, for *development* and *deployment*.

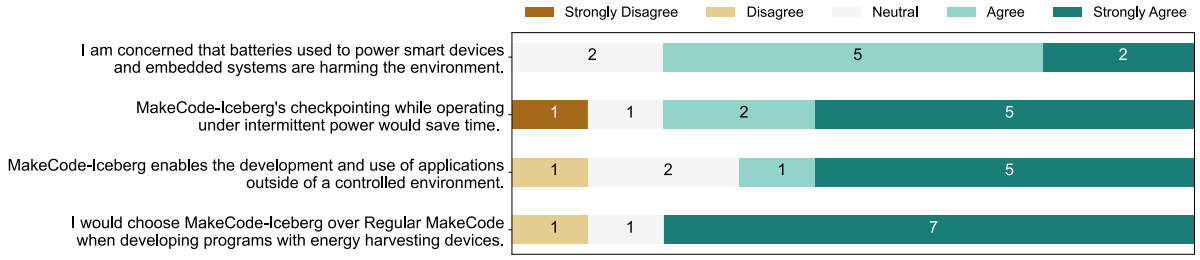


Fig. 17. Participant responses on various issue around usability and application for MakeCode-Iceberg.

one participant responded "Strongly Disagree" to the question about MakeCode-Iceberg's ability to save time as their program for Tasks 1 and 2 was dependent on measuring the actual runtime of a program, this runtime is effected by power failures, and in fact, MakeCode-Iceberg does not yet recover this variable as in other work that maintains time across power failures like TICS [34] and InK [64]. MakeCode-Iceberg therefore caused their run time to increase and the desired program to no longer work properly. Additionally, the negative response on the "I would choose MakeCode-Iceberg...with energy harvesting devices" came from a misunderstanding that MakeCode-Iceberg *only* supported block based programming. The participant commented that if they could use Python, JavaScript, or an alternative language, they would use MakeCode-Iceberg (which they can use, but misunderstood). Overall, these two outliers show that we have (1) a bug that needs to be solved for future version of the work, to capture the internal timekeeping library state and update it, and (2) make clear that MakeCode-Iceberg can be used with any of the supported languages of MakeCode.

Participants also provided responses to open-ended questions about MakeCode-Iceberg, such as possible use cases, suggestions for applications, strong and weak points of MakeCode-Iceberg, as well as further remarks. The full remarks are available in Appendix A. We include highlights below:

Based on the responses to both qualitative and open-ended questions (Figure 18), MakeCode-Iceberg enables users to design and develop creative intermittent-computing applications, which positively supports **RQ1**.

7.1.4 Limitations of the User Experience Study. While the user experience study results were positive, we recognize there are several limitations to this study. The main one is the sample size and a lack of participants across age and experience level that would mimic the full audience for MakeCode-Iceberg. Further, some of these participants had prior experience with programming with embedded systems and mentioned that the block-based

What are possible use cases for these battery-free embedded devices?

- *"Bird monitors/trackers powered by micro-wind turbines."*
- *"Device looking for leaks in the town plumbing system that broadcasts pressure data once every 5 minutes."*
- *"A package notifier which is solar powered."*

What are strong points about Makecode-Iceberg?

- *"Very easy to use, other than some annoyances and repetitiveness of block based programming"*
- *"It seems to work seamlessly, meaning there isn't really a downside to having it in every program."*
- *"Automatic state saving, you don't need to know how it works to use it."*

Did MakeCode-Iceberg change your perception of the applications of energy harvesting devices?

- *"I was already aware of the challenges, but this definitely would be a powerful tool to get into the hands of young developers."*
- *"Yes, I didn't realize that there were such a wide variety of applications."*
- *"Yes. This makes intermittent power/energy-harvesting devices seem like something I could actually use as a programmer, without being that experienced in hardware."*

Fig. 18. Selected participant responses to free-form questions.

programming interface was more challenging to use than lower-level IDEs. Finally, participants in the study were using an early version of MakeCode-Iceberg that did not fully support all MakeCode blocks, which led to bugs that prevented them from running their programs.

7.2 Interview

To answer question **RQ2**, we conducted a guided interview with a K-8 STEM Teacher from a majority Indigenous (Native Hawaiian) public school in Hawaii, responsible for the STEM curricula of over 250 K-8 students. We had collaborated with the teacher for the past year, so the interview served as a summarization of discussions and goals. The teacher detailed their thoughts on the usefulness of the system and future extensions for pedagogy. This discussion also helped inform the design of the system. The interview lasted roughly 50 minutes and included an introduction of MakeCode-Iceberg, a demo and explanation of its use cases and possible applications, and a semi-structured interview about current computing education standards and MakeCode-Iceberg's potential in the classroom.

The feedback from the educator was positive, particularly about MakeCode-Iceberg's ability to enable students to ask more critical questions about sustainability. One of the remarks was that "we should be having things like Makecode-Iceberg running systems around our school," such as using solar power to display the electronic lunch menu in the cafeteria, which "would allow the kids to think about what kind of energy is powering the systems around them." Additionally, Iceberg would allow the students to shift their questions away from questions about whether or not their data would be lost to conversations about what sources of sustainable energy to use. Such as the lunch menu in the cafeteria, which could run on solar power since it only needs to be used in the daylight. Having MakeCode-Iceberg would allow the students to identify which conditions are required for specific programs and sustainable energy sources.

The educator also remarked that MakeCode-Iceberg would allow education to influence the industry by lessening the conceptual gap between the work students are doing in the classroom to things happening at the industry level. The educator stated that "the big technology that's going to be happening in the next 5/10 years is

sustainable computing." MakeCode-Iceberg would allow students to work with industry-level technology early on, allowing them to consider sustainable computing in their interactions with technology throughout the rest of their lives. Based on this interview, we believe that MakeCode-Iceberg has the potential to enable a new pedagogy and curricula centered around sustainable computing for K-12 students experiencing programming for the first time, which positively supports **RQ2**.

8 RELATED WORK

This paper merges two domains: expert-oriented intermittent computing, with novice-friendly programming languages focusing on ease-of-use and availability. This work, distilling existing intermittent computing systems, is the first to target source-to-source compilation schemes, which has the impact of enabling programming of intermittently powered devices with online block-based languages.

8.1 Intermittent Computing Runtimes

Existing state-of-the-art intermittent computing runtimes can be divided into two broad categories: automatic checkpointing, and task-based. Automatic checkpointing approaches [2, 25, 29, 34, 37, 40, 42, 55, 59] mainly target C/C++ runtimes, and use compiler transformations and static analysis to determine the best method for checkpoint insertion, to provide efficient forward progress despite power failures. Task-based approaches [13, 23, 39, 64] wrap atomic and idempotent portions of code into tasks, which are then connected in a task graph. The edges of these graphs often have constraints or rules which define how data, timekeeping, and energy are handled. Other runtime systems build on top of both methods and add mechanisms for adaptation and dynamism, often to support new usage (i.e., machine learning), or to increase performance. Many approaches use a hardware module to trigger a checkpoint [8, 9]. machine learning [26, 35] Other systems rely on environmental signals for general adaptation of tasks [7, 41, 43, 62]. Regardless of their category, all the existing runtime systems in intermittent computing require expertise in C/C++ and in-depth domain knowledge of the challenges posed by intermittent computations. None besides BFree [33] were specifically designed for novices, makers, or early students.

8.2 Intermittent Computing Platforms

Building off a rich tradition in the wireless sensor networks community of platform building [1], intermittent computing hardware platforms have sprung up to tackle energy monitoring [63], rapid prototyping (Flicker [19]), energy management (Capybara [15]), and timekeeping [24]. MakeCode-Iceberg is hardware independent, which has the benefit of enabling broad access, but loses out on the advanced hardware techniques that can enable more reactive and higher performance. In the future, a dedicated platform to match with MakeCode, might be beneficial to students.

8.3 Battery-free Applications

A number of applications have been demonstrated in batteryless domains, including wearable health [38, 54, 56], environmental monitoring [22], mobile Skype calls [58], a battery-free Game Boy [17], and nano-statelites [15]. These devices harvest various types of energy depending upon their deployments, such as sunlight, human movement, and vibrations. The represent the limits of the state of the art of the field and prove the capability of the batteryless approach. However, these systens are bespoke, built by experts with knowledge of computing and electrical engineering. By opening up intermittent computing to more programmers and makers in this domain, we can expand the canvas of battery-free applications. MakeCode-Iceberg is an effort in this direction.

8.4 Novice-Friendly Programming

Many tools and systems to increase access and applicability of computing for novices inspire this work. Well known systems like Logo, Scratch, Processing, and Arduino represent programming environments designed with a low learning curve and directed toward makers, artists, designers, and inexperienced programmers. Platforms like Codeable Objects [27] extend programming to the physical world. Bifröst [44], WiFröst [45] and Scanalog [57] help with debugging complex hardware and software embedded systems. Blockly [51], Scratch [50], and MakeCode [47] all provide online programming environments for students learning to code. MakeCode-Iceberg leverages these works for broader impact and access. Instead of building from scratch, MakeCode-Iceberg relies on an existing community and offers that community an alternative way to engage with computers.

9 DISCUSSION

We note a few limitations of our current version of MakeCode-Iceberg, as well as future plans.

9.1 Currently Unsupported Features

Arrays in MakeCode micro:bit comprise many functions of typical data structures (such as a stack and queue). We support arrays that are not modified (i.e., those used in the combination lock) but do not currently track changes to arrays. We are planning to add support for this in the future. Additionally, MakeCode supports a type of parallelism (despite running on a single core), where a round-robin scheduler underneath the runtime schedules a type of thread, known as a fiber, and triggers events. While this is easy to avoid as a programmer in MakeCode, we anticipate exploring how to enable this type of parallelism in the future. Programming in blocks steers users towards the micro:bit version of parallelism, which MakeCode-Iceberg does not support. However, all of these can be recreated using if-else and control statements in the main block. We also found that some libraries are incompatible because of the low pin count of the micro:bit. However, this is not a problem specific to MakeCode-Iceberg, but a typical thing associated with any embedded computing platform (i.e., you run out of pins).

9.2 Optimizing Energy-Efficiency vs. Hardware Independence and Accessibility

We chose to use the micro:bit version of the MakeCode platform for our demonstration, however, nothing stops our approach to be used in other MakeCode embedded hardware, for example, maker.makecode.com. All instantiations of MakeCode build on Programming eXperience Toolkit (PXT), which is what we modify. To test this, we ran our MakeCode-Iceberg extensions as the core module for the maker instance and found that it generated the correct IR to checkpoint. All we needed to do to give full compatibility was rewrite the FRAM library to support the slightly different pin configuration API. This development is exciting as it means that any MakeCode platform can support energy harvesting and battery-free programming.

9.3 Rapid Prototyping for Expert Users

A high majority of our participants from the user study knew how to program in either C++ or Python. This shows that the majority of our participants were already skilled developers. Since they were able to execute the tasks assigned in the user study, we can conclude that prototyping for an expert user would not be a problem in MakeCode-Iceberg. Another indicator is the ease of development as rated by the participant. As per the numbers, our ease of development is comparable with the regular MakeCode. So any difficulties faced by the users in development would also be present in the existing MakeCode.

9.4 Future Studies

By releasing an MakeCode-Iceberg instance online, we plan to assist educators at all levels in conducting sustainable programming curricula. This work lays the foundation for new K-12 and college-level teaching applications for sustainable physical computing.

10 CONCLUSION

Intermittent computing no longer belongs to experts because of MakeCode-Iceberg. Students and makers alike can now program battery-less devices with ease and low cost through a series of compiler transformations with Microsoft's popular MakeCode for Micro:bit. These compiler transformations utilize a driver for external, non-volatile storage to analyze and transform a program into an energy-harvesting agnostic one. We evaluate MakeCode-Iceberg on various benchmarks, real-world use cases, and in a user study to show that it is correct, reliable, and easy to use. While MakeCode-Iceberg is not without its limitations, our design shows that the middle-end of the compiler is ripe for future work in battery-less systems. By achieving our goals of accessibility, correctness, low user burden, hardware independence, and energy efficiency, MakeCode-Iceberg democratizes access to intermittent computing.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Cohen for illustrations and visual design for figures, Amanda Nelson for discussions around the role of technology in education, Marcelo Worsley for application inspiration and discussions, and the member of Ka Moamoa for help and support during the experimental stages of the project. This research is based upon work supported by the National Science Foundation under award number CNS-1850496, and CNS-2137784. 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] Kofi Sarpong Adu-Manu, Nadir Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. 2018. Energy-harvesting Wireless Sensor Networks (EH-WSNs): A Review. *ACM Transactions on Sensor Networks* 14, 2 (July 2018), 10:1–10:50.

[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] Saad Ahmed, Naveed Anwar Bhatti, Martina Brachmann, and Muhammad Hamad Alizai. 2021. A survey on program-state retention for transiently-powered systems. *Journal of Systems Architecture* (2021), 102013.

[4] Arduino. 2021. UNO WIFI. https://store.arduino.cc/usa/arduino-uno-wifi-rev2. [Online; accessed 02-March-2021].

[5] 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].

[6] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli De Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. 2020. The BBC Micro:Bit: From the U.K. to the World. *Commun. ACM* 63, 3 (Feb. 2020), 62–69. https://doi.org/10.1145/3368856

[7] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5, 3, Article 87 (sep 2021), 42 pages. https://doi.org/10.1145/3478077

[8] 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).

[9] 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).

[10] 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].

[11] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A Syed, and Luca Mottola. 2016. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks (TOSN)* 12, 3 (2016), 1–40.

[12] 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.

[13] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[14] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*. 116–127.

[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] Paul Curzon, Mark Dorling, Thomas Ng, Cynthia Selby, and John Woollard. 2014. Developing computational thinking in the classroom: a framework. (2014).

[17] 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.

[18] 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.

[19] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems* (Nov. 6–8). ACM, Delft, The Netherlands, 19:1–19:13.

[20] Josiah Hester and Jacob Sorber. 2017. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–6.

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

[22] Josiah Hester and Lanny Sitanayah Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 14th ACM Conference on Embedded Networked Sensor Systems* (Nov. 1–4). ACM, Seoul, South Korea, 5–16.

[23] 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.

[24] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P Burleson, and Jacob Sorber. 2016. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 4 (2016), 1–28.

[25] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[26] Bashima Islam and Shahriar Nirjon. 2020. Zygarde: Time-Sensitive On-Device Deep Inference and Adaptation on Intermittently-Powered Systems. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–29.

[27] Jennifer Jacobs and Leah Buechley. 2013. Codeable Objects: Computational Design and Digital Fabrication for Novice Programmers. In *Proc. CHI* (Apr. 27 – May. 2). ACM, Paris, France, 1589–1598.

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

[29] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quick Recall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* (2015).

[30] Karen H Jin, Howard Eglowstein, and Mihaela Sabin. 2018. Using Physical Computing Projects in Teaching Introductory Programming. In *Proceedings of the 19th Annual Sig Conference on Information Technology Education*. 155–155.

[31] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor CM Leung, and Yong Liang Guan. 2015. Wireless energy harvesting for the Internet of Things. *IEEE Communications Magazine* 53, 6 (2015), 102–108.

[32] Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J Zhao, Kevin Ma, and David E Culler. 2018. System architecture directions for post-soc/32-bit networked sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 264–277.

[33] 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.

[34] 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.

[35] 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.

[36] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent computing: Challenges and opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[37] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices* (2015).
[38] Yunfei Ma, Zhihong Luo, Christoph Steiger, Giovanni Traverso, and Fadel Adib. 2018. Enabling deep-tissue networking for miniature medical devices. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 417–431.
[39] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* (2017).
[40] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.
[41] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1005–1021. https://doi.org/10.1145/3385412.3385998
[42] Andrea Maioli and Luca Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys 2021)*. 261–273.
[43] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2020), 1–24.
[44] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems Across Hardware and Software. In *Proc. USIT* (Oct. 22–25). ACM, Québec City, QC, Canada, 299–310.
[45] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. 2018. WiFröst: Bridging the Information Gap for Debugging of Networked Embedded Systems. In *Proc. UIST* (Oct. 14–17). ACM, Berlin, Germany, 447–455.
[46] Microbit. 2021. Meet the new BBC micro:bit. https://microbit.org/new-microbit/. [Online; accessed 02-March-2021].
[47] Microsoft. 2020. MakeCode: Hands on computing education. https://www.microsoft.com/en-us/makecode. [Online; accessed 02-March-2021].
[48] Microsoft. 2021. MakeCode Maker. https://maker.makecode.com/. [Online; accessed 02-March-2021].
[49] 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].
[50] Stefanos Nikiforos, C Kontomaris, and K Chorianopoulos. 2013. MIT Scratch: A Powerful Tool for Improving Teaching of Programming.
[51] Erik Pasternak, Rachel Fenichel, and Andrew N Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 21–24.
[52] Powercast. 2021. Wireless energy harvesting and RF Power. https://www.powercastco.com/. [Online; accessed 02-November-2021].
[53] R Venkatesha Prasad, Shruti Devasenapathy, Vijay S Rao, and Javad Vazifehdan. 2013. Reincarnation in the ambiance: Devices and networks with energy harvesting. *IEEE Communications Surveys & Tutorials* 16, 1 (2013), 195–213.
[54] Damith C. Ranasinghe, Roberto L. Shinmoto Torres, Alanson P. Sample, Joshua R. Smith, Keith Hill, and Renuka Visvanathan. 2012. Towards Falls Prevention: a Wearable Wireless and Battery-less Sensing and Automatic Identification-Tag for Real Time Monitoring of Human Movements. In *Proc. EMBC* (Aug. 28 – Sep. 1). IEEE, San Diego, CA, USA, 6402–6405.
[55] 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.
[56] 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.
[57] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. 2017. Scanalog: Interactive Design and Debugging of Analog Circuits with Programmable Hardware. In *Proc. USIT* (Oct. 22–25). ACM, Québec City, QC, Canada, 321–330.
[58] 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.
[59] 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*.
[60] Nicolas Villar and Steve Hodges. 2010. The Peppermill: A human-powered user interface device. In *Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction*. 29–32.
[61] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article 3 (oct 2017), 25 pages. https://doi.org/10.1145/3089799
[62] Harrison Williams, Xun Jian, and Matthew Hicks. 2020. Forget failure: Exploiting sram data remanence for low-overhead intermittent computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 69–84.
[63] Harrison Williams, Michael Moukarzel, and Matthew Hicks. 2021. Failure sentinels: ubiquitous just-in-time intermittent computation via low-cost hardware support for voltage monitoring. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*

Table 8. Open ended responses in the user study.

Questions	Responses
What are possible use cases for these battery-free embedded devices?	→A device that records time spent washing hands. →Device looking for leaks in the town plumbing system that broadcasts pressure data once every 5 minutes. →A package notifier which is solar powered. →A ball that keeps track of how many times it's been hit/thrown for use in sports.
What are strong points about Makecode-Iceberg?	→Very easy to use, other than some annoyances and repetitiveness of block based programming →It seems to work seamlessly, meaning there isn't really a downside to having it in every program. →It can be used outside of a controlled environment in an energy efficient way →Automatic state saving, you don't need to know how it works to use it
What are weak points about Makecode-Iceberg?	→Hard to debug to see what is stored in FRAM →Needs to save 100% of the variables 100% of the time, otherwise it's basically useless →It might not work for all applications outside of a controlled environment due to extreme conditions →Startup time after regaining power is a bit slow →Not all the functions in Make code are covered in Iceberg, like loops and plotting.
Did MakeCode-Iceberg change your perception of the applications of energy harvesting devices?	→I was already aware of the challenges, but this definitely would be a powerful tool to get into the hands of young developers to get them thinking about the new doors this opens. →Yes, I didn't realize that there were such a wide variety of applications →No →Yes. This makes intermittent power/energy-harvesting devices seem like something I could actually use as a programmer, without being that experienced in hardware.
Do you have any other remarks about Makecode-Iceberg?	→Very cool software, I hope it gets widely used →Great idea, I hope it becomes a common toggle option in novice prototyping IDEs.

(ISCA). IEEE, 665–678.

[64] 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.

A APPENDIX: USER STUDY QUESTIONNAIRE AND RESPONSE

We present the full responses to the portion of the user study questionnaire on usability and applications in Table 8, which augments the description in Section 7.