Effortless Intermittent Computing

Implicit Power Failure Resilience with Task-Oriented Programming

Software Science Master's Thesis

January, 2025

Cas Visser

Supervisor
dr. Mart Lubbers

Second Reader
dr. Peter Achten

Radboud Universiteit

Abstract

The *Internet of Things* is growing rapidly, to over a trillion (10^{12}) devices in the coming decade. Powering all of these devices with lithium-ion batteries would put significant strain on the environment. *Energy harvesting* is an increasingly popular alternative, but leaves devices with an unreliable, i.e. intermittent, power supply. In order to be effective despite losing power frequently and unexpectedly, new programming techniques are required: intermittent computing.

We show that the Task-Oriented Programming (TOP) paradigm is well-suited for intermittent computing. TOP is a programming paradigm in which programs consist of tasks, which are combined with task combinators. The mTask language is a DSL embedded in Clean which follows the TOP paradigm. This work presents an extension to mTask making mTask programs resilient to intermittency without additional effort from the programmer. We achieve this by implementing checkpointing infrastructure, logic for when to resume from a checkpoint, and by configuring communication using MQTT appropriately.

This extension makes mTask the first TOP language with built-in intermittent computing functionality. Additionally, the mTask system is more flexible than other intermittent computing frameworks. Moreover, combining intermittent computing with multithreading is considerably more straight-forward using mTask.

Acknowledgements

I am immensely grateful to all those who supported me in the process of writing this thesis. It was not an easy task. I especially want to thank the following people:

- Dr. Mart Lubbers for sacrificing a ridiculous amount of time to meet with me every week. You have not only taught me a great deal about computing science, but also about scientific writing and, surprisingly, a lot about typography – for which I am just as grateful.
- Dr. Peter Achten for agreeing to be second reader for this thesis. I hope it does not make you regret sharing an office with Mart.
- Paul for introducing me to the Typst language, saving me from endless LaTeX struggles, and for answering countless questions about the Typst struggles that I ran into instead.
- My parents for their endless patience with my long academic journey and for enabling me to undertake it.
- Annemiek, Aron, Elise, and Maaike for being amazing people.

Contents

1. Introduction **1**
2. Task-Oriented Programming **2**
3. TOP for the IoT **5**
3.1. The mTask system 6
3.2. Execution model of mTask 7
3.3. Task trees 8
4. Intermittent Computing **8**
4.1. Use Cases and Relevance 8
4.2. Challenges 9
4.3. Strategies 11
4.4. With Task-Oriented Programming 13
5. Extending mTask with Implicit Intermittent Computing **14**
5.1. Implementation 14
5.2. Evaluation 18
6. Related Work **18**
6.1. Task-Oriented Programming 18
6.2. Intermittent Computing 19
7. Future Work **20**
7.1. Atomic Tasks 20
7.2. Voltage Test 20
7.3. Functional Reactive Programming 21
7.4. Other Intermittent Computing Concepts 21
8. Conclusion **22**
Abbreviations **23**
References **23**

1. Introduction

From fridges to cars and from phones to thermostats: the number of devices that is connected to the internet is increasing rapidly. This collection of devices, often called the Internet of Things (IoT), is as varied as it is vast. The number of IoT devices is expected to surpass one trillion (10^{12}) in the next decade [1]. IoT devices need electrical power to operate. Currently, this power is often supplied using a lithium-ion battery, which can store a large amount of power for a long time using little space. However, they can only be recharged a limited number of times, meaning they typically last just 5 to 10 years, and cannot easily be recycled. Powering the expected one trillion devices with lithium-ion batteries would therefore put a significant strain on the environment.

An alternative to using lithium-ion batteries, is to rely on Energy Harvesting (EH): opportunistically drawing electrical power from the environment, for instance using a solar panel or using the kinetic energy from a person walking. Although these methods offer cheap energy, they come with an additional challenge. In most cases, they cannot provide power consistently. A solar panel only provides power when there is enough sunlight. A podometer harvesting kinetic energy can only do so while the person wearing it, is walking. When no energy is harvested, the device shuts down.

Due to the way computer memory works, a failing power supply causes significant issues. Typically, computers use what is called *volatile memory* to operate. This memory constantly requires power in order to retain its data. When power is lost, the memory is erased. Computers may also have non-volatile memory (NVM), but this is often slower and sometimes can only be rewritten a limited number of times, meaning some amount of volatile memory is required.

As a result, when power is interrupted, a device cannot simply continue what it was doing before the interruption. By default, it will start over from the beginning of its program. If power interruptions are more frequent than the time it takes to run to completion, the device will not be able to complete its program. *Intermittent computing* is the field of research that aims to find solutions for this challenge.

A key strategy in intermittent computing is to periodically store the information we have in volatile memory in NVM, i.e. make a *checkpoint*. When power returns, we can continue from this checkpoint, instead of having to start over from the beginning. This is not a trivial task however. For instance, creating a checkpoint takes time and electrical power, reducing effectiveness of the device, so we do not want to do this too frequently. On the other hand, the more time has passed since the last checkpoint, the more work is lost. Furthermore, it is not trivial to ensure that relying on checkpoints does not break correctness of the program.

Moreover, the implementation of strategies that deal with power failures is similar over different applications, so ideally we want to separate the required code. As an example, we could collect the instructions for creating a checkpoint into a separate function. This is not enough though, as calls to this function would still be littered throughout the codebase. This makes the code hard to read and thereby hard to maintain and prone to errors.

Several frameworks exist that attempt to help the programmer to make their programs *intermittent aware*: resilient to unreliable power supply. However, they all come with their own challenges and limitations, from requiring major programming expertise to not supporting multithreading. We apply intermittent computing to Task-Oriented Programming (TOP), a relatively new programming

paradigm, that allows for creating a system where programs are made intermittent aware without any additional effort from the programmer. More concretely, we extend mTask, a TOP language, with intermittent computing functionalities.

The remainder of this work is structured as follows. We first give an introduction to TOP (Section 2) and mTask (Section 3), a TOP language for the IoT. Then, we discuss intermittent computing, what its challenges are, and the current state of the art (Section 4). Next, our main contribution, an extension to mTask that makes mTask programs intermittent aware without additional effort from the programmer (Section 5). We discuss how mTask compares to other intermittent computing solutions (Section 6) and further research opportunities (Section 7).

2. Task-Oriented Programming

TOP is a programming paradigm where program logic and GUI are generated from a single source. Other approaches require the programmer to define these manually, or even use multiple different languages for different parts of an application. TOP allows for creating interactive and distributed programs while using a declarative programming style.

In TOP, programs consist of tasks, which are combined using task combinators. A task represents a piece of work that needs to be done. A task in progress has an unstable value or no value. Once a task completes, its value is stable and can no longer change. Tasks can observe task values of other tasks through task combinators. There are many different task combinators, which make it possible to, for instance, only start the next task once the current task has a stable value, select a next task based on the value of the previous task, or to run tasks concurrently.

Besides task values, tasks communicate using Shared Data Sources (SDSs). A SDS can be, for example, a part of a file system, a stream of random bytes, or a clock. Tasks can read from and write to SDSs, giving them a way to communicate to the outside world. In addition, SDSs form another way for tasks to communicate with each other besides using task values.

The iTask system [2] is an implementation of TOP. It is in active use, both in the academic world, where it has been actively researched for more than 20 years, and in the industry, where it is for example used to power coast monitoring software VIIA [3]. The iTask system implements the TOP paradigm as an embedded Domain-Specific Language (DSL) in host language Clean [4]. The system runs a webserver with multi-user and distributed computing capabilities to present a GUI. The GUI consists of interactive webpages, generated from the source code.

We further demonstrate iTask using an example, namely a program that converts between written and sounding note names [5]. The remainder of this section goes into some music-theoretical detail, but the important part is that musicians have different names for the same sounds, and this app can help clarify this difference.

Due to historical and practical reasons, some instruments, e.g. saxophone and trumpet, are *transposing instruments*, which means the names they give to notes are shifted compared to non-transposing instruments. For instance, when someone who plays a B♭-instrument plays what they call a C, the note they play will sound like a B♭ to non-transposing instruments [6]. Converting between written and sounding pitches requires some counterintuitive arithmetic which confuses even experienced musicians, so a tool that helps with this may be of significant value.

```
1 :: Pitch = A | B | C | D | E | F | G
2 :: Accidental = Flat | Natural | Sharp
3 :: Key = {pitch :: Pitch, accidental :: Accidental}
```

Listing 1: Data types for modeling tones/keys.

```
1 transposeUp :: Key Key -> Key
2 transposeUp key transposeBy =
3   keyFromInt ((keyToInt key) + (keyToInt transposeBy))
4
5 transposeDown :: Key Key -> Key
6 transposeDown key transposeBy =
7   keyFromInt ((keyToInt key) - (keyToInt transposeBy))
```

Listing 2: Functions for transposing (`keyToInt` and `keyFromInt` ommitted).

To start, we define some custom data types to represent tones (Listing 1). For brevity, we use the same data type for tones and keys. We can convert each tone to its index in the 12-tone chromatic scale, i.e. how many semitones it is above C natural, and use this to transpose one tone *by* another tone (Listing 2). Doing this naively means we cannot differentiate between enharmonically equivalent tones, i.e. A♯ and B♭, but we accept this shortcoming for the sake of keeping the example compact.

For the actual application, we want the user to be able to specify a source key and one or more keys to calculate written names for (Listing 3). We define two SDSs (lines 1–5), with C as the default value for the source key (line 2). For each of these data sources, we define a task that updates it (lines 7–9 and 11–17 respectively). On line 9, we use the built in task `updateSharedInformation` to change the value of the SDS that holds the source key. We use the `<<@` modifier to add a title to the input form. On line 14, we use the `>>*` task combinator to specify a list of possible task continuations. This combinator first starts the task on the left side. On the right side is a list of continuations, which consist of a condition over a task value. When a condition holds, that task starts.

In this case, we specify two `OnAction` continuations, which show up as buttons in the final application. One continuation simply provides a way to end the program (line 17). The other continuation (lines 14-16) adds the user input to the SDS and then uses the `>-|` task combinator to recursively call this task again. This combinator first starts the task on the left and once this task has a stable value, continues with the task on the right. It ignores the result from the task on the left. Restarting the task causes the input fields to be reset and allows the user to enter another key.

The main functionality of this program is to generate a table showing the different names (Listing 4). We generate the table as a customly defined HTML table (lines 3–6). For brevity we omit the exact details of building the table.

Finally, we tie everything together to form a single program (Listing 5). We use the parallel task combinator `-||` to combine the previously defined tasks. This combinator starts both tasks in parallel and has as result the result of the task on the side with the minus. We use the view from Listing 4 to transform the data from the SDSs (line 6). The `|*|` operator combines the SDSs to give us a single SDS that matches the type of `viewAsTransposingTable` (line 7).

```
1  sourceKeySDS :: SimpleSDSLens Key
2  sourceKeySDS = sharedStore "sourceKey" {pitch=C, accidental=Natural}
3
4  transposingKeysSDS :: SimpleSDSLens [Key]
5  transposingKeysSDS = sharedStore "transposingKeys" []
6
7  selectSourceKey :: Task Key
8  selectSourceKey =
9    updateSharedInformation [] sourceKeySDS <<@ Title "Select source key"
10
11 addTransposingKey :: Task ()
12 addTransposingKey =
13   enterInformation [] <<@ Title "Select transposing keys"
14     >>* [ OnAction (Action "Add key") (
15                  hasValue \k-> upd (\ks -> [k:ks]) transposingKeysSDS
16              >-| addTransposingKey)
17         , OnAction (Action "Quit") (always (shutDown 0)) ]
```

Listing 3: SDSs and tasks to gather user input.

```
1 viewAsTransposingTable :: (Key, [Key]) -> HtmlTag
2 viewAsTransposingTable (sourceKey, transposingKeys) =
3   TableTag []
4     [ TheadTag [] [ TrTag [] header ]
5     , TbodyTag [] (map row transposingKeys)
6     ]
7 where
8   header :: [HtmlTag]
9   row :: Key -> HtmlTag
```

Listing 4: Task for generating the transposing tones table.

Figure 1 shows a screenshot of the resulting application. The UI elements are all automatically generated from the source code discussed above. The appearance may be a bit austere, but this can be modified using custom HTML and CSS if desired.

```
1 calculator :: Task Key
2 calculator =
3       selectSourceKey
4   -|| addTransposingKey
5   -|| viewSharedInformation
6         [ViewAs viewAsTransposingTable]
7         (sourceKeySDS |*| transposingKeysSDS)
8
9 Start w = doTasks calculator w
```

Listing 5: Main task for the transpose calculator.

Figure 1: Webpage generated by iTask transpose calculator.

3. TOP for the IoT

The IoT is a term used for devices that collect data using sensors and communicate this data over the internet. IoT devices are typically small, cheap, and limited in processing power and available energy. Examples of IoT devices include smart watches, RFID tags to identify parts in an industrial plant, and wirelessly communicating sensors of a climate control system [7]. It is estimated that the number of IoT devices will grow to over one trillion (10^{12}) in the coming decade [1].

IoT applications typically consist of a number of sensing nodes and a single server, where the sensing nodes are usually cheap and have little computational power while the server is generally more powerful. The sensing nodes collect data, which they relay to the server, which processes and stores it. IoT applications with this setup can typically be considered as consisting of several layers.

1. Perception layer. Data is gathered from the environment with sensing nodes, for instance by reading out a temperature sensor, recording sound using a microphone, or measuring light intensity.
2. Network layer. The sensing nodes communicate with the server, for instance to forward data or receive commands.
3. Application layer. The server processes the data, for instance storing it in a database or sending commands based on a certain reading.
4. Presentation layer. To allow humans to interact with the system, the server provides an interface, often in the form of webpages, where information about the system is presented, for instance a live view of recently gathered data.

In most cases, different parts of an IoT application are implemented using several different languages and paradigms, e.g. C for programming resource-constrained sensing nodes, SQL for the database, HTML and PHP for the webpages, and Python to glue it all together. This is also called tiered programming. In tiered programming, a considerable amount of code is needed to interface between the different parts of the system.

The alternative to tiered programming is *tierless* programming, which the TOP paradigm is specifically designed for and which has several advantages. In a tierless approach, all of the layers mentioned above can be created using a single paradigm and a single source. In addition, a large part of the code that programmers have to write manually in traditional tiered approaches is generated automatically in TOP, for instance the HTML code and the code for communication. This leads to a smaller codebase, which in turn leads to fewer errors and lower maintenance effort [8].

The remainder of this section explores the mTask system, which is a TOP language made specifically for the IoT.

3.1. The mTask system

The mTask system [9] is a DSL embedded in the pure, lazy functional language Clean. Microcontrollers are typically not powerful enough to run a high-level system such as iTask, whereas mTask is designed specifically for the IoT and can work with as little as 2 KiB of RAM. It supports interrupts and features a scheduler, allowing for minimal power consumption [10]. The mTask system integrates with the iTask system, which means it is possible to extend iTask programs with mTask tasks. This setup allows for powerful, useful applications where parts of the application are executed on edge devices, with all code generated from a single source file. To demonstrate this, a prototype smart campus application was built using mTask [8].

The mTask system works with clients and a server. The client is typically a resource-constrained microcontroller on which a light-weight Runtime System (RTS) is installed, which runs mTask bytecode. The server compiles the mTask program to bytecode and sends this to the client. The client executes the bytecode and communicates the result back to the server. Although the byte code is generated at runtime, type-safety is ensured by the Clean compiler. Once a device is set up with the RTS, it can execute any mTask program without needing to be reprogrammed. Figure 2 shows how the different components of the mTask system work together.

Similar to iTask, programs written in mTask consist of tasks, combined with task combinators. A subset of the task combinators from iTask is also present in mTask, though they are syntactically separated. Like in iTask, tasks in mTask either have no value, or a stable or unstable value.

Listing 6 shows a Clean program that uses iTask and mTask to blink an LED at a specified interval. It was written for a *Wemos D1 Mini*, for which General Purpose Input-Output (GPIO) pin `D4` is connected to the built-in LED. The program consists of a single iTask task `mainTask`, which asks the user to specify a device. When this specification is entered, the mTask task `blinkTask` is compiled to bytecode and sent to the device using the build in `liftmTask` function.

The task `blinkTask` first specifies that pin `D4` will be used as output (line 12). Then, it defines a function `blink` taking one argument `st` (line 13), which consists of the following three basic tasks.
- Write `st` to the previously specified pin (line 14).
- Wait 500 milliseconds (line 15).
- Call `blink` recursively (line 16).

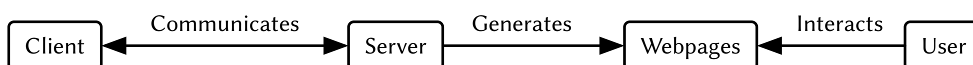


Figure 2: Components of the mTask system.

```
1  Start w = doTasks mainTask w
2
3  mainTask :: Task ()
4  mainTask = enterDevice
5     >>? \spec -> withDevice spec (\dev -> liftmTask blinkTask dev)
6     >>* [
7           OnAction (Action "Reset") (always mainTask)
8          , OnAction (Action "Stop") (always (shutDown 0))
9          ]
10
11 blinkTask :: Main (MTask v Bool) | mtask v
12 blinkTask = declarePin D4 PMOutput \ledPin->
13     fun \blink=(\st->
14               writeD ledPin st
15          >>|. delay (lit 500)
16          >>|. blink (Not st)
17     ) In {main=blink true}
```

Listing 6: iTask and mTask code for blinking an LED.

These three basic tasks are combined with the `>>|.` combinator, which starts the next task once the previous task is stable. Finally, line 17 specifies that the main function of this task consists of a call to `blink` with argument `true`.

In the call to `delay`, the number of milliseconds is wrapped in `lit`, which lifts it into the mTask language. In the recursive call, the state is toggled using the mTask function `Not`.

3.2. Execution model of mTask

After startup, the mTask client enters into the main program loop, which consists three separate phases: communication, execution, and memory management.

Communication During the communication phase, the client processes any messages it has received from the server, which can be e.g. a task to execute, a new value for a shared data source, or a command to reset. The client acknowledges the server's messages and sends any return values it has obtained.

The mTask system does not fix a specific communication method. Rather, it provides an interface, which can be implemented to support different communication methods. Wi-Fi, TCP, and serial communication come with the system.

Execution Tasks are executed using an interpreter and a rewriter. An mTask task starts as an expression. Expressions are evaluated by the interpreter, which results in a task tree. The rewriter tries to rewrite task trees, which may result in further expressions to be evaluated by the interpreter. After a rewrite step, the task value is updated. Task trees are discussed in more detail in Section 3.3.

During the execution phase, one rewrite step is performed for each task for which this is expedient, i.e. which is not sleeping or otherwise blocked. The steps are small, so tasks can be executed seemingly in parallel by interleaving tasks. A scheduler determines which tasks should be executed, balancing timely execution and energy efficiency.

Memory Management The mTask RTS reserves a single block of RAM, within which it applies its own memory management. Task trees are allocated in high address space, task information in low address space. The task information includes a pointer to the root of the corresponding task tree, the current task value, and the byte code of the task. The rewriter and interpreter build their stack on top of the task information, but it does not persist between steps. That is, at the end of the execution phase, the interpreter stack is empty, so at this point it is safe to change the task information.

Task trees of finished tasks and unreachable task tree nodes are garbage collected during this phase.

3.3. Task trees

A task in progress is represented by a task tree. The nodes of the tree are task combinators and the leaves are basic tasks, e.g. `delay`, `rtrn`, or writing to a GPIO pin.

For instance the task `blinkTask` from Listing 6 starts as the expression `blink true`. After the interpreter evaluates this expression, we obtain the task tree shown in Figure 3(a). Applied to this tree, the rewriter first encounters the `>>|.` combinator in the root node. It then first handles the left child, where it uses the interpreter to evaluate `writeD ledPin st`, which as a result toggles the LED. After this task completes, we can rewrite to obtain the task tree shown in Figure 3(b), which concludes this rewrite step. Next up is the evaluation of `delay (lit 500)`, after which the scheduler will not schedule this task until the delay has run out.

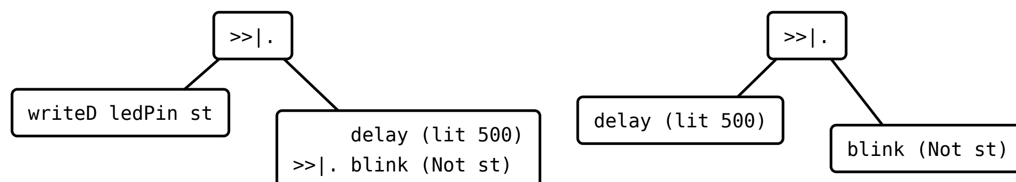
4. Intermittent Computing

Intermittent computing is about computing with an unreliable, i.e. intermittent, power supply. By default, a computer starts the program it is running over from the beginning, meaning it cannot run to completion when there are frequent power failures. What we want instead is when power returns, for the device to continue as if a power failure never occurred. This section goes over the use and importance of intermittent computing, as well as its main challenges.

4.1. Use Cases and Relevance

As discussed in Section 3, the number of IoT devices is expected to exceed one trillion (10^{12}) by the next decade [1]. Many of these devices are powered using a lithium ion battery. Obtaining lithium from the earth requires a chemical process that is harmful to the environment. Additionally, these batteries have a limited lifetime before their performance starts to degrade. Moreover, they are expensive and difficult to recycle [11].

It is therefore clear that we should try to limit the number of batteries needed. One way to do this, is to rely on EH: obtaining useful energy from the environment. This can be done in numerous



(a) Initial task tree. (b) Task tree after rewrite step.

Figure 3: First two task trees for `blinkTask` from Listing 6.

ways, for instance using a solar panel, harvesting kinetic energy, or even harvesting energy from stray radio waves. EH devices typically store the energy they harvest in a capacitor. A capacitor stores electrical energy just like a lithium-ion battery can, but is cheaper, can be charged faster, is far less environmentally unfriendly to produce, and has a much longer lifetime. On the other hand, capacitors are less energy dense and lose up to 20% of their stored energy per day, making them unsuitable for replacing batteries outright.

The power supply from EH is not as reliable as from a power grid or battery, but it is freely available, comparatively environment-friendly, and EH devices may keep working for decades without maintenance.

Several products using EH are already available to consumers. For instance, a button that harvests energy from being pressed. The harvested energy is used to communicate with a centralized system, allowing the button to be placed anywhere, without needing any additional wiring or other installation work and without requiring a battery [12]. There is also a smart watch that combines energy harvested from body heat and solar energy, so that it never requires charging and can be worn indefinitely [13].

A possible criticism of intermittent computing is that, even with state-of-the-art technology, a batteryless device cannot offer the same reliability as the same device with a battery. After all, there are always periods of time when no energy can be harvested and during these times, a battery-free device cannot operate while a device with a battery can. In many devices that monitor the environment however, data collection is only interesting at moments when EH is possible. For instance, a weather station measuring solar intensity only needs to work when the sun is visible, meaning there is solar energy to harvest. A step counter only needs to work when the user is making steps from which kinetic energy can be harvested. Another way to frame this is to say that absence of power may be considered part of data collection.

4.2. Challenges

There are several challenges to using EH instead of relying on batteries [14], but this work focuses on the software challenges. In particular, it focuses on creating effective, efficient, and maintainable software for devices that may lose power regularly and unexpectedly.

The main challenge comes from *volatile memory*, which is used by virtually all computers for RAM and registers. Volatile memory needs power to retain data. This means that as soon as the power fails, the program counter, the system clock, and program memory are lost. Hence, when power is restored, the device cannot determine what it was working on or what time it is. There also exists NVM, for instance solid state drives, SD cards, FRAM, and EEPROM, but these types of memory have much lower reading and writing speed and sometimes also a limited number of write cycles, meaning we almost always need to use some volatile memory as well.

For small, simple applications a solution could be to periodically store all important variables in a file in NVM and read them from this file at startup: a checkpoint. However, doing this manually quickly becomes infeasible for larger applications. As the number of program states grows, it may become difficult to determine where the program should continue. Which variables need to be stored may change over time. Most importantly, the code will become bloated with infrastructure ensuring the volatile memory is saved at the right time. This is further demonstrated in Section 4.3.1.

For larger applications, it is clear we need an organized way to handle power failure. Ideally, we also want to take some responsibility away from the programmer and build this resilience into the compiler, instead of asking the programmer to solve the same problems again for each program.

We can break the challenge of handling intermittent power up into several sub-challenges.

Progress If the checkpointing interval is larger than the power failure interval, our program will start from the same point every time there is power and never run to completion. We want to be able to guarantee progress under as many circumstances as possible.

Correctness Not all state can safely be checkpointed. Connections to other devices may need to be re-established and peripherals may need to be re-initialized. Some operating systems use Address Space Layout Randomization (ASLR), in which case stored pointers are no longer correct after rebooting. Additionally, the timing of writing the checkpoint can cause incorrectness. If an instruction has non-volatile side effects, but no checkpoint is made after it is executed, undefined behavior may occur. This is sometimes referred to as a Write-After-Read (WAR) dependency [15]. See also Section 4.2.1. The intermittent computing infrastructure that is automatically added should take all of this in account and preserve the correctness of the code provided by the programmer.

Timeliness Some information is only useful for a limited time. For instance, a temperature measurement may only be accurate for a few minutes. If a device measures temperature and then loses power for some time, it should use not use the old measurement to make decisions, as it is no longer reliable. This requirement is called *freshness*. Alternatively, instead of the time between an event and now, we may instead have a requirement for the maximum time between two events. For instance, a temperature sensor and humidity sensor read at the same time can provide an estimate for wet bulb temperature. This requirement is called *temporal consistency* [16]. We want the programmer to be able to specify these constraints. A power failure not only delays execution, it may also erase the system clock, meaning there is no means of knowing how old a recovered value is. Some implementations use dedicated hardware to estimate the amount of time passed. One way of doing this, is by measuring the amount of power a capacitor has lost, as this happens reasonably consistently over time. Another method is to save some power to keep powering the real-time clock, which only takes a tiny amount of power, e.g. ~20 µA on the ESP8266 [17].

Communication If a message is sent to an intermittently powered device while it is off, the message is lost. Moreover, if two intermittently powered devices want to communicate, the probability of them having power at the same time drops drastically as the proportion of uptime decreases. Specialized communication protocols are needed for effective transmission of data.

Efficiency Writing to NVM takes up processor cycles, meaning the device has less time to do actual work. Additionally, this consumes power, which is scarce in EH devices. This means we want to limit how often we make checkpoints as much as possible. On the other hand, as more time has passed since the last checkpoint at the moment of a power failure, more work is lost and less progress is made. Hence, this is a delicate balance.

Usability Apart from timeliness constraints, power failure logic is largely separate from program logic. In the majority of cases, we simply want our program to continue as if no power failure has occurred. Therefore, it makes sense to separate the intermittent computing functionality into a resuable, program-agnostic framework, e.g. a library with intermittent computing tools, or a

```
1 x = 0
2 ...
3 checkpoint()
4 x += 1
5 if x > 1:
6   ...
7 checkpoint()
8 ...
```

Listing 7: Checkpointing non-volatile state leads to errors.

declarative programming style that abstracts away the intermittent computing functionality. This way, the programmer is concerned only with the *what*, and the source code does not get bloated with the *how*.

4.2.1. Checkpointing Errors

As discussed in Section 4.2, it is not trivial to preserve correctness when creating and restoring checkpoints, especially when volatile and non-volatile memory are mixed. One solution is to divide code into *atomic sections* that are *idempotent*. An atomic section is a section of code that must be executed without interruption and without power failure. If a power failure occurs during an atomic section, execution should resume from the beginning of the section. A section of code is idempotent if it can be executed multiple times without changing the outcome of the program.

To illustrate this, see Listing 7, where `x` represents some non-volatile state. On line 4, `x` is incremented between checkpoints. If a power failure occurs between these two checkpoints, execution resumes from the first checkpoint and `x` is incremented again, meaning the final value of `x` is determined by power circumstances. Since `x` is used in a conditional statement on line 5, this in turn affects the flow of the entire program.

This problem is sometimes called a Write-After-Read (WAR) dependency: variable `x` is written (line 4) after it is read (line 1). A possible solution is to execute the read and write as an atomic section, meaning no checkpoint can occur between line 1 and line 4. Together, lines 1 and 4 are idempotent. After executing them, the value of `x` is 1. Conversely, line 4 on its own is not idempotent. Every time it is executed, the resulting value of `x` is different.

4.3. Strategies

Creating intermittent computing software remains an open challenge. Over the past decade, several intermittent computing strategies have emerged, focusing on different challenges. The strategies can be roughly divided into three categories, discussed in the following three sections.

4.3.1. Explicit Intermittent Computing

It is possible to manually make a program power-failure resistant. Listing 8 demonstrates this via a Python implementation of a power-failure resistant counter. Upon startup, the program tries to restore the counter from a file. If this fails, it resets the counter to 0. Though this works, it is easy to see how even in a high-level language such as Python, the complexity of the code would quickly reach unmanagable levels for more interesting programs. Basic requirements such as keeping track of what state needs to be included in the checkpoint, making checkpoints at suitable times, and

```python
def load_checkpoint():
  try:
    with open("cp", "r") as f:
      return int(f.read())
  except Exception as e:
    return 0

def save_checkpoint(state):
  with open("cp", "w") as f:
     f.write(state)
```

(a) Checkpointing functions.

```python
def main():
  # Try to restore checkpoint
  # Returns 0 upon failure
  counter = load_checkpoint()
  while True:
    counter += 1
    save_checkpoint(counter)
    ...


```

(b) Main function.

Listing 8: Simple intermittent Python program.

not making too many checkpoints to conserve power, would already clutter the codebase beyond a usable level.

To further illustrate this point, we discuss BOOTHAMMER, which provides some infrastructure for checkpointing [18]. It builds on the Arduino framework, in which programs are based around a setup function and a loop function. The setup function is run once, after which the loop function is repeated indefinitely. The BOOTHAMMER tool inserts itself in the compilation chain to add checkpointing.

Although BOOTHAMMER guarantees safety and correctness, programmers need to manually specify where in their code a checkpoint is saved. The only checkpoint that BOOTHAMMER adds automatically, is at the end of the loop function. However, the loop function may take an arbitrary amount of time, for instance if the loop waits for communication. Moreover, to ensure correctness, BOOTHAMMER creates a checkpoint of the entire RAM, though it does offer an option to only checkpoint the stack. Without expert programming knowledge, BOOTHAMMER adds major overhead, increasing runtime hundredfold in some cases.

The limitations of BOOTHAMMER are representative for this approach. It is clear then, that not all applications can be made battery-free with explicit intermittent computing with reasonable effort, and that more involved frameworks are needed.

4.3.2. Task-Based Programming

In task-based programming (not to be confused with TOP), programmers arrange their code into tasks [19]. This task structure exists in addition to what the program should do. In this paradigm, a task is a section of code that should be executed atomically. Additionally, tasks should be idempotent, so that it is safe to make a checkpoint before and after a task. In addition to dividing code into tasks, the programmer needs to specify a control flow, i.e. in what order the tasks need to be executed. In some cases, the programmer also needs to define how tasks can share data [20].

Based on this task structure and control flow, the framework then ensures progress and timeliness. In task-based programming, it is also possible to schedule tasks dynamically, taking into account current power availability [20].

4.3.3. Implicit Intermittent Computing

There are also frameworks which add the intermittent functionality implicitly [21], [22]. These build on an existing programming language and extend the compiler or the interpreter to add checkpointing infrastructure based on static analysis.

In some cases, some language constructs are added as well for finer control, for instance allowing users to manually specify atomic regions [16].

4.4. With Task-Oriented Programming

The TOP paradigm is well suited for implicit intermittent programming. In functional programming languages, variables are immutable. This means e.g. the problem from Listing 7 does not appear as such in TOP. Tasks form an abstraction over side effects. That is, tasks themselves may have side effects, but each step happens atomically.

To illustrate this, consider Listing 9, which shows how the snippet from Listing 7 might be implemented using iTask. Instead of a variable `x`, we have a SDS `xSDS` which holds an integer value. We write 0 to the SDS on line 1 and on line 3, we increment the value by 1. The `upd` function has the value that gets written as result. This is captured by the `>>*` combinator on line 4, which has a continuation which is only activated if this value is larger than 1. This corresponds to the if-statement on line 5 in Listing 7.

The important difference compared to Listing 7 is that the update on line 3 happens atomically. If power fails after the update, but before a checkpoint is made, the SDS is restored to the state saved in the previous checkpoint. This does not work for external state. If instead of updating a SDS, we e.g. activate a motor which pushes over a line of dominos, there is no obvious way to restore this from a checkpoint. This is not something we can solve or want to solve automatically, but rather something that requires active input from a programmer, for instance in the form of atomic sections, which are discussed in Section 7.1.

Besides checkpoint safety, the TOP paradigm boasts several other advantages for intermittent computing. For instance, after each rewrite step, only the task information, the task trees, and some auxiliary data are in RAM. This results in checkpoints being quite small compared to a checkpoint of the stack and heap of a program written using an imperative language. Moreover, the rewrite steps are small, so checkpointing opportunities are very frequent. This means we can postpone making a checkpoint until power gets quite low, without having to fear power running out unexpectedly. Furthermore, the mTask RTS can be restarted quickly when power comes back. This is due to the small checkpoint size and the low overhead.

```
1     set 0 xSDS
2 >-| ...
3 >-| upd ((+)1) xSDS
4 >>* [ OnValue (ifValue ((>)1) ...), ... ]
```

Listing 9: Updating state does not lead to a WAR dependency in TOP.

5. Extending mTask with Implicit Intermittent Computing

Our main contribution is an extension of the mTask system such that mTask programs can operate intermittently without additional effort from the programmer. We describe our implementation in Section 5.1 and verify that it works in Section 5.2.

5.1. Implementation

The mTask RTS implementation features a range of software interfaces to separate components and to tailor the build for different systems. Additionally, features can be altered using compile-time macros. Currently, it can be built to run in a PC environment or to integrate with the Arduino ecosystem.

The Clean library `gentype` is used to serialize objects, needed for saving and sending. As an added benefit, the `gentype` library also generates code to deserialize objects as C objects, which we need as the RTS is implemented in C and the server is implemented in Clean.

As discussed in Section 4.4, we get some intermittent computing challenges *for free*. The challenges that remain, are tackled in the sections below. First, in Section 5.1.1, we configure the communication so that the mTask server and client can communicate despite the client being unreachable periodically. Then, in Section 5.1.2, we implement creating and restoring a checkpoint. Lastly, in Section 5.1.3, we implement a policy to use on startup to determine whether we should try to resume from a checkpoint. All our changes are implemented in the mTask client.

5.1.1. Communication

The mTask RTS provides an interface that allows implementing additional communication methods. One of the available implementations is an implementation of the MQTT protocol, which is a communication protocol designed for IoT-like systems. We modify this implementation to work with intermittent computing.

The MQTT protocol is centered around a *message broker*. Instead of communicating directly, participants of the network send messages to and receive messages from the broker. To this end, MQTT clients — the *mTask* server and client are both *MQTT* clients — specify *channels* on which they *publish* messages, i.e. send them to the broker. Other MQTT clients may *subscribe* to a channel. Upon receiving a message on a channel, the MQTT broker forwards the message to all MQTT clients subscribed to this channel. Figure 4 shows a diagram of how the mTask client and server communicate using MQTT, extending Figure 2.

If a subscriber cannot be reached when a new message is sent, the broker registers this. This is where the added value of the protocol lies. The broker sends the message as soon as the subscriber comes back online, thereby ensuring all subscribers receive all messages, regardless of imperfect connections or power failures.

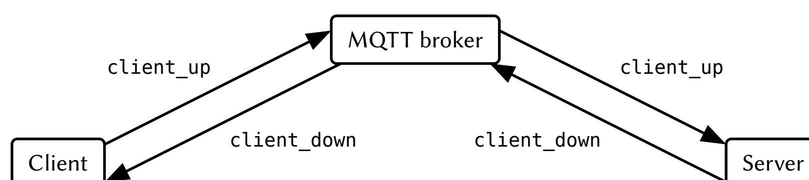


Figure 4: The mTask system with MQTT (webpages and user ommitted).

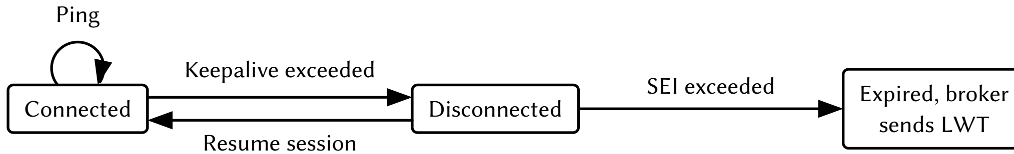


Figure 5: The progression of an MQTT session.

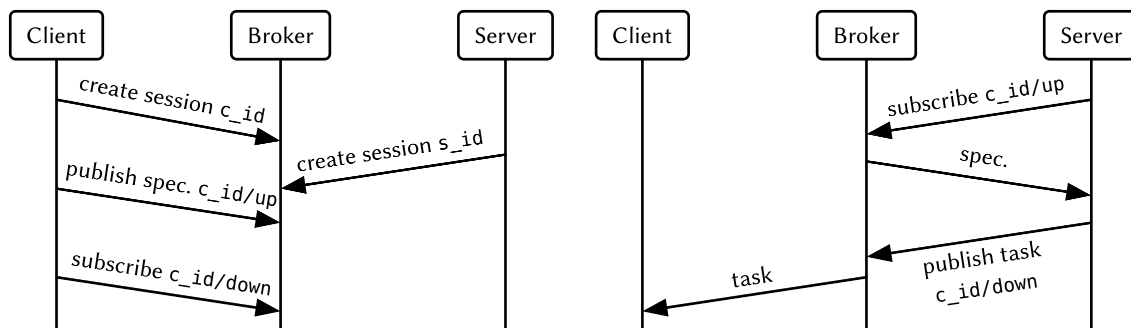
Furthermore, the client specifies a Last Will and Testament (LWT) and a Will Delay Interval (WDI) when setting up a session with the broker. The client specifies on which channel the LWT should be sent. When the client runs out of power or loses connection for another reason, the broker waits for the WDI to pass, after which it sends the LWT to all subscribers of the channel and discards the session.

The MQTT protocol moreover allows for specifying a Session Expiry Interval (SEI). If the SEI is larger than the WDI, the client may resume its session, meaning e.g. that the broker remembers its subscriptions, even though the LWT has already been sent. This however is not used in the current mTask implementation, where the SEI is always set to be the same as the WDI. Figure 5 shows a diagram of how an MQTT session progresses.

The LWT specified by the client is a disconnect message. This means that upon losing connection, the client has as long as the specified WDI to come back online before the server discards the session and sends the LWT, in which case a new session will have to be established. When this happens, the client discards any state recovered from its NVM and starts up as a *blank* mTask client.

In the case of mTask, there is a channel for messages from the client to the server, named `id/up`, and for messages from the server to the client, named `id/down`. Here, `id` is a unique identifier associated with the client. Figure 6 shows how the mTask server and client set up communication over MQTT. Here, `c_id` is the identifier used by the client and `s_id` is the identifier used by the server. The server needs to be provided the client's identifier in order to use the appropriate channels. The server's identifier is only needed for the broker to identify the server.

Figure 6(a) shows how the client and server establish connection with the broker. It does not matter which connects first. The client publishes a message — *spec.* in the figure — on the `up` channel with info about the device, e.g. number of pins and mTask version. As soon as the server subscribes to the `up` channel, the broker forwards this message to the server. Then, the server publishes the task on



(a) Establishing connection with broker. (b) Sending task to client.

Figure 6: Messages sent for setting up mTask client and server communication with MQTT.

```
1  prop = MqttClient_PropsAdd(&connect.props);
2  prop->type = MQTT_PROP_SESSION_EXPIRY_INTERVAL;
3  prop->data_int = WILL_DELAY_INTERVAL;
4  rc = MqttClient_Connect(&mqtt_client, &connect);
5  if (connect.ack.flags & MQTT_CONNECT_ACK_FLAG_SESSION_PRESENT) {
6      // Resuming session successful
7      ...
8  } else {
9      // Broker did not have a stored session
10     ...
11 }
```

Listing 10: Code excerpt for setting up MQTT connection.

the `down` channel, which the broker then forwards to the client. Now, the client may start executing the task.

Listing 10 shows a relevant fragment from the implementation. The RTS uses the WolfMQTT library which provides helper functions for building, sending, and reading messages. On lines 1–3 the WDI is specified. line 4 tries to establish a connection. The result is written to the connection struct `connect`. We inspect this result on line 5 to determine whether we were able to resume an existing session, or whether we need to reset, e.g. because we were offline for too long and the broker has already sent our LWT to the server.

5.1.2. Checkpointing

We add automatic checkpoints to mTask. Due to the nature of mTask, we can safely create a checkpoint after each rewrite step (see Section 4.3.3). Checkpoints are written contiguously without interruption. To restore a checkpoint, we then read the same objects in the same order. Listing 11 shows the implementation for creating and restoring checkpoints. The print and parse functions are generated by the `gentype` library. Some details of the implementation have been omitted for clarity.

Seen side by side, it is clear that restoring a checkpoint mirrors creating a checkpoint. A notable difference is that when restoring a checkpoint, we may need to *patch* the pointers to correct for ASLR. We do this by calculating the offset in address space (lines 23–24) and adding this offset to each pointer (line 25, ommitted). Additionally, when restoring a checkpoint, we also initialize the peripherals corresponding to the task (line 26, ommitted). Moreover, restoring a checkpoint may fail if memory is corrupted (lines 2–3) or if one of the peripherals throws an error. In this case, the restore function returns false and the client resets.

The data included in a checkpoint is as follows.

- The address of our data, needed for ASLR correction (lines 3–4),
- the task pointer (line 6),
- the heap pointer (line 7),
- the events pointer (line 8),
- the task memory (lines 9–10),
- the task tree memory (lines 11–12), and
- the scheduling queue (lines 13–22).

```c
void nvm_save(void) {
  nvm_open_write();

  print_VoidPointer(mtmem_real)

  print_UInt16(mem_task);
  print_UInt16(mem_heap);
  print_VoidPointer(events);
  for (i=0; i<mem_task; i++)
    nvm_put(mtmem[i]);
  for (i=mem_heap; i<memsize; i++)
    nvm_put(mtmem[i]);
  print_UInt8(queue_start);
  print_UInt8(queue_end);
  for (
    uint8_t i = queue_start;
    i != queue_end;
    i = (i+1)%TASK_QUEUE_SIZE
  ) {
    print_VoidPointer(
      task_queue[i]);
    }




  nvm_close_write();

}
```

(a) Function for creating a checkpoint.

```c
bool nvm_restore(void) {
  if (!nvm_open_read())
    return false;
  void *mtmem_real_old =
    parse_VoidPointer();
  mem_task = parse_UInt16();
  mem_heap = parse_UInt16();
  events = parse_VoidPointer();
  for (i=0; i<mem_task; i++)
    mtmem[i] = nvm_get();
  for (i=mem_heap; i<memsize; i++)
    mtmem[i] = nvm_get();
  queue_start = parse_UInt8();
  queue_end = parse_UInt8();
  for (
    uint8_t i = queue_start;
    i != queue_end;
    i = (i+1)%TASK_QUEUE_SIZE
  ) {
    task_queue[i] =
      parse_VoidPointer();
  }
  pd = ((intptr_t)mtmem_real -
        (intptr_t)mtmem_real_old);
  // Patch pointers
  // Initialize peripherals
  nvm_close_read();
  return true;
}
```

(b) Function for restoring a checkpoint.

Listing 11: Checkpoint saving and restoring functions side by side.

It is possible for power to fail while writing a checkpoint. In this case, we are left with a partial, invalid checkpoint. To avoid ending up with no usable checkpoints, we apply double buffering. That is, only after finishing writing the new checkpoint, we invalidate the old checkpoint. We reserve one byte before writing data to represent the validity of the checkpoint. This way, we always have a valid checkpoint, at the cost of needing more NVM.

5.1.3. Resuming a session

When the client is powered on, it does not know for how long it has been off. Therefore, we assume that we should continue from the latest checkpoint. When the client runs a program to completion, it invalidates both checkpoints, so that it starts as a *blank* client on the next startup.

If restoring a checkpoint from NVM fails, we check whether the MQTT broker still has a session for this client. If it does, we end the session, so that the server is informed that the client cannot continue the task.

If restoring a checkpoint from NVM passes, we try to resume the MQTT session. If there is no session, we reset. The broker will have already sent the LWT, meaning we do not need to inform the server anymore that the client is not resuming the session.

To reset, we invalidate the checkpoints and clear task memory, so that we end up in the *blank* state and can accept a new task from a server.

Figure 7 shows how the client decides whether to resume a previous session or to reset.

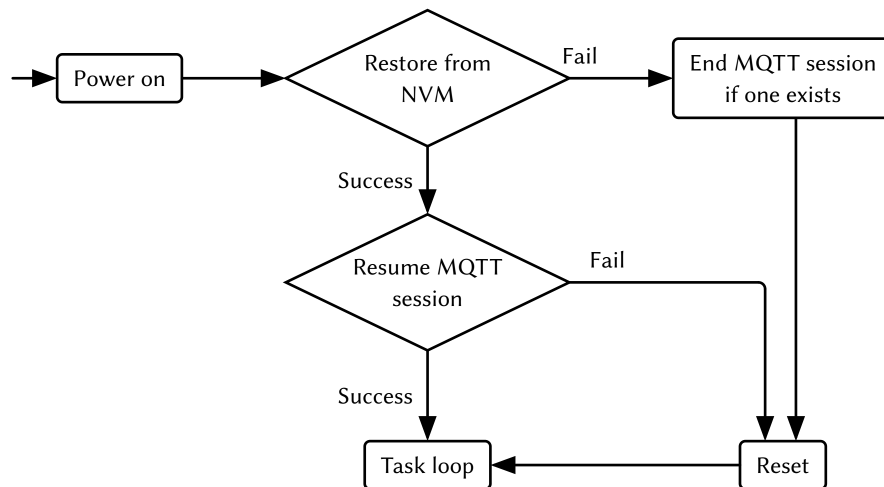


Figure 7: Decision diagram for restoring a checkpoint.

5.2. Evaluation

We use a build of the mTask RTS for a PC environment and an example program (Listing 12) to demonstrate that the extension works. In this program, we use a SDS (line 4) to store a number. The server displays this number on a webpage. Every half second, the edge device increments the counter. We run the mTask client and server on a laptop running Linux. To simulate power failure for the client, we use system signal 9 (kill) to suddenly stop execution. As MQTT broker, we use mosquitto [23], running on the same laptop.

When running mTask with this setup, the mTask client does indeed restore its state from disk, after which it continues counting from where it was before being shutdown. The MQTT connection stays open for the specified amount of time and continues the session as designed.

6. Related Work

In this section, we describe this work's place in literature and describe the advancements it presents in the fields of TOP and intermittent computing.

6.1. Task-Oriented Programming

With mTask, TOP has become suitable for the IoT [9]. The present work extends the mTask ecosystem such that programs can be made power-failure resilient with minimal to no additional work from the programmer. Though other TOP implementations exist, e.g. TopHat [24], this is the first TOP language expanded with dedicated intermittent computing functionality.

```
1  Start :: !*World -> *World
2  Start w = doTasks (main <<@ ApplyLayout frameCompact) w
3
4  counterShare = sharedStore "counter" 0
5
6  main :: Task ()
7  main = enterDevice
8     >>? \spec->withDevice spec deviceTask
9     >>* [ OnAction (Action "Stop") (always (shutDown 0))
10        , OnAction (Action "Reset") (always main)
11        ]
12 where
13   deviceTask :: MTDevice -> Task ()
14   deviceTask dev = liftmTask count dev
15                 -|| viewSharedInformation [] counterShare
16
17 count :: Main (MTask v ()) | mtask, lowerSds v
18 count = lowerSds \counter=counterShare
19  In fun \countfun=(\n->
20           setSds counter (n +. lit 1)
21      >>|. delay (ms 500)
21      >>|. countfun (n +. lit 1))
23  In {main=countfun (lit 0)}
```

Listing 12: Counter program in mTask.

6.2. Intermittent Computing

With mTask, it is possible to obtain an intermittent-aware program without writing any additional code. All required checkpointing infrastructure is added by the compiler. This is also possible using the BFree framework [21], though it requires re-implementing a large part of the CircuitPython language, making it more costly in terms of work to update and maintain. Since mTask is a TOP language, it is suitable for intermittent computing by nature and only needs to be extended, not altered, to enable intermittent computing.

Most intermittent computing frameworks do not support multithreading [19], [20], [21], [22]. Frameworks that do either require a significant amount of additional code [25] or have implemented multithreading with polling, which is not efficient in terms of energy and computation, i.e. ImmortalThreads [26]. Additionally, ImmortalThreads is based on C, where creating a safe multithreaded program is rather convoluted compared to how this is done in TOP. In mTask, multithreading is achieved through interleaving tasks. The scheduler only schedules tasks which can progress, meaning mTask does not waste CPU-cycles on busy waiting.

The presented framework allows edge devices to be reprogrammed on the fly thanks to the client-server setup. Other intermittent computing frameworks require either including the program with the device on e.g. an SD card [21], or completely fixing the behavior of the device at compile time [19], [22]. There does exist a kernel targeted at intermittent computing which allows for adjusting priorities of work based on current power availability, but the way priorities are adjusted is still fixed at compile time [20].

7. Future Work

7.1. Atomic Tasks

Much of this work is concerned with alleviating the programmer of the burden of intermittent computing. There are some scenarios however where we do want the programmer to actively think about how to deal with unreliable power supply, for instance when it comes to temporal consistency.

As an example, consider a motor that we want to activate when a button is pressed. If power fails right after the button is pressed, and only comes back an hour later, we probably do not want to still activate the motor.

One way to avoid this in mTask is to allow the programmer to specify *atomic tasks*. If an atomic task is interrupted by a power failure, it starts over from the beginning of the task, instead of from the last checkpoint. Consequently, atomic tasks need not be checkpointed, although the task info and bytecode should still be stored in NVM. Adding atomic tasks to mTask allows more applications to operate without battery, thereby increasing the utility of the framework.

One way to implement this behavior is to add a task combinator that takes a task and executes that task atomically. Listing 13(a) shows the type for this combinator. This can be implemented by altering the rewriter so that rewriting an atomic task is a single step. This way, the atomic task completes as quickly as possible, as no other tasks are interleaved. On the other hand, this limits functionality, as atomic tasks cannot be executed in parallel with other tasks with this method.

Alternatively, the atomic task combinator can save a copy of its task, similar to the `rpeat` combinator. After a power failure occurs, the atomic task combinator starts from the beginning of its task, using the saved copy. When restoring a checkpoint, the client should check for instances of the atomic task combinator and remove any task trees in progress.

This approach allows for multithreaded, *normal* execution of tasks when power is available, but requires some additional thought with regards to state. Any updates to SDSs are not reverted. We could save the state of relevant SDSs with the copy of the task, but this also reverts updates made by other tasks. A safe solution is to disallow atomic tasks to update SDSs, though this limits functionality. If an atomic task is the only task to update an SDS, reverting to a saved state is safe, but this is non-trivial to implement.

7.2. Voltage Test

Surbatovich et al. [16] show that atomic sections suffice to specify timeliness for internal state, but external state poses a more fundamental challenge for intermittent computing. If power fails while altering the physical world, e.g. while powering some pump, reverting to a previous state may be undesirable or even impossible.

Ideally, we only start such a task when enough power is available to complete it. With static or just-in-time analysis of power consumption we can estimate whether we are able to complete a task with the power currently available. Such a check can be added automatically for atomic tasks. In addition, basic tasks similar to those in Listing 13(b) can be added to allow the programmer to have more control over program flow. Many devices allow obtaining the currently available voltage by

```
atomic :: MTask v t -> MTask v t
```

(a) Atomic task operator.

```
1 remainingPower :: MTask v Real
2 remainingPowerMs :: MTask v Long
```

(b) Minimum voltage condition.

Listing 13: Additional mTask language constructs for intermittent computing.

```
1 Obs.range(1, :infinity)
2 |> Obs.each(fn i -> GPIO.D4.write (even i))
3 |> Obs.delay(500)
```

Listing 14: Potato-based pseudo-code for blinking an LED.

reading a pin (line 1). Combined with analysis of power consumption, this can be used to express the remaining power as an estimate for when, i.e. in how many milliseconds, power will run out (line 2).

7.3. Functional Reactive Programming

TOP is often compared to Functional Reactive Programming (FRP) [27]. FRP focuses heavily on events, making it a good candidate for programming for the IoT. Listing 14 shows what code for blinking an LED may look like in FRP. The code is roughly based on the Potato framework. On line 1, an infinite stream of integers is generated, wich is used on line 2 to toggle an LED. The `|>` operator applies the function on the right-hand side to the argument on the left-hand side.

Potato is an FRP framework for the IoT [28]. It is likely that the insights from this work can also be implemented in Potato and other FRP frameworks. Potato builds on the Elixir programming language and it would be interesting to see if it is possible to add intermittent computing to this system.

7.4. Other Intermittent Computing Concepts

Checkpoint size Although checkpoints in the mTask system are already quite small compared to other frameworks, the amount of data that is written for each checkpoint could still be reduced. Each byte takes power and time to checkpoint, so reducing checkpoint size may significantly improve performance. In particular, data that has not changed since the last checkpoint can be skipped when writing a new checkpoint.

An obvious candidate for this in the mTask system is the bytecode, which, in the current setup, is included in every checkpoint, even though it never changes. Additionally, if instead of check-pointing the memory as a whole, we checkpoint each task separately, we can skip tasks that have not changed since the last checkpoint, e.g. tasks that are waiting for a certain condition to hold.

How to determine which data to checkpoint is not obvious. Since mTask uses double-buffering, we want to checkpoint all data that has changed since two checkpoints ago, not just the data that has changed since the last checkpoint. When implementing a more complex checkpoint structure, using a file system is advisable. This way, checkpoints for different tasks can be organized as different files, thereby also splitting up the double-buffering into smaller chunks.

For further checkpoint size reduction, inspiration may be drawn from state-of-the-art techniques [29].

Sleep strategy The EarlyBird technique shows that waking up the processor before the capacitor has fully recharged leads to better results [30]. The mTask system can be expanded with this technique to allow it to spend less time waiting for energy to harvest.

8. Conclusion

We presented an extension to mTask that makes programs intermittent aware without any additional effort from the programmer. This is the first TOP language with built-in intermittent computing.

The mTask approach offers several advantages over other intermittent computing frameworks. Thanks to the nature of TOP, the intermittent computing functionality can be implemented as an extension to mTask and does not require significantly changing mTask, meaning it does not require a large dedicated codebase like e.g. BFree [21]. Furthermore, the client-server setup of mTask allows reprogramming edge devices on the fly and without exhausting the limited rewrite cycles of flash memory. Moreover, thanks to the task-tree based evaluation, checkpoints can be made frequently and without needing the programmer to specify suitable opportunities. Lastly, multithreading in combination with intermittent computing is supported by mTask and easy to use.

The current implementation does not support programmer-defined atomic tasks, meaning some requirements cannot easily be implemented using mTask. Furthermore, the mTask system can benefit from additional functionality, most importantly reading remaining voltage and reduced checkpoint overhead.

Abbreviations

***ASLR* – Address Space Layout Randomization**: Technique where the location in memory of objects is randomized to protect against attacks.

***DSL* – Domain-Specific Language**: Programming language tailored to a specific application, e.g. SQL which is designed to express database operations, in contrast to a general-purpose programming language, e.g. Python.

***EH* – Energy Harvesting**: Collection of techniques to obtain electrical power from the environment, e.g. using a solar panel to harvest solar energy or harvesting the kinetic energy from a person walking.

***FRP* – Functional Reactive Programming**

***GPIO* – General Purpose Input-Output**: A GPIO pin can be controlled by software to be used for different purposes.

***IoT* – Internet of Things**: Term used to describe devices with sensors or actuators that communicate over the internet.

***LWT* – Last Will and Testament**: Message specified by an MQTT client that the MQTT broker sends when the client is lost, i.e. when the client does not communicate for a specified amount of time.

MQTT: MQ Telemetry Transport. Communication protocol that is often used for the IoT.

***NVM* – non-volatile memory**: Computer memory that does not lose its data upon power loss.

***RTS* – Runtime System**

***SDS* – Shared Data Source**

***SEI* – Session Expiry Interval**: Time the MQTT broker keeps a session alive after losing connection.

***TOP* – Task-Oriented Programming**

***WAR* – Write-After-Read**

***WDI* – Will Delay Interval**: Time the MQTT broker waits after losing connection before sending the LWT.

References

[1] "An Update on Arm's AI Journey Toward a Trillion Connected Devices." Accessed: Nov. 12, 2024. [Online]. Available: https://newsroom.arm.com/news/an-update-on-arms-ai-journey-toward-a-trillion-connected-devices

[2] P. Achten, P. Koopman, and R. Plasmeijer, "An introduction to task-oriented programming," *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers 5*, pp. 187–245, 2015.

[3] "VIIA." Accessed: Nov. 23, 2024. [Online]. Available: https://www.top-software.nl/VIIA.html

[4] R. Plasmeijer, P. Achten, and P. Koopman, "iTasks: executable specifications of interactive work flow systems for the web," *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 141–152, 2007.

[5] C. Visser, "Transpose Calculator." Accessed: Dec. 05, 2024. [Online]. Available: https://gitlab.com/cas_visser/transposecalculator

[6] Wikipedia contributors, "Transposing instrument — Wikipedia, The Free Encyclopedia." Accessed: Dec. 14, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transposing_instrument&oldid=1183299795

[7] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010, doi: https://doi.org/10.1016/j.comnet.2010.05.010.

[8] M. Lubbers, P. Koopman, A. Ramsingh, J. Singer, and P. Trinder, "Could Tierless Languages Reduce IoT Development Grief?," *ACM Trans. Internet Things*, vol. 4, no. 1, Feb. 2023, doi: 10.1145/3572901.

[9] M. Lubbers, "Orchestrating the Internet of Things with Task-Oriented Programming," Radboud University Press, 2023.

[10] S. Crooijmans, M. Lubbers, and P. Koopman, "Reducing the Power Consumption of IoT with Task-Oriented Programming," in *International Symposium on Trends in Functional Programming*, 2022, pp. 80–99.

[11] C. Church and L. Wuennenberg, "Sustainability and second life: the case for cobalt and lithium recycling, 2019," *International Institute for Sustainable Development.*, [Online]. Available: https://www.iisd.org/system/files/publications/sustainability-second-life-cobalt-lithium-recycling.pdf

[12] "Casambi Ready 4-button Smart Switch." Accessed: Nov. 11, 2024. [Online]. Available: https://casambi.com/ecosystem/casambi-ready-4-button-smart-switch/

[13] "Perceptive - MATRIX: Self-Powered Solutions." Accessed: Nov. 11, 2024. [Online]. Available: https://www.matrixindustries.com/perceptive

[14] D. Ma, G. Lan, M. Hassan, W. Hu, and S. K. Das, "Sensing, computing, and communications for energy harvesting IoTs: A survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1222–1250, 2019.

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

[16] M. Surbatovich, L. Jia, and B. Lucia, "Automatically enforcing fresh and consistent inputs in intermittent systems," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 851–866.

[17] "ESP8266 Low Power Solutions." Espressif Inc., 2016. Accessed: Dec. 10, 2024. [Online]. Available: https://www.espressif.com/sites/default/files/9b-esp8266-low_power_solutions_en_0.pdf

[18] C. Kraemer, W. Gelder, and J. Hester, “User-directed Assembly Code Transformations Enabling Efficient Batteryless Arduino Applications,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 8, no. 2, pp. 1–32, 2024.

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

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

[21] V. Kortbeek, A. Bakar, S. Cruz, K. S. Yildirim, P. Pawełczak, and J. Hester, “Bfree: Enabling battery-free sensor prototyping with python,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 4, no. 4, 2020.

[22] C. Kraemer, A. Guo, S. Ahmed, and J. Hester, “Battery-free MakeCode: Accessible Programming for Intermittent Computing,” *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 6, no. 1, Mar. 2022, doi: 10.1145/3517236.

[23] “Eclipse Mosquitto.” Accessed: Jan. 28, 2025. [Online]. Available: https://mosquitto.org/

[24] T. Steenvoorden and N. Naus, “Dynamic TopHat: Start and Stop Tasks at Runtime,” in *The 35th Symposium on Implementation and Application of Functional Languages*, 2023, pp. 1–13.

[25] Y. Wu, B. Min, M. Ismail, W. Xiong, C. Jung, and D. Lee, “IntOS: Persistent embedded operating system and language support for multi-threaded intermittent computing,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 425–443.

[26] E. Yıldız, L. Chen, and K. S. Yıldırım, “Immortal threads: Multithreaded event-driven intermittent computing on {Ultra-Low-Power} microcontrollers”, in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 339–355.

[27] J. Stutterheim, P. Achten, and R. Plasmeijer, “Maintaining separation of concerns through task-oriented software development,” in *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*, 2018, pp. 19–38.

[28] C. de Troyer, J. Nicolay, and W. de Meuter, “Building IoT Systems Using Distributed First-Class Reactive Programming,” in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pp. 185–192. doi: 10.1109/CloudCom2018.2018.00045.

[29] Y. Kim, Y. Lim, and C. Lim, “LACT: Liveness-Aware Checkpointing to reduce checkpoint overheads in intermittent systems,” *Journal of Systems Architecture*, vol. 153, p. 103213, 2024, doi: https://doi.org/10.1016/j.sysarc.2024.103213.

[30] H. Reymond, J.-L. Béchennec, M. Briday, and S. Faucou, “EarlyBird: Energy belongs to those who wake up early,” in *2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2024, pp. 1–10. doi: 10.1109/RTCSA62462.2024.00011.