

POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Computer Science and Engineering



From Energy to Throughput in Intermittent Computing

Supervisor:

PROF. LUCA MOTTOLA

Master Graduation Thesis by:

FRANCESCO BERTANI

Student Id n. 875585

Academic Year 2018-2019

To my parents.

To those who are trying to figure it out.

No matter the result.

ACKNOWLEDGMENTS

Thanks to everybody who helped me to understand how to do scientific research, to ask me the right questions, and to thoughtfully and responsibly looking for answers in data.

Thanks to my advisor, Professor Luca Mottola, who made me aware of such a challenging research field, guided me in this process, and taught me the elements of style¹.

Thanks to Professor Antonio Miele who, even if on a different topic, introduced me for the first time to scientific writing and publication.

Thanks to my lab colleagues and friends Andrea, Francesco, Fulvio and Pietro. You have been an invaluable resource in times of trouble and wonderful fellows in times of happiness.

Thanks to Federica, you always know the right words to give me balance.

Thanks to my family, your unconditional support is my biggest strength.

¹ The Elements of Style by William Strunk and E. B. White

ABSTRACT

The ability to power small embedded devices with just the energy harvested from the environment, extends their scenarios of adoption. Though, the instability of these power sources, often times results in drops of power supply. To mitigate this issue, we can use capacitors as energy buffers.

Once the capacitor voltage reaches a given activation threshold, the microcontroller powers on and the computation starts. If the energy intake is lower than the outtake, the capacitor voltage drops, potentially to a point where it is below the minimum device's operating voltage. If that happens a power failure occurs: the computation stops and the device powers off, charging until the activation threshold is reached again. This causes losing processor's state, hence the progress in computation is lost and it must be restarted from scratch at the next activation. We call this behavior *intermittent computing*.

Unlike mainstream platforms where software can benefit from OS support, on our constrained devices we execute code on bare hardware and power failures are not managed. It is evident that we need a mechanism to save the state on non-volatile supports, when we are close to a power failure, so that the computation can resume from where it was when the device switched off, ensuring code completion. Different approaches are described in the literature when it comes to saving state. Each of them proposes a different take on how to structure, instrument and execute code. Though, all of them consider the maximization of energy efficiency as the ultimate goal that guides their decisions. This allows to operate in situations of scarce energy intake. We proved with experiments that being too conservative produces rigid solutions that do not adapt well to the opposite situation of sudden bursts of high energy provisioning.

In this document we propose an important shift in perspective. We suggest a different paradigm: to focus back on throughput requirements instead of energy efficiency. We propose to consider energy efficiency not as a goal, but as a mean to satisfy throughput requirements, that we argue should be the ultimate goal. In light of this conceptual leap, we propose a framework to support intermittent computing, composed by a new programming abstraction and a dynamic scheduler. Our scheduler, for the first time, manages at runtime both workload and activation voltage threshold, to pursue our new goal: the satisfaction of throughput.

SOMMARIO

La possibilità di alimentare dispositivi di piccole dimensioni solamente con l'energia raccolta dall'ambiente, ne estende il campo d'azione. L'instabilità di queste sorgenti rende frequenti i cali di alimentazione e può essere mitigata utilizzando condensatori come buffer energetici.

Il microcontrollore si attiva quando il voltaggio del condensatore raggiunge una determinata soglia. Se durante la computazione il bilancio energetico è negativo, il voltaggio del condensatore cala, potenzialmente raggiungendo un valore incompatibile con l'operatività del dispositivo alimentato. Se ciò accade si verifica una *power failure*: la computazione si interrompe e il dispositivo si spegne, caricandosi fino al nuovo raggiungimento della soglia di attivazione. Lo spegnimento improvviso causa la perdita dello stato del processore che al riavvio dovrà eseguire da capo tutte le istruzioni. Chiamiamo questo comportamento *computazione intermittente*.

In questi dispositivi il codice viene eseguito accedendo direttamente all'hardware e le power failure non sono gestite, a differenza delle piattaforme di calcolo comuni in cui il software beneficia del supporto del sistema operativo. Emerge la necessità di implementare meccanismi di salvataggio dello stato su supporti non volatili, in prossimità di una power failure, per poter riprendere la computazione interrotta, garantendo il completamento del lavoro. In letteratura sono descritti svariati approcci al problema del salvataggio dello stato che offrono molteplici strade per strutturare ed eseguire il codice destinato a queste piattaforme. Esiste però un comune denominatore a tutte queste proposte: il fatto di considerare la massimizzazione dell'efficienza energetica l'obiettivo primario che guida le scelte prese. Questo approccio permette di operare in condizioni di scarsità energetica, ma abbiamo sperimentato che produce soluzioni rigide che non si adattano a scenari opposti di improvvisi picchi di alimentazione.

In questo documento proponiamo un importante cambio di prospettiva. Sugeriamo un diverso paradigma in cui si riporta al centro dell'attenzione il soddisfacimento dei requisiti non funzionali, e in particolare del throughput minimo di esecuzione. Proponiamo di rendere l'efficienza energetica, non più un obiettivo, ma uno strumento per raggiungere la performance desiderata. Alla luce di questo cambiamento, proponiamo un framework per dispositivi a computazione intermittente, composto da una nuova astrazione di programmazione e, per la prima volta, da uno scheduler dinamico che gestisce a runtime il carico di computazione e le soglie operative di voltaggio, per perseguire quello che consideriamo il nuovo obiettivo principale: il soddisfacimento dei throughput richiesti.

CONTENTS

1	INTRODUCTION	1
1.1	Protect Against Power Failures	2
1.2	Time Related Requirements	4
1.3	Energy Awareness	4
1.4	From Energy to Throughput	5
1.5	Our Contribution	7
1.6	Structure	9
1.6.1	Introduction to Intermittent Computing	9
1.6.2	State of the Art Analysis	10
1.6.3	Contribution	10
1.6.4	Evaluation	11
2	INTRODUCTION TO TRANSIENTLY POWERED COMPUTING	13
2.1	Intermittent Computation	14
2.2	Forward Execution Problem	15
2.2.1	Checkpoint Inconsistencies	17
2.3	Time Without a Clock	20
2.4	From Problems to Solutions	21
3	HIDING POWER FAILURES	23
3.1	Checkpoint Based Solutions	23
3.1.1	Static Checkpoint Solutions	24
3.1.2	Dynamic Checkpoint Solutions	29
3.2	Task-Based Solutions	32
3.2.1	Constraints and Goals for Task-Based Systems	33
3.2.2	Task-Based Solutions Overview	35
3.3	Why Two Solutions to the Same Problem?	44
4	SETTING THE THRESHOLD	47
4.1	The Conundrum of Threshold Selection	49
4.2	EPIC results	51
4.3	Overview of Threshold Management Solutions	53
4.4	Shifting Perspective in Threshold Management	59
5	ENABLING MULTITENANCY	63
5.1	Fundamental Concepts	64
5.2	Tasks and Applications	64
5.2.1	Tasks	65
5.2.2	Applications	67
5.3	Data Dependencies	67
5.3.1	Greenhouse Example	69
5.3.2	Data Dependency Semantics	71
5.3.3	Task Operating on Machine	76
5.3.4	Task Operating on Shared Phenomena Controlled by the World	77

5.3.5	Task Operating on Shared Phenomena Controlled by the Machine	78
5.4	Memory Model	81
5.4.1	Write Output Data	82
5.4.2	Read Input Data	84
6	SCHEDULING TASKS	87
6.1	Chapter Overview	89
6.2	Minimum Throughput and Applications Priority	90
6.3	Dynamic workload management	91
6.4	Scheduler Initialization	94
6.5	Task Selection and Deadlines Management	97
6.6	React to Throughput's Drifts	101
6.6.1	Managing Over-Performing Applications	103
6.6.2	Managing Under-Performing Applications	104
6.7	The Γ parameter	106
6.8	Scheduler Fairness	107
6.9	Complete Overview	109
7	IMPLEMENTATION	113
7.1	Define Tasks	114
7.2	Define Applications	117
7.3	Creating a Task	119
7.4	Scheduler Implementation	119
8	EVALUATION	123
8.1	Evaluation Environment	123
8.2	Extending SIREN Simulator	124
8.2.1	Capacitor Simulator	126
8.2.2	Extended SIREN Commands	128
8.3	Evaluation Scenario	130
8.4	Evaluation Baseline	132
8.5	Outputs and Metrics	132
8.6	Evaluation Results	134
8.6.1	Stable Energy Source	134
8.6.2	Underpowered Execution	136
8.6.3	Fairly Stable Source With Energy Failures	139
8.6.4	Gamma and Fairness Interaction	144
8.6.5	Scheduler Stability	146
9	CONCLUSION AND FUTURE WORKS	149
9.0.1	Future Works	150
	BIBLIOGRAPHY	153

LIST OF FIGURES

Figure 1.1	A typical intermittent execution pattern.	2
Figure 1.2	Code execution with or without a save state mechanism in presence of power failures.	3
Figure 1.3	Example of an application that manages a HVAC system	8
Figure 2.1	Simulation of an intermittent execution pattern.	15
Figure 2.2	Example of an inconsistency caused by intermittent execution.	18
Figure 3.1	HarvOS Control Flow Graph (CFG) slicing. . .	27
Figure 3.2	Different programming abstractions induce different code structures.	44
Figure 4.1	Flicker power management circuit.	47
Figure 4.2	Capacitor voltage over time.	48
Figure 4.3	Simulation of Hibernus to understand how different selections of the activation threshold affect system's performance.	50
Figure 4.4	The selection of the activation threshold affects the performance of task based systems.	52
Figure 4.5	Effects of threshold change.	53
Figure 4.6	Voltage supply has an impact on both power consumption and clock speed. Taken from [1]	54
Figure 4.7	Dependencies among quantities involved in the computation of execution time of a piece of code in Transiently Powered Computation (TPC).	55
Figure 4.8	Flowcharts of the routines to calibrate hibernation and activation thresholds in Hibernus++ [2].	56
Figure 4.9	A conceptual view of Flicker federated energy storate, taken from [14].	58
Figure 5.1	Example of a partitioned DAG that shows tasks, applications and dependencies.	65
Figure 5.2	Task access to Non Volatile Memory (NVM) is mediated by the framework.	66
Figure 5.3	A TPC device is deployed in a greenhouse to control the irrigation system.	69
Figure 5.4	Dependency graph for the greenhouse example presented in Section 5.3.1	71
Figure 5.5	Example of a application with tasks operating on shared phenomena controlled by the world.	77
Figure 5.6	Possible scheduling trace the scenario described in Section 5.3.4	78

Figure 5.7	Dependency graph for scenario described in Section 5.3.5.	79
Figure 5.8	Flowchart representing the steps followed by the runtime environment to persist task's output on NVM.	84
Figure 5.9	Example of the update of the NVM data record.	85
Figure 6.1	Two applications with inter dependency among them.	93
Figure 6.2	Example of deadline update after the execution of a task.	98
Figure 6.3	Deadline management example	100
Figure 6.4	Example of an application with data dependencies that cause the shrinking of the enabled tasks set after a task execution.	101
Figure 6.5	Slack time computation	103
Figure 6.6	Applications DAG for scheduler final overview	109
Figure 7.1	Overview of the structure of the proposed solution	114
Figure 7.2	Application layout for YAML example in Listing 7.1	116
Figure 7.3	Applications layout for YAML example in Listing 7.2	118
Figure 7.4	Example of macro expansion in task's code	119
Figure 8.1	Example of a Ekho solar IV surface. Taken from Furlong et al. [11]	125
Figure 8.2	Overview of SIREN	126
Figure 8.3	SIREN UML class diagram	127
Figure 8.4	Sequence diagram for SIREN main execution loop	128
Figure 8.5	Dependency graph for the evaluation scenario	131
Figure 8.6	Comparison between static and dynamic scheduler with a stable solar energy source.	135
Figure 8.7	Comparison between static and dynamic scheduler with a highly unstable RF energy source.	137
Figure 8.8	Different selections of Γ parameter can lead to different performance with the same energy source.	138
Figure 8.9	Comparison between static and dynamic scheduler with a fairly stable source with sudden drops in provided energy.	140
Figure 8.10	Detail of the initial 5000 milliseconds of the simulation results shown in Figure 8.9b	141
Figure 8.11	Static scheduling with a single application leads to wasted energy	143
Figure 8.12	Interaction between Γ selection, throughputs and fairness.	145

Figure 8.13	Dependency graph of the modified scenario, implemented to measure scheduler stability	146
Figure 8.14	Comparison between different secondary applications with the same main application and energy trace.	147
Figure 8.15	Detailed comparison between the throughput of Application 1 with different secondary applications. The original scenario is described in Section 8.3, while the modified one is in Section 8.6.5	148

LIST OF TABLES

Table 2.1	Measured Ferroelectric RAM (FRAM) and Flash characteristics. Taken from [4]	17
Table 3.1	Overview on different checkpoint solutions to guarantee forward execution.	24
Table 3.2	Overview on different task-based solutions to guarantee forward execution.	35
Table 5.1	Data dependency semantics and corresponding graphical conventions.	72
Table 5.2	Possible trace for the scenario presented in Section 5.3.5	80
Table 8.1	Mean throughput, variance, correctness and fairness obtained when powered by a stable source with no power failures. The table shows the metrics for the main application Application 1	136
Table 8.2	Mean throughput and fairness with design-time, runtime with Γ that maximizes App 1 throughput, and with lowest Γ parameter. . .	139
Table 8.3	Average throughput, correctness and fairness with design-time and dynamic runtime scheduling, against an energy source with sudden drops.	143
Table 8.4	Comparison between static scheduling with a single application, and runtime scheduler with multi-tenancy.	143

ACRONYMS

CFG	Control Flow Graph
CRFID	Computational RFID
DAG	Direct Acyclic Graph
DRAM	Dynamic Random Access Memory
FRAM	Ferroelectric RAM
GPR	General Purpose Registers
ISR	Interrupt Service Routine
LPM	Low Power Mode
MCU	Micro Controller Unit
MISD	Minimum Inter Sample Delay
NVM	Non Volatile Memory
PC	Program Counter
SRAM	Static Random Access Memory
SR	Status Register
SP	Stack Pointer
TPC	Transiently Powered Computation
WAR	Write After Read
WSN	Wireless Sensors Network

INTRODUCTION

Let us imagine a system of tiny invisible networked computers to support humanity. Batteries are the single greatest threat to this vision [15]. They are expensive, bulky and hazardous, and replacing them when they wear out poses a serious environmental issue. By leaving batteries behind, relying exclusively on energy harvested from the environment, we can enable this vision of a multitude of devices deployed and forgotten, maintenance free for decades [16, 22, 32].

We can use different energy harvesters to collect energy from a wide variety of sources. Some of them, such as those extracting energy from wind, can provide large amounts of power in short bursts; some others, like small photovoltaic cells operating from indoor light, steadily provide small amounts of power; energy harvested from RFID readers is subject to voltage fluctuations that are highly dependent on the operating environment and device's physical orientation [4, 29]. If we want to switch to batteryless computation, we need to face the inconsistency of energy harvesting that is typically characterized by extreme variations in supply voltage [1]. To mitigate this erratic supply pattern we can buffer energy in capacitors as a replacement for batteries. Still, they can store a small amount of energy, they charge faster than batteries, but their discharge phase is even faster leading to continuous power failures. Unlike mainstream computation platforms in which software can benefit from OS support, on this class of computationally constrained devices we execute code on bare hardware and power failures are not managed. Each failure causes losing processor's state, hence the progress in computation is lost.

In conclusion, this new class of devices, powered with harvested energy, buffered in small capacitors, operate intermittently as energy is available, alternating small burst of unknown length of computation, to period of inactivity of unknown length. We call this behavior *Intermittent Computing*. Figure 1.1 shows a typical intermittent execution pattern. The capacitor charges up to a voltage that triggers Micro Controller Unit (MCU) activation; the execution rapidly discharges the capacitor to a point where the stored energy is below the threshold needed to sustain the micro controller activity, hence the device switches off until the end of the next charging phase.

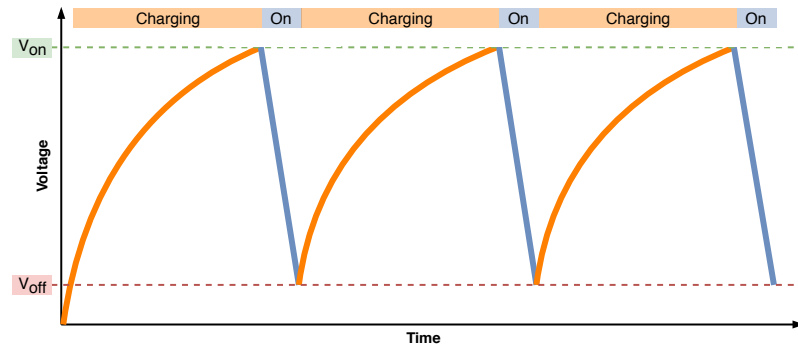


Figure 1.1: A typical intermittent execution pattern. The capacitor charges up to a voltage that triggers Micro Controller Unit (MCU) activation; the execution rapidly discharges the capacitor to a point where the stored energy is below the threshold needed to sustain the micro controller activity, hence the device switches off until the end of the next charging phase.

This new paradigm poses many challenges. Given the unpredictability of energy sources, the developer has to address unpredictable executions flows, since a power failure can occur at any time. At reboot, in the absence of a state saving mechanism, the program counter is reset to the initial value, internal state and peripheral configuration are lost. To support the adoption of this family of devices, we need to build a software stack that helps developers overcome the issues posed by intermittent computing.

1.1 PROTECT AGAINST POWER FAILURES

In our scenario of tiny batteryless devices, deployed and forgotten in the environment, we have tight constraints in terms of computational power, available memory, and available energy. Most importantly, due to the unpredictability of the energy source, the stability of power supply is not guaranteed and power failures are far from being a rare event. When they occur, the content of registers and main memory is completely lost.

Let us imagine a program that requires the execution of X instructions to complete, as shown in the example in Figure 1.2. Suppose that the energy buffered in the capacitor is enough to execute approximately up to $X/2$ instructions. As we can see in Figure 1.2a, every time the program starts, a power failure causes a reset right in the middle of the computation. Every time a reset happens the execution starts from the beginning, without any chance to complete.

As depicted in Figure 1.2b, if we want to guarantee forward execution, we need a way to save the state on Non Volatile Memory (NVM), so that it persists across power failures, allowing the device to continue the computation from the last successfully saved state. We call the saved state a *checkpoint*. Saving a checkpoint is a costly operation:

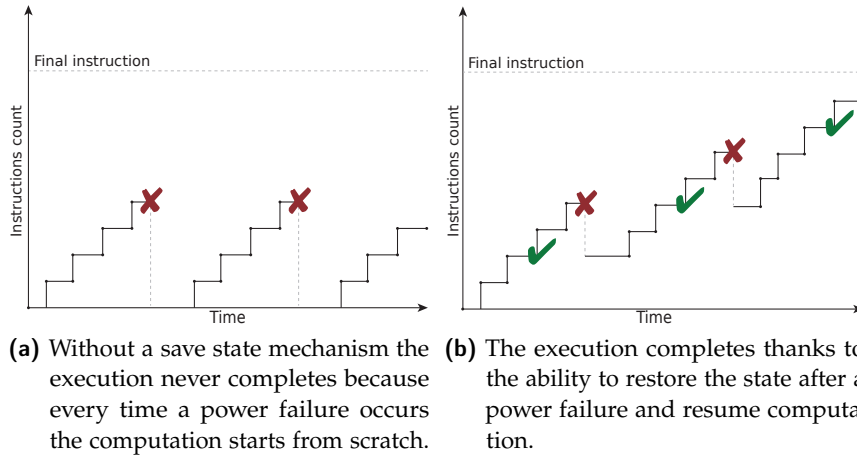


Figure 1.2: When a power failure occurs \times , the program counter is reset to the initial value, internal state and peripheral configuration are lost, hence the progress in computation. In the absence of a state save mechanism the computation must restart from scratch, potentially never reaching the final instruction 1.2a. With a state save mechanism, when a power failure occurs, the computation can be resumed from the last successfully saved state \checkmark , ensuring code completion 1.2b.

it introduces an overhead both in terms of energy and time. This overhead depends both on the size of the checkpoint and on the technology used to implement *NVM*, the most common on these platforms is Ferroelectric RAM (*FRAM*). For instance, saving a state on this class of memories can require up to $5.7\mu\text{J}$, where a single clock cycle on average consumes 1nJ [3].

There are different solutions to save checkpoints and guarantee forward execution in presence of intermittent computation [29, 34, 3, 2, 20]. Each solution in this class proposes a different take on the two main questions of checkpoint saving: what to include in a checkpoint and when to execute it, trying to guarantee forward execution, while minimizing the overhead.

There exists a second class of solutions to address power failures that, instead of saving checkpoints to adapt regular code to this class of devices, proposes to decompose code in a set of smaller portions of atomic computation called *tasks* [24, 25, 16, 35]. By atomic computation we mean that either the code of a task is completely executed without being interrupted by a power failure, or it must be entirely re-executed. Moreover, each task should exhibit transactional semantics: it should not produce persistent effects unless its execution successfully completes. With task-based solutions, the runtime executes one task at a time. The execution proceeds as in a pipeline where data go through different stages represented by these atomic tasks. The system saves partial results between tasks on *NVM* to guarantee forward execution.

Different solutions implement different ways to communicate between tasks through data passing.

With checkpoint or a task-based solutions we can guarantee code completion. Still, this is not the only challenge posed by the new paradigm of intermittent computing. During the inactivity time the board is completely off, obviously without a clock, so keeping time on these batteryless devices is a complex task.

1.2 TIME RELATED REQUIREMENTS

Sensing activities usually require some kind of guarantee on the freshness of data. Let us imagine for instance a device that is controlling the air conditioning based on temperature. If the device senses data and then powers off, we need to know at resume if these data are still relevant to the specific application. Implementing some sort of time keeping mechanism, in the absence of a persistent clock is a complex task. Some existent solutions rely on the natural decay of Static Random Access Memory (SRAM) cells to estimate the length of the off period [28]; some other rely on the discharge of a small capacitor [17]. Both of these solutions need an effort from the developer, who has to add explicit checks to consider data expiration, sensing rates, temporal signal properties, resulting in programs difficult to debug, maintain, and understand [16]. Once again we see the importance of a framework that hides some of this complexity to developers.

In conclusion, power failures pose a threat to the execution of code on this class of devices. Moreover, sensing activities usually require some kind of insurance on the freshness of data. To truly enable the usage of batteryless devices, we need a software stack that guarantees forward code execution and supports the definition of time related requirements.

1.3 ENERGY AWARENESS

Energy efficiency is important because through energy efficiency we can maximize the amount of code that can be executed with a given energy budget. Checkpoints are a way to protect the execution against power failures: we add a checkpoint to save the state across failures, so that we can resume the work at reboot. Given a workload, the more of it we can fit in a single execution, without power failures, the fewer checkpoints we need. The same goes for tasks: a higher energy efficiency allows us to fit more code within a single atomic task, that must execute without power failures in between. For this reason, all the existent systems developed to enable intermittent computing, naturally consider energy efficiency as the main optimization target.

Our device harvests energy from a wide variety of sources, each with different characteristics, buffering energy in a capacitor. The MCU

activates at a given threshold V_{on} , and the computation continues until the turn off threshold V_{off} is reached. This second threshold is usually set to the minimum voltage required to operate the device. For instance TI MSP430, one of the most common platform for intermittent computing, requires 1.88V to sustain computation [27].

The activation threshold is a powerful knob that impacts the system behavior and, in particular, its energy efficiency. The higher is the voltage, the more energy is buffered on wakeup, hence the device can work for a longer time interval. Obviously, reaching a higher voltage requires a longer charging time, and, due to the capacitor characteristic, this time increases more than linearly. The correct selection of the activation threshold is not an easy task. In fact, increasing it ensures more energy and a longer execution time, but increases the charging time; decreasing it allows the device to turn on earlier, but increases the chances of a power failure because the buffer stores less energy.

The capacitor, due to the unpredictable nature of the energy source, may discharge and recharge several time. Therefore, the MCU's operating voltage varies several time during the execution of an application. Ahmed et al. proved that power consumption and clock speed significantly change as the operating voltage spans different values during the execution. Their experiments showed a reduction of power consumption per clock speed by a factor up to 363.36% when the voltage is at its minimum value of 1.88V, compared to a capacitor voltage of 3.6V [1]. These results prove the importance of the correct selection of voltage thresholds as a mean to obtain energy efficiency. This selection is a quest to find the correct balance. A higher voltage threshold results in a longer execution span, fewer power failures, and therefore a lower overhead caused by a lower number of checkpoint restores; a lower threshold results in shorter charging time intervals and lower power consumption. The unpredictability of the energy source makes this selection even more complex.

Some solutions propose a refinement process that tries to calibrate voltage thresholds at runtime [2, 6]. None of them takes in consideration the changes in power consumption and clock speed at different voltages. All of them are designed around checkpoint based systems, and we still lack an efficient solution to deal with activation threshold on task-based systems.

1.4 FROM ENERGY TO THROUGHPUT

As we mentioned earlier, systems to enable intermittent computing, try to structure code and support its execution in a way that guarantees its completion, either through checkpoints or task decomposition.

Existent solutions are static: they do not adapt their behavior at runtime based on the energy intake, but propose an approach defined at compile time that must adapt to all energy scenarios. In particular,

they do not manage workload at runtime. With checkpoints code is structured in a single stream of instructions, while in existent task-based systems, tasks are executed one after the other in a sequence decided at compile time. Some solutions adapt the activation threshold at runtime [2, 6] to maximize energy efficiency, but even if the threshold refinement is performed at runtime, they exhibit a static behavior for what concerns workload management.

The development of a static system: one that is not able to dynamically adapt to current energy intake, forces unconditionally to look for energy efficiency maximization. In fact, if such a system wants to be robust, then it must deal with the worst case scenario of scarce energy. Still, being too conservative produces rigid solutions that do not adapt to the opposite scenario of sudden bursts in energy provisioning that we proved with experiments are not isolated cases.

Another possible solution, would be to adapt the workload at runtime, so that the system can make the best use of the available energy. When the harvester provides a high amount of energy the system could increase the workload, decreasing it when the energy intake drops. As we said, checkpoint solutions are based on a single stream of instructions interleaved with checkpoints to protect against power failures. With such a code structure, it is difficult to reason in terms of workload management: either we continue the execution of the code, or we perform a checkpoint.

Task-based solutions are more suited to this new approach in which the system decides at runtime what to execute based on the current energy budget. As we saw, in these systems code is structured as a collection of atomic pieces of computation. These fragments offer a unit of computation that can be run independently, therefore a collection of tasks can be the right tool if we want to implement a new solution that manages workload at runtime. We can envision a system in which we select at runtime what to execute from a set of available tasks, based on the current energy intake.

Developers build their applications around data, especially on this class of devices particularly suited for Wireless Sensors Network (WSN). These applications sense data from the environment, filter, classify, store them; they take decisions based on data and they may operate actuators based on these decisions. Data is all it matters and is often time sensitive. Data capture events that change at a given frequency, or that last a given amount of time, such as the concentration of a given pollutant, heart rate, humidity level, temperature, movement. The rate at which these data are produced is a relevant requirement and impacts the whole application, that again is often time data centric. Since data drive application's execution, the number of relevant data produced over time is directly associated to the application's throughput.

We argue that the scenario of intermittent computing can benefit from a shift in perspective: from energy efficiency, to the satisfaction of throughput requirements. The ultimate goal, when developing software stack for intermittent computing, should not be energy efficiency per se, but energy efficiency should be considered as a way to minimize the overhead and increase code execution to maximize the throughput. Current systems are designed around energy efficiency maximization, and, only as an emergent feature they try to maximize code execution. We propose a shift: manage energy budget in a way that guarantees throughput requirements, considering these requirements as a goal, and energy and workload management as a way to satisfy that goal. The satisfaction of throughput requirements should be *the* main goal, and energy management should be guided by that goal, and not necessarily by the quest for maximum efficiency. If we design a new system that is able to accept minimum throughput requirements, we can direct our focus on their satisfaction, instead of indirectly reaching them through maximum energy efficiency.

To define throughput we need a unit of computation, tasks are a better fit for this view. Their atomicity makes it easier to define throughput as number complete iterations over time. Moreover, as we said before, we can select tasks at runtime, based on the throughput requirement and on the current energy budget.

1.5 OUR CONTRIBUTION

As we discussed in the previous Section, we want to focus on throughput as our main goal, developing a framework for intermittent computing that has this requirement at its core. For this reason, we select for our proposal a task-based approach.

Let us discuss briefly how a throughput requirement maps to an example scenario. Let us consider a device used to perform activity recognition, based on data sensed from an accelerometer. We can easily imagine a task-based structure where a task senses data, and a task produces the classification. In this scenario, the throughput is more meaningful if applied to the entire process: for instance, the developer can ask to produce a classification once every second. So, the developer should be able to define the throughput for a higher level function: activity recognition, without being forced to consider the throughput for each single task.

For this reason we introduce the concept of *applications*, new to this class of devices. An application is a collection of tasks, necessary to provide a higher level service. For instance, a task that senses

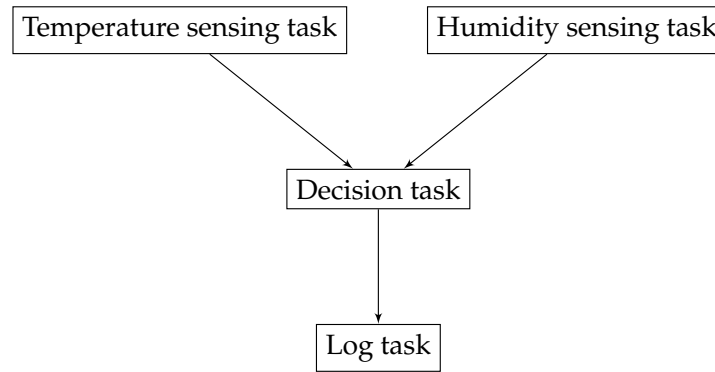


Figure 1.3: Example of an application that manages a HVAC system. Applications are collections of tasks, grouped to offer a higher level function. Tasks communicate among them through data exchanges. Applications can be represented as graphs in which tasks are nodes, and edges represent data dependencies among them.

temperature can be part of an application that manages an HVAC system. This application can collect four tasks:

- a task that senses temperature;
- one that senses humidity;
- one that decides if activate the ventilation and operates on an actuator;
- one that logs the decision, sending it to a data collection point.

An application is defined by its tasks, and these tasks communicate among them through data passing. We can imagine an application as a graph in which tasks are nodes, and edges represent data exchanges among them. Figure 1.3 represents the graph of data dependencies for the application for HVAC management. The code deployed on our system may be composed by several tasks, grouped in several applications, each one with minimum throughput requirement.

We said that we want to focus on throughput, also by managing workload at runtime. This structure of applications and tasks provides us the ideal setting to pursue our goal. In fact, in this scenario we can implement a runtime scheduler that selects at every wake up a set of tasks to be executed with the current energy budget. Each task in the execution plan is selected as part of an application, and in particular, as an iteration of a given application.

Existent task-based solutions, such as Alpaca [25], or Mayfly [16], statically schedule tasks at compile time. There is no runtime task selection and the execution plan is simply composed by a sequence of task executions, statically determined at compile time once and for all, no matter the runtime energy intake.

For the first time on these devices, we propose a dynamic scheduler that selects which task to execute taking into account, not only the current energy budget, but also the desired throughput of applications.

As we said, voltage thresholds management impacts deeply on system performance. For this reason, our system implements runtime management of activation threshold, once again as a way to reach the desired throughput. Thresholds are increased or decreased as a response to deviation to the desired value of this requirement.

In this document we present our contribution that is threefold:

1. we propose a *new programming abstraction*, where tasks are grouped in applications, with explicit non functional requirements in terms of minimum desired throughput;
2. thanks to this new way to describe software for this class of devices, we build a *dynamic scheduler* that is energy aware and reacts to changes in the amount of the harvested energy, with the goal to satisfy the throughput requirements of the applications;
3. we build an *adaptive threshold management system*, first in the task-based scenario, that adapts the activation threshold, based on the scheduler decisions.

1.6 STRUCTURE

The rest of this document is structured in four main parts. The description of intermittent computing scenario, the analysis of state of the art solutions, the description of our contribution and of its implementation, and finally its evaluation.

1.6.1 Introduction to Intermittent Computing

In the first part, we provide an extensive description of the intermittent computing scenario. In particular, in **Chapter 2** we describe its peculiarities and challenges. To guarantee forward execution, we must save the state across power failures, and restore it on reboot. As mentioned earlier, we call this saved state a checkpoint. Code execution results should be indistinguishable between standard and intermittent computation. We say that we have an inconsistency every time the results differ due to power failures. In this chapter we analyze the cause and nature of these inconsistencies. Moreover, we provide an insight on the consequences of not having a persistent clock, and what this entails from the developer's point of view.

We conclude the Chapter with a description of the constraints and goal that should guide the design of a system to support intermittent computing.

1.6.2 *State of the Art Analysis*

The second part of this document presents state of the art solutions.

- In **Chapter 3** we describe how different checkpoint and task-based solutions address the challenges previously presented. For each one of them, we analyze how they map to the constraints and goals that we listed in Chapter 2. We conclude the Chapter with a discussion on the fundamental differences between the two classes of solutions: checkpoint and task-based ones.
- In **Chapter 4**, we discuss activation threshold management, starting from an analysis of its importance, showing the results of simulations that prove its impact on system performance. We describe the reasons that make threshold management a particularly difficult task, and we present how the most relevant state of the art solutions tackle this problem.

1.6.3 *Contribution*

In this part we present our contribution and its implementation.

- In **Chapter 5** we present our programming abstraction. We describe how we structure code in tasks and applications, in a way that allows the developer to ask for minimum throughput requirements at application level. As we mentioned earlier, tasks communicate among them through data exchanges. These communications describe data dependencies between tasks. Two tasks that are connected by a dependency have a producer consumer relationship. Through an example, we show the necessity of an extended semantics for these data dependencies, to capture the functional requirements that can arise when describing an application for our class of devices. Therefore, we introduce our extended set of semantics, we describe the most common dependencies patterns, and finally we propose an overview on how they can be used to address different scenarios. Our proposed abstraction is orthogonal to scheduler implementation.
- In **Chapter 6** we introduce the main component of our contribution: a dynamic scheduler that manages workload and activation threshold at runtime, to satisfy throughput requirements. We start with a discussion on what it means to offer a guarantee on minimum throughput in intermittent computing, and why it is necessary to implement the ability to change the workload at runtime to adapt to the current energy intake. Then we analyze the internals of the scheduler, from its initialization to the discussion on how it selects tasks, always looking for the satisfaction of throughput requirements. Finally, we describe the reactive

nature of our scheduler. The inconsistency of power sources, and consequent power failures can affect throughputs. We show how our proposed system manages applications that, at runtime, exhibit a throughput that does not satisfy the requirements, and we analyze how the concept of fairness can be adapted to our setting. Finally, we present the knobs offered by our system to the developer to fine tune the behavior of the scheduler.

- In **Chapter 7**, we succinctly present an overview on the implementation of our system, and on the workflow that allows one to obtain a firmware from the description of tasks and applications.

1.6.4 Evaluation

In **Chapter 8** we present an evaluation of our system.

To properly evaluate our contribution, we need a way to conduct repeatable experiments. With our class of devices, this means that we also need a way to reproduce a given energy source. We call *energy trace* the series of energy values provided by a given source over time. We describe an evaluation environment that is able to execute code on a simulated device as if it was powered by a given trace. We also present our contribution to an existent evaluation platform, which we extended to better support our evaluation.

We present the scenario, the metrics and the baseline. Finally, we describe the results of experiments conducted by powering the device with different energy traces, to show the performance of our solution in different energy harvesting conditions. These results include both an evaluation on how well our system can satisfy throughput requirement, compared to the baseline, and on how the knobs offered to the developer to tune the scheduling policy, actually affect system's performance.

Our dynamic approach leads to an increase in application's throughput of more than 32% compared to a static solution, when the board is powered by a source that exhibits sudden drops in power supply, as well as unpredictable peaks in energy provisioning. In case of scarce energy provisioning, our solution reaches an application's throughput over 43% higher than a static one, by dynamically managing both the activation threshold and workload at runtime.

We call *correctness* the sum of time intervals during which the requested throughput of an application is satisfied, over the complete execution time. Our proposed solution reaches a correctness over 80% higher than the static approach.

2

INTRODUCTION TO TRANSIENTLY POWERED COMPUTING

“Smart dust”: a system of many tiny invisible networked computers to support humanity. Envisioned decades ago, not yet a reality. As pointed out by Hester et al. [15] batteries are the single greatest threat to this vision. Expensive, bulky and hazardous, replacing them when they wear out poses a serious environmental issue. A real huge scale deploy-and-forget scenario is impossible as long as batteries are the only viable option.

Energy can be harvested from the environment: solar, radio frequency, kinetic are just some examples of easily accessible energy sources, still we need a new place to store the buffered energy.

Capacitors may be the answer to our concerns as they overcome battery issues, yet they are far from being a silver bullet. They can buffer a small amount of energy, charge faster than a battery, but their discharge phase is even faster leading to continuous power failures. Moreover their package and form factor may pose a constrain to the design of our tiny devices. Still they are the most promising solution to finally enable a real “smart dust” [15].

Due to the capacitors nature and the unpredictability of energy sources, these devices violate one of the most basic assumption of computing: a stable power supply, alternating small burst of unknown length of computation, to periods of inactivity of unknown length.

Different hardware platforms are already available to implement batteryless solutions: Flicker [14] and WISP [31] are two examples. The average Micro Controller Unit (MCU) has a very limited amount of memory and storage, an example of these constrained MCU is Texas Instruments MSP430.

To support the adoption of this family of devices we need to build a software stack that helps the developer overcome the issues posed by this new paradigm of computation.

In this chapter we present the challenges posed by Transiently Powered Computation (TPC), starting from the cause of them all: the *intermittent computation*.

2.1 INTERMITTENT COMPUTATION

Thanks to the significantly higher lifespan and resistance of capacitors, batteryless devices can be deployed in a huge variety of scenarios and therefore be equipped with many different harvesters, scavenging energy from different sources. Each energy source has a different volume and different availability profiles.

Bhatti et al. [4] propose a systematic analysis of energy harvesting techniques, distinguishing between the relevant energy sources and the corresponding extraction techniques. The energy source is the environmental phenomena from which one may extract energy, thanks to harvesting mechanisms.

With bio-chemical sources, biological or chemical energy is extracted through chemical reactions. Energy can be extracted from thermal sources by altering the thermodynamic equilibrium of an object to produce an energy flow, usable to harvest electric energy. Vibrations, mechanical stress and sound wave are popular sources of kinetic energy [4], that can be harvested thanks to piezoelectric, electromagnetic or electrostatic effects. Kinetic energy can also be harvested from the flow of common fluids like air and water through micro turbines. Visible light, is probably the most commonly known radiant energy source, extracted thanks to photovoltaic cells. Another relevant example of radiant source is the energy extracted from RF transmissions, commonly used to power RFID devices.

Biochemical and thermal sources are those with the lowest reported performance in terms of harvested power, typically in the μV range. Energy harvesting from kinetic sources is vastly employed in Wireless Sensors Network (WSN), and its performance varies greatly in terms of power, spanning from few μV to tens of mW [4]. The extraction of radiant energy is strongly influenced by the size of the harvesting device, and by the distance from the source.

While some of these aforementioned phenomena are fairly stable, for instance outdoor solar radiation, we can not safely assume the absence of fluctuations. This instability may cause power failures, and therefore it may result in intermittent computation.

Figure 2.1 shows a typical intermittent execution pattern. For simplicity, in this example obtained with a simulation, the device is powered by a source with a square wave profile. The energy buffer is a $16\mu\text{F}$ capacitor. The MCU activates at a given voltage threshold V_{on} and starts the computation, draining energy from the capacitor. The capacitor charges as long as the balance between energy intake and outtake is positive. As soon as the power source stops providing energy, the MCU computation rapidly discharges the capacitor to a point where the voltage is below the minimum operating value V_{off} , hence the device switches off. During the off interval, when not powered by the source, the capacitor slowly discharges due to current leakage.

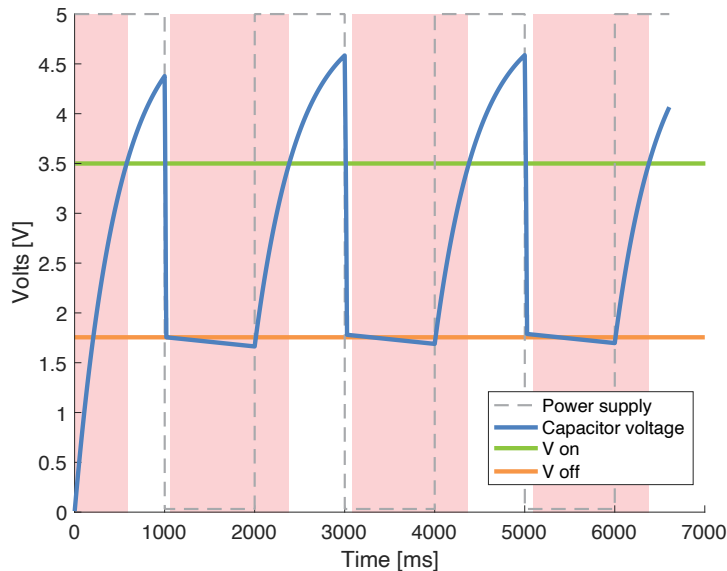


Figure 2.1: A typical intermittent execution pattern obtained with a simulation. For simplicity, the device is powered by a source with a square wave profile. The energy buffer is a $16\mu\text{F}$ capacitor. The MCU activates at a given voltage threshold V_{on} and starts the computation, draining energy from the capacitor. The capacitor charges as long as the balance between energy intake and outtake is positive. As soon as the power source stops providing energy, the MCU computation rapidly discharges the capacitor to a point where the voltage is below the minimum operating value V_{off} , hence the device switches off. During the off interval, when not powered by the source, the capacitor slowly discharges due to current leakage.

Capacitors with higher capacities increase the amount of energy that can be stored, but also the time needed to reach the activation threshold.

Given the unpredictability of energy sources, the developer has to address unpredictable executions flows, since a power failure can occur at any time. At reboot, in the absence of state saving mechanisms, the program counter is reset to the initial value, internal state and peripheral configuration are lost. Overcoming this problem is far from being trivial, posing a severe threat to a batteryless vision, we call this the *forward execution problem*.

2.2 FORWARD EXECUTION PROBLEM

Imagine that we need to implement a back-end server to support an online sales platform. We probably want to satisfy a given availability requirement, and to reach that goal we usually deploy a redundant system. In fact a failure in the infrastructure may make unreachable our server, so instead of relying on a single instance, we orchestrate multiple copies of our server, potentially deployed in different regions,

to minimize the probability that they all incur in the same failure at the same time. In that way the system as a whole reaches an availability higher than the single component.

Relevant informations on internal state are kept on Non Volatile Memory (NVM), potentially distributed across the system. The redundancy should be transparent to the user, hence these multiple data repositories must be consistent, so that the result of an interaction with our system is independent from the replica that we contact. Given that a complete power failure is a rare event, thanks to our redundant infrastructure, we accept different degrees of consistency [33], potentially looser than the strict one [21], and we accept to potentially deal with state reconciliation to restore a consistent global state after a failure.

Now let us scale down our scenario from servers distributed across the internet, to distributed tiny batteryless devices, deployed and forgotten somewhere in the environment, with high constraints in terms of computational power, available memory and available energy.

On batteryless platforms, the stability of power supply is the main problem, power failures are far from being a rare event and when they occur the content of main memory is completely lost. Coordinating a redundant infrastructure of connected devices, coping with local failures is currently an unfeasible solution. Given the scarce power budget each access to NVM has a big impact on execution. For the same reason accesses to radio equipment to exchange messages between nodes, or to a gateway, must be reduced as much as possible. These constraints, in addition to the scarce computational resources, make not applicable any of the consistency models, or state reconciliation algorithm, since they require the collection of a global state through several accesses to NVM and the exchange of numerous messages among nodes [33]. In other words we can not see the system as a whole and therefore count on features emerging from the coordinated effort of redundant replicas. Instead we must look for ways to deal with the problem of failures at single node level.

Let us consider the pattern of execution shown in Figure 2.1. Imagine that the deployed program requires 300.000 clock cycles, from the initial instruction to the final return, with just a single possible execution flow. Suppose that the capacitor, when fully charged, can provide energy to execute up to 100.000 clock cycles. Every time the program starts a power failure causes a state reset at one third of the complete execution and we never reach the final instruction. Capacitors can be replaced with bigger capacitors, but obviously we are not eradicating the problem once and for all. We need a software solution to collect the current state and guarantee forward execution, a convenient place to store that state locally and a way to restore that state when the board reboots after a power failure.

PLATFORM	OPERATION	TIME			CURRENT
		1B	256B	512B	
FRAM	Read	0.15ms	6.47ms	12.8ms	360 μ A
	Write	0.18ms	7.52ms	14.9ms	360 μ A
Flash	Read	n.a.	0.02ms	n.a.	12.4mA
	Write	n.a.	212ms	n.a.	12.4mA

Table 2.1: Measured FRAM and Flash characteristics. Flash is page-programmable with page size of 256 bytes. The write operation also includes energy required to erase a page, as necessary before rewriting. Taken from [4]

In Chapter 3 we will present different solutions to guarantee forward execution in presence of intermittent computation. All of them need some sort of NVM to store that state persistently across power failures. TI MSP430 family boards rely on Ferroelectric RAM (FRAM), similar to Dynamic Random Access Memory (DRAM), realized with ferroelectric material to implement non volatility. All the accesses to NVM are costly operations, and that overhead depends on the specific technology. For instance on TI MSP430FR boards, with FRAM as NVM, saving the complete state requires 5.7 μ J [2], where a single clock cycle on average consumes 1nJ. On this class of devices the access to FRAM is synchronous as long as the clock is lower or equal to 4Mhz [27]. If the NVM is implemented with Flash technology we would have a time and energy overhead orders of magnitude higher. Table 2.1 summarizes the performance of these two class of technologies.

We call the saved state a *checkpoint* in the computation, and we say that we *save a checkpoint* when we execute the operations to collect and save that state.

Not only we must reduce the frequency of these checkpoint save, in order to reduce energy consumption, but we must also ensure that each checkpoint represents a *consistent* state.

2.2.1 Checkpoint Inconsistencies

The goal of a checkpoint is to hide power failures and guarantee forward progress. Code execution results should be indistinguishable between the standard computation and TPC. We say that we have an inconsistency every time the results produced by a TPC device differs from those obtained without interruptions caused by power failures.

Following the framework proposed by Fuggetta et al. [10] we call *execution segment* the part of a process containing the current execution state and it usually includes .bss, .data, .stack, .heap, Program Counter (PC), Stack Pointer (SP), Status Register (SR) and General Pur-

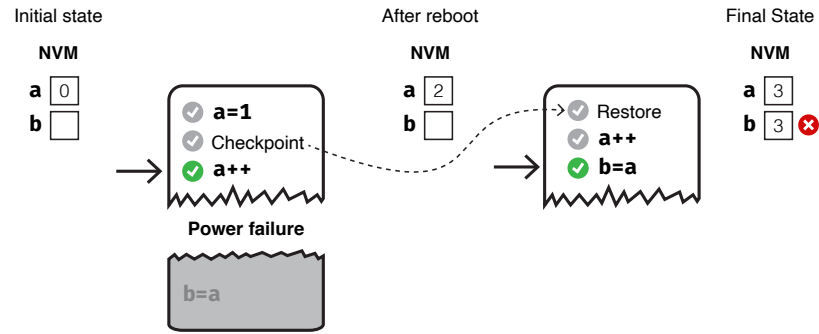


Figure 2.2: Intermittence causes an inconsistency. The variable `a` is stored on `NVM`. A checkpoint is placed before the increment instruction, the program counter, pointing to the increment instruction, is saved. A power failure causes a reboot *after* the increment. On wakeup the checkpoint is restored and with it the program counter. The next instruction is the increment of variable `a`. Since the variable is stored on `NVM`, its value persisted across restore, resulting in a double increment of its value.

pose Registers (`GPR`). To support `TPC` we must guarantee its integrity across power failures.

A possible solution would be to use `NVM` as main memory and checkpoint only registers. Thanks to its persistence across failures, memory content would be preserved, though, as discussed before, due to its implementation, the access to this class of memories consumes more energy compared to standard `SRAM`, the cost per bit is higher and its size is limited.

Instead of working directly on `NVM`, we can save the complete execution segment before a failure and restore it on volatile main memory on reboot. This could be done efficiently interleaving code instructions with saving routines, and implementing a restore process on wake up after a power failure.

The cost of `NVM` read and write suggests a third solution: divide the execution segment between `NVM` and volatile memory and save and restore only the non volatile portion.

Moreover the energy cost of a checkpoint is not constant as it depends on its size. The size can change during the execution as the stack grows and shrinks, so it would be convenient to perform the checkpoint save when the size is at a local minimum, to consume less space and energy.

Let us consider the example shown in Figure 2.2. In this example we use an hybrid solution in which a portion of the execution segment is stored on `NVM` and therefore does not need to be saved across power failures.

Variable `a` is on `NVM` and its value is initialized to 0. A checkpoint save routine is called after the initialization of `a`. The content of the execution segment is persisted on `NVM` and with it the program counter, pointing to the next instruction: the increment of variable `a`. The incre-

ment is performed and the new value for *a* is 2, this value is stored on *NVM* and therefore it will persist across reboots. A power failure occurs and the board switches off. Once the capacitor reaches a voltage high enough to support computation the board powers on again and the checkpoint is restored. The program counter, restored from the checkpoint, points to the increment instruction. The instruction is executed and the new value of variable *a* is now 3. This value is assigned to variable *b*.

This behavior is inconsistent, since it differs from the one we would have without power failures. The fact that the checkpoint saved the program counter *before* the execution of the increment instruction, together with the fact that the variable value persisted across reboots, caused a double increment. *With* power failures the final value of variable *b* is 3, *without* power failures the final value would be 2, hence the inconsistency.

In this example variable *a* is an integer, that can be stored in a 16-bit long word. Depending on the platform the write access to *NVM* is an atomic operation at different degrees of granularity. Let us suppose that the selected platform can write atomically up to 16 bits on *NVM*. This means that, as long as the variable's size is less or equal than 16 bits, in presence of a power failure, either the variable gets successfully written on memory, or it does not get written at all. TI MSP430 implements a technique called *charge pump* where an additional small capacitor serves as energy buffer to guarantee the completion of a word write, even in presence of a power failure caused by the complete discharge of the main capacitor. Nevertheless, if a variable stores a structured data, or its size is bigger than a word, it may happen that in presence of a power failure, its value is only partially written on memory, causing the corruption of its content. Moreover the complete checkpoint requires to write several words, depending on the portion that must be persisted, and for sure the charge pump can not guarantee its completion in case of power failure. Therefore the checkpoint save must be performed with a mechanism that resembles *Two Phase Commit* to ensure its atomicity.

To summarize saving a checkpoint is a delicate operation:

- it involves accesses to *NVM*, these memories are slower than regular ones, and the access require more energy, therefore its frequency must be reduced as much as possible;
- the size of a checkpoint depends on the portion of data that must be persisted, and it varies during the execution as the stack grows and shrinks, therefore we must try to save it when its size is at a local minimum;
- mixing persistent and volatile state may cause data inconsistencies, and result in wrongful computation when compared with continuous execution;

- finally we must ensure the atomicity of the save operation, so that we do not end up with partial, and therefore corrupted, checkpoints.

These issues clearly highlight the need for a carefully written checkpoint routine, and an even more careful placement of calls to this routine. Such a complex task can not be demanded to the developer, but must be part of a framework built to support the batteryless scenario.

After a power failure, during the charging phase of the capacitor, the board is powered off, until the voltage reaches the activation threshold. During this interval we have no clock, therefore no timers. The absence of precise time measurement adds a new brick in the wall between us and the envisioned batteryless scenario: the problem of dealing with timely execution requirements.

2.3 TIME WITHOUT A CLOCK

During inactivity time the board is completely off, and keeping time on these batteryless devices can be challenging. Harvested energy is often variable and difficult to predict, making not applicable any possible model that links time and energy. Moreover many devices can store only enough energy in a tiny capacitor for a few seconds of operation, leading to short bursts of computation, breaking any possible Real-Time-Clock implementation [17]. Still having some kind of knowledge on the length of inactivity intervals is important, not just for security reasons [28], but also to support sensing activities that usually require some kind of guarantee on the staleness of data. If a sensor gathers data and then powers off, we need to know at resume if those data are still relevant, to prevent energy costly operations of data processing on useless informations.

Two alternative batteryless techniques for keeping time on intermittently powered batteryless devices using remanence decay are presented in [17]: TARDIS, a software-only technique that checks the percentage of decayed Static Random Access Memory (SRAM) cells in an array to estimate the duration of a power failure, and CusTARD, a solution that relies on an ad hoc small capacitor charge decay, providing a finer grained timing at the cost of an hardware modification.

Even with these solutions time keeping is non trivial since, as pointed out by Hester et al. [16]

As developers add explicit checks that consider data expirations, sensing rates, and temporal signal properties, their programs become difficult to debug, maintain and understand.

Imagine a solution where a sensor reads temperature and, based on the value, operates some kind of actuator. If the board switches off

right after the sensing stage, on activation the data may be irrelevant and we may end up with a wrong effect on the world. Even with this simple example the effort to trace the staleness of the data is not trivial.

2.4 FROM PROBLEMS TO SOLUTIONS

Batteryless devices may be the enabling technology for our “Smart dust” vision. To power them we rely on many different energy sources and harvesting techniques, each one with different performance. The instability of power supply may cause power failures. Hence, to support this technology we must build a software stack that supports **TPC** and adapts to different energy profiles.

Many challenges arise from **TPC**. Power failures pose a severe threat on forward code execution and data consistency. Moreover, the absence of a persistent clock makes difficult to track time passing and to address time related requirements, often times connected to sensing activities.

We must react implementing countermeasures to solve these challenges on behalf of the developer, to support the batteryless vision, making it an attractive platform. As suggested by Maeng et al. [25], we propose a schematic overview of correctness requirements (**C1–2**) that we must meet to address the challenges introduced by **TPC**, and goals (**G1–4**) that should guide the development of the software stack.

C1 A program must **preserve progress**.

The system must be able to successfully complete the checkpoint operation, once started, and restore the computation without losing the progress, in presence of power failures causing the lost of volatile state. If the system autonomously places checkpoints, then it must guarantee that the next checkpoint is reachable. In fact, if the distance between two checkpoints is such that the energy required to execute the instructions between them is higher than the energy that can be buffered in the capacitor, the program would end up re executing the same set of instructions, without being able to progress. If the placement of checkpoint save operations is performed by the developer, then the system must signal any execution path whose worst case energy consumption exceeds the energy budget that can be stored on the capacitor.

C2 A program must have a **consistent view** of its state across volatile and non-volatile memory [25].

The system must save the state in a way that prevents the inconsistencies presented in Section 2.2.1.

G1 The system should **minimize the amount of wasted energy**.

Solutions should minimize the amount of energy spent to exe-

cute work whose results are lost due to power failures. Ideally the checkpoint should be done when the energy is just enough to save the state. Performing a checkpoint too early, would waste energy because the board could have executed more instructions before saving the state; delaying a checkpoint too much, would result in a power failure during the state save. Moreover the system should reduce the number of accesses to *NVM*, since it is a costly operation in terms of energy.

G2 The system should require **minimum user intervention**. Ideally the solution should require minimum input from the user, and should work transparently.

G3 The system should **minimize the size of persisted data**. Since the energy overhead of a checkpoint depends on the size of the persisted data, then its size should be minimized.

G4 Applications should be able to **respect time related constraints**.

Often time sensing activities require that data sampling is performed in a timely manner, and the application may require information on the staleness of data. As discussed in Section 2.3 time keeping is not trivial on this kind of devices, hence the system should assist the developer.

In Chapter 3 we propose an overview on different existent solutions, implemented to meet these constraints.

3

HIDING POWER FAILURES

On batteryless platforms the stability of power supply is not guaranteed. The fact that capacitors can store only enough energy for small bursts of computation, together with the unpredictability of power sources, leads to multiple power failures during code execution.

As in regular platforms the content of main memory is lost on shutdown: stack, heap and global variables; together with registers, stack pointer and program counter. Not only the result of computation is lost, but also the progress in code execution. To preserve data and progress we must save relevant informations on Non Volatile Memory (NVM). Moreover we must save them in such a way that guarantees their consistency, as on different platforms, NVM access mode is atomic at different word sizes. For instance on TI MSP430FR [27], NVM writes are atomic at 16-bit word level and power failures may happen while writing a data longer than 16-bit, causing data corruption.

Different solutions have been proposed to address the aforementioned problems. We can classify the systems developed to assist TPC in two broad categories: *checkpoint based solutions* and *task based solutions*. Section 3.1 presents those belonging the former class, while Section 3.2 those belonging to the latter. In Section 3.3 we discuss on the differences between the two categories.

3.1 CHECKPOINT BASED SOLUTIONS

In Section 2.2 we introduced the concept of checkpoint: a consistent snapshot of the execution segment. These checkpoints must be collected, saved on NVM and restored on reboot, to support TPC and guarantee forward progress in presence of power failures.

When dealing with checkpoints solutions we have to answer two main questions: what to checkpoint and when to execute checkpoints. While the answer to the first question is simply the complement of the portion of the execution segment that is stored on NVM, to answer the question on when to place checkpoints we need to consider far more aspects. Data inconsistencies may arise and we need to ensure that the buffered energy at the time of checkpoint is enough to support the save operation. This energy depends on the size of the portion of the

SOLUTION	C1	C2	G1	G2	G3	G4	REFERENCE
Mementos	✗	✓	✗	✗	✎	✗	Section 3.1.1 page 24
HarvOS	✓	✓	✓	✓	✓	✗	Section 3.1.1 page 26
Ratchet	✗	✓	✗	✓	partially	✗	Section 3.1.1 page 28
Hibernus	✎	✓	✎	✗	✗	✗	Section 3.1.2 page 30
QuickRecall	✎	✓	✎	✗	✓	✗	Section 3.1.2 page 31

Table 3.1: Overview on how different checkpoint solutions match constraints and goals presented in Section 2.4. “✎” means that the constraint or goal is satisfied requiring some sort of user’s intervention.

segment that must be saved, that varies during execution as the stack grows or shrinks.

There exists in literature different ways to answer to the placement question, these approaches can be broadly divided in two categories. In Section 3.1.1 we describe *Static Checkpoint Solutions*, in which checkpoint are statically placed at compile time. In Section 3.1.2 we present techniques where the decision on whether perform a checkpoint is taken at runtime, we call these *Dynamic Checkpoint Solutions*.

For each of them we discuss in what way they meet the requirements introduced in Section 2.4.

- ✓ Means that the Constraint **C**, or the Goal **G** is satisfied by the examined solution.
- ✗ Means that the Constraint **C**, or the Goal **G** is *not* satisfied by the examined solution.
- ✎ Means that user’s intervention is required to satisfy the Constraint **C**, or the Goal **G**.

Table 3.1 summarizes the mapping between solutions, goals and constraints.

3.1.1 *Static Checkpoint Solutions*

This class of checkpoint solutions statically place at fixed points in the code calls to functions that save the execution segment. Checkpoint functions are implemented differently, depending on the solution. Some of them execute a checkpoint every time the function is called, some other check a condition to decide whether to save the state or not. Still, in all of them the maximum number and location of checkpoints is determined at compile time.

Mementos

Mementos is a static checkpoint mechanism proposed by Ransford et al. [29]. The system is implemented on MSP430 family and does not

require any hardware modification. It is composed of a checkpoint routine exposed as a function call, and a suite of compile-time *LLVM* passes that automatically insert these calls and wrap the program `main()` function with code to restore the execution from an available checkpoint.

At each call the function measures the energy buffered in the capacitor by checking its voltage. If the voltage is below a given threshold V_{trsh} , it saves the checkpoint and returns. This threshold is determined through emulation experiments, conducted simulating the execution against different energy traces supplied by the user [11]. It corresponds to the voltage at which the capacitor buffers the average amount of energy that is needed to save the checkpoint. Once selected, the threshold does not vary at runtime. In Chapter 4 we propose a discussion on the importance of the selection of voltage thresholds in systems that support *TPC*, and the issues connected with these decisions.

Even if the number of checkpoints ultimately depends on the results of the energy checks, their placement and number is predetermined at compile time, thus we can consider *Mementos* a static checkpoint solution.

To support a wide range of applications, *Mementos* offers the following three different strategies to place calls to the checkpoint routine.

LOOP-LATCH MODE *Mementos* places a call at the end of each loop body, before the instruction to jump back to the condition evaluation, resulting in an energy check after each iteration.

FUNCTION RETURN MODE The calls are placed after every function call, resulting in a check after every return.

TIMER-AIDED MODE This mode is designed to reduce the frequency of checkpoints, given their high energy cost. It works in combination with either of the previous modes and requires a hardware timer interrupt that raises a flag at predetermined intervals. Each call executes the checkpoint if the voltage is below the threshold *and* the flag is up. The flag is lowered after each checkpoint for the next trigger point.

Beside these strategies for automatic checkpoint placement, the programmer can opt out to any automatic compilation pass and place calls manually.

The proper selection of the threshold does not guarantee by itself that the system has enough energy to complete the checkpoint, and therefore save progress [✗ C1].

Let us suppose that the developer opts for loop-latch mode. Suppose that, at the end the first iteration of a loop, the voltage is checked and it is above the threshold, so the execution continues without saving the checkpoint. Suppose that, after this first iteration, the harvester

does not receive any energy from the source. The voltage drops below the threshold during the next iteration, due to the energy consumed by the instructions in the loop body. At the end of this second iteration the voltage is checked, the checkpoint routine starts, but it does not complete successfully due to the energy shortage. The board switches off, because of the power failure. Now the harvester receives some more energy, and the board reboots. At reboot the system restores the last valid checkpoint, discarding the partial one, hence the progress is lost. Nothing prevents that this repeatedly happens, causing a starvation of the program. Moreover this would cause the re-execution of instructions, and the waste of energy to perform computation whose results are lost. Mementos does not address energy overhead minimization [✗ G1].

In theory the developer could autonomously place checkpoints, where the size of the stack is minimal, therefore minimizing the amount of data that must be persisted [✎ G3]. Still this would require a great effort, resulting in a solution far from being transparent to the user [✗ G2].

Mementos does not consider hybrid solutions in which the execution segment is partially stored on *NVM*, therefore each checkpoint must save registers, stack and globals. To overcome the problem of corrupted checkpoints due to power failures during the write instructions, as highlighted in Section 2.2.1, Mementos checkpoint routine never overwrites the last valid checkpoint, instead it uses deallocated space on *NVM*. Moreover the last word that it writes during checkpoint save is a known magic number, used to validate upon restore. If the developer does not autonomously store variables on *NVM*, re-execution does not produce data inconsistencies, since the whole execution segment is restored. This, together with the aforementioned saving techniques, guarantees that the content of a checkpoint represents a consistent state [✓ C2].

Mementos does not provide any support for timely execution [✗ G4].

HarvOS

HarvOS is a static checkpoint solution proposed by Bhatti et al. [5]. Checkpoints are obviously an overhead compared to the normal computation. Energy measurements are costly operations whose number should be reduced as much as possible. Finally not all checkpoints are equal: the more data must be saved, the higher is the cost in terms of energy and time. Starting from these observations, the authors of HarvOS propose to analyze the Control Flow Graph (CFG) with static code analysis techniques, to find the placement of triggers to checkpoint routines. This results in a placement tailored to the specific application.

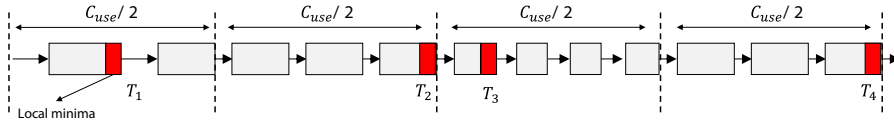


Figure 3.1: In HarvOS the CFG is sliced into sub-graphs containing instruction for up to $C_{use}/2$ clock cycles. The picture, taken from [5], considers a linear CFG for simplicity.

Thanks to the CFG analysis HarvOS obtains an estimation of the highest checkpoint energy cost $E_{CKP_{max}}$ at any point in the program’s execution, that depends on the size of the execution segment. With this input HarvOS computes the maximum number of clock cycles that can be used for program’s computation C_{use} . This value is computed analyzing the worst case scenario in which the board wakes up with a freshly charged energy buffer and it does not receive energy during computation. Therefore C_{use} is the number of cycles that can be executed with an amount of energy equal to $E_{wake-up} - E_{CKP_{max}}$.

The CFG is then sliced into sub-graphs containing instruction for up to $C_{use}/2$ clock cycles. At least one trigger point must be placed within each sub-graph. In fact the distance in terms of clock cycles between two calls should be lower than C_{use} , in order to have enough energy to reach the next call and perform the checkpoint. Suppose that the trigger call is placed at the beginning of a sub-graph, and the next one is at the end of the next sub-graph, for example T_3 and T_4 in Figure 3.1 [5]. Then to obtain a distance lower than C_{use} , the sub-graph size must be up to $C_{use}/2$.

The fact that the static code analysis performed by HarvOS considers the worst case energy scenario, together with the aforementioned checkpoint placement, guarantees that the save routine has enough energy to complete and guarantee forward progress [✓ C1].

Moreover the distance between two consecutive checkpoints guarantees that the next checkpoint is always reachable, and that it can complete, even with the only energy stored in the capacitor and no intake from the harvester. This prevents the starvation scenario described in Mementos, and guarantees that all the results of operations between two trigger calls are always included in a checkpoint [✓ G1]. Finally HarvOS checkpoints include the whole execution segment, therefore none of the data inconsistencies mentioned in Section 2.2.1 can arise [✓ C2].

To minimize the size of the checkpoint, and therefore its energy cost, HarvOS identifies in each sub-graph the block that corresponds to the minimum size of allocated memory and places a trigger call at the end of it [✓ G3].

As in Mementos the function call actually executes the checkpoint depending on a condition on the current buffered energy amount. This threshold depends on the specific trigger point, since it is computed

Listing 3.1: A fragment of C code presenting a **WAR** hazard between line 5 and 6.

```

1 | int a = 1;
2 | int b;
3 |
4 | int function(){
5 |     b = a + 2;
6 |     a = 3;
7 |     return b;
8 | }
```

as the energy needed to reach the next call, plus the energy needed to execute the checkpoint at that call.

Thanks to the aforementioned techniques, HarvOS proposes a custom solution based on the specific application, since the placement of the trigger calls and the threshold is decided at compile time using the **CFG** as an input. This placement is automatic and does not require user's intervention [✓ **G2**].

HarvOS does not provide any support for timely execution [✗ **G4**].

Ratchet

Ratchet is a static checkpoint solution proposed by Van Der Woude et al. [34]. A checkpoint is executed at every call without an energy check mechanism.

As described in Section 2.2.1 data inconsistencies may arise due to the unpredictability of the execution flow when in presence of power failures. In particular let us consider the Write After Read (**WAR**) hazard between two instructions. Listing 3.1 presents a fragment of code with this hazard between line 5 and line 6. A **WAR** consists in a couple of memory accesses in which the first one reads a value that is an operand of an instruction, and the second perform a write of the same value. If a power failures happens after the second instruction and the code is re-executed, the first instruction reads the result of the second, as the main memory is non volatile, causing a data inconsistency.

In particular instruction at line 4 reads the memory location corresponding to variable *a*, while instruction at line 5 writes the same memory location. Without power failures the function returns 3. The correct placement of checkpoints in presence of **WAR** is crucial to preserve a correct execution. In the aforementioned fragment of code, the only checkpoint placement that is guaranteed to preserve data consistency is between line 5 and 6. In fact a checkpoint placed before line 5 or after line 6, combined with a power failure after line 6, would cause the re execution of line 5 with a wrong value of variable *a*: 3 instead of 1, and a return value of 5 instead of 3. On the contrary let us consider a checkpoint between line 4 and 5. Any power failure would

not cause the re-execution of instruction at line 4, breaking the **WAR** data dependency.

For this reason Ratchet statically analyzes the code to find all pairs of instructions involved in **WAR** and statically places a checkpoint between the read and the write to break the dependency and prevent threats to data consistency [✓ **C2**]. The analysis is performed automatically and does not require any intervention of the developer [✓ **G2**].

With Ratchet main memory can be stored with any configuration: completely on volatile memory, partially or completely on **NVM**. Given the high frequency of **WAR** hazards Ratchet's approach results in the introduction of many checkpoints. Hence to reduce the overhead, Ratchet is best suited with **NVM** used as main memory. Moreover, to further mitigate the overhead, Ratchet reduces the portion of registers to persist on **NVM**, including only those actually involved in the computation that produced the **WAR**. This addresses the checkpoint size reduction goal only partially, because, unlike HarvOS, it does not consider any analysis of memory size when main memory is volatile, and therefore must be included in the checkpoint [✓ **G3 - partially**].

Since Ratchet needs to compare at compile time memory locations between reads and writes to identify **WAR**, it needs to statically know these addresses, therefore the usage of the heap is not considered as a viable option during the computation.

Ratchet statically places checkpoints without any energy related analysis. This does not guarantee, that the checkpoint can complete with the energy budget, and that the progress is successfully preserved [✗ **C1**].

It may happen that the distance between two **WAR** hazards, and therefore between two checkpoints, is too long and power may fail repeatedly before the next checkpoint is reached, causing the starvation of the program. To mitigate this, Ratchet introduces a timer that triggers an interrupt. At each interrupt, it checks if at least one checkpoint completed within the timer interval, and forces a checkpoint otherwise.

All the instructions executed between the last valid checkpoint and the power failure are not included in any checkpoint and therefore must be re-executed, wasting energy. Ratchet does not consider energy related conditions for checkpoint placement [✗ **G1**].

Ratchet does not support timely execution [✗ **G4**].

3.1.2 *Dynamic Checkpoint Solutions*

With dynamic checkpoint solutions, checkpoint routine calls are not statically decided at compile time, instead these solutions implement the checkpoint save routine as an Interrupt Service Routine (**ISR**) that is triggered by events at runtime.

Hibernus

Hibernus [3] is a checkpoint based mechanism implemented by Balsamo et al. over a MSP430 architecture.

In Hibernus, checkpoints are performed as soon as the capacitor voltage reaches a given low threshold called V_H . Once the checkpoint is completed, the board is put in Low Power Mode (LPM) or *hibernation state*. In this state it consumes a considerably lower amount of energy, since any computation is stopped. The computation is then resumed as soon as the capacitor reaches a second threshold $V_R > V_H$, implementing an hysteresis mechanism. Thanks to LPM lower energy requirement, the balance between the energy intake and outtake should let the capacitor charge, preventing power failures between the checkpoint and the computation resume. In any case the checkpoint ensures the ability to successfully restore the computation if the harvested energy does not balance the LPM consumption, causing a failure.

To implement Hibernus the board must be equipped with an on-chip voltage comparator. This comparator triggers an interrupt as soon as the capacitor reaches a given voltage threshold.

Hibernus checkpoints the whole execution segment. This prevents data inconsistencies [✓ C2], but does not implement any mechanism to minimize the size of persisted data [✗ G3].

Two routines are offered as a library to the developer: *Hibernate* that performs the checkpoint and put the board in LPM and *Restore* that reactivates the computation and restores the checkpoint, if needed. In particular:

HIBERNATE

- saves the complete execution segment on NVM;
- activates LPM;
- sets the voltage comparator reference to $V_R > V_H$;
- sets *Restore* as the interrupt service routine to the interrupt generated by the comparator;

RESTORE

- deactivates LPM;
- restores the checkpoint from NVM if needed;
- sets the voltage comparator reference to V_H ;
- sets *Hibernate* as the interrupt service routine to the interrupt generated by the comparator;
- resumes computation.

On initialization the voltage threshold that triggers the interrupt is set to V_H , and *Hibernate* is the corresponding interrupt handler. The

checkpoint is restored from *NVM* by the *Restore* routine if a power failure occurred while the board was in *LPM*, otherwise there is no need to restore the execution segment, since in Low Power Mode the main memory does not lose its content.

Setting the threshold V_H is critical to the stability of the system. This threshold must be set to a value that corresponds to a buffered energy higher than the amount needed to support a complete checkpoint E_σ [✎ C1]. Setting it to a voltage at which the energy is lower than E_σ does not allow the *Hibernate* routine to complete the checkpoint. Moreover, the buffered energy should be large enough to sustain the *LPM* energy requirements throughout the hibernation, otherwise a power failure would occur. In fact, even if the checkpoint ensures forward execution, restoring a checkpoint consumes energy that could be used to perform actual computation. A proper setting of these thresholds minimizes the amount of wasted energy, in fact the checkpoint would be executed when the energy is strictly sufficient to execute the save routine and to sustain hibernation [✎ G1].

In *Hibernus* the thresholds must be set and calibrated by the developer [✗ G2]. To overcome the problem of finding the correct values, Balsamo et al., propose *Hibernus++* [2]. An overview on energy management solutions, including *Hibernus++*, is presented in Chapter 4.

Hibernus does not address timely execution requirements [✗ G4].

QuickRecall

Jayakumar et al. [20] propose *QuickRecall*, a dynamic checkpoint solution. As already described in 2.2, to successfully perform computations across power cycles, the execution segment must be persisted on *NVM*. Conventionally, the linker maps *.bss*, *.data*, *.stack* and *.heap* on volatile *SRAM*.

The proposal of *QuickRecall* is to modify the linker so that these sections are mapped on *NVM*. With this different allocation, registers are the only content that must be saved and persisted on *NVM*. This significantly reduces the overhead of checkpoints, at the cost of a higher impact on *NVM*, that acts as conventional RAM [✓ G3]. Using *NVM* as main memory guarantees the absence of data inconsistencies [✓ C2].

As in *Hibernus*, *QuickRecall* needs an on-chip voltage comparator that triggers an interrupt when the voltage of the capacitor reaches a given threshold. In *QuickRecall* this threshold, called V_{trig} , is set by the developer. It must be carefully calibrated such that it guarantees that the checkpoint operation is successfully completed, even in case the board receives energy to just switch on and switch off immediately [✎ C1]. Setting the threshold to the minimum requested energy value, allows to continue computation up until the very last moment, still guaranteeing the completion of checkpoint operation, but requires user intervention for voltage trigger calibration [✎ G1]. The corre-

Algorithm 3.1. QuickRecall ISR

```

1 procedure QUICKRECALL_ISR
2   store(GPR)
3   store(SR)
4   store(SP)
5   checkpoint_flag = True
6   store(PC)
7   if checkpoint_flag == True then
8     wait for  $V_{dd} > V_{trig}$ 
   return

```

sponding Interrupt Service Routine (ISR) is implemented following Algorithm 3.1.

The execution of the ISR either completes with a return, once the voltage is back to a value higher than V_{trig} , or a power failure happens while waiting for the capacitor recharge (instruction at line 8 in Algorithm 3.1). If no power failure happens then the return instruction executes a context switch and the computation resumes.

On reboot, after a power failure, if the checkpoint flag is set to True, the board restores SR, GPR, SP, sets the checkpoint flag to False and restores the PC, to resume computation. Otherwise memory and registers are initialized and the computation is restarted from scratch. This flag check is needed to support the initial boot, where there is no need to restore registers.

It is important to notice that when the routine stores the Program Counter (PC), this points to the next instruction within the code of the ISR, which is the condition at line number 7 of Algorithm 3.1. This means that on reboot the computation is restored within the context of the service routine. When that happens the condition on checkpoint flag is not satisfied, thanks to the flag reset performed on reboot, so the return from ISR is executed and the computation resumes from the next instruction in the context of the developer's program.

As in Hibernus, the setting of the voltage threshold V_{trig} is critical since the corresponding energy, buffered in the capacitor, must be high enough to support the store/restore operations. The developer has to deal autonomously with threshold calibration [✗ G2].

QuickRecall does not address time related requirements [✗ G4].

3.2 TASK-BASED SOLUTIONS

Often times in computer science we implement solutions to hide underlying limitations, and present a higher level of abstraction to the end user.

For instance with speculative execution we speed up the throughput of a processor, completely hiding the mechanism to developers; we implemented the cache to pretend to have a bigger and faster main memory, once again with a solution that is completely transparent

to the user; registers renaming is just another example of pretending to have more resources than we actually have; we use schedulers to pretend that multiple processes are actually running together.

Developers produce their code, working at a higher level of abstraction, leveraging on properties obtained through techniques implemented on lower levels.

This encapsulation and abstraction process plays a key role in pushing the development of computer science, as “The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise” Dijkstra [8].

Checkpoints based solutions follow this approach: we implement checkpoint systems to hide power failures from the developer. By doing so we create a new layer in which the developer can abstract from the issues of intermittent computation. Still we may argue that this time this is no longer the correct approach. In fact power failures are far from being an isolated event. Moreover, due to data inconsistencies and deviations from the intended control flow, their impact on execution is not totally concealed to the developer, even with checkpoints.

There exists a second class of solutions to address power failures that do not try to completely hide them to developers, instead asking for their contribution to guarantee forward code execution and data consistency. These solution proposes an approach in which the application is decomposed in smaller portions of atomic computation, called *tasks*. Either the code of a task is completely executed within a computation burst, or it must be entirely re-executed. A task should not produce persistent effects unless its execution successfully completes. In other words the system does not checkpoint intermediate results within a task. This results in a *transactional semantics*:

- a task must be executed atomically;
- task’s effect must be visible only after its successful completion;
- the system must be in a consistent state even when the task fails due to power failure;
- task’s effects must persist after its completion.

Tasks can be viewed as atomic functions that ingest data, and produce outputs that change the internal state, or produce effects on the surrounding world through actuators. Each task can rely on data produced by other tasks, so we can connect them through data dependencies. This task decomposition is performed by the developer.

3.2.1 Constraints and Goals for Task-Based Systems

For each checkpoint solutions, presented in Section 3.1, we referred to constraints and goals introduced in Section 2.4.

In Constraint [C1], we stated that the distance between two consecutive checkpoints must be such that the second is reachable from the first one, with the energy that can be buffered in a capacitor charge, otherwise the system may be unable to save progress, and the program may be unable to complete. We can redefine this constraint to adapt to task-based systems.

In a task-based solution we expect that tasks run atomically. If the energy requirement of a given task is higher than the maximum amount that can be buffered by the capacitor, it may happen that the task can never complete, depending on the energy source. This blocks the ability to save progress, in fact the system would keep failing the execution of the same task, without saving any result, and without being able to complete the program. Hence, in a task-based solution, the system must deal with tasks whose energy requirement is higher than the maximum amount of energy that can be buffered in the capacitor, either proposing a different task decomposition, or emitting a warning.

Goal [G1] states that the system should minimize the amount of wasted energy by executing a checkpoint when the energy is just enough to save the state. With task-based solutions we waste energy every time the system can not complete a task because the current energy budget is not sufficient. In fact, given the atomicity requirement, the task should be re-executed, and the results of the partial execution would be useless. We redefine this goal for task-based solutions by saying that the system should minimize the wasted energy, by temporarily preventing the execution of a task, when the current energy budget does not match the energy required to run the task.

Goal [G2] states that the system should require minimum user intervention. Task-based systems require a shift in code structure, that requires user intervention, who has to work with a different abstraction: transactional semantic. Therefore we argue that this goal does not fit this class of solutions.

In Goal [G3], we stated that the system should minimize the size of persisted data, in order to reduce the overhead of checkpoints. In task-based solution the size of these data depends on task's outputs. The system must persist task's results, and their size depends on developer's choices. Hence, we argue that the minimization of the size of persisted data does not compete to the underlying task-based solution.

In conclusion, the analysis of the task-based solutions should consider this modified list of constraints and goals.

C1T A program must preserve progress.

The system must persist results of successfully completed tasks.
The system must deal with the fact that the energy requirement of a task may be higher than the maximum amount of energy that

SOLUTION	C1T	C2T	G1T	G2T	REFERENCE
DINO	✗	✓	✗	✗	Section 3.2.2 page 35
Alpaca	✗	✓	✗	✗	Section 3.2.2 page 37
MayFly	✗	✓	✗	✓	Section 3.2.2 page 40
InK	✗	✓	✍	✓	Section 3.2.2 page 42

Table 3.2: Overview on how different task-based solutions match constraints and goals presented in Section 3.2.2. “✍” means that the constraint or goal is satisfied requiring some sort of user’s intervention.

can be buffered in the capacitor. The solution must either propose a different task decomposition, or it must emit a warning.

C2T A program must have a **consistent view** of its state across volatile and non-volatile memory [25].

The system must persist task’s results in a way that prevent the inconsistencies presented in Section 2.2.1.

G1T The system should **minimize the amount of wasted energy**. Solutions should minimize the amount of energy spent to execute work whose results are lost due to power failures. The system should minimize the wasted energy, by temporarily preventing the execution of a task, when the current energy budget does not match the energy required to run the task.

G2T Applications should be able to **respect time related constraints**.

Often time sensing activities require that data sampling is performed in a timely manner, and the application may require information on the staleness of data. As discussed in Section 2.3 time keeping is not trivial on this kind of devices, hence the system should assist the developer.

3.2.2 Task-Based Solutions Overview

Different solutions exist in literature to support task-based TPC. We present the most relevant ones in this Section. Table 3.2 summarizes the mapping between solutions and the redefined set of goals and constraints presented in Section 3.2.1.

DINO

DINO is a solution proposed by Lucia et al. [24]. In DINO the programmer explicitly slices the code into tasks by means of boundaries implemented as calls to a function `DINO_task()`. Thanks to DINO implementation, the code within two boundaries is effectively a task,

Listing 3.2: An example of task decomposition with DINO programming model.

```

1  void main(){
2      s = rd_sensor();
3      DINO_task( );
4      c = classify(s);
5      upd_stats(c);
6      DINO_task( );
7  }
8
9  upd_stats(class c){
10     if(c==CLASS1){
11         DINO_task( );
12         c1++;
13     } else {
14         DINO_task( );
15         c2++;
16     }
17     total++;
18     assert(total==c1+c2);
19 }

```

its execution is atomic and its side effects are seen by other tasks only if the task effectively completes. Listing 3.2 shows an example of task decomposition with `DINO_task()` taken from [24].

Task definition in DINO is not explicit: the developer slices a monolithic code into tasks by means of fences (i. e. the `DINO_task()` function), instead of structuring the application as a set of functions connected by data dependencies. This makes DINO a solution in between checkpoint and tasks: it exhibits task based semantic, with an implementation that resembles checkpoints. Tasks are paths between boundaries. Traversing the control flow graph, each `DINO_task()` is the beginning or the end of a task. For instance in Listing 3.2 there are 3 tasks. The first one is composed by instructions at lines 3, 4, 5, 10, 11; the second one by those at lines 11, 12, 17, 18, 6; the third one contains instructions at lines 14, 15, 17, 18, 6. DINO saves the execution segment at every boundary that represents the beginning of a task. Though this is not enough to guarantee the atomicity of tasks.

For instance let us consider once again the Listing 3.2. Instruction at line 12 is part of the task composed by instructions at line 11, 12, 17, 18 and 6. Suppose that variable `c1` is stored on `NVM`. The increment of its value is a complex instruction presenting a `WAR` hazard. Its execution actually consists in three basic instructions: *reading* the current value of the variable from memory, incrementing it and *writing* the updated value in memory. Suppose that a power failure occurs *after* the execution of instruction at line 12 and before the next task instruction, that is instruction at line 17. Since the task completion failed, at wake up the execution is resumed from its beginning, in accordance with task

semantic. Now the second iteration of the increment at line 12, reads the value already incremented by the previous partial execution of the task, since `c1` is on `NVM`, resulting in a double increment. This violates the atomicity of tasks since the effects of a task (the increment of `c1`) are visible even if the task does not complete its execution.

To overcome the potential inconsistencies caused by `WAR` hazards, DINO implements the so called *versioning* of variables: an important concept to maintain data consistency across power failures with task based solutions, implemented also in our proposed framework.

In particular, at compile time, DINO analyzes the `CFG` to look for *write* instruction on variables stored on `NVM`. For each one of these instructions, it looks for all the tasks that include the write, traversing backward the graph to find a `DINO_task()` instruction that represents the beginning of a task. For each one of these tasks, it looks for a *read* instruction of the same variable, that precedes the write instruction. This process identifies all the `WAR` hazards within a task, involving variables stored on `NVM`.

To break the hazard and prevent inconsistencies caused by power failures [✓ **C2T**], DINO saves the variable value at the beginning of the task and restores it, together with the execution segment, in case of a resume after a power failure. This operation is performed by function `DINO_version()` automatically added at compile time as part of the `DINO_task()` instruction, only for those variables that are involved in `WAR` hazards, reducing the size of data that must be written on `NVM`.

In the previous example with variable `c1` from Listing 3.2, DINO saves a version of that variable *before* instruction at line 12, so that at every re execution of the task after a power failure, the read part of the increment instruction always reads the correct value and not the result of increments from previous failed executions.

DINO is implemented as a runtime library that includes instructions to perform checkpointing and versioning, and a set of LLVM passes that analyze the source code to obtain the `CFG`, look for potential inconsistencies and to translate `DINO_task()` instructions into calls to the runtime library. Moreover DINO analysis exposes the cost of each task, so that the developer can check if a task can fit the energy budget. Though this check must be executed autonomously by the developer and the system does not emit warnings [✗ **C1T**]. Moreover DINO simply executes tasks following code structure, without taking into account the current energy budget that may be insufficient to reach the next boundary [✗ **G1T**].

DINO does not address time related requirements [✗ **G2T**].

Alpaca

Alpaca is a task based solution proposed by Maeng et al. [25]. With Alpaca, the developer structures the code as set of atomic tasks and statically connects them by means of a library instruction called

Listing 3.3: Alpaca privatization and commit of scalar variables.

```

1 | TS int a, b, c;
2 | NV int c_priv; //added by ALPACA
3 | task example_1() {
4 |     c_priv = c; //added by ALPACA
5 |     a = 3;
6 |     int d = b;
7 |     e++; //removed by ALPACA
8 |     c_priv++; //added by ALPACA
9 |     pre_commit(&c_priv, &c, sizeof(c)); //added by ALPACA
10 |    transition_to(example_2);
11 | }

```

`transition_to(task)`, that switches the execution from the currently running function to the destination task. The execution flow starts from a specific task, decorated with the `entry` keyword, and proceeds task by task following the flow induced by the `transition_to(task)` instructions. Alpaca keeps a non-volatile pointer to the currently selected task, and after a power failure the execution resumes from that task. Tasks are void functions that share data among them by means of non-volatile global variables stored on `NVM`.

An Alpaca task is “a user-defined region of code that executes on a consistent snapshot of memory, and produces consistent set of outputs”. To satisfy this definition Alpaca implements data privatization. In fact, as already discussed, writes to global variables stored on `NVM`, in presence of power failures and `WAR` hazards may lead to wrong computation. With data privatization each access to the variables that potentially cause such inconsistencies is performed on a local version, that is copied into the global one only on task successful commit. To inform the compiler analysis on which variables are shared and therefore must be protected against `WAR`, the developer must decorate the declaration of such globals with the `TS` keyword. Alpaca performs a compiler analysis to identify if these shared variables are involved in `WAR`. In that case it modifies the developer’s task code to add the declaration of a non volatile (`NV`) private copy of such variable and to replace each access to the original variable with an access to the private copy. Finally it adds a `pre_commit` instruction that copies back the updated private copy into the original shared variable, so that other tasks can effectively access the updated value.

Listing 3.3 shows an example of how the Alpaca code instrumentation replaces each access to the variable `c` with an access to the privatized copy of that variable `c_priv`. In particular, Alpaca adds a private version only of shared variable `c`, since it is the only one involved in a `WAR` hazard, due to the increment instruction at line 7. This increment is replaced with the increment of the local copy `c_priv`

and the `pre_commit` instruction at line 9 reconciles the values of the copy private and the shared one.

The values of the private copies are written at the original shared variable locations with a two-phase commit. In fact a power failure may interrupt this process. With two-phase, first the system lists all the outputs that must be persisted, then saves them and exits from the commit only when they are all successfully written, so that all the variables are saved together, preserving the transactional semantic of tasks, even if this process is interrupted at any stage. In particular a `pre_commit` instruction is added for each private variable, and each invocation adds the variable to a non-volatile list of variables that must be committed, called `commit_list`. This list is statically sized for each task at compile time, so that it can accommodate the maximum number of privatized variables. After the last `pre_commit` instruction and before the transition to the next task, Alpaca adds an instruction to set a non volatile `commit_ready` bit. This specifies that the second phase of the commit can take place.

The second phase of the commit is implemented in the Alpaca runtime library by a void function `commit`. This function iterates on the `commit_list` and for each entry it copies the value of the privatized copy at the address of the shared variable. Once all the variables in the list are successfully committed it clears the `commit_ready` bit. After a successful commit the `transition_to` instruction updates the pointer to the current task to point to the next one.

On reboot the `commit_ready` value is checked. If it is set it means that the power failed during the commit, but after the `pre_commit`. In that case the commit operation can proceed following the `commit_list`. If it is not set than either the commit completed, and the execution can proceed to the next task, or the task was not completed due to a power failure during its execution and it must be re executed. The initialization of the private copy to the shared value, as shown at line 4 of Listing 3.3, ensures that each re-execution does not produce wrong results due to **WAR** hazards. This effectively ensures the consistency of the saved state [**✓ C2T**].

With these aforementioned techniques Alpaca offers a task-based solution that requires a minimal effort to the developer, who only has to decorate tasks with informations on shared variables and transition instructions. Still Alpaca does not address energy related optimizations, resulting in a static succession of tasks that can not be altered at runtime to address various energy profiles [**✗ G1T**]. Instead our proposed solution dynamically schedule tasks to best fit the real energy profile at runtime. Moreover in Alpaca the programmer must ensure that tasks size is such that they do not require more energy than the amount that the target device can buffer, and the system does not support this activity [**✗ C1T**]. Finally, with Alpaca the developer has to autonomously address any requirements that deals with timely

Listing 3.4: Alpaca privatization and commit of arrays.

```

1  int vbm_test(v){ v == cur_version; }
2  int vbm_set(v){ v = cur_version; }
3
4  TS int A[30], B[30], C[30];
5  NV int C_priv[30]; //added by ALPACA
6  NV int C_vbm[30]; //added by ALPACA
7  task example_1(){
8      int r=0;
9      for(int k=0; k<15; k++) {
10         r = rand()%30;
11         A[r] = 3;
12         int d = B[r];
13         if(!vbm_test(C_vbm[r])) //added by ALPACA until l. 24
14             C_priv[r] = C[r];
15         C[r]++; //removed by ALPACA
16         C_priv[r]++;
17         if(!vbm_test(C_vbm[r])){
18             vbm_set(C_vbm[r]);
19             pre_commit(&C_priv[r], &C[r], sizeof(C[r]));
20         }
21     }
22     transition_to(example_2);
23 }

```

execution or time constraints on sensed data, as Alpaca only offers versioning and task transition instruments [✗ G2T].

MayFly

Mayfly is a task-based solution proposed by Hester et al. [16]. Similarly to our solution, it encompasses a language and a runtime.

Mayfly not only addresses the issue of forward code execution, but also deals with the problems of timely execution in presence of power failures discussed in 2.3, given that often times data must be generated in a timely manner, as their freshness affects their utility [✓ G2T]. Mayfly makes the passing of time explicit, by timestamping data and keeping track of time throughout power failures thanks to decay based solutions [17].

Data staleness is not the only time related constrain: real applications usually process windows of data, producing meaningful and accurate results as long as the window is of a given size and the inter sample delay between two consecutive reads falls within some specific boundaries. For instance, let us consider this example [16].

A pedometer’s step-counting algorithm may call for 30 accelerometer samples collected at 10Hz. Depending on how it detects steps, that same algorithm may give accurate results as long as the 30 readings fall within a 4s window

Listing 3.5: Mayfly syntax example.

```

1 // Task definition
2 task_name (TYPE input, ...) -> (TYPE output, ...)
3
4 // Flow for activity recognition
5 sample -> compute -> send
6
7 // Predicate
8 // compute -> (int error, int activity)
9 sample -> compute[_ ,RUN] -> send
10 compute[_ ,WALK] -> log

```

and as long as no two readings are taken within 80ms of each other.

Mayfly proposes a data flow programming abstraction in which tasks are connected by simple data dependencies, decorated with explicit time requirements, so that the developer can focus on the high level application sensing goals, leaving the burden of satisfying these requirements in presence of intermittence to the underneath runtime.

The developer provides to the Java Mayfly compiler two components: a set of *tasks* written in Embedded-C, and a set of dependencies that describe the data relationship between tasks, called *flows*. Tasks are atomic units of computation, and flows connect them in a graph of data dependencies. Flow dependencies may be decorated with *conditions* and *timely constraints*.

Conditions activate or deactivate the flow dependency based on the evaluation of a predicate, allowing conditional execution between tasks. For instance let us consider three tasks *A*, *B*, *C* and a boolean predicate *p*. Task *A* may be connected with a flow to both task *B* and *C*. These flows may be decorated with the predicate so that if the result of the evaluation of *p* is *true*, then the flow to *B* is active and the one to *C* is not, otherwise if the evaluation of *p* is *false*, task *C* is enabled and *B* is disabled. In the example shown at line 9 and 10 in Listing 3.5 the compute task classifies the data coming from an accelerometer to recognize the activity and has two outputs: the error of the classification and the actual classification. In this example the classification can either be RUN or WALK. The predicate of the example, enables the flow to task send if the result is RUN, or to task log if the result is WALK.

Timely data constraints add three possible time related constraints to flows: *expires*, *Minimum Inter Sample Delay (MISD)* and *collect*. *Expires* disables a dependency if the flowing data is older than a given value; *MISD* allows the developer to specify the minimum length of the time interval between two consecutive data flowing through the same dependency; *collect* allows the specification of the size of the window of data that must be processed in batch by the downstream task.

Since tasks cannot alter system or non-volatile memory, Mayfly avoids consistency issues associated with mixed memory volatility systems [✓ C2T].

Thanks to task’s code and flow metadata the Java Mayfly compiler builds a firmware that:

- persists on *NVM* task data output with timestamps;
- takes care of timestamps update after power failures;
- builds a task graph and *statically* schedules tasks in accordance with flows and conditions.

As in our solution, in Mayfly timestamped data are persisted on *NVM* with a double buffer technique, so that old data are never overwritten to prevent data corruption in presence of power failures during write operations. Moreover, tasks have no access to *NVM* and their result is persisted only on return.

The Mayfly runtime loop selects the next task accordingly to the static schedule, checks if all the incoming flows are enabled in accordance to the conditions and time constraints, and executes the task.

During the compilation, Mayfly does not emit any energy related warning to help the developer to understand which tasks exceeds the maximum amount of energy that can be buffered by the device [✗ C1T].

Given that the task schedule is built at compile time, unlike our solution, Mayfly firmware can not address variation of energy provisioning. The next task selection does not take in consideration the amount of current buffered energy and a persistent drop in the amount of harvested energy can lead to continuous reboot and re-execution of the same, statically selected next task, leading to the starvation of the rest of the application [✗ G1T]. On the contrary, our solution builds a *dynamic* schedule that leverages real time energy state as an input of the next task selection, reacting to energy shortage so that the device can guarantee a minimum set of functionality. Moreover, in our proposed solution, we enrich the semantics of data dependencies taking into consideration different producer and consumer patterns.

InK

InK is a task based solution proposed by Yıldırım et al. [35]. It is the first attempt to introduce dynamic scheduling in *TPC* as it offers a solution that reacts to runtime events. In InK tasks are collected in sequences called task threads. Each thread is then linked to specific runtime events that trigger their execution. Each thread can be decorated with a priority.

The execution of threads is atomic at task level, and partial results are committed at the end of each task within the thread. As Alpaca,

also InK privatizes task variables before task execution, guaranteeing the consistency of the internal state [✓ C2T].

InK supports three kinds of runtime events: events related to energy threshold, events related to timers [✓ G2T], implemented thanks to the hardware solutions described in [17], and general hardware interrupts.

The InK firmware includes a scheduler that operates as follows. At each scheduling loop iteration, the scheduler

1. selects the enabled thread with the highest priority;
2. selects the next task within that thread;
3. initializes the local private version of the task variables as in Alpaca;
4. executes the task;
5. commits the task modifications;
6. suspends the thread and re-enters the loop.

Each thread has a non volatile event queue, implemented as a circular buffer. *ISR* corresponding to the aforementioned events, add the event data in the event queue of the corresponding thread, the thread execution is enabled and the interrupted task continues its execution. At the next scheduling loop a new thread will be available for scheduling and the execution, depending on the priorities, may be diverted to the newly enabled thread. The *ISR* only performs the aforementioned operations, as the actual event processing is executed by a corresponding thread.

While this implements a solution that responds to real-time events, it is not a dynamic scheduling solution, since the conditions that trigger these events are statically set at design time. Moreover, we argue that the way in which interrupts are served could be misleading. In fact the solution proposed in InK introduces a potential delay of unbound length, between the actual event occurrence and the instant at which the code related to the event is executed. Let us imagine that an energy event happens right after the first instruction of the currently running task. The event is added to the corresponding event queue and the execution of the task is resumed. The actual response to the event only happens after the complete execution of the currently running task, as threads can be preempted only at task granularity. This potentially invalidates the precondition to the execution of the thread linked to the energy event. In fact, the rest of the task execution consumes energy and the energy value that triggered the energy event may be invalid. This forces the developer to check if the event condition still holds at the beginning of the thread, ultimately resulting in a polling mechanism, instead of a real interrupts and events model.

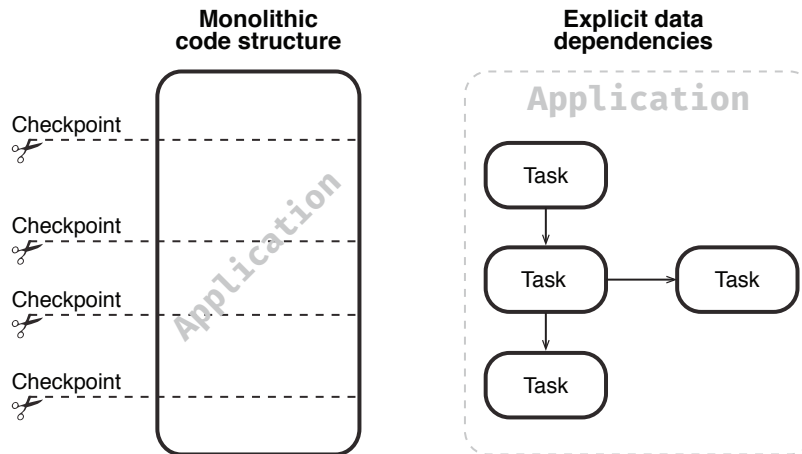


Figure 3.2: Checkpoints and task decomposition are two different abstractions and they promote two different code structures. While checkpointing slices a standard monolithic code, task decomposition pushes the developers to reason about their application in terms of data dependencies between functions.

The developer can prevent the execution of a task when the energy budget is below its requirement, by properly setting the voltage threshold that triggers the interrupt activating the task. Still this operation requires user intervention [✍️ G1T].

InK does not support correct task decomposition, by signaling tasks exceeding the maximum amount of energy that can be buffered by the device [✖️ C1T].

The concept of tasks grouping in thread is similar to what we propose with the concept of applications.

3.3 WHY TWO SOLUTIONS TO THE SAME PROBLEM?

Numerous solutions have been proposed in literature on how to ensure forward execution and data consistency on systems that experience frequent power outages. They either rely on checkpointing, or on explicit decomposition in atomic tasks, potentially providing a way to specify data dependencies between tasks like in Mayfly.

One may argue that at its core checkpointing and task-based solutions are in fact the same thing, or even that tasks are just syntactic sugar on top of standard checkpoints. To be fair this is partially true, but only if we stand at a low level of abstraction. No matter of the specific implementation, ultimately task based solutions rely on mechanism to save values and restore them when needed, resembling checkpointing. DINO and its boundaries are an example on how thin is the border between the two worlds.

Transiently powered devices are usually embedded systems that often times deal with sensing activities. Applications read data coming from different sensors and these data are usually processed to extract

informations. Let us suppose to deploy a health care application, on a wearable device. The device is equipped with a heart rate monitor, a sensor to measure electrodermal activity and a motion sensor. Each one of these sensors may require a different amount of energy to sample data, moreover these data usually need to be filtered and processed to extract some features. The final outcome of the application depends on data coming from different sources, each one with different requirements and processing techniques. We may add sophistication to this scenario by including actuators that perform activities on the outer world based on application results, again with different energy requirements and different semantic based on the actuator.

The aforementioned description suggests a model in which data are produced by a sensor, traverse a series of transformations, and are ultimately reduced to produce one or more outputs. This data pipe can be intuitively obtained dividing the application logic into simpler, possibly stateless, functions, connected together by means of data dependencies. In this abstraction each functional component ingests some data, and produces a result that can be consumed by others.

Usually in this scenario data are relevant within a given interval. Processing stale data produce wrong or not relevant outputs, and therefore a waste of energy. Managing time sensitive data on batteryless devices is a difficult task by itself, as already discussed in Section 2.3, managing conditional execution of portion of code, based on time conditions with standard checkpoint adds additional complexity to the code. Structuring code as a set of tasks forces the developer to think in terms of data dependencies.

Ultimately we may want to sample different data at different rates, maybe giving different priorities to outputs. Once again tasks and data pipes are better suited to easily describe and support this feature.

Since tasks are supposed to run atomically, their worst-case energy consumption must be compatible with the amount of energy that can be stored in the capacitor. This requires an effort by the developer that must be aware of energy related issues. We think that this is necessary in such a constrained scenario.

Our solution proposes a programming abstraction guided by these aforementioned principles, centered around data dependencies. Our goal is to provide to the developer a model in which she can easily describe the non functional requirements presented so far, together with an execution environment that hides the complexity of energy management and problems related to transiently powered devices, while keeping her aware of energy constraints.

Tasks may be seen indeed as syntactic sugar on checkpoints, but we think that they promote an abstraction that is better suited to meet the specific needs of developers dealing with this family of devices.

 SETTING THE THRESHOLD

Transiently powered devices harvest energy from a wide variety of sources, each with different characteristics. Some of these energy sources, such as wind energy harvester, can provide large amounts of power in short burst; some other, like small photovoltaic cells operating from indoor light, steadily provide small amounts of power; energy harvested from RFID readers is subject to voltage fluctuations that are highly dependent on the operating environment and device's physical orientation [29].

This energy is buffered in decoupling capacitors, placed in parallel with the harvester. The MCU activates at a given threshold V_{on} and the computation continues until the turn off threshold V_{off} is reached. This deactivation threshold V_{off} is usually set to the minimum voltage required by the considered platform, for instance TI MSP430 requires 1.88V to sustain computation [27]. V_{on} must be strictly higher than V_{off} : if $V_{on} = V_{off}$ the board would not be able to perform any computation when the power intake is lower than the outtake, neutralizing the effect of an energy buffer.

Figure 4.1, taken from Flicker [14] schematics, shows a typical harvester and power management configuration, where the decoupling capacitance C1 buffers harvested energy, and the voltage regulator U1 sets the thresholds.

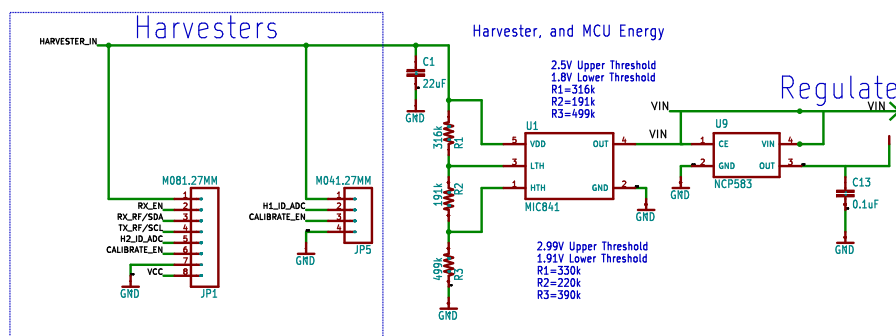


Figure 4.1: This figure presents the portion of Flicker [14] board in charge of power management. In particular we see a decoupling capacitor C1 of 22 μ F that serves as energy buffer, and a voltage regulator U1 that, depending on the value of resistances R1, R2 and R3 sets the on and off threshold.

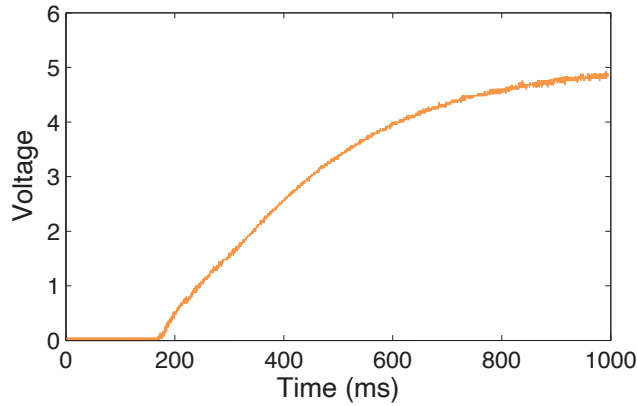


Figure 4.2: Capacitor voltage over time. A $10\mu\text{F}$ capacitor in an Intel WISP node [31] charges with energy harvested by an RF antenna when the source is at a 2m distance.

Figure 2.1 in Chapter 2 shows a typical intermittent execution pattern. The time interval between a deactivation and the next activation depends on different parameters: the profile of the energy source, the capacitor charging characteristics and, of course, the value of V_{on} .

The high variability of energy sources makes any prediction on the expected energy intake unreliable. Still, the turn on threshold V_{on} is a powerful knob that can significantly impact the system's behavior. In fact, given a certain energy source, rising the activation threshold results in a higher amount of stored energy, and therefore in longer execution time, but requires a longer charging time and therefore a longer period of inactivity. Unfortunately, buffering a large amount of energy to deal with potential shortage, increase more than linearly the length of the period of inactivity, due to capacitors characteristic [6].

Figure 4.2 shows a typical charging curve for a capacitor. Capacitor charging profiles are not linear: as the capacitor voltage approaches the voltage supplied by the harvester, the charging current decreases to zero, resulting in a faster charging phase when the capacitor voltage is lower.

As we discussed in Chapter 3, in a task-based model we need to ensure the transactional nature of tasks. Therefore, the result of completed task must be durable, hence we must save tasks outputs on **NVM** to retain them across power failures. This means that we can not avoid an **NVM** write after each task. On the other hand, we can reduce the overhead by allowing tasks to read their inputs from volatile main memory. Still, after each power failure, we need to restore these data from **NVM** to main memory.

Decreasing V_{on} decreases the time needed to charge the capacitor, but increases the rate of on-off cycles. As we said, the transactional nature of tasks does not allow us to reduce **NVM** writes, but an increase in on-off rate cycles results in a higher number of **NVM** read accesses for state restore, as we need to read persisted data to properly reboot

the board after each power failure. These **NVM** accesses are costly operations in terms of energy, and solutions like HarvOS [5], presented in Section 3.1.1, actively try to reduce the size of persisted data, to reduce the **NVM** access overhead. We can positively affect the execution performance, by reducing the frequency of these **NVM** read, thanks to a proper selection of the activation threshold.

Increasing V_{on} increases the off time, decreasing it, increases the energy wasted to restore the state after a power failure, and this makes the quest to reach a correct balance a particularly challenging task. The road to the perfect V_{on} selection is further endangered by an elusive feature of this class of devices, as presented by Ahmed et al. in [1] and in the next Section of this document.

In Section 4.1 and 4.2 we discuss the challenges connected to the selection of the activation threshold; in Section 4.3 we present an overview on how existing solutions tackle the problem of threshold selection; in Section 4.4, given the results presented in previous Sections, we propose a new take on this problem.

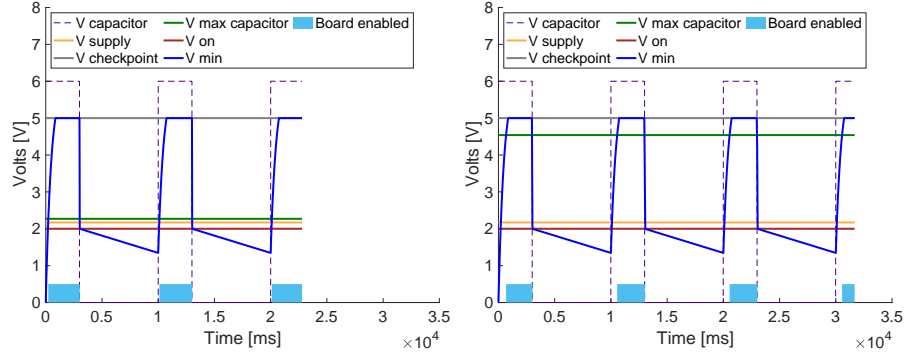
4.1 THE CONUNDRUM OF THRESHOLD SELECTION

As we discussed in previous Sections, **TPC** relies on a great variety of energy harvesting mechanisms, often characterized by significantly different performance and unpredictable dynamics across space and time [1, 4]. Given that our ultimate goal is to leverage **TPC** as an enabling technology for a battery-less Internet of Things, and finally realize our “Smart Dust” vision, we must support this high variance in terms of energy sources.

As presented in Chapter 3, programmers may alternatively rely on checkpoint based mechanisms or on task-based programming abstractions. The selection of V_{on} affects the performance of both these approaches.

For instance let us consider Hibernus [3], a checkpoint based system presented in Section 3.1.2. In Hibernus, an interrupt is generated every time the capacitor voltage drops below a certain threshold V_H , this interrupt triggers a checkpoint. After every checkpoint, the board enters in low power mode and sleeps until the voltage reaches a second threshold V_R , higher than the checkpoint one, implementing an hysteresis mechanism. Once the capacitor voltage rises up to this second threshold, the board exits from sleep mode. On wakeup the checkpoint is restored, if needed, and the computation continues. The checkpoint is restored if the voltage of the capacitor dropped below the minimum value needed to operate, during the sleeping interval.

To better understand the effect of V_{on} selection on the system performance, we simulate Hibernus running on a board with a decoupling capacitor of $16\mu\text{F}$. In each simulation we run 10 iterations of an activity recognition kernel, powering the board with a square wave voltage



(a) Setting $V_{on} = 2.27V$ results in an execution time of 22.787ms (b) Setting $V_{on} = 4.54V$ results in an execution time of 31.672ms

Figure 4.3: Simulation of Hibernus to understand how different selections of the activation threshold affect system’s performance. The simulation consists in the execution of 10 iterations of an activity recognition kernel, on a board with a decoupling capacitor of $16\mu F$, minimum voltage to support computation $V_{min} = 2V$, $V_H = 2.17V$, maximum capacitor voltage $V_{max} = 5V$, board frequency 8Mhz, with a constant energy consumption of $1nJ$ per clock cycle.

trace, with a period of 10s and duty cycle of 30%, and by varying the activation threshold V_{on} . A lower threshold results in a lower execution time to execute the same number of iterations of the kernel. By halving V_{on} from 4.54V to 2.27V we obtain an execution time that is 1.39 times faster: from 31.672ms to 22.787ms. Still the activation threshold V_{on} can not be simply lowered as much as possible, in fact this would result in an increased number of checkpoints and therefore in a higher computation and energy overhead. Figure 4.3 presents the results of the simulation.

We would face the same issue when dealing with task based systems. As we discussed in Section 3.2, tasks are atomic pieces of computation. This means that they must be fully executed within an execution burst, without being interrupted by a power failure, otherwise they must be re executed completely. Moreover, their output must be durable, so we need to perform an NVM write after each completion to retain the output. Existent task-based solutions does not allow to dynamically change the activation threshold. For this reason, they must ensure to buffer enough energy to sustain the computation of the task with the highest energy consumption, hence the threshold must be set to a value that corresponds to a buffered energy that is higher or equal to that maximum energy request. A proper selection of the activation threshold can reduce the number of NVM read accesses. In fact, the system must restore significant data every time a power failure occurs between two tasks with a data dependency, resulting in an increased overhead given the numerous NVM reading accesses.

Let us suppose that two tasks, T1 and T2, are connected by a data dependency: T1 produces some data, and T2 consumes these data to perform its computation. Let us suppose that T2 is the task with the highest energy consumption. If a power failure occurs after the execution of the first task, we would need to access NVM to restore its output, for the execution of T2. If we set the activation threshold to a voltage that exactly fits T2, but not the execution of both tasks, we would need to restore T1 data to properly execute T2, as they are connected by a data dependency and the second task needs data coming from the first one. On the other hand if we set the threshold to accommodate both the tasks, we would not need to restore data between the execution of the producer T1 and the consumer T2, but buffering this higher amount of energy would require a longer charging interval. Figure 4.4 presents a description of this aforementioned scenario.

Once again, increasing the threshold results in a lower overhead due to NVM accesses, but as we previously discussed, reducing it results in a lower execution time.

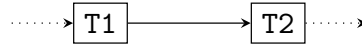
We argued that reducing the threshold produces a shorter execution time, while increasing it results in a longer time needed to reach that voltage. But how much shorter and how much longer? To properly discuss threshold selection we would want to quantify this gain in execution time, or the length of the time spent in buffering energy. Unfortunately, these quantities can not be computed once and for all.

A temporary increase in the amount of harvested energy causes an increase of the charging curve slope, while a temporary slowdown on energy intake results in an increased charging time, and therefore in an increased inactivity interval. These changes can not be predicted, as aim to support instantaneous variation on energy sources. But this is not the only reason why we can not easily quantify these numbers.

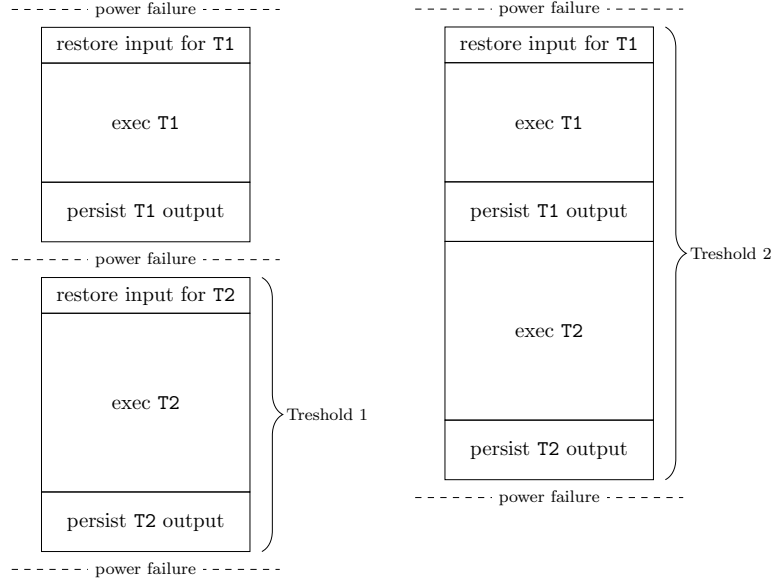
The decoupling capacitors, due to the unpredictable nature of the energy source, may discharge and recharge several times during a single application run. We call *power cycle* the time interval needed to discharge the capacitor from a full charge, down to the minimum operating voltage. A capacitor may end up performing 17 power cycles during a single CRC code run, when powered by a RF harvester [29, 1]. Therefore the MCU's operating voltage may rapidly vary several times during a single execution of straightforward algorithms. This causes important side effects on MCU's performance, as we discuss in the next Section.

4.2 EPIC RESULTS

Ahmed et al. [1] present in their work the results of experiments conducted on a TI MSP430G2553 board. They noticed that, when running at 1Mhz, the power consumption reduces in a single power cycle by a factor up to 363.36%, while the clock speed increases by a



- (a) Task T1 and task T2 are connected by a data dependency. T2 needs data produced by T1 to perform its computation.



- (b) The activation threshold V_{on} is set to a value that corresponds to a buffered energy equal to the energy consumption of task T2.

- (c) The activation threshold V_{on} is set to a value that corresponds to a buffered energy equal to sum of the energy consumption of task T1 and T2.

Figure 4.4: The selection of the activation threshold affects the performance of task based systems. Increasing the threshold minimizes the number of **NVM** accesses needed to restore persisted data after a power failure, hence minimizing the system overhead.

factor of up to 3.42%. This means that, without changing the nominal frequency, the same instruction takes different times depending on the supply voltage at the time it is executed. Moreover, these experiments showed that a single clock cycle consumes 1.59nJ when the capacitor voltage is 3.6V, and 0.33nJ when the voltage is 1.8V. This happens regardless of the system load and the software has no control on it [1]. Figure 4.6, taken from [1], presents the aforementioned results.

We can not easily predict the length of the charging phase due to energy source fluctuations, and we can not easily predict either the execution time, or the exact energy consumption of a fragment of code.

Ultimately the total time needed to run an application on a **TPC** board T_{tot} , is the sum of T_{charge} : the time needed to charge the capacitor up to V_{on} , and the time to actually execute the code T_{exec} .

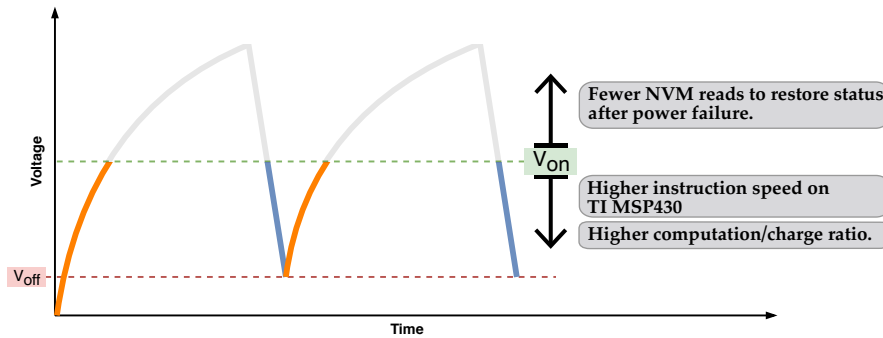


Figure 4.5: Threshold selection must be the result of a refinement process, as both increasing and lowering it have pros and cons.

T_{charge} depends on the current energy profile E_{profile} and on the current capacitor voltage, as the charging phase is not linear. Without loss of generality, we can say that T_{exec} depends on instructions count and on clock speed that is affected by the contingent voltage [1]. The voltage drops because of the energy drain E_{tot} from the capacitor to execute instructions, this energy obviously depends on the energy needed to execute a single clock cycle E_{CC} , that, again, depends on the contingent voltage [1]. This results in the cycle of dependencies showed in Figure 4.7 and in the difficulty to produce a simple model that predicts execution time or exact energy consumption, and therefore to exactly quantify the benefit of a specific threshold selection.

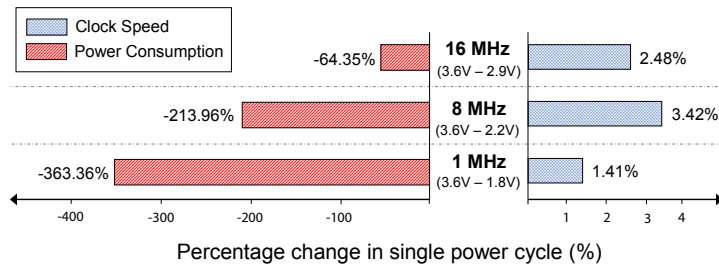
In conclusion, we saw that the activation threshold affects system performance, yet its selection is complex, and reasoning at compile time on the correct value is exceedingly difficult as many of the variable involved in the process can not be predicted. Therefore, we better address the threshold selection conundrum at runtime, implementing a refinement process that hopefully converges to a proper value. In Section 4.3 we present an overview on how existent solutions refine the activation threshold at runtime.

4.3 OVERVIEW OF THRESHOLD MANAGEMENT SOLUTIONS

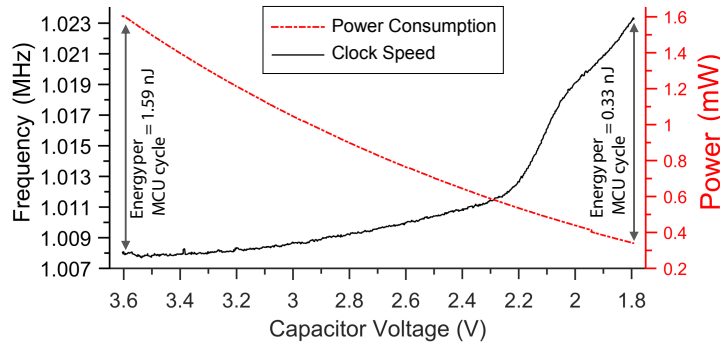
As we discussed so far, the difficulties of threshold selection suggest a runtime refinement process. In this Section we present an overview on how different existent solutions tackle this issue.

Hibernus++

As previously discussed in Section 3.1.2, Hibernus [3] performance depend on the correct setting of two voltage thresholds: V_H that is the voltage level that triggers the hibernation stage, and V_{on} that is the threshold at which the computation resumes after the hibernation. Ideally the system should hibernate at the last possible moment before



(a) Voltage supply variation impact both power consumption and clock speed on TI MSP430G2553, in a single power cycle. The effects are different depending on the selected frequency.



(b) Impact of supply voltage variation on TI MSP430G2553 power consumption and clock speed. Energy consumption is a product of power and execution time, which is a function of clock speed. This graph shows that the energy cost of a single MCU cycle varies by up to $\approx 5\times$ depending on the instantaneous supply voltage [1]

Figure 4.6: Voltage supply has an impact on both power consumption and clock speed. Both graphs are taken from [1].

supply failure and resume at the earliest optimal point, but, once again, the correct balance of these thresholds is a complex process that requires a runtime refinement process. A non optimal selection of V_H would result in either an increased overhead due to too many restores after power failures, or in wasted energy, in case the hibernation happens too early. The same goes for V_{on} , in fact a wrongful selection would result in too many hibernations, or in wasted time to buffer too much energy. To support the selection of these parameter, Balsamo et al. introduced Hibernus++ [2].

Hibernus++ adds two components to standard Hibernus: a self-calibration procedure that selects and refine V_H , and a supply test that results in the selection of V_{on} . The self-calibration routine is an iterative process that is executed at the first boot. It waits for the supply voltage to reach a calibration threshold V_{cal} , initially set to a default value. Once this voltage is reached, the energy harvester is disconnected so that the board is powered only by the decoupling capacitor. At that point the routine saves a test checkpoint to *NVM*, then reads the capacitor voltage V_{meas} and reconnects the harvester.

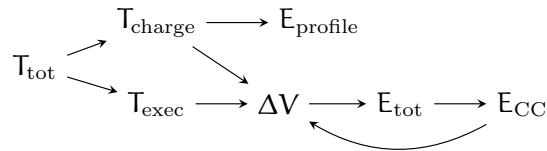


Figure 4.7: Dependencies among quantities involved in the computation of execution time of a piece of code in TPC. In particular the cyclic dependencies render exceedingly difficult the description of a simple closed model to link together these parameters.

The value $V_{\text{cal}} - V_{\text{meas}}$ gives the voltage drop due to hibernation. At the end of a successful calibration the hibernation threshold V_H is set to $V_{\text{min}} + (V_{\text{cal}} - V_{\text{meas}})$, where V_{min} is the minimum operating voltage. The initial value of V_{cal} is set as low as possible, equal to the nominal voltage level that corresponds to MCU on. In case a power failure occurs during this process, V_{cal} is increased and the routine restarts.

The supply test routine sets the activation threshold V_{on} . This routine is executed at the end of the calibration procedure and after every power failure. As we previously discussed, the energy source's profile is deeply connected to the selection of the activation threshold. For this reason, with the supply test process, Hibernus++ tries to classify the sources in two categories and performs different operations depending on the class.

The test starts at a known voltage level V_{test} . It determines whether the energy source is able to supply enough energy to sustain the operation of the MCU or not, by executing a short segment of code. In particular the voltage V_{check} is read at the end of the code execution, if $V_{\text{check}} \geq V_{\text{test}}$ the source sustained the computation and it is classified as *high-power*, otherwise the harvester is unable to supply enough power and the source is classified as *low-power*. In presence of a high-power, source the system restores immediately the computation to take advantage of the abundant power. If the source is classified as low-power the system tries to buffer as much energy as possible to prevent repeated cycling between hibernation and restore. To do this, the system relies on two interrupts: one that detects increasing capacitor voltage, and a timer that acts as time-out. As long as the voltage is increasing the system continues to reset the timer; if the voltage stops increasing and keeps its level throughout the timer, the test stops and the system restores the computation, since there is no benefit in further waiting.

Figure 4.8 shows the flowcharts of the calibration and test routines.

Dewdrop

Dewdrop is a task based solution proposed by Buettner et al. [6]. It is specifically tailored to Computational RFID (CRFID) and, unlike other task based solutions presented in Section 3.2, multiple tasks

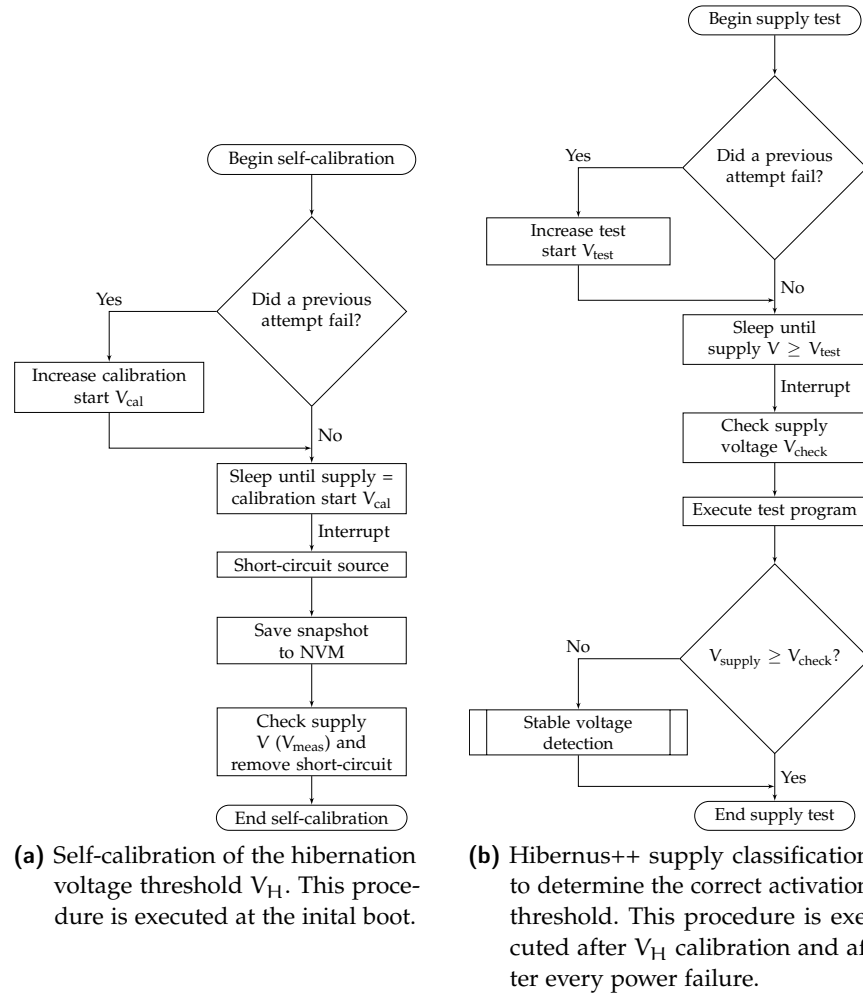


Figure 4.8: Flowcharts of the routines to calibrate hibernation 4.8a and activation thresholds 4.8b in Hibernus++ [2].

are isolated as they do not share any data. The solution has been developed on Intel WISP platform [31].

In Dewdrop task selection depends on messages coming from the RF antenna. The selected task starts as soon as the capacitor voltage reaches a given threshold. This threshold is refined iteratively by applying a heuristic.

In particular, as discussed so far, delaying task's start, at a higher energy level, and therefore at a higher voltage, decreases the time wasted due to task failing, but increases the time wasted in overcharging the capacitor. In fact, given the atomic semantic of a task, it must be executed from scratch every time it is interrupted by a power failure, and all the energy spent for the partial execution is wasted. Moreover the exact power consumption of a task is not constant, as different tasks have different requirements that may vary depending on inputs.

To reach the correct balance, Dewdrop relies on a probabilistic model. Let:

$P(\text{FAIL}|V_{\text{on}})$ be the probability that the selected task will fail, given the activation threshold V_{on} ;

t_{UNDER} be the time to charge back to V_{on} after a failure;

t_{OVER} be the time spent charging beyond the energy level that would have been sufficient to execute the task.

Then the wasted time t_{wasted} is represented by the following equation:

$$t_{\text{wasted}}(V_{\text{on}}) = P(\text{fail} | V_{\text{on}})t_{\text{under}} + (1 - P(\text{fail} | V_{\text{on}}))t_{\text{over}} \quad (4.1)$$

An exhaustive exploration to find the value of V_{on} that minimizes the wasted time defined in Equation 4.1 would be impractical. In fact the device would need a large series of execution attempts, moreover the analysis would need to be repeated periodically as the energy source may vary through time.

To avoid this extensive search Dewdrop implements a refinement procedure to find an approximate solution. Let P_f be the failure rate at a given default V_{on} ; T_{under} be $P_f \cdot t_{\text{under}}$, and T_{over} be $P_f \cdot t_{\text{over}}$. If $T_{\text{over}} \gg T_{\text{under}}$, then the selection is too conservative and the task's execution has been delayed too much, as the capacitor spent time to buffer more energy than what needed to execute the task. Otherwise if $T_{\text{over}} \ll T_{\text{under}}$, then tasks are failing too often and the activation threshold should be increased.

Dewdrop uses this heuristic to find the balance point by slowly updating V_{on} . In particular the system maintains two estimates of T_{under} and T_{over} , updating them with an exponentially weighted moving average, after each task complete or partial execution. The activation threshold is then updated depending on these estimates, adjusted by β in the appropriate direction.

Let:

V_e be the voltage at the end of a running task;

V_0 be the minimum MCU operating voltage;

ε be a small voltage.

A task successfully completes if and only if $V_e \geq V_0 + \varepsilon$. Equations 4.2 detail the process to update the activation thresholds.

$$\begin{aligned} T_{\text{over}} &= \begin{cases} (1 - \alpha)T_{\text{over}} + \alpha t_{\text{over}}, & \text{if } V_e \geq V_0 + \varepsilon \\ (1 - \alpha)T_{\text{over}}, & \text{if } V_e < V_0 + \varepsilon \end{cases} \\ T_{\text{under}} &= \begin{cases} (1 - \alpha)T_{\text{under}}, & \text{if } V_e \geq V_0 + \varepsilon \\ (1 - \alpha)T_{\text{under}} + \alpha t_{\text{under}}, & \text{if } V_e < V_0 + \varepsilon \end{cases} \\ V_{\text{on}} &= \begin{cases} V_{\text{on}} - \beta, & \text{if } T_{\text{over}} > T_{\text{under}} \\ V_{\text{on}} + \beta, & \text{if } T_{\text{under}} > T_{\text{over}} \end{cases} \end{aligned} \quad (4.2)$$

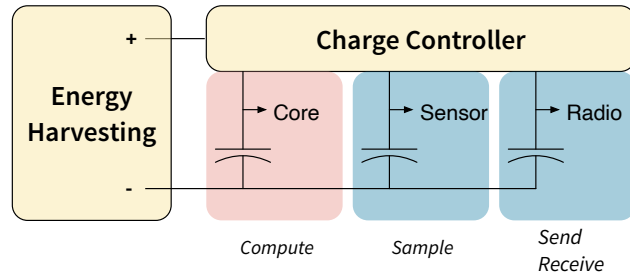


Figure 4.9: A conceptual view of Flicker federated energy storage, taken from [14]. Energy storage is separated per peripheral, and each peripheral maps to a set of sensing tasks.

Dewdrop solution is specifically tailored on WISP [CRFID](#), that currently does not feature a voltage supervisors that can trigger an interrupt and start the execution at the selected V_{on} [6]. This means that the voltage level must be checked using a software polling approach. With Dewdrop the board sleeps while energy is being harvested and samples the voltage occasionally. To reduce the energy impact of polling, the sampling rate is exponentially adapted, applying the Equation 4.3, where V is the measured voltage and V_r is the voltage gained since the last woke up. The polling rate is lowered when the capacitor voltage is increasing rapidly, so that the device does not miss the opportunity to execute tasks. Equation 4.3 shows the update process of the polling rate.

$$t_{next} = \begin{cases} 2t, & \text{if } V_{on} - V > 2V_r \\ t/2, & \text{if } V_{on} - V < V_r/2 \\ t, & \text{otherwise} \end{cases} \quad (4.3)$$

Flicker

Flicker [14], proposes a hardware solution to threshold management through reconfigurable federated energy management.

This approach assigns a dedicated capacitor for every hardware peripheral. Each capacitor is sized so that it can buffer enough energy for a single task using the corresponding hardware peripheral.

Thanks to this hardware support it is possible to set a different threshold for each task, in order to prevent the starvation of low energy tasks when a task with high energy request keeps failing. Moreover it is possible to use smaller capacitors, which charge faster. In fact each component only needs to harvest the energy it needs, not the energy to support all components. Figure 4.9, taken from [14] presents a conceptual view of federated energy management.

The programmer can assign the activation threshold and the priority of charging for each capacitor, and therefore for all the tasks that are linked to that capacitor.

As shown in Figure 4.9, the energy from the harvester is stored at first stage in a core capacitor. Then a charge controller pours energy from the core capacitor, to peripheral capacitors at different thresholds, hence introducing a priority. Finally an interrupt is generated when a capacitor reaches a given voltage, and that interrupt informs the MCU that a task can be executed.

4.4 SHIFTING PERSPECTIVE IN THRESHOLD MANAGEMENT

Threshold management is mostly an unexplored territory, in particular in presence of task based solutions. If Hibernus++ proposes an efficient routine to calibrate thresholds with checkpoint systems, Dewdrop is specifically tailored to the CRFID scenario and the knowledge on the nature of the energy source allows to make strong assumptions that are not valid for an arbitrary source, hence it can not be seen as a general solution for tasks systems. Flicker federated energy management offers a promising platform, but does not define any software stack to select, manage and refine the thresholds, as it is outside of its scope.

We still lack an efficient solution to manage activation threshold on task-based systems, that is independent from both the platform and the profile of the energy source.

The main focus when dealing with batteryless devices is to structure code and support its execution in a way that ensures its completion. Current solutions address this issue by trying to maximize energy efficiency and, as we discussed so far, the correct selection of voltage thresholds is a fundamental step toward this efficiency.

The development of a static system, or, in other words, a system that is not able to dynamically adapt to the current energy intake, forces to push unconditionally toward that energy efficiency maximization. In fact such system must be robust and behave well in the worst case scenario of scarce energy.

We argue that the ultimate goal when developing software stack for TPC, though, is not energy efficiency per se, instead efficiency is a way to minimize the overhead and increase code execution to maximize the throughput.

If we develop a system that is able to accept minimum throughput requirements, we can direct our attention to the satisfaction of that parameter, which we argue that is the ultimate goal, instead of indirectly reaching it by looking for the maximum energy efficiency. Tasks are a better fit for this view, as their atomicity makes it easier to define a unit of computation and therefore a throughput as a number of iteration over time.

To support this vision we need:

1. an efficient way to describe tasks, their connections and to formulate throughput requirements;
2. a way to select tasks that is aware of the current energy scenario;
3. a way to manage thresholds accordingly.

If we do so we can implement a dynamic solution: one that is able to address a sudden peak in harvested energy, while being able to support execution with scarce energy intake, always addressing the throughputs requests.

Moreover, as it emerges from the overview on existing task-based solutions presented in Section 3.2.2, and summarized in Table 3.2, none of the existent solutions autonomously responds to runtime variation of the energy budget when selecting which task to run, in order to make the best possible use of such a scarce resources.

Let us consider a device that senses data from an accelerometer and recognizes the activity based on these data. We can easily imagine a structure with multiple tasks: at least one to sense data and one to produce the classification. In this scenario the developer should be able to ask for a minimum throughput for the entire process, without being forced to consider the throughput for each single task. Therefore, a system that accepts throughput requirements, should let the developer ask for a given throughput for higher level functions that encompass the execution of several tasks. Let us call *application* this structure that groups tasks to execute a higher level function.

Of course, different applications can have different energy requirements that derives from the requirements of their tasks. Let us suppose that a single application is running on a device, and that the energy harvested from the source is high enough to allow a computation without power failures. In this scenario the device would keep executing several times the single application, potentially exceeding the throughput request. We argued that the ultimate goal of a developer is to reach a given throughput. Executing an application with a throughput higher than its request, can be seen as a waste of energy. In fact, the system could make a better use of the energy that is consuming to run this excess of iterations, to execute additional workload. Still, with just one application there is no additional workload. For this reason we can envision a multi tenant system, where multiple applications provide additional workload to make a better use of the energy in case of high energy provisioning.

In the rest of this document we present a new task-based framework whose goal is threefold:

1. propose a *new programming abstraction*, based on tasks, that are grouped in applications, with explicit data dependencies among

them, and explicit non functional requirements in terms of minimum desired throughput per application;

2. thanks to this new way to describe software for [TPC](#), build a *dynamic scheduler* that enables multi tenancy, is energy aware and reacts to changes in the amount of harvested energy;
3. build an *adaptive threshold management system* that adapts thresholds based on the scheduler decisions.

We introduce our proposal starting from Chapter 5, where we describe this new programming abstraction, in which tasks are connected among them by means of explicit data dependencies, and are grouped in applications.

5

ENABLING MULTITENANCY

As discussed in Section 4.4, our first goal is to propose a new programming abstraction, based on tasks, with explicit data dependencies.

Tasks, are the basic blocks of our new model. As discussed in Section 3.2, they are atomic unit of computation and their execution follows a transactional semantic. As we said in Section 4.4, we group them in applications, a concept similar to InK [35] task threads, presented in Section 3.2.2.

Applications are a collection of tasks, necessary to provide a higher level service. For instance, a task that senses temperature may be included in an application that monitors cold chain equipment. Applications support the definition of non functional requirements in terms of minimum desired throughput.

This infrastructure let multiple applications to coexist on the same device, enabling multitенancy. The scheduler can dynamically change the workload, enabling and disabling applications, to react to changes in provided energy. In presence of a peak in energy supply, the number of active applications is increased to capitalize a higher amount of energy. With a single application, the only effect would be an increased throughput, that potentially exceeds the requests. On the other hand, thanks to multitенancy, we can use the increased energy to run additional applications and expand the capability of our device. Moreover, given an application, our system can select from a pool of available tasks the one that fits best the current energy budget, discarding those that can not complete due to energy limitations. We will discuss how the system manages applications and schedules tasks in Chapter 6.

In this Chapter, we begin the description of our new proposal, starting from the programming abstraction. In particular in Section 5.1 we present a formal introduction to the new programming abstraction, in Section 5.2 we provide a detailed description of tasks and applications, in Section 5.3 we investigate data dependencies nature and issues, finally in Section 5.4 we present the memory model associated to the proposed abstraction.

5.1 FUNDAMENTAL CONCEPTS

Formally the scheduler works in an environment where code is organized according to the following list of elements.

A set of tasks T

Each element $\tau \in T$ is a piece of atomic computation, whose execution follows a transactional semantics.

A set $IN(\tau)$

It represents the set of input data for a given task τ .

A set $OUT(\tau)$

It represents the set of output data of a given task τ .

A set of applications A

Each application $a \in A$ is a set of tasks and $A \subset \mathcal{P}(T)$.

A dependency function

$Dep(\tau) : T \rightarrow \mathcal{P}(T)$, such that $\tau' \in Dep(\tau) \iff IN(\tau') \subseteq OUT(\tau)$. In other words, a task τ' is in the dependency set of a task τ if τ produces some output that is relevant to the execution of τ' . The transitive closure of this function must be non reflexive.

We present a detailed description of data dependencies in Section 5.3.

These previous elements can be used to organize the codebase in a partitioned Direct Acyclic Graph (DAG), or a forest of partitioned DAG, in case one or more applications do not have inter applications dependencies. In this graph tasks are nodes, while the edges are couples in $\tau \times Dep(\tau)$. Figure 5.1 shows an example of this structure. In the following Section we introduce the basic blocks of our new programming abstraction: tasks and applications.

5.2 TASKS AND APPLICATIONS

Tasks are pieces of atomic computation: either their code is completely executed within a computation burst, or they are entirely rescheduled for execution in a further power cycle, in other words the system does not checkpoint intermediate results within a task.

Applications are a collections of tasks, grouped to provide a higher level service, for instance an application may include tasks needed to provide proper irrigation in a greenhouse, thanks to the analysis of humidity and solar radiation sensors, another application deployed on the same device may include tasks to actuate the ventilation based on the greenhouse temperature.

Within an application, each single task is responsible for a subset of functions, varying from reading sensor data, to data processing or

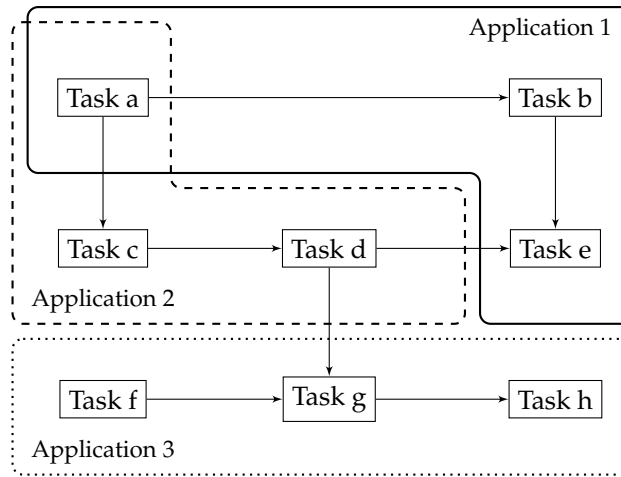


Figure 5.1: Example of a partitioned DAG induced by the definition of the framework elements.

actuators management. These tasks may be connected by means of data dependencies as shown in Section 5.3.

A task may be part of multiple applications since its specific function may be necessary to provide more than one higher level service. For instance, a task that analyzes the humidity may be useful both to provide a sensible irrigation service, and to deploy a parasites prevention service.

5.2.1 Tasks

A task is characterized by the following elements:

- a unique identifier;
- a set of required input data IN ;
- a code segment;
- a data output OUT ;
- one or more exit points within the code segment;
- a prediction on worst case task's energy consumption E_{wc} ;

To preserve memory consistency in presence of power failures, all input and output accesses are mediated by the framework that provides the requested input data, and writes task's output in a way that ensures the consistency of the internal state. In particular a task *can not* directly output data to *NVM* during its execution, but it can only do that through the runtime support, providing the output variable as a parameter to its exit point. This operation is performed by an instruction with the same semantic of C return statement. Section 5.4 presents how the framework manages *NVM* on behalf of tasks.

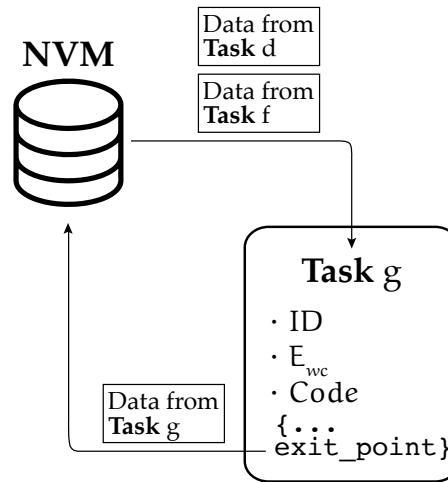


Figure 5.2: Task access to *NVM* is mediated by the framework.

Energy consumption E_{wc} must take into account the execution of a potential *NVM* write of the output value, even if in practice this activity is executed by the scheduler framework and not by the task itself. Thanks to this parameter the framework can temporarily prevent the execution of a task whose energy request is higher than the current energy budget and address the goal [G1T] on energy efficiency, presented in Section 3.2.1. To quantify this energy consumption it is possible to rely on the work of Ahmed et al. [1] that takes into account the variation in clock speed and energy drain mentioned in Section 4.2.

A task is considered successfully executed when one of its exit point is reached and, if provided, the data output is successfully written to memory. If the energy budget is insufficient and, either no exit point is reached, or the data output is not properly executed, then the task must be entirely rescheduled. Since the only way to write a data output to the shared memory is through an exit point, multiple writes are not allowed within a single execution.

The previous limitations ensure that in case of a power failure the system is rebooted to a consistent state, as long as the task code does not produce non idempotent effects through actuators, further examined in Section 5.3. In fact:

- if the execution fails before an exit point then for sure no data has been persistently changed on *NVM*;
- if the execution fails after an exit point and the data has been persistently written *NVM*, then the task code does not contain any other instruction able to affect the *NVM*, since multiple writes are not allowed.

5.2.2 Applications

An application is a collection of tasks linked by data dependencies that are necessary to provide a higher level service. They are characterized by the following elements:

- a unique identifier;
- a set of tasks;
- an initial task;
- a desired minimum throughput X_{\min} in terms of complete iterations over time, a complete iteration corresponds to one execution of all its task.

Since applications represent higher level services, their execution is supposed to be continuous.

The code deployed in the system may be composed by several tasks grouped in several applications. The scheduler selects at every wakeup a set of tasks to be executed with the current energy budget. Each task in the execution plan is selected as part of an application and, in particular, as a step in the execution of an iteration of a given application. In Chapter 6 we presents a detailed description of the execution model that guides tasks selection.

One iteration of an application consists in the execution of one iteration of all its tasks, starting from the initial one, and in an order compatible with the dependencies.

5.3 DATA DEPENDENCIES

Tasks communicate with each other by sharing data. The framework mediates this communication, collecting output data from completed tasks and providing these data to tasks that need them as an input to their computation, in accordance to data dependencies. This prevents task's direct access to *NVM* and keeps the internal state consistent. In fact, tasks work on volatile memory and their results are persisted only when they complete, the framework collects results and writes them on *NVM*.

In our proposed framework, data dependencies specify producer-consumer relationship among tasks, and they can be decorated with a set of Boolean conditions. A dependency is active, or in other words the consumer can access to producer's data, if and only if either the set of conditions is empty, or *all* the conditions in the set are evaluated as *true*. These conditions can capture temporal requirements, as requested by goal [G2T]. For instance they can be used to specify data expiration, or Minimum Inter Sample Delay (MISD) as in Mayfly [16].

We argue that simply allowing to specify these conditions, can not prevent semantics inconsistencies with a solution, like the one that we

aim to implement, that dynamically selects tasks at runtime, as we will discuss in Chapter 6. Let us consider the following example.

An application to control and log accesses to a facility is deployed on a transiently powered device. For the sake of simplicity we assume that the device has only one **NVM** memory location, that is managed as follows:

- a **STORE** `<data>` instruction saves `<data>` in the memory location;
- a **LOAD** instruction reads the data stored on **NVM**, providing it to the task and *removes* it from the memory location.

The application is composed of two simple tasks:

- a **SENSE** task that monitors the status of the entrance door;
- a **LOG** task that takes care of the transmission of the opening event to a collection point.

Requirement Each time a sensing event is produced by the first task, a log *must* be generated by the second one.

SENSE task produces data, while **LOG** task consumes them, hence they are connected by a data dependency. A simple data dependency poses no constraints on how the consumer task can access the producer's data. The output of the sensing task enables the log task: until no data is produced by the sensing task there is no need to schedule the log task. A dependency with no constraints on how and when the consumer can access these data, does not prevent the following execution trace where two consecutive iterations of the sensing task, not interleaved by a log task, violate the aforementioned requirement.

Iteration	Enabled tasks	Selected task	LOAD/STORE instruction	Memory on exit
1	SENSE	SENSE	STORE data1	data1
2	SENSE, LOG	SENSE	STORE data2	data1 , data2
3	SENSE, LOG	LOG	LOAD (reads data2)	data1 , data2

On initial boot, the only enabled task is **SENSE**: there is no data to log, therefore the device selects and executes this task. This execution of **SENSE** enables the **LOG** task that can now access to the sensed data, still nothing prevents a re execution of **SENSE**. For instance, let us suppose that the current energy budget does not allow the execution of the log task. Instead of waiting until the capacitor buffers enough energy to execute **LOG**, the runtime decides to use the currently available energy to execute another iteration of **SENSE**. This would overwrite the output of the previous iteration of the sensing task, violating the requirement.

To prevent the violation of the requirement, we need to allow the developer to specify a different access pattern for the consumer task,

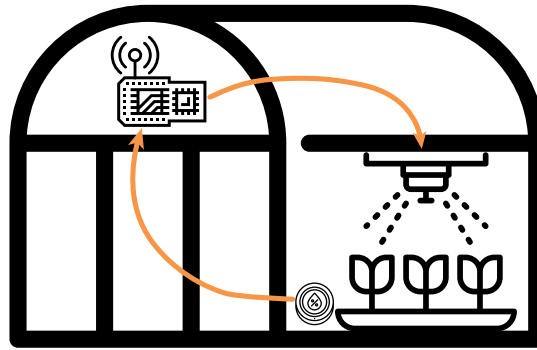


Figure 5.3: A TPC device is deployed in a greenhouse to control the irrigation system. The device operates a sprinkler based on data gathered from a humidity sensor. Each activity is sent to a collection point for logging purposes.

hence we need to enrich the semantic of data dependencies. This example does not cover all the potential data access patterns that produce the need for a new data dependency semantics. To prove that, we now introduce an more complex example that will motivate and guide us throughout the definition of this enriched set of semantics.

5.3.1 Greenhouse Example

In the example depicted in Figure 5.3, a greenhouse is equipped with an autonomous irrigation system that provides the correct amount of water to plants, based on sensed humidity. The system is controlled by a TPC device, powered with solar energy. The device is connected to a humidity sensor and sends commands to a timer that switches on a sprinkler for the desired amount of time. The sprinkler timer accepts one radio command `add_time <10m|20m|30m>`. If the sprinkler is off, then the command starts the irrigation and keeps it open for 10, 20 or 30 minutes. If the sprinkler is on, then command *increases* the duration of the irrigation for 10, 20 or 30 minutes. The application decides the amount of time that the sprinkler should be on, based on humidity. In particular, it needs a series of three consecutive humidity reads, to confirm the value and take the proper decision. Each decision is transmitted to a collection point for logging purposes.

The application is composed by four tasks:

Humidity senses humidity;

Decide decides the correct amount of irrigation time, based on humidity data;

Sprinkler sends the `add_time` command to the sprinkler, based on the decision;

Log transmits the log to a collection point.

Figure 5.4 presents the dependency graph for this scenario.

Let us examine each dependency one by one, starting from dependency ① that connects the task that senses humidity to the decision task. As we said, **Decision** task computes the time for the sprinkler, based on the last three humidity data. Let us imagine that the **Humidity** task completes its first execution. With a simple data dependency, without any enriched semantics, nothing prevents the execution of the **Decision** task, right after this single run of **Humidity**. To preserve the requirement, the developer should implement an internal check to verify that there are at least three humidity reads, and skip the execution otherwise. This situation happens every time a task has to access a set of data to extract a feature. Imagine an activity recognition task that decides based on a set of reads coming from an accelerator, or a task that implements a filter. A new kind of data dependency that allows to specify that the consumer task must receive a window of data, instead of a single item, would simplify the description of these situations and provide a useful tool to the developer.

Dependency ② connects the **Decision** task and **Sprinkler** task. Imagine that **Decision** task computes that the sprinkler should be active for 10 minutes. After that, **Sprinkler** task reads this decision and sends the command to the sprinkler that activates for 10 minutes. With a simple data dependency, nothing prevents the re execution of the **Sprinkler** task on the same decision. For instance it may happen that after a while, within these 10 minutes of sprinkler activity, there is only enough energy for **Sprinkler** task, and all the other tasks require a higher energy buffer. The runtime that controls task's execution could decide to execute it once again, since it is the only one that matches the energy budget. **Sprinkler** task would read once again the same decision and wrongfully increase the sprinkler activity for other 10 minutes. An enriched semantics on this data dependency could prevent the re execution of a consumer task based on an already read data.

With the dependency ③ that connects **Decision** and **Log** task, it happens the same issue that we described in the introduction to Section 5.3. The simple data dependency does not prevent that a new execution of **Decision** overwrites the result of the previous one, *before* that **Log** task sends it to the collection point. This would result in a missing item in the log. For this reason, we need a new dependency semantics that prevents a new execution of the producer task, until the consumer successfully reads the data. We can relax this requirement, if the system is able to store more than one version of the produced data, and provide the correct one to the consumer. Imagine that three consecutive executions of **Decision** can store three consecutive decisions on a buffer. In that case, the system could execute three times **Log** and provide it respectively the first, the second and the third

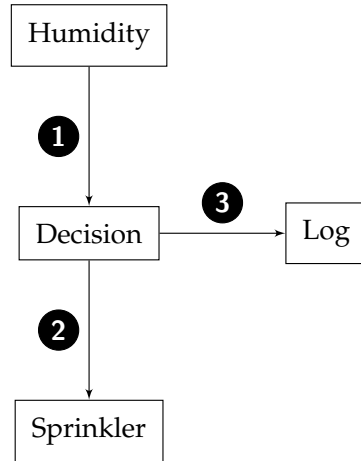


Figure 5.4: Dependency graph for the greenhouse example presented in Section 5.3.1. The application is composed by four tasks.

decision. To allow that we need a new dependency semantics that allows versioning of the producer data.

An enriched set of dependency semantics enables a simpler description of functional requirements, and allows the developer to program each task independently, under the guarantee that the runtime interleaves tasks executions in an order that is compliant with these requirements.

In the rest of this Section we present the details of each new dependency semantics that emerged from the aforementioned example.

5.3.2 Data Dependency Semantics

Our proposed framework enriches the semantic of the data pipes connecting producer and consumer tasks. In particular, let us consider two tasks τ and τ' , where $\tau' \in \text{Dep}(\tau)$, or, in other words, τ' needs some data coming from τ . The developer can specify one of the following expected behavior for the data dependency that connects τ and τ' .

- Simple dependency
- At most one read dependency
- At most one write dependency
- Add version dependency
- Window dependency

As we discussed in Section 5.3.1, the proper selection of the dependency type is fundamental to guarantee the intended application semantics. In the rest of this Section we present the semantic of these

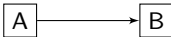
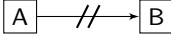
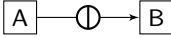
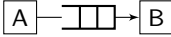

DEPENDENCY TYPE	ARROW	EXECUTION EXAMPLE
Simple dependency		$A(w_1), A(w_2), B(r_2), B(r_2)$
At most one read		$A(w_1), A(w_2), B(r_2), A(w_3), B(r_3)$
At most one write		$A(w_1), B(r_1), A(w_2), B(w_2)$
Add version		$A(w_1), A(w_2), B(r_1), B(r_2)$
Window		$A(w_1), A(w_2), A(w_3), B(r_1, r_2, r_3)$

Table 5.1: Data dependency semantics and corresponding graphical conventions. $T(wx)$ means that task T writes data x , while $T(rx)$ means that task T receives x as an input

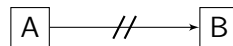
new dependencies by means of regular expressions, and we introduce the graphical conventions used in this document to represent each of them. The overview is summarized in Table 5.1. For the sake of simplicity, we consider an application composed by two tasks A and B , connected by a data dependency from producer A to consumer B .

Simple Data Dependency



Default data dependency in which each execution of A produces an output that overwrites the previous one, and each execution of B is always allowed to read the most recent data. A *simple data dependency* poses no constraints, either on the ability of B to read A output, or on the ability of A to produce a new output that overwrites the previous ones. This dependency produces a interleaving between A and B executions that follows the pattern $(A^n B^m)^*$ where $n > 0$ and $m \leq n$.

At Most One Read Data Dependency



The data produced by A *can not* be read multiple times by consecutive executions of B . Each execution of the consumer task *must* receive a fresh data. Multiple consecutive executions of the producer are allowed, each one overwrites previous data.

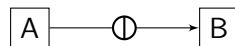
An *at most one read* dependency forces the execution of at least one iteration of task A between two consecutive executions of the

consumer task B, producing an interleaving between them that follows the pattern $(A^+B?)^*$ where:

- each application iteration starts with the producer task A;
- application iterations interleaving is allowed, resulting in consecutive executions of task A, each one overwriting the previous data;
- consecutive executions of A can be interleaved with one execution of the consumer B;
- consecutive executions of B are not allowed, since they must be interleaved with at least one execution of A that produces a fresh data.

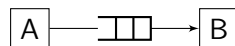
This dependency produces a stricter constrain on task selection compared to the simple one, since its pattern is a special case of $(A^nB^m)^*$ with $n > 0$ and $m \leq 1$.

At Most One Write Data Dependency

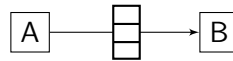


Each data produced by A *must* be consumed by one execution of B. In other words the data produced A can not be overwritten by a new execution, until the consumer B successfully reads it. An *at most one write* data dependency between A and B produces an interleaving of the two tasks that follows the pattern $(AB)^*$, since each iteration starts with an A and two consecutive iterations of task A must be interleaved with one execution of B that consumes the data. This dependency type produces the strictest constrain on task selection, in fact the language produced by $(AB)^*$ is a subset of the language produced by $(A^+B?)^*$.

Add Version Data Dependency



Each execution of the producer task creates a new version of the data. The scheduler provides the correct version to the consumer task depending on the iteration. This dependency allows the interleaving of multiple iterations, while keeping an *at most one write* semantic. The gain in task selection freedom, comes at the cost of a higher memory consumption and a more complex memory management. In fact the framework must store more data, one per version, and must execute logic to provide the correct version to the consumer.

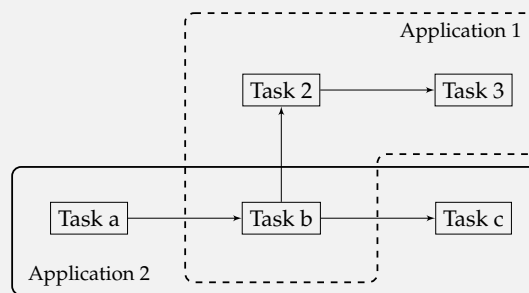
Window Data Dependency

This dependency type forces the execution of at least a given number of iteration of the producer task, in order to provide to the consumer task access to a sliding window of data, each one with an “*at most one read*” semantic. This could be useful in applications where a task must extract a feature from a collection of data.

Example 5.3.1 presents another scenario where the appropriate selection of the dependency’s semantic is fundamental to capture the functional requirements.

Example 5.3.1

The code deployed on the system is composed by two applications sharing one of their task as shown in the following graph.



Task A is the initial task for Application 1, while Task B is the initial task for Application 2; Task C and Task 3 are the final task for their respective applications.

For the sake of simplicity let us suppose that:

- Application 1 is simpler than Application 2 and its execution is less energy demanding, given that its tasks are shorter and require a small amount of energy;
- task’s execution never fails, since energy predictions are always correct;
- there are no conditions on data dependencies.

Let us assume that, at every wake up, a dynamic scheduler selects some tasks to execute, let us suppose that the following trace represents the selection operated over five consecutive wakeup:

1. Task A — Task B — Task C

2. Task A — Task B — Task 2 — Task 3
3. Task 2 — Task C — Task 3
4. Task A — Task B — Task C — Task 2
5. Task 3

Given the dependencies depicted in the graph, each execution of Task B enables a new iteration of Application 2, in particular:

- Application 1 performs three iterations over five wakeups;
- Application 2 performs three iterations over five wakeups, interleaving the first one that starts at step 1 and ends at step 2, with the second one that starts at step 2 and ends at step 3.

Depending on application semantic, it may happen that the interleaving of different instances of the second application, results in an unwanted behavior. In fact, if we suppose that the intended semantic for Application 2 is:

Each data produced by Task B *must* be processed by Task 2.

the proposed trace breaks the constraint, since the data produced by Task B in step 1 is overwritten by the one produced by Task B in step 2, therefore in both the first (step 2) and second iteration (step 3), the same data is processed by Task 2 (i.e. the one produced in step 2).

To prevent this issue the programmer must specify the intended semantic using the appropriate dependency type.

As we discussed so far, different dependencies induce different semantics. Often times, these semantics are strictly connected to task's nature. For instance a task could be a consumer responsible for data transformation, or it could operate an actuator. Here we propose a systematic classification of tasks, to help understand how different tasks and scenarios correspond to different potential violation to semantic constraints, and therefore to different dependency types selection. To classify tasks, we refer to the framework presented by Michael Jackson in "The World and the Machine" [18] and we divide them in three classes:

- those that operate exclusively on the machine;
- those that operate on shared phenomena controlled by the world;
- those operating on shared phenomena controlled by the machine.

Given this categorization, we consider possible consistency threats and semantic constraints violations, and, through an example scenario, we propose techniques to address them, setting proper data dependency patterns. In these examples we refer to an hypothetical scheduler that selects tasks satisfying the constraints induced by the dependencies. This scheduler works under some simplified assumptions, depending on the context. The real scheduler logic, implemented in the proposed solution, is presented in Chapter 6.

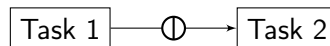
5.3.3 Task Operating on Machine

This first category of tasks is composed by all the tasks that perform some kind of data analysis or transformation on data stored on *NVM*. Possible examples of this category are tasks performing a convolution on an already captured image, tasks filtering acquired data, or tasks computing a Fast Fourier transform.

These kind of tasks do not pose a consistency threat to internal state, in case of a power failure, since write access to the shared memory is mediated by the scheduler that will reschedule the task if the write is not successfully performed. Still the usual violation of the semantic constraints, already presented in previous examples, are possible. To give a further example, let us consider the following graph representing two tasks with a simple producer-consumer relationship.



The execution trace Task 1 — Task 1 — Task 2 — Task 2 is allowed, still both the executions of Task 2 will process the same data produced by the second instance of Task 1. To prevent that the simple data dependency must be replaced with an at most one write dependency, producing the following graph:



The same result could be obtained using an *add version* dependency, in that case multiple iterations of the application could be interleaved. For instance, let us suppose that the system can store up to three consecutive versions, then three iterations can be interleaved and the following schedule could be selected. The subscript is the iteration counter, *w_x* means that the task writes *x*, *r_x* means that the task receives *x* as an input:

Task 1₁(w1) — Task 2₁(r1) — Task 1₂(w2) — Task 2₂(r2) — Task 1₃(w3) — Task 2₃(r3)

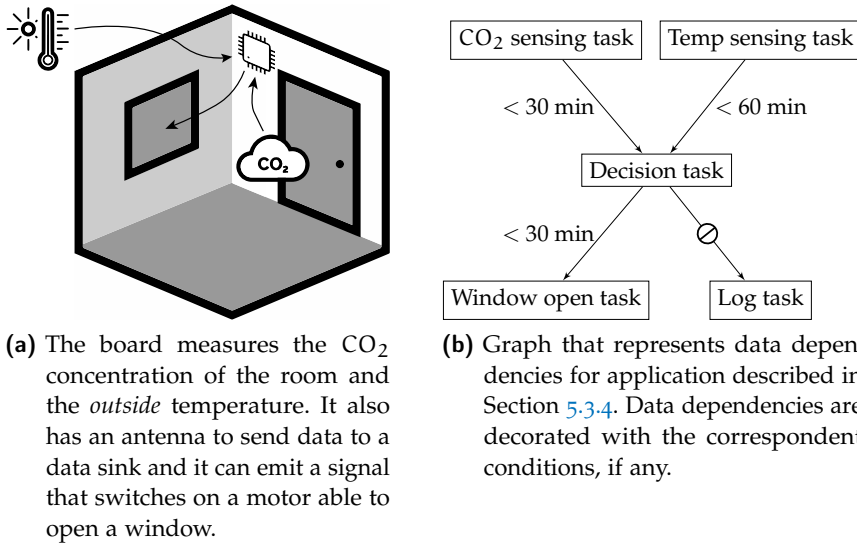


Figure 5.5: Example of an application with tasks operating on shared phenomena controlled by the world. An intermittently powered board, equipped with temperature and CO₂ concentration sensors is deployed in a room.

5.3.4 Task Operating on Shared Phenomena Controlled by the World

This second category is composed by all the tasks responsible of sensor reads. Usually these task are followed by a task of the previous category: a sensing task produces a data stored on *NVM*, consumed by a task performing some kind of computation on it.

When dealing with tasks in this category, data timeliness issues may arise since a data collected by a sensor may be of no use after a certain amount of time. Let us consider the following scenario.

As depicted in Figure 5.5a, an intermittently powered board, equipped with temperature and CO₂ concentration sensors is deployed in a room. The board measures the CO₂ concentration of the room and the *outside* temperature. It also has an antenna to send data to a data sink and it can emit a signal that switches on a motor able to open a window.

If the level of CO₂ is over a certain threshold and the outside temperature is *higher* than a certain value, then the board must send the signal to open the window. If the exterior temperature is too low, then the window must not be opened. *Every* decision must be logged by sending it to the data sink. The CO₂ concentration and the temperature level are supposed to change at two different frequencies, therefore a CO₂ must be considered stale after 30 minutes, while a temperature data is meaningful if taken within the last hour. The window should be opened only if the decision was taken no longer than 30 minutes ago.

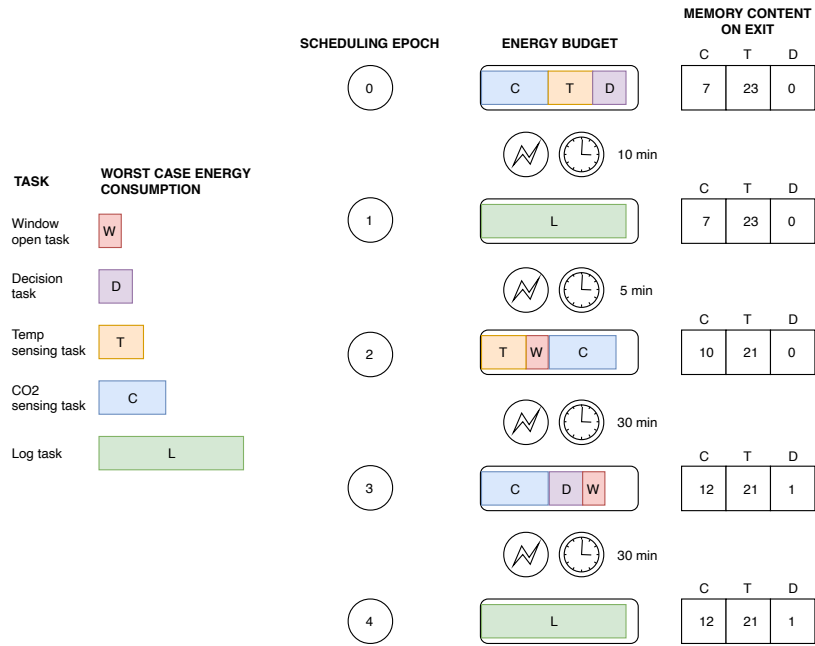


Figure 5.6: Possible scheduling trace the scenario described in Section 5.3.4

The graph in Figure 5.5b represents the application. Data dependencies are decorated with the correspondent conditions. The dependency spanning from the decision task to the log task, forces to send the decision, whatever it is, to the data sink before taking a new one, producing execution traces where two decision tasks cannot be executed without a log task in the middle. The staleness conditions on data dependency edges, force the scheduler to re-execute the upstream task if the condition is no longer satisfied (i.e. the data is older than the time limit). In particular the condition on the data dependency edge between decision and actuation tasks, works under the assumption that if the actuation task is not executed within 30 minutes, then the CO₂ concentration may have changed and a new decision must be performed with fresher data.

Figure 5.6 shows a feasible trace for the first five scheduling epochs. The window open task *W* in the scheduling epoch 2 is the last task of the first iteration of the application, while tasks *C* and *D* belong to the second iteration. Scheduling epoch 3 starts with task *C* because the sampled value for CO₂ concentration has expired, since it is 30 minutes old.

5.3.5 Task Operating on Shared Phenomena Controlled by the Machine

This category is mainly composed by tasks dealing with actuators, including the ones that manage radio output devices from the example in Section 5.3.4.

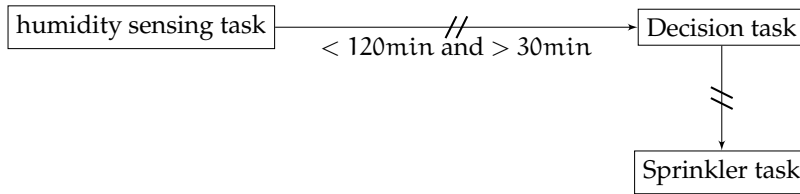


Figure 5.7: Graph that represents data dependencies and timeliness conditions, for application described in Section 5.3.5.

So far we introduced a dependency pattern to prevent data loss due to re-execution of the upstream task. Another peculiar issue arises with this class of tasks. In fact, we may experience an inconsistent behavior, when we execute multiple times a task that performs non-idempotent actions on the world. To prevent this, it is necessary to bind the task producing the data that informs the decision on the actuator, to the one that actually performs the action, with an *at most one read* dependency. To clarify this potential problem, let us consider a slightly modified version of the example presented in Section 5.3.1.

An intermittently powered device is deployed in a greenhouse to manage the irrigation system. The board is connected to a humidity sensor and it can send commands to a timer that switches on a sprinkler for the desired amount of time. The sprinkler timer accepts one radio command `add_time <10m|20m|30m>`. If the sprinkler is off, then the command starts the irrigation and keeps it open for 10, 20 or 30 minutes. If the sprinkler is on, then command *increases* the duration of the irrigation for 10, 20 or 30 minutes. The application senses humidity, decides the amount of time that the sprinkler should be on, and consequently sends the appropriate command to the sprinkler timer. The data from the humidity sensor are stale after 120 minutes, while two consecutive samples are useless if they have an inter sample rate shorter than 30 minutes.

The application is composed by three tasks: the initial one senses the humidity, the second one performs the decision and the third sends the command to the sprinkler timer.

Let us assume the following worst case energy requirements E_{wc} .

- $E_{wc}^{sense} = 10$ units;
- $E_{wc}^{decide} = 5$ units;
- $E_{wc}^{sprinkler} = 50$ units.

The graph in Figure 5.7 summarizes tasks and dependencies.

The *at most one read* dependency between decision and sprinkler activation, prevents the execution of two consecutive sprinkler activations based on the same decision, causing an unwanted increase of the active time for the sprinkler. Likewise, the dependency between sensing and decision tasks prevents two consecutive decisions on the same data.

Let us consider the execution trace shown in Table 5.2, where B is the energy budget of the current activation, H, D and S respectively stand for an iteration of the humidity sensing task, the decision task and the sprinkler command task. The subscripts represent the worst-case energy requirement of the task. The initial task is H and we suppose that the device activates when a threshold of 30 energy units is reached. For simplicity, let us suppose that task selection is performed once, at the beginning of the scheduling epoch, then all selected tasks are executed.

EPOCH	ENERGY BUDGET B	SELECTED TASKS
0	30	H ₁₀ D ₅
40 minutes inactive		
1	30	H ₁₀ D ₅
10 minutes inactive		
2	30	⊗ set wakeup to B = 50
10 minutes inactive		
3	50	S ₅₀
5 minutes inactive		
4	30	⊗ wait
5 minutes inactive		
5	45	H ₁₀ D ₅
10 minutes inactive		
6	30	⊗ set wakeup to B = 50
10 minutes inactive		
7	50	S ₅₀

Table 5.2: Possible trace for scenario presented in Section 5.3.5

Let us examine the trace summarized in Table 5.2.

Epoch 0 Initially the board activates with 30 energy units, accordingly to the activation threshold parameter. Task H is selected since it is the only one with no pending dependencies to be satisfied. The execution of H would unlock D, so the scheduler selects that task too. After H and D the budget would be 15 energy units. No other task from the current application iteration fits the budget, and a new iteration can not start since the condition on $H \rightarrow D$ data dependency sets the inter sample delay to 30 minutes. Selected tasks are executed, then the board stays inactive and harvests more energy till the threshold is reached.

Epoch 1 The budget is not enough to execute S and complete the first iteration, but 40 minutes passed from the first epoch, so the

condition on the inter sample delay is satisfied and the scheduler can start a new iteration of the application.

- Epoch 2** A new iteration can not start due to the inter sample delay, the only possibility is to sleep until $B = 50$ to execute S and complete the first iteration.
- Epoch 3** The budget is enough to execute S and the first iteration of the application, started in the first epoch, is now complete. This execution of S from the first application iteration is based on the data written by the last execution of D that belongs to the second application iteration, still this is ok with the application semantic.
- Epoch 4** The scheduler can not complete the second iteration, not even by setting a new wakeup threshold to 50. In fact the dependency forces the execution of a new D before an S , a D would require an H for the same reason, but H can not be executed due to the inter sample delay condition and the last data sampling happened 25 minutes ago. A new iteration can not be started due to the same problem with inter sample delay condition. The only feasible option is to wait until the inter sample delay condition is satisfied.
- Epoch 5** The data dependency condition on inter sample delay is now satisfied, so in order to complete the second iteration the scheduler re schedule a new iteration of D preceded by H .
- Epoch 6** Same as epoch three.
- Epoch 7** S can be executed and the current last application iteration is now complete.

As already discussed so far, a task's code does not have direct access to [NVM](#), neither to read, nor to write data. All these accesses are mediated by the runtime framework that manages memory to ensure its consistency, as requested by constraint [C2T]. In the next Section we present this memory access model.

5.4 MEMORY MODEL

[NVM](#) is managed *exclusively* by the framework that enforces its consistency. Each task may require access to a certain data, in accordance to dependencies specified by the developer. Tasks may produce output data that must be persisted on [NVM](#), as discussed in Section 5.2.1. These data are provided as parameters to their exit point. The runtime framework collects and persists them.

5.4.1 Write Output Data

Let us revise the requirements that our solution should follow, in light of our current goals: to provide a managed write access to *NVM*.

In Section 3.2.1, constraint [C1T], we stated that the system must persist result of successfully completed tasks. In other words, the persist process is deeply connected to task successful execution. To ensure the transactional semantics, tasks do not have access to *NVM* during their execution, since their effects must be visible and persistent only when the execution successfully completes.

Let us imagine that a task completes its execution, asks to persist data and that this process is interrupted by a power failure. On reboot all task's results are lost, hence the task must be re-executed. So a task successfully completes when it reaches one of its exit points *and* the framework successfully writes output data on *NVM*.

In constraint [C2T], we stated that the solution must ensure memory consistency. On task-based solutions, this implies that the internal state must not be affected by partial executions of a task. Let us suppose that a task must be executed two times. The first execution successfully completes, while the second one fails. On reboot, after the second unsuccessful execution, the results of the first one must be available and uncorrupted.

To summarize:

- the process to write the output data is performed by the runtime framework, but it must be considered as a mandatory step of a successful task's execution;
- to preserve consistency, write operations must be part of the transactional semantics too: either they complete successfully, or the internal state must be as if they never occurred at all.

We stated in Section 5.2.1 that tasks carry a worst case scenario energy consumption prediction E_{wc} . To address the first requirement we ask that this prediction includes the energy required to complete the persist process. In Chapter 6 we will examine how the scheduler selects tasks. In general, a task is selected only if its energy requirement fits the current budget. By including the energy needed to write *NVM*, we inform the scheduler to avoid the selection of tasks that potentially can not complete the persist phase.

To guarantee consistency in case of power failures during *NVM* write, the framework persists data with an *old master* — *new master* technique, where the old copy is used as backup. The new value is never overwritten on the current value, instead the write is performed on a second memory location, and the pointer to the current value is updated only after a successful write.

In particular, each task is associated to a *NVM* record, and to a global volatile variable. The global variable is useful to reduce the impact of

the read phase, presented in the next Section, while the non-volatile record is composed by the following elements:

Task ID Unique identifier of the task.

Versions array Circular array of $n + 1$ cells, where n is the number of versions for the output variable. The number of versions can be greater than 1 if the variable is involved in an *add version* data dependency, as specified in Section 5.3.

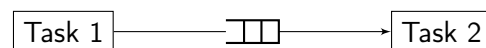
Timestamp array Circular array of $n + 1$ cells, where each cell stores a logical timestamp of the last successful write performed on the correspondent version array cell. This timestamp is useful to track time related constraints [G2T], as discussed in Chapter 6.

Write index Index of the cell that must be used to store the next version. Its value starts from the first cell of the version array, and the pointer is moved to the next cell *after* each write.

Given the circular structure of the versions array, the index to the oldest version of the stored data moves as we insert new data. Therefore, in addition to these values that are written on *NVM*, the framework computes the **window begin index**. This value represents the index of the oldest valid element stored in the version array. Its value is obtained from the value of the write index. It points to the first cell of the version array until the array is full, then it points to the next cell with respect to the one pointed by the write index. For instance let us suppose that the version array stores 3 values, hence its size is 4. Window begin index points to cell 0 until the fourth write. From the fourth write on, it points to the next cell with respect to the write index. If the write index is 2, then its value is 3, if the write index is 3, then its value is 0, because the version array is circular.

Figure 5.8 summarizes the steps followed by the framework to persist output data on behalf of tasks. In particular the persist process never overwrites the old value, to maintain consistency in case of power failure during the write. The update of the write index corresponds with the successful execution of the whole process.

To clarify this process let us consider the following example. The application represented in the following graph is deployed on a transiently powered device.



Let us assume that:

- the framework is set to keep 3 versions of Task 1 output.
- for sake of simplicity we omit the update of the timestamp field.

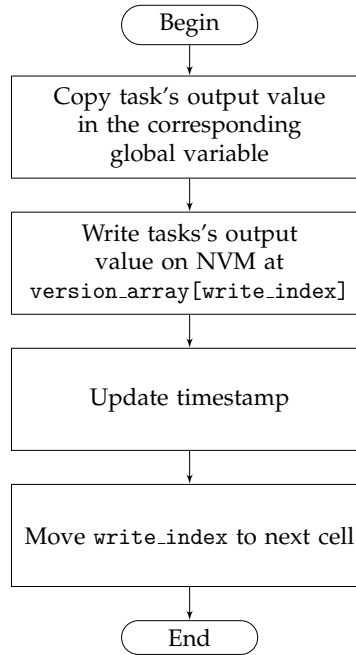


Figure 5.8: Flowchart representing the steps followed by the runtime environment to persist task's output on *NVM*.

Figure 5.9 presents the status of Task 1 *NVM* data record during 8 consecutive iterations, starting from an empty version array.

This process implements a two-phase-commit like mechanism, similar to the one used in Alpaca [25]. This mechanism guarantees that the internal state is consistent even in presence of power failures during the write phase. In fact:

- if a power failure happens before that the output is written on the location indexed by the **write index**, then *NVM* is unaltered;
- if a power failure happens after the write, but before the update of the **write index**, then on reboot the status is as if no write happened, and the three versions are unaltered.

5.4.2 Read Input Data

As stated before, in our solution tasks do not have direct access to *NVM*. In the previous Section we discussed how the framework manages write access to persistent memory, in this Section we present how our proposed solution manages read access.

Let us suppose that a task τ is selected for execution. If $IN(\tau)$ is not empty, then the system must collect all data in the set from *NVM*, and provide them to τ as an input to its computation. If the incoming dependency between τ and the producer task is not an *add version* dependency, then the framework simply provides access to the current value. Otherwise the framework must select the appropriate version

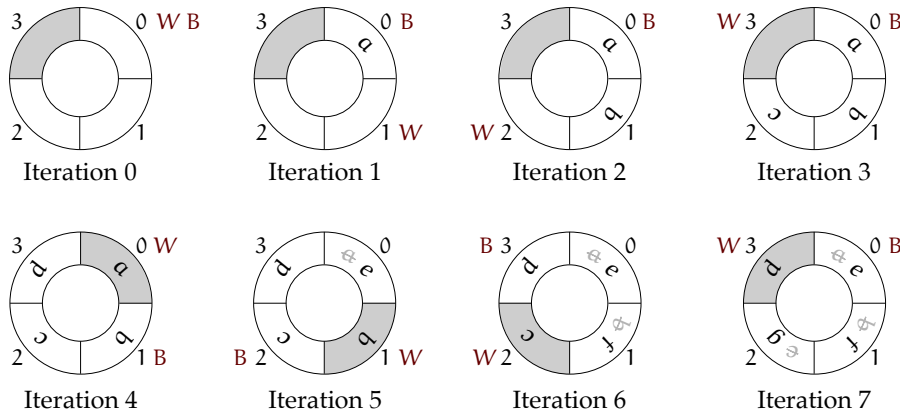


Figure 5.9: Example of the update of the *NVM* data record. This picture represents the first consecutive iterations of a Task involved in an *add version* data dependency, on a system set to keep 3 consecutive versions. *W* represents the **write index**, *B* is the **window begin index**, while letters from *a* to *g* represent consecutive outputs of the task.

that corresponds to the correct iteration of the producer, as presented in Section 5.3.

As we discussed in Section 2.2, read and write accesses to *NVM* are costly operations and any solution, implemented to support *TPC* and guarantee forward execution, must reduce the number of these operations to preserve energy.

Without power failures causing the lost of the volatile state, we could simply store the output of a task in a global variable associated to the producer task, and access this value from the consumer task. This would prevent any access to volatile memory, reducing energy overhead. Unfortunately this is not always the case, since power failures may occur between the execution of producer and consumer tasks.

As discussed in Section 5.4.1, each task is associated to a global variable stored on volatile memory. As presented in the flowchart in Figure 5.8, the first step when persisting a task output is to copy the value in this global variable.

Let us call *power cycle* the time interval between two consecutive power failures. If the producer and the consumer tasks run within the same power cycle, and the dependency between them does not involve versioning, the consumer can simply access to the volatile global variable, without any read access to *NVM*. Otherwise, if they run in two different power cycles, or the consumer requires a version different from the last one, then:

1. the requested version of the data is copied from the versions array on *NVM*, to the correspondent global variable;
2. the consumer task is executed and it can access to the correct value, reading the volatile global variable.

This implies, as stated in Section 5.3 that an *add version* data dependency adds an energy overhead, since it requires more NVM accesses, one per version, to fetch the correct version.

This process effectively minimizes energy consumption. In fact:

- if no power failure happens between producer and consumer we avoid any read access to NVM;
- if more than one task need the same data as an input to their computation within a power cycle, then the value is read from NVM only once, then it can be accessed from volatile global variable.

As we discussed in Chapter 4, the selection of a correct activation threshold has a relevant impact on system performance. Here we see yet another proof of that: by selecting a threshold that corresponds to an energy budget that fits both producer and consumer in a single power cycle we reduce the overhead of the framework.

So far we discussed how code is structured in tasks and applications, how developers can connect tasks by means of data dependencies and how these dependencies can inform the framework on application constraints through the selection of different semantics. Dependencies can be decorated with conditions to capture time related constraints [16]. We also presented how the framework provides managed access to NVM.

Our programming abstraction is *orthogonal* to the implementation of the dynamic scheduler. In fact, it can serve as a base to any task-based solution in which task's execution plan is either statically selected at compile time, or dynamically at runtime.

In the next Chapter we introduce another core component of our solution: the dynamic scheduler.

6

SCHEDULING TASKS

In this Chapter we present the main component of our solution: the dynamic scheduler, but first let us revise what we introduced so far and what is still missing. In Section 4.4 we stated that our goal is threefold:

1. ✓ propose a new *programming abstraction*;
2. build a *dynamic scheduler*, energy aware, that is able to react to changes in the amount of available energy;
3. manage *activation threshold* at runtime.

In Chapter 5, we presented the three components of the programming abstraction: tasks, applications and dependencies, together with the description on how our solution manages memory to ensure its consistency and preserve progress. We still miss scheduling and threshold management.

In Chapter 4, we discussed the importance and the challenges of the selection of the activation voltage threshold, understanding that changes to this threshold affect system's performance. In particular, we saw that finding the right balance is not trivial, as both an increase and a decrease of this threshold may produce positive and negative impacts on energy efficiency and system's performance.

As shown in Figure 4.5 in Chapter 4, a higher threshold corresponds to a higher energy budget, since energy is directly proportional to voltage, following this equation.

$$E = \frac{1}{2}CV^2 \quad (6.1)$$

A higher energy budget let the system run multiple tasks within a power cycle. Let us suppose that task A and B are connected by a simple data dependency from A to B. If the threshold is set so that the corresponding energy allows the system to run both tasks within the same power cycle, then, in accordance to the memory model presented in Section 5.4, there would be no need to restore the output of A from Non Volatile Memory (NVM), since B can read it from volatile memory as no power failure happened after A. As shown in Figure 4.4 in Chapter 4, this would reduce the overhead, increasing energy efficiency.

On the other hand, as already discussed in Chapter 4, the capacitor charges faster when its voltage is lower, due to its non linear charging characteristic, hence a lower threshold reduces the length of the time intervals during which the board is inactive while accumulating enough energy to reach the activation threshold. Moreover, as already discussed, on the considered platform the MCU exhibits a higher energy efficiency when powered with a lower voltage [1]. This complexity, together with the unpredictability of the energy source, makes the selection of thresholds a particularly difficult task, and suggests that threshold management should be a refinement process to be performed at runtime.

In Section 4.4, we argued that programmer's ultimate goal is not generic energy efficiency, but the satisfaction of throughput-related requirements, which depend on specific scenarios. In other words, the programmer's goal is not to minimize energy consumption, but to produce outputs, or effects on the world, with a throughput that falls within given boundaries. Reading a sensor at a given pace, extracting a feature from sensed data with a throughput compatible with sensing rate, guaranteeing that the device can capture a change in the world that happens at a given frequency. Once again, our ultimate goal is not energy efficiency per se, instead efficiency is a way to minimize the overhead and increase code execution to obtain the desired throughput.

For this reason, we structured our programming abstraction so that the developer can divide the code base in multiple applications, allowing to assign a desired minimum throughput to capture specific non-functional requirements. Given this structure, our scheduler tries to satisfy these requirements at runtime, selecting the appropriate task to run.

As we saw, performance is deeply connected with threshold management. Hence, a scheduler interested in reaching certain performance to satisfy throughput requirements, must be involved in threshold selection. For this reason, in our proposed solution, threshold selection is part of the scheduling activity.

The runtime scheduler is the main component of our solution and its description requires an articulate discussion of several aspects, from workload management, to task selection policies and threshold management. Hence, to simplify the navigation of this Chapter, we propose in the next Section an outline, while in Section 6.9 we conclude with a final overview of all the features of the scheduler. Moreover, the most relevant features of the scheduler, presented in the following Sections, are highlighted with a **Property** box.

Property X

Example of property box, used to highlight important scheduler properties.

6.1 CHAPTER OVERVIEW

Section 6.2: We said that our goal is to satisfy throughput requirements. For this reason, in Section 6.2, we start with a precise discussion of what this means, as this goal guides the whole scheduler definition.

Section 6.3: To simplify fulfilling throughput requirements and reduce the scheduling overhead, in presence of unstable energy sources, the scheduler focuses on one application at a time. Moreover, different amounts of energy allow different workloads: a high amount of energy buffer intuitively allows us to run more applications. For these reasons, the scheduler is able to dynamically change the workload to adapt to different energy scenarios, as described in Section 6.3.

Section 6.4: Given these considerations on fulfilling throughput requirement, and on the ability of the scheduler to dynamically adapt the workload, in Section 6.4 we describe the scheduler initialization phase. We examine how the scheduler manages the sets containing references to tasks and applications. We discuss on the initial selection of the activation threshold.

Section 6.5: Once initialized, the scheduler starts selecting tasks for execution. In this Section we describe how they are selected at runtime.

Section 6.6: It may happen that applications exhibit a throughput that is not compliant with their requests. In this Section we describe how the scheduler reacts at runtime to these events.

Section 6.7: Long inactivity periods, caused by charging phase after power failures, may produce a relevant decrease of application's throughput. The frequency of these periods is related to the stability of the power source. Some can be characterized as fairly stable, like those provided by solar harvester, others are more subject to continuous drops and unpredictable stability, like kinetic or RF sources. Since the scheduler reacts at runtime to throughput changes, these power failure may cause a sequence of actions that affect the overall performance of the system, as described in Section 6.6. Knowing the supposed stability of the

energy source that powers the device, the developer can inform the scheduler on this parameter to adapt its policy to the real scenario of deployment. In this Section we describe how this can be done.

Section 6.8: Fairness is an important concept related to scheduling policies. In this Section we propose a discussion on fairness indexes and how they relate to our proposed solution.

Section 6.9: Finally, we propose a complete overview on the concept described in previous Sections.

So let us now start with our first step: a precise discussion on applications throughput, that is the main parameter that guides the scheduler decisions.

6.2 MINIMUM THROUGHPUT AND APPLICATIONS PRIORITY

The minimum throughput for an application A $X_{\min}(A)$ [$\frac{\text{iterations}}{\text{second}}$] is the minimum expected number of complete iterations over a time interval. A complete iteration corresponds to one execution of all application's tasks, starting from the initial one, and in an order that is compatible with data dependencies. As we saw in Section 5.2.2, the programmer specifies the minimum desired throughput for each application. This requirement must be intended as best effort, due to the nature of the techniques to implement timing on batteryless devices [17], and the unpredictability of energy sources.

In a scenario where multiple applications are deployed on a single board, this parameter induces a priority of execution. In fact, if an application has a higher requested throughput compared to the other ones, then its tasks must be executed more often to achieve a higher number of complete executions, hence they should be selected with a higher priority. Moreover, by selecting minimum desired throughputs, the developer can inform the scheduler on the relevance of the applications deployed. Let us suppose that a transiently powered device is deployed on a road sign. This device implements three functions, therefore three applications A, B and C:

A: takes a picture of the road, it analyzes the picture to understand the traffic level, and it sends this data to a data sink;

B: measures $PM_{2.5}$ concentration and sends the data to a data sink to monitor pollution;

C: compresses the last road picture and sends it to the data sink.

We assume that traffic congestion changes at a higher frequency than particulate concentration. Moreover, we suppose that sending a picture to the data sink, even if properly compressed, consumes

more energy than sending a single data item. By selecting minimum requested throughput such that $X_{\min}(A) > X_{\min}(B) > X_{\min}(C)$, we inform the scheduler that:

- application A is our main goal and should be the one with the highest priority;
- application C implements functions that are least relevant, compared to those implemented by applications A and B.

From these assumptions we derive an important property of our runtime scheduler.

Property 1

The minimum requested throughput X_{\min} induces a total order among applications, that are ordered by decreasing X_{\min} . The scheduler tries to satisfy throughput's requirements by decreasing X_{\min} order: an application's X_{\min} requirement is considered only when the X_{\min} requirements of the applications with a higher value are satisfied.

As we discuss in the next Sections, from **Property 1** we derive important features of our scheduling algorithm.

We will see that, to enforce **Property 1**, the scheduler activates and deactivates applications execution, dynamically changing the workload. So, let us discuss the details of this scheduler feature.

6.3 DYNAMIC WORKLOAD MANAGEMENT

In Section 6.2 we introduced **Property 1**, that states that our proposed runtime scheduler tries to satisfy throughput's requests one application at a time, starting from the application with the highest X_{\min} and then by decreasing X_{\min} order.

This assumption reduces the complexity of the scheduling algorithm, in presence of continuously changing surrounding conditions due to the instability of power sources. Reducing the complexity, we reduce the scheduling overhead.

To enforce **Property 1**, the runtime activates one application at a time, starting from the one with the highest priority and adding a new application only if the X_{\min} requirement is satisfied for all currently active applications. In fact, each application is composed by a set of tasks and one iteration of the application corresponds to one iteration of all its tasks. The more applications we activate, the more tasks we must run in order to complete the respective iterations.

Since the energy source is unpredictable, the energy intake may change over time, resulting in a higher number of power failures and,

consequently, in a higher number of inactivity intervals due to the charging phase. Moreover, the duration of these interval may change and increase, when the harvester provides fewer energy. This may result in the violation of X_{\min} request for an already active application.

Let us suppose that we have two applications A and B and that $X_{\min}(A) > X_{\min}(B)$, hence A has a higher priority than B. The system is able to satisfy A's throughput requests, therefore it activates B and, after an initial transitory phase, both throughputs are satisfied. Suddenly, a drop in the provided energy causes a sequence of power failures, followed by long charging phases. This obviously impacts both applications throughput, which we remember is the number of iterations per second. In this case the system disables application B, so that the global workload is reduced and the throughput of the first application A may increase. These activation or deactivation operations are performed iteratively one application at a time.

From these observations we derive a second property of our runtime scheduler.

Property 2

Given a set of applications $\{A_1, A_2, \dots, A_n\}$, where $X_{\min}(A_i) \geq X_{\min}(A_{i+1}) \quad \forall i \in [1; n]$:

- initially only A_1 is active;
- A_{i+1} is activated only if the X_{\min} throughput request is satisfied for each A_j with $1 \leq j \leq i$;
- given an active application A_i , if during system's activity the measured throughput of A_i is below the requested $X_{\min}(A_i)$, then the system iteratively deactivates, one at a time, as many application A_j with $i < j \leq n$ as needed, until the measured throughput of A_i matches $X_{\min}(A_i)$.

In Section 5.2, we said that applications are composed by tasks. These tasks are the basic block that the developer groups in applications, in order to achieve higher-order services. Activating an application means scheduling for execution all its tasks.

We also said that a task is characterized by its input set, which is the set of the incoming dependencies, or, in other words, the set of data that are needed for its execution. As shown in Figure 6.1, these data may be produced by a task included in a different application. Task 2, included in Application 1 needs a data produced by Task b from Application 2 to complete its execution. This example may be generated by a scenario where, for instance, the device is deployed in a room, Application 1 implements a service to operate the air conditioning system, while Application 2 measures and logs presence. Task 2 could

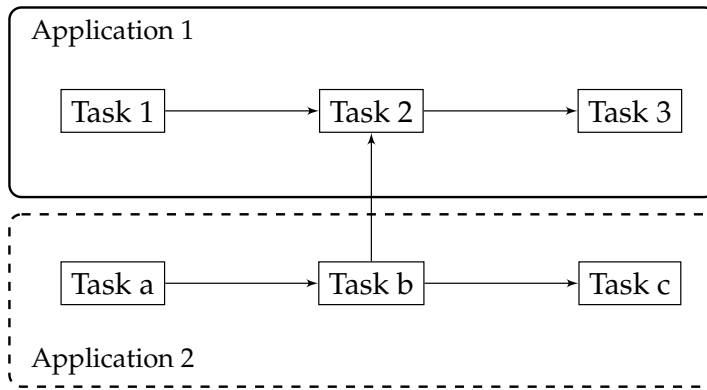


Figure 6.1: In this example if we want to activate Application 1, then we must schedule for execution all of its tasks plus Task b, even if it comes from a different application. Task b itself has an incoming dependency from Task a, so this second task must also be scheduled for execution, otherwise Task b would not have all the inputs it needs to complete and, consequently, neither Task 2.

be the decision task, that selects the command for the air conditioner, based on the temperature coming from Task 1 and the number of persons currently in the room, coming from Task b.

Let us suppose that we want to activate Application 1, then we must schedule for execution all of its tasks plus Task b, even if it comes from a different application. Task b itself has an incoming dependency from Task a, so this second task must also be scheduled for execution, otherwise Task b would not have all the inputs it needs to complete and, consequently, neither Task 2. In other words, every time we schedule for execution a task, we must travel backwards its incoming dependencies, to ensure that all the tasks we visit are also scheduled for execution. In the example from Figure 6.1, if we want to activate Application 1, then, without taking in consideration the order, the set of tasks scheduled for execution must be {Task 1, Task 2, Task a, Task b, Task 3}.

The same goes when we deactivate an application. Its tasks must be removed from the workload, except for those that are relevant for other applications that are still active. Let us consider once again Figure 6.1, and let us suppose that Application 2 was active and now must be deactivated, while Application 1 remains active. Then we must remove from the workload only Task c, because Task a and Task b are still necessary to execute Application 1.

We saw that scheduler enables the execution of an application only when the requirements of the applications with a higher priority are satisfied. Once enabled, an application can be disabled if the requests of the applications with higher X_{\min} are no longer satisfied. These changes in workload let the runtime scheduler focus on a requirement at a time, simplifying its computational effort and enforcing the aforementioned **Property 1**.

This behavior may intuitively lead to application's starvation. In fact, let us suppose that two applications A and B are deployed and that $X_{\min}(A) > X_{\min}(B)$. Following **Property 1**, this means that the priority of application A is the highest, hence the scheduler will first try to satisfy its throughput requirement, and only after B's one. It may happen that, due to energy source characteristic, A's requirement can not be satisfied, hence B is never activated, leading to its starvation.

Depending on the specific scenario this can be considered an acceptable behavior, or not. In fact, if the application with the highest priority is by far the most relevant one, and its throughput request must be considered stringent, than this behavior is meaningful. On the contrary, if the scenario demands for a more fair distribution of execution time among applications, this may not be an acceptable solution. Since this depends on the specific scenario, we implement techniques, described in Section 6.7, to tune the willingness of the scheduler to activate a new application, allowing premature activations of new applications to obtain a fairer behavior.

With **Property 1** we stated that the scheduler tackles one throughput request at a time, with **Property 2** we said that initially the only active application is the one with the highest X_{\min} . With these considerations in mind, let us go through the scheduler initialization stage.

6.4 SCHEDULER INITIALIZATION

The runtime scheduler maintains the following sets:

ActiveApps is a set that contains all the applications that are currently running, sorted by decreasing X_{\min} order;

ActiveTasks is a set that contains all the tasks belonging to active applications, or needed to satisfy the dependency of a task of an active application.

EnabledTasks is a set that contains all the task from **ActiveTasks** whose dependencies are satisfied and, therefore, can be executed.

As already discussed in Section 5.3, a dependency spanning between two tasks from τ_1 to τ_2 , is satisfied when τ_1 successfully completes, and τ_2 can use its output according to the semantics of the specific dependency. The difference between *active* and *enabled* tasks is that the first may have unsatisfied incoming dependencies, while the second can be actually executed because all their incoming dependencies are satisfied.

As presented by Algorithm 6.1, the initialization of the scheduler begins with the activation of the application with the highest X_{\min} . All its tasks are activated, together with those that are in their input set, but belong to other applications.

Algorithm 6.1. Activation of the first application**Global Variables:**

- A : $\{a_1; a_2; \dots; a_n\}$ Set of deployed applications sorted in decreasing requested throughput order ($a_1.Xmin > a_2.Xmin > \dots > a_n.Xmin$).
- $ActiveApps$: Set of running applications.
- $ActiveTasks$: Set of tasks to be scheduled.
- $EnabledTasks$: Set of active tasks with satisfied incoming dependencies.

Body:

```

1 ActiveTasks = {}
2 ActiveApps = {}
3 ActiveApps = ActiveApps  $\cup$   $\{a_1\}$ 
4 ActiveTasks = ActiveTasks  $\cup$   $a_1.tasks$ 
5 do
6   NewTasks = {}
7   for all  $\tau \in ActiveTasks$  do
8     NewTasks = NewTasks  $\cup$  ( $\tau.InSet \setminus ActiveTasks$ )
9   ActiveTasks = ActiveTasks  $\cup$  NewTasks
10 while NewTasks  $\neq \emptyset$ 
11 for all  $\tau \in ActiveTasks$  do
12   if  $\tau.Inset = \emptyset$  then
13     EnabledTasks = EnabledTasks  $\cup$   $\{\tau\}$ 

```

As we stated in the introduction to this Chapter, the scheduler is also responsible for the selection and refinement of the activation threshold. Therefore, after this setup, the scheduler must set the initial voltage threshold.

Let us revise all the observations made so far on threshold management, to understand how to select its initial value.

- TI MSP430 MCU exhibits a higher energy efficiency at lower voltages [1];
- capacitors charging speed decreases as they accumulate energy, due to their characteristic therefore a lower threshold results in a shorter inactivity interval needed to reach that threshold;
- a higher voltage threshold corresponds to a higher energy buffer, allowing us to fit more tasks within a power cycle, and therefore to reduce NVM read accesses according to Section 5.4;
- the complexity of voltage threshold selection demands for an iterative process that refines the value at runtime, as discussed in Section 4.2.

Given these observations, we apply a conservative approach to select the initial threshold. We select the lowest possible value to maximize the instruction speed, reduce the interval needed to reach the threshold, and start the first execution as soon as possible. Another component of the scheduler, described in the following Sections, is in charge of successive refinements.

Algorithm 6.2. Voltage threshold initialization

Global Variables:

- `ActiveTasks`: Set of tasks to be scheduled. requested throughput order.

Global Constants:

- E_s : Average energy consumption of a one scheduler iteration execution;
- E_r : Energy needed to copy a word from `NVM` to stack.

Body:

```

1 for all  $\tau \in \text{ActiveTasks}$  do
2   prediction =  $\tau.E_{wc} + E_r \times \tau.WC$ 
3   EnergyPredictions = EnergyPredictions  $\cup$  prediction
4 setThreshold(max(EnergyPredictions) +  $E_s$ )

```

This threshold is selected in order to fit the active task with the highest energy consumption E_{wc} , that, as stated in Section 5.2.1, is a parameter provided as an input to the framework, specified for every task.

Let us consider two generic tasks connected by a data dependency spanning from the producer τ_p to the consumer τ_c . In accordance with the memory model, presented in Section 5.4, without any power failure between the two tasks, τ_c can read the data produced by τ_p from the volatile memory. Otherwise, if a power failure happens, the runtime must access `NVM` to read τ_p data and offer them to τ_c . When selecting the threshold we must account for this second worst-case scenario. Therefore, when we compute the energy consumption for a generic task, we must add to E_{wc} the overhead needed to potentially restore input data from `NVM` to main memory.

In conclusion, the predicted energy consumption $E_p(\tau)$ for each task τ is computed as follows:

$$E_p(\tau) = E_{wc}(\tau) + E_r \times WC(\tau) \quad (6.2)$$

Where E_r is the energy consumed to read one word from `NVM`, and $WC(\tau)$ is the words count of the input set for task τ .

Finally, the threshold value must also take into account the overhead E_s introduced on average by the scheduler at each task execution. This value can be affected by the number of tasks and applications that are deployed, therefore it should be obtained through empirical experiments. Algorithm 6.2 summarizes the selection of the initialization threshold.

Now the scheduler initial setup is complete and the execution can start. As stated before, the scheduler starts by executing the first application, trying to satisfy its minimum requested throughput.

We still miss some features, though. We still do not know how the scheduler selects at runtime which task to execute from the set of enabled tasks. Moreover, in Section 6.3, we said that the scheduler dynamically changes the workload. It activates a new applications when

all the active applications throughputs are satisfied, and it deactivates the last added application, if *any* of the currently running application is under performing. But let us suppose that we have two applications A and B currently active, with A on pace with its throughput requests, while B is under-performing. We do not have any application that can be deactivated to reduce the workload, and we still miss a way to deal with this event. The same goes for a system where there is just one currently active application that is under performing.

In Section 6.5, we describe how the scheduler selects which task to run among the enabled ones, while in Section 6.6 we present how the runtime scheduler reacts to under-performing or over-performing applications.

6.5 TASK SELECTION AND DEADLINES MANAGEMENT

After the initial setup, we have:

- *one* active application: the one with the highest X_{\min} ;
- all its tasks in the active tasks set, plus all those in their input sets;
- all the active tasks without incoming dependencies in the enabled set.

Since our main goal is the satisfaction of throughput requirements, we need to know the execution time of an application iteration, so that we can check if this requirement is satisfied or not. In particular, based on the minimum throughput requirement X_{\min} iterations per seconds, we can say that if the application executes an iteration within $1/X_{\min}$ seconds, then on average the throughput requirement is satisfied. So we can say that the deadline for an application with X_{\min} minimum throughput requirement is $1/X_{\min}$.

We also need a way to prioritize the selection of tasks belonging to applications that are closer to their deadline. If we have this information, our scheduler can select tasks based on how close to the deadline is the application that contains them.

Let us imagine an application composed by two tasks: τ_1 and τ_2 connected by a simple data dependencies from τ_1 to τ_2 . Let us imagine that they are grouped in an application with $X_{\min} = 1/6$ iteration per seconds. This means that the deadline for one iteration of the application is 6 seconds.

We need a way to propagate this information along the sequence of tasks executions. To do that, we can start by associating the first task of the application with this value, so we associate τ_1 with the application deadline value of 6 seconds. This means that, currently, the task is part of an application iteration that must be completed within 6 seconds. This value does not represent the deadline for the

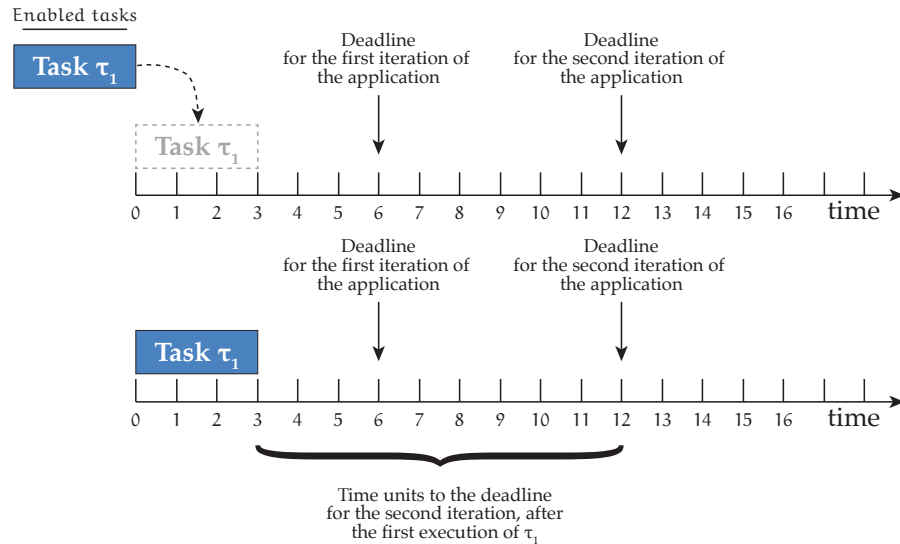


Figure 6.2: Example of deadline update after the execution of a task.

execution of the specific task, but for the current iteration of the entire application that contains the task.

Let us suppose that τ_1 completes after 3 seconds. Given the dependency, its execution enables τ_2 . Now we have two enabled tasks: we can execute τ_2 as a second step of the first iteration of the application, or we can start a new iteration by executing τ_1 again. In fact, we remind that, unless a particular data dependency pattern prevents it, the runtime can decide to start a new iteration before that the current one completes. For instance, this can happen in case the next task of the current iteration does not fit the energy budget, but the first task of the next iteration does.

So, we need a mechanism to:

- reset the application's deadline value associated to τ_1 , after it completes.
- associate a value to τ_2 at the end of τ_1 execution.

As shown in Figure 6.2, since τ_1 took 3 seconds to complete, then the first iteration of the application now has 3 seconds left before its deadline. Moreover, since the throughput requirement imposes one iteration every 6 seconds, we now have 9 seconds to complete the second iteration of the application. Therefore, we can reset the value associated to τ_1 to 9 seconds, and associate 3 seconds to τ_2 . In fact:

- a new execution of τ_1 , now would be part of the second iteration of the application, that must complete within 9 seconds;
- an execution of τ_2 is part of the first iteration of the application that must complete within 3 seconds.

The scheduler can select *earliest deadline first*, taking into consideration these values. In our example, this would result in the selection of τ_2 .

Let us suppose that τ_2 completes in 2 seconds. By subtracting this execution time to its associated value of 3 seconds, we know, at the end of the first iteration of the application, that the iteration completed 1 second ahead of its deadline, therefore the requirement is satisfied, and the application is currently running ahead of its deadline.

In practice, we can obtain the aforementioned behavior with the steps described in the following property.

Property 3

The initial tasks are associated to the application deadline, which is $1/X_{\min}$. Then, after each execution, deadlines are updated as follows:

1. on task completion, the application deadline is propagated to the newly enabled tasks;
2. the deadline for the initial task is reset to twice the initial value;
3. every time a task completes, its execution time is subtracted to all the deadlines.

Moreover, on restore after a power failure, the TARDIS [17] time is subtracted to all the deadlines.

Figure 6.3 summarizes these steps using the previous example with tasks τ_1 and τ_2 .

As we mentioned in Chapter 5, a task can be part of more than one application, because its function may be relevant to more than one service. In that case, the initial application deadline for enabled tasks is set to the minimum value among the deadlines of the applications that include them.

Task selection is summarized by the following property.

Property 4

Each task is associated to an *application deadline*. The scheduler selects and executes one task at a time by *earliest deadline first*. The selected task is the one with the lowest deadline, whose energy consumption prediction E_p fits the current energy budget. If none of the enabled tasks fits the current energy budget, then the device enters low power mode until the current activation threshold is reached.

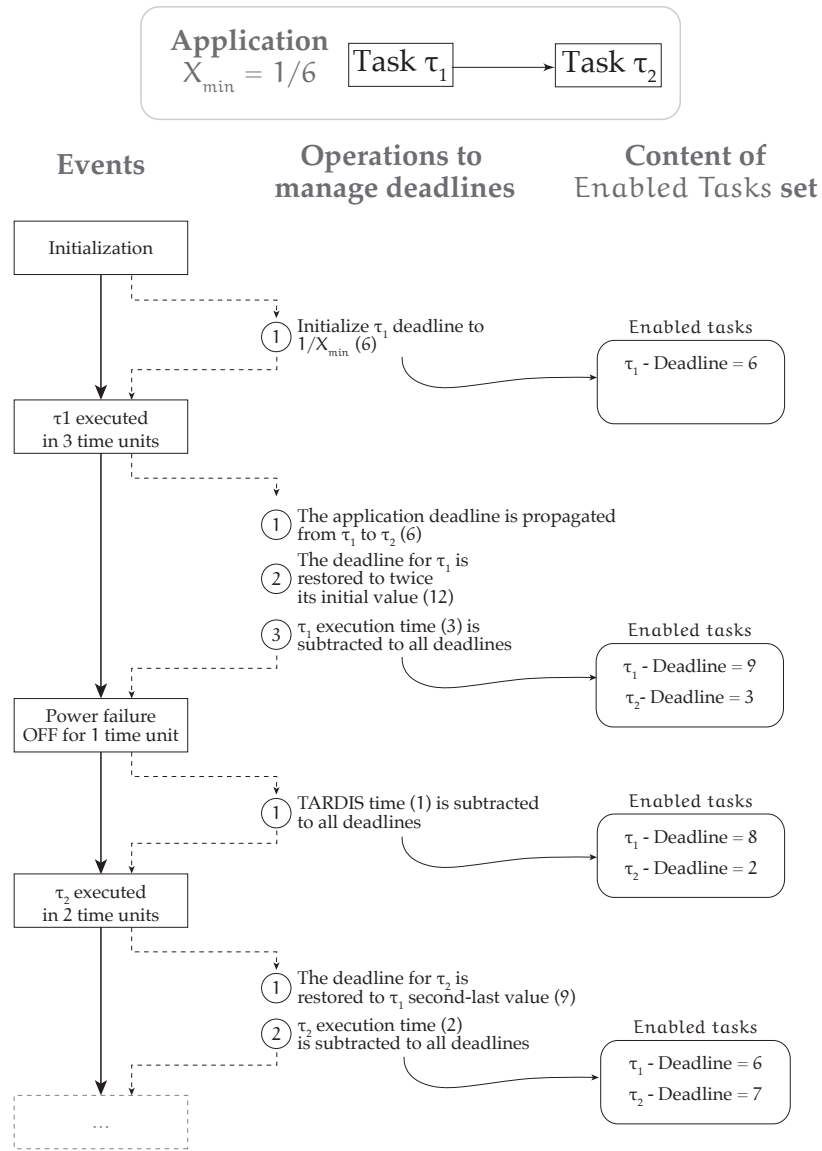


Figure 6.3: Example showing how deadlines are updated in accordance with **Property 3**

After each execution some tasks may become enabled, as the executed task produced some data that satisfies their dependency. For instance, in the previous example τ_1 enables τ_2 . It is important to notice that set of enabled tasks does not necessarily increase its length, but may also shrink after the execution of a task.

Let us consider the configuration shown in Figure 6.4. Task a and Task b are connected by an *at-most-one-write* dependency, while Task b and Task c are connected by an *at-most-one-read* dependency.

Initially the set of enabled tasks is {Task a}. After Task a execution the set becomes {Task b}, since Task a can not be executed until Task b completes due to the semantic of the dependency. After the execution of Task b the set becomes {Task a; Task c}.

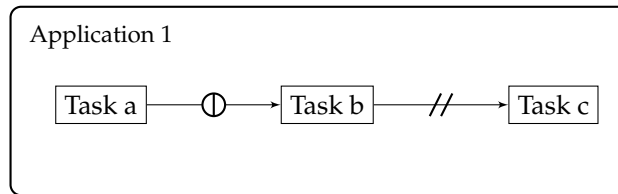


Figure 6.4: Example of an application with data dependencies that cause the shrinking of the enabled tasks set after a task execution. For instance the *at-most-one-write* dependency between Task a and Task b, forces the removal of Task a from the enabled set after its execution. Task a is enabled again after the completion of Task b, in accordance with the dependency semantic.

In fact:

- Task a can be executed because Task b consumed its previous output;
- Task b can not be executed because its previous execution consumed Task a output;
- Task c can be executed because Task b completed.

If Task c is executed next, the set becomes {Task a}, because the dependency between Task b and Task c does not allow multiple reads of the same data.

In the next Section we examine how the runtime scheduler deals with under-performing and over-performing applications.

6.6 REACT TO THROUGHPUT'S DRIFTS

As we saw in Section 6.5, the runtime scheduler manages tasks deadlines with an algorithm that allows to know if an application is running on pace, or if its execution rate is below or over the requested minimum throughput X_{\min} .

In this Section we present the techniques to deal with drifts in application's throughput.

In **Property 1** and **Property 2**, we said that applications are sorted by increasing X_{\min} , and that this value induces a priority: the higher is the requested throughput, the higher is the priority. Moreover, after the initialization, only the application with the highest priority is active. Finally, the scheduler considers one application requirement at a time, activating a new application only if all those that are currently active are on pace.

To deal with mismatches between desired and measured throughputs, we introduce a variable called *slack* for each active application. This variable persists across power failures and stores the advance or delay in application's execution. To compute the slack of an applica-

tion we take the deadline of the final task before its execution and we subtract its execution time.

In the example in Figure 6.3 the slack is 0, since the deadline of the final task Task b, before its execution is 2 and its execution time is 2. The value of the slack accounts also for the inactivity time, since, as described in Section 6.5, their duration is subtracted to tasks deadlines and it propagates through the deadline's update process.

Imagine that an application has a requested throughput of 2 iterations per second. Assuming that all the iterations take the same time, this means that each iteration must complete within half a second, hence the deadline for the application is 0.5s. If the iteration took 0.2s, then, in absence of other applications or power failures, two complete iterations would require 0.4s, corresponding to a throughput of 2.5 iterations per second, which is higher than the request. This means that we can use the slack time of 0.3s for other workload for each iteration, potentially without affecting the throughput of the first application.

If the same application runs in 0.8s, then two iterations would require 1.6s resulting in 0.625 iterations per second, which is below the request. In this case we would have a negative slack of $-0.3s$ for each iteration.

A positive slack means that the application is running ahead of its deadline, hence it would complete more iterations per second, than those requested by X_{\min} . A negative slack corresponds to a delay in application's execution.

The value is cumulative, as the slack value computed at the end of the iteration number i of application A is summed to the value computed at the end of the iteration number $(i - 1)$ of the same application. This means that, if in our previous example the application has a slack value of 0.3s after the first iteration, and during the second iteration it finishes 0.2s earlier, then the slack after this second iteration is 0.5s. On the other hand if the application accumulates a delay of $-0.3s$ after the second iteration, then the slack after this iteration would be $-0.1s$.

Let us call D_i the deadline of the final task before its execution during iteration i , and T_i the execution time of this final task during iteration i . Then, given what we said so far, the slack after iteration i is computed as follows.

$$\text{slack}_i = \text{slack}_{i-1} + (D_i - T_i) \quad (6.3)$$

As we said, a positive slack value accounts for the time that we can spend to run other applications, without violating the requested throughput of the ones with higher X_{\min} . Let us consider Figure 6.5 and let us suppose that the set of active applications, sorted by decreasing X_{\min} is $\{A_1; A_2; A_3\}$. Then, $\text{slack}(A_1)$ is the time we can spend to run A_2 and A_3 without violating $X_{\min}(A_1)$, while $\text{slack}(A_2)$ is the

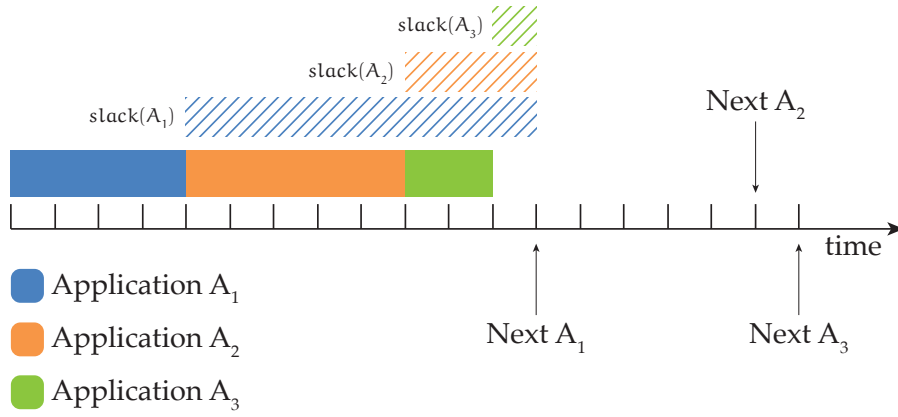


Figure 6.5: Example of slack time computation.

time we can spend to run A_3 without violating both $X_{\min}(A_2)$ and $X_{\min}(A_1)$.

Let us suppose that the slack value of A_2 is greater than the one of A_1 . In this case, if we use all A_2 slack to execute A_3 , and then we go back to execute A_1 , we would miss A_1 's deadline. In fact, the slack of A_1 is the maximum amount of time that we can spend to execute other applications without violating its throughput request. If A_2 slack time is greater, and we use it all to execute other applications, then we would go back to A_1 in a time greater than the slack of A_1 , violating the constraint.

Hence, the slack for an application A_i , except for the initial one A_1 , cannot exceed the slack value of all the applications A_j with $j < i$. Therefore the slack is computed following this equation.

$$\text{slack}(A_i) = \min\{\text{slack}(A_i); \text{slack}(A_{i-1})\} \quad \forall \quad i > 1 \quad (6.4)$$

Let us now examine how the scheduler deals with over-performing applications, or, in other words, with positive slack values.

6.6.1 Managing Over-Performing Applications

Let us consider, once again the example in Figure 6.5 where three applications A_1 , A_2 and A_3 are deployed on the device, with $X_{\min}(A_1) > X_{\min}(A_2) > X_{\min}(A_3)$. The scheduler knows that A_1 must be executed every $1/X_{\min}(A_1)$ time units, hence, at the end of the execution of A_1 it knows how much time can be used to run other applications: the slack time. Within this slack time it starts executing the second application A_2 . The same happens for the execution of A_3 .

In the same example, let us consider the case in which only A_1 is currently active. At the end of its execution the scheduler checks the application's throughput and computes its slack. If it is positive the scheduler can start A_2 execution as before, but before that it must

activate the application. The activation of an application follows the same procedure to activate the initial one, described in Algorithm 6.1. Once the new application is active, the scheduler must also update the activation threshold, as described in Algorithm 6.2, since it may happen that a task included in the newly activated application demands for a higher energy budget.

Within the slack time, the scheduler starts selecting enabled tasks from the next application. Once the slack time is finished, the execution goes back to the previous application.

Property 5

A positive slack allows us to run the next application in decreasing X_{\min} order.

Each application, except the first one, recursively run *entirely* within the slack of the previous one.

It may happen that during the execution a sudden drop in harvested energy causes several power failures, negatively affecting the throughput of the applications. In the next Section we describe how the runtime scheduler deals with under-performing applications.

6.6.2 Managing Under-Performing Applications

Once again, let us consider the usual scenario with three applications A_1 , A_2 and A_3 , where $X_{\min}(A_1) > X_{\min}(A_2) > X_{\min}(A_3)$, all active.

Let us suppose that, at the end of A_1 execution, its performance is checked and it results that the application is under-performing: its measured throughput is below the requests, hence its slack time is negative.

In accordance with **Property 2**, the first technique to address this issue is to decrease the number of running applications, since apparently the system, under the current energy conditions, can not deal with the current number of applications. The system deactivates one application at a time, starting from the last one. So, in our example it deactivates application A_3 , removing all its tasks from the active tasks set. After the deactivation, the voltage threshold is updated to fit the current set of active tasks. In fact, the threshold is set to fit the task with the highest energy demand and, by removing some of them, this value may decrease. Now the system can proceed with just two active applications: A_1 and A_2 . If A_1 can not satisfy the requirements, even after the deactivation of A_3 , A_2 is deactivated too. The system deactivates just one application at a time, and proceeds with the next deactivation only if the performance requirements are still violated.

This behavior is justified by the intent not to perturbate too abruptly, especially given the unpredictable nature of the power source.

Let us now suppose that, after the deactivation of A_3 , the performance of A_1 is still below the request, and that even the deactivation of another application (A_2) does not solve the issue. A_1 is the only active application, so we must implement a different technique to deal with its under-performance.

We remember, from Figure 4.4 in Chapter 4, that by increasing the activation threshold, we increase the number of tasks that can run within a power cycle. As described in Section 5.4, when two tasks with a producer-consumer dependency among them run within the same power cycle, the consumer can access producer's output by avoiding costly read access to **NVM**.

From these observation we derive the second way to deal with under-performing applications. We can try to increase A_1 performance by increasing the activation threshold, in order to minimize **NVM** read accesses. Of course, this increase must be iterative, since we must find the correct balance between the pros and cons of increasing the activation threshold, as described in Chapter 4.

Let us now review how and when the system changes the activation threshold.

- The voltage threshold is set during the initialization phase, to match the request of the tasks in A_1 , as described in Section 6.4;
- the threshold is potentially increased any time a new application is activated, in order to match the demands of the newly added tasks;
- the threshold is potentially decreased any time an application is deactivated, in order to exactly fit the reduced set of tasks;
- the threshold is iteratively increased when an application is under-performing and there are no other applications that can be deactivated.

There are no guarantees that the throughput requirement can be satisfied just by increasing the threshold, since the voltage is bounded by the maximum value supported by the capacitor. If this upper bound is reached and the application is still under-performing, our system does not have any other resource to improve the situation. The developer can either replace the capacitor with one that supports a higher energy buffer, or recalibrate the throughput requirements.

We can summarize the technique implemented to manage under-performing applications with the following property.

Property 6

Given a set of applications $\{A_1, A_2, \dots, A_n\}$, where $X_{\min}(A_i) \geq X_{\min}(A_{i+1}) \quad \forall i \in [1; n]$, when an application A_j measured throughput is lower than $X_{\min}(A_j)$:

- if $j < n$ the system deactivates A_n , applications are deactivated one at a time to avoid abrupt perturbations of the system;
- if $j = n$ the system increases iteratively the activation threshold until, either the performance are satisfied, or the maximum value for the considered capacitor is reached.

Running an application with a pace equal to the application period, does not necessarily guarantee the fulfillment of the throughput requirement. In fact our energy consumption prediction may be inaccurate, or the harvester may be unable to provide enough energy, and power failures may severely affect the throughput. For these reasons we introduced the slack variable, that accounts for a cumulative measurement of the advance or delay in application's execution, including those caused by power failures.

Long off time intervals though, may decrease a negative slack up to a point where the budget earned by running an application faster than its period, can no longer cope with the accumulated delay.

Due to this delay caused by the inactivity, it may happen that an application is locally running with the requested throughput, while globally it is under-performing. Therefore secondary applications are never scheduled, leading to their starvation.

To deal with this issue, we introduce a parameter called Γ , described in the next Section.

6.7 THE Γ PARAMETER

As already discussed, different energy sources have different profiles: some can be characterized as fairly stable, like those provided by solar radiation, some others are more subject to continuous drops and unpredictable stability, like kinetic or RF sources.

Different sources demand different importance to the old slack value, when computing the new one: an inactivity interval, when powered by a solar source, can be viewed as an isolated event and a second drop is not likely to happen in the near future; we can not make the same assumption with an RF source, where power failures are more likely and unpredictable.

Suppose that we are running powered by a solar harvester and that we experience an unexpected drop in provided energy that causes a long inactivity interval. The length of this inactivity interval consistently increases our delay; after the power failure the board switches on again and keeps running stably, given the nature of the solar source. The main application is executing ahead of its deadline, increasing the slack iteration after iteration, but, due to the length of the off time, the slack stays negative. Speculating on the stability of the energy source, we can safely assume that the slack will be back to a positive value in the future, still, depending on the duration of the off period, it may take a long time. We can speed up the process of increasing the slack back to a positive value, by modifying slack computation showed in Equation 6.3. We introduce a parameter $\Gamma \in [0; 1]$. This parameter is an intuitive measure of the source instability: a higher Γ factor means that the source provided energy oscillates at a high frequency, a lower Γ factor corresponds to a source that is less prone to sudden drops.

Given this factor, in Equation 6.5, we propose an updated version of Equation 6.3 to compute the slack. The slack after iteration i is computed as follows.

$$\text{slack}_i = \Gamma \times \text{slack}_{i-1} + (D_i - T_i) \quad (6.5)$$

A low Γ promotes multi tenancy, since it is easier for an application to accumulate slack time, a high Γ maximizes the throughput of the main application, but can lead to starvation of secondary applications.

We said that a lower Γ promotes the activation of new applications and reduces the chances of starvation. In Chapter 8, we propose an evaluation of how the selection of Γ impacts the so called *fairness* of our scheduler.

So, let us now introduce formally this concept of fairness in the next Section.

6.8 SCHEDULER FAIRNESS

The essence of a scheduler is to allocate, with a given logic, a scarce resource. On intermittently powered devices such scarce resource are the clock cycles available for computation before the next power failure.

Thanks to the Γ factor described in Section 6.7 we can modify the scheduling policy:

- by increasing Γ we increase the time needed to overcome a possible delay in application execution, this would limit the activation of other applications, hence maximizing the throughput of the active ones;

- by decreasing Γ we reduce the time needed to overcome a possible delay in application execution, this promotes application's activations, obtaining more balanced throughput among them.

The two aforementioned diverging directions, affect the so called scheduler *fairness*: a measure of the compliance of the obtained allocation to a given optimal resources sharing.

In our scenario, we have the highest fairness when each application receives a share of the total available clock cycles that is proportional to its minimum requested throughput: if the first application should reach a minimum throughput two times higher than the second, then it should receive two times the clock cycles of the other one.

Different fairness indexes have been proposed in literature, for instance: mean and variance; coefficient of variation; Min-Max ratio; normalized distance from optimal; Jain Index. Among those, we selected *Jain Index* proposed by Jain et al. [19], due its intuitive relationship with user perception, and the fact that the output is a continuous number between 0 and 1, that can easily be seen as a percentage: if $y\%$ of the applications are treated fairly and $(100 - y)\%$ are treated unfairly, then the fairness index is $y\%$.

Let us consider the following variables:

- Tp_i is the *measured* average throughput for application i ;
- O_i is the optimal average throughput that can be obtained by the fairest allocation described before;
- x_i is $\frac{Tp_i}{O_i}$
- n is the number of applications.

Then the index is computed with the following equation:

$$\text{Fairness Index} = \frac{(\sum x_i)^2}{n \sum x_i^2} \quad (6.6)$$

On batteryless devices we may not want to maximize the fairness, instead we may try to maximize the throughput of one specific application, that in our system we assume is the one with the highest X_{\min} , potentially starving the others. The Γ factor offers a knob to the developer to decide whether to lean toward fairness, or the maximization of the throughput for the application with the highest minimum throughput request. In Chapter 8 we show an empirical evaluation on the interaction between Γ and fairness.

We have now completed the description of the runtime scheduler. In the next Section we propose an overview of the concepts described so far.

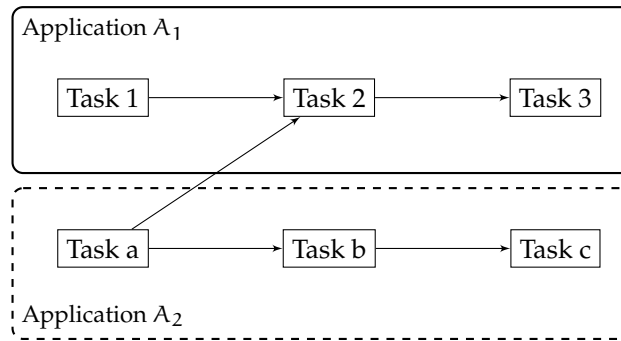


Figure 6.6: Applications DAG for scheduler final overview. $X_{\min}(A_1) = 5$, $X_{\min}(A_2) = 2$

6.9 COMPLETE OVERVIEW

In light of all the concept and property described so far we propose a final overview on the runtime scheduler.

In this overview let us consider two applications A_1 and A_2 . With $X_{\min}(A_1) = 5$ and $X_{\min}(A_2) = 2$ iterations per second. The applications are composed as shown in Figure 6.6.

Applications are sorted by decreasing minimum requested throughput X_{\min} , so the sorted set of applications is $\{A_1; A_2\}$. The scheduler tries to satisfy the throughput requests one application at a time, starting from the one with the highest demand: A_1 .

Applications can be active or deactivated. Initially this first application in X_{\min} order is the only one active, while the others are deactivated. Therefore, the initial set of active applications is $\{A_1\}$. The scheduler activates one application at a time, when *all* the requests of the previous ones are satisfied. In our example A_2 is activated only once A_1 request is satisfied. Activating an application means to schedule for execution all its tasks.

Tasks can be non-active, active or active and enabled. Tasks belonging to deactivated applications are non-active. Tasks belonging to active applications are active. An active task is also enabled when all its incoming dependencies are satisfied. Every time a task becomes active all the tasks in its IN set must be activated too, even if they are included in different applications, since they produce data relevant for its execution. In our example, at startup the set of active tasks is composed by all tasks from application A_1 , plus Task a, that produces data relevant also for Task 2. When A_2 is active too, the data produced by Task a can be used by both Task 2 and Task b.

The set of enabled tasks is initially composed by all the active tasks with no incoming dependencies, in our example this set is $\{\text{Task 1}; \text{Task a}\}$. The execution of an enabled task may enable other tasks as it may satisfy their dependencies, for instance the execution of Task 1 enables Task 2.

Each active task has a deadline. At startup the deadline for enabled tasks is set to $1/X_{\min}$, therefore in our example the initial deadline for Task 1 is 0.2s. The same deadline is set also for Task a, since it initially contributes to A_1 execution. Deadlines are updated as described in Section 6.5. In particular inactivity times are subtracted to all deadlines on wakeup.

The runtime scheduler starts executing enabled tasks one at a time, selecting earliest deadline first. Once again we remember that the set of enabled tasks changes during the execution, for instance initially this set is {Task 1; Task a}, after the execution of Task 1 *and* Task a, the set is {Task 1; Task a; Task 2}, after the execution of Task 2 the set is {Task 1; Task a; Task 2; Task 3}.

The deadline update process propagates the partial execution time and inactivity times along the application task graph, so that it is possible to compute the complete execution time of the application, once its final task is executed.

Let us suppose that A_1 completes in 0.1s, hence 0.1s earlier than its deadline. If the application completed earlier, it means that it is running ahead of its pace and that it would exhibit a throughput higher than the requested one. The difference between this execution time and the application deadline, that is $1/X_{\min}$, accounts for the advance or delay in application execution, and is called slack.

If an application keeps completing earlier than its deadline its slack increases, while any delay decreases this value. If the slack is positive, it means that the application is over-performing and new applications can be activated and executed. To maintain the performance of the already running ones, each new application must run within the slack value of the previous ones.

Every time a task is selected, the runtime scheduler checks if the energy requested for its execution is lower or equal to the current energy budget. If the energy requirement exceeds the budget, the task is skipped and the next one is selected. If no task matches the budget, then the scheduler puts the board to sleep mode until the activation threshold is reached.

As described in Section 6.6.2 the scheduler reacts to drops of application throughputs by reducing the set of active applications, or by increasing the activation threshold. If the throughput of A_1 drops below the request while both applications are active, then A_2 is deactivated. If the throughput of A_1 drops below the request and A_1 is the only active application, then the activation threshold is iteratively increased until either the throughput is correct, or the threshold reaches its maximum value. The activation threshold is increased also if the activation of an application adds to the active task set a task whose energy budget is higher than the one of the currently active tasks. The threshold is decreased if the task with the highest energy

demand is removed from the active task set, due to the deactivation of an application.

The slack of an application accounts for the delays or advances in its execution time. A long inactivity caused by a power failure may produce a long delay and a negative slack with a high absolute value. The slack can increase back to a positive value, by iteratively accumulating advances in execution. If the delay is high, then it may stay negative for a long time, starving secondary applications. To mitigate this effect, the developer can reduce the importance of old delay by setting a discount parameter between 0 and 1, called Γ .

This concludes our description of the proposed dynamic scheduler. In the next Chapter we present some details on the implementation of our solution.

7

IMPLEMENTATION

In this Chapter we present the implementation of our proposed solution. As shown in Figure 7.1 the system relies on a preprocessor that takes as input the description of the application's layout and tasks parameters described in Section 5.2.1. Given these inputs, the preprocessor produces two outputs: a custom include file and the main C firmware code. The .h include file contains a set of macros for task's code, while the main firmware code implements the scheduler.

The overall philosophy that justifies this structure is to reduce as much as possible the complexity of the runtime code, to reduce the scheduler overhead.

The main firmware code that implements the scheduler, is generated automatically based on the specific scenario. For this reason, we can avoid the execution of computationally intensive instruction at runtime, like those that involve graph traversal.

For instance, as stated in Section 6.3, when an application is activated, we must traverse the task's graph to activate all the tasks from other applications that are relevant for its execution. This set of tasks does not change at runtime, as the graph is fixed at compile time. Hence, the computation of this set is performed by the preprocessor, that generates specific C code to perform the activation.

The preprocessor is implemented as a Python code that parses the input files, rebuilds the DAG associated to the deployed scenario, and based on a set of C templates builds the aforementioned outputs.

Task's functions can be implemented as standard C functions, by simply including the macros produced by the preprocessor. The only restrictions are those connected with NVM access, as stated in Section 5.4.

The set of .h include file and .c tasks code and main firmware can be compiled with the standard TIMSP430 compiler without any additional requirement.

The preprocessor can produce a test and debug version of the aforementioned files. This version contains a set of debug instructions that interface with the SIREN cycle accurate simulator, developed by Furlong et al. [11], that we extended to increase its features and improve the support to our solution. The extensions to this simulator are described in Chapter 8.

In Section 7.1 we specify how to describe tasks; in Section 7.2 we present how to map applications DAG to the definition file; in Section 7.3 we describe how to structure task's code; in Section 7.4 we describe the implementation of the scheduler.

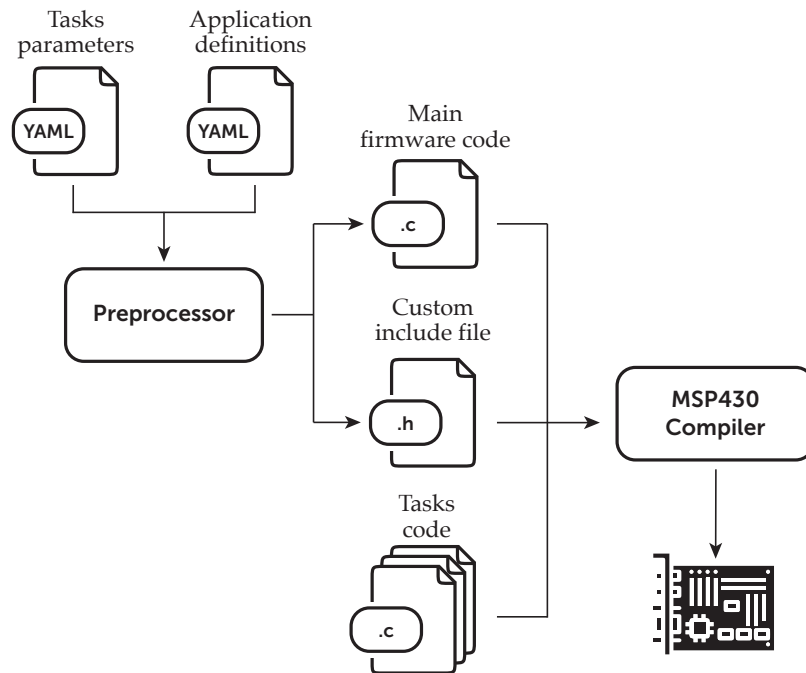


Figure 7.1: Overview of the structure of the proposed solution. The preprocessor builds a custom set of macros and a main firmware code, based on the description of the applications.

7.1 DEFINE TASKS

The developer provides tasks parameters by means of a configuration YAML file. These parameters derive from the description of tasks presented in Section 5.2.1.

In particular the developer must specify for each task:

- a unique task identifier `id`;
- a set of input dependencies `in_set`;
- a reference to its output variable `output`;
- its predicted worst case energy consumption `e_wc`.

Here we present the description of each parameter. Listing 7.1 presents a complete example of a YAML configuration file for the tasks shown in Figure 7.2.

Listing 7.1: Example of tasks YAML file for DAG in Figure 7.2

```
1 TASKS:
2   - id: task_a
3     in_set:
4       - task_id: task_b
5         dependency_type: 1r
6     output:
7       name: a_output
8       type: int
9     e_wc: 81
10  - id: task_b
11    in_set: []
12    output:
13      name: b_output
14      type: float
15    e_wc: 33
16  - id: task_c
17    in_set:
18      - task_id: task_a
19        dependency_type: simple
20      - task_id: task_b
21        dependency_type: 1w
22    output:
23      name: c_output
24      type: int*
25    e_wc: 51
26  - id: task_d
27    in_set:
28      - task_id: task_c
29        dependency_type: simple
30    output:
31      name: d_output
32      type: int
33    e_wc: 16
```

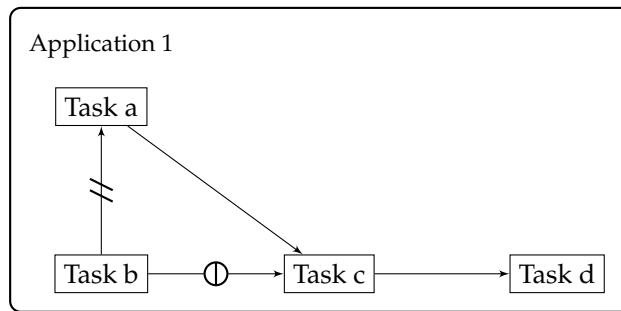


Figure 7.2: Application layout for YAML example in Listing 7.1

id

Unique id for the task, it *must* correspond to the name of the function that implements the task.

in_set

Sequence of incoming dependencies. The sequence can be empty in case the task has no incoming dependency. Each element of this sequence is structured as follows.

task_id Unique id of the task that produces the input data. It must correspond to the `id` field of a task included in the same YAML.

dependency_type Name of the dependency type selected among those described in Section 5.3, represented by the following strings:

simple simple data dependency;

1r at most one read data dependency;

1w at most one write data dependency;

qN add version data dependency, where N is the number of versions that must be stored;

wN window data dependency, where N is the number of elements in the sliding window.

output

Identifier and type of task's output. As described in Section 5.4.1 this value is persisted on NVM by the framework, and will be offered as an input to all the tasks connected by a data dependency. This field has two key-value pairs:

name identifier of the output variable, it must match the identifier of a variable declared in task code;

type output variable type, it must match the type of the variable whose identifier is specified in the name field.

e_wc

Worst case energy consumption prediction in μJ .

The developer provides a second YAML file containing the definition of applications, described in the following Section.

7.2 DEFINE APPLICATIONS

The developer provides a representation of the applications DAG, by means of a YAML file. Each application is also decorated with the corresponding parameters, as presented in Section 5.2.2.

In the rest of this Section we provide the description of each YAML field, while Listing 7.2 presents a complete example of the YAML file representing the applications shown in Figure 7.3.

id

Unique id of the application.

tasks

Set of the unique identifiers of the tasks included in the application. They must match one of the identifiers for the tasks described in tasks YAML.

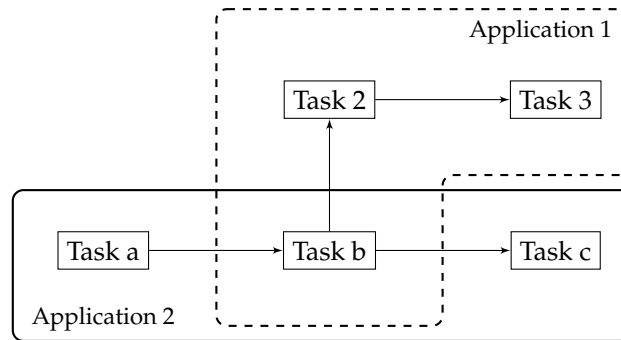


Figure 7.3: Applications layout for YAML example in Listing 7.2

Listing 7.2: Example of applications YAML file for DAG in Figure 7.3

```

1 | APPLICATIONS:
2 |   - id: app_1
3 |     tasks: [task_b, task_2, task_3]
4 |     initial_task: task_b
5 |     final_task: task_3
6 |     x_min: 5
7 |   - id: app_2
8 |     tasks: [task_a, task_b, task_c]
9 |     initial_task: task_a
10 |    final_task: task_c
11 |    x_min: 2
  
```

initial_task

Unique identifier of the application's initial task. It must match one of the identifiers in tasks set.

final_task

Unique identifier of the application's final task. It must match one of the identifiers in tasks set.

x_min

Desired minimum throughput X_{\min} for the application, in terms of iterations per second.

This configuration file concludes the set of inputs for the preprocessor, thanks to these informations the preprocessor builds a set of macros and the scheduler firmware. In the next Section we present the set of macros that allows the definition of tasks code.

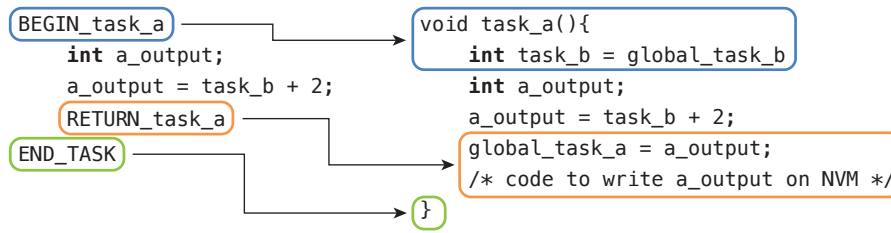


Figure 7.4: Example of macro expansion in task’s code. The instructions are placed directly in the task, instead of relying on library functions, to prevent the overhead of function calls. The identifier of the output variable is specified in the task YAML file.

7.3 CREATING A TASK

Tasks are implemented as C functions. In accordance with the specifications on memory access from Section 5.4, they do not have direct access to *NVM*. So, as shown in Figure 7.4, their input parameters, specified as incoming dependencies in the YAML file described in Section 7.1, are injected through the expansion of a custom macro generated by the preprocessor.

It is important to notice that the developer can use data from its incoming dependencies, through a local variable named after the producer’s id. For instance, in the example from Figure 7.4, Task a can use Task b output, accessing a variable called `task_b`. The local scope of this variable preserves its consistency, unless the value is a pointer. In that case the consistency must be guaranteed by the developer. The preprocessor emits a warning if any of the input variables type is declared as an address.

In Section 5.4, we state that tasks can not directly persist their output data. For this reason, their signature identifies them as void functions, and instead of the standard C return instruction, the developer uses another custom macro that expands into the set of instructions to complete the operations described in Section 5.4.1.

The macro expansion mechanism allows to simplify developer’s code. Moreover, instead of calling library functions that implement the aforementioned operations, the instructions are expanded right into the task’s code, preventing the overhead of a function call.

7.4 SCHEDULER IMPLEMENTATION

The scheduler is implemented as an infinite loop composed by three steps:

1. task selection;
2. task execution;
3. tasks enabling;

4. performance parameters update.

We remind that the scheduler code is generated automatically by the preprocessor, based on the description provided with the YAML files.

Task selection is performed in accordance with the steps described in Section 6.5. In particular each task is associated to a `struct`, stored on `NVM`, that holds all task's parameters, plus the function pointer to their implementation. `ActiveTasks` and `EnabledTasks` sets, described in Section 6.4, are implemented as an array of pointers to the corresponding task struct.

As described in Chapter 6, tasks are selected by earliest deadline first. Hence, the task array is sorted by deadline and the scheduler scans the array from the beginning, until it finds a task that matches the current energy budget.

Once selected, to execute a task, the scheduler simply calls the corresponding function. As stated in Section 7.3, input dependencies are injected into tasks code by the expansion of a macro. This macro adds a variable and initializes it to the value of the global variable storing the input data. In fact, as specified in Section 5.4.2, tasks read input variables from stack. This means that the scheduler implements instructions to copy the value from `NVM` to the global variable, in case a power failure happens between the producer and the consumer.

After the task function completes, the scheduler enables tasks whose dependency are satisfied by task execution. To reduce the overhead, leveraging on the fact that the scheduler code is generated automatically from tasks and applications definition, this operation is performed through a `switch` statement, where each case corresponds to the execution of a task, and contains the instructions to enable the specific tasks that rely on its output.

Since dependencies may be decorated with boolean conditions that control their activations, as specified in Section 5.3, task's execution may be prevented when they are evaluated as false. For instance let us suppose that task A produces a value, task B consumes it, and the dependency among them is decorated by a timeliness condition that prevents the execution of B if A's output is stale. If task B is selected as next task, but the condition is not satisfied, then the scheduler must remove it from the enabled set, and select another task. Once again, to reduce the overhead, this is implemented with a `switch` statement. If a task is involved in a dependency with a boolean condition, than a corresponding case catches this selection and implements instructions to disable the execution, if needed.

Finally, deadlines and slack time are updated.

It is important to notice that, since a power failure may happen during scheduler's execution, all the instructions to update any of the support structures, such as enabled and active task sets, tasks `struct`

Listing 7.3: Algorithm to capture power failures

```

1 |
2 | /* NON VOLATILE VARIABLES */
3 | int resets = -1;
4 | int seen_resets = 0;
5 |
6 | void scheduler(){
7 |     if(seen_resets != resets){
8 |         /* a reset occurred */
9 |         seen_resets = resets;
10 |        /* subtract off time to deadlines */
11 |    }
12 |    /* scheduler code */
13 | }
14 |
15 | int main(){
16 |     resets++;
17 |     if(resets == 0){
18 |         /* initial boot */
19 |     }
20 |     scheduler();
21 | }

```

and slack values are implemented with *two-phase commit*, as described in Section 5.4.1.

As described in Section 6.3, the scheduler dynamically activates and deactivates applications. Once again, thanks to the automatic generation of a custom scheduler code, these operations are performed by specific instructions tailored to the application. For this reason, instead of an inspection of the task graph, to understand at runtime which tasks to deactivate or activate, as a consequence of an application deactivation or activation, the generated code contains the specific instructions based on the specific application.

As described in Section 6.5, on reboot after a power failure, the inactivity time, obtained by implementing SRAM decay mechanism [28], is subtracted to all the deadlines. Moreover, in case of a power failure between a producer and a consumer, the scheduler must restore data from NVM main memory. This means that the scheduler must be aware that a power failure happened. To obtain this information, we use a technique suggested by Maioli et al. [26]. In particular, the scheduler implements the algorithm described in Listing 7.3.

In particular this technique allows to capture both the initial boot, and any power failure, by using a data inconsistency as a source of knowledge. In fact, the increment of the resets variable is the first instruction performed on reboot, and the inconsistency between its value and the variable seen_resets, highlights an unexpected failure during the execution of the scheduler function.

8

EVALUATION

In this Chapter we present the evaluation of our proposed solution.

To properly evaluate our contribution, we need a way to conduct repeatable experiments, and compare the results of our system with those obtained by existing state of the art solutions. In other words, we need a way to reproduce *all* the surrounding conditions so that we can repeat the same experiment with the same inputs, with both our solution and the baseline.

In Transiently Powered Computation (TPC), the concept of surrounding conditions is fundamental: we can compare the performance of two TPC solutions *only* if the devices that execute them are powered by *exactly* the same energy source, with the same provided energy and the same evolution over time. Relying on real energy sources would be an impractical solution, given that we need to replay them whenever we run a test. Therefore, we need:

- a way to describe an energy trace: the energy provided by a source over time;
- an environment that can execute our firmware, exhibiting the same behavior of a TPC device;
- a way to replay a given energy trace to power this environment.

In Section 8.1 and Section 8.2 we present the evaluation environment, and our contribution to an existent simulation tool to make it more suitable to our needs; in Section 8.3 we describe the evaluation scenario; in Section 8.4 we present the baseline against which we compare the performance of our solution; in Section 8.5 we detail the metrics for the evaluation and, finally, in Section 8.6 we present the results of this process.

8.1 EVALUATION ENVIRONMENT

The target platform for our evaluation is a Texas Instruments board powered by a MSP430 family MCU. In particular, our reference MCU is MSP430FR6989 [27], mounted on an MSP-EXP430FR6989 development board.

The platform has the following specifications:

- 16-bit RISC architecture;
- up to 16MHz clock, 1MHz in our simulation;
- minimum supply voltage of 1.88V;
- 2KB of volatile [SRAM](#);
- 128KB of Non Volatile Memory, implemented with Ferroelectric RAM ([FRAM](#)) technology.

The device is powered by the energy buffered in a 47 μ F capacitor that supports a maximum voltage of 5V. We selected this platform because its [MCU](#) is widely used in literature as a reference for evaluation.

As we discussed so far, the behavior of transiently powered devices is highly dependent on harvested energy. To understand the performance of our proposed solution we need an efficient way to conduct repeatable experiments. Hence, we need a way to record and replay energy supply.

Ekho, a tool proposed by Hester et al. [13], allows to record and emulate energy harvesting conditions, using the abstractions of I-V surfaces. Figure 8.1 shows an example of these surfaces. They are made up of I-V curves, each of them represents how an energy harvester will behave for a given load.

I-V surfaces provide a digital representation of harvesting conditions, and they provide a general energy harvester abstraction that can be used by a simulator — Hester et al. [13]

There are various simulator for this class of devices, such as MSPSim [9], TOSSim [23] and SIREN [11]. Among these we selected SIREN as it is the only one that allows to simulate the actual energy harvesting environment from recorded traces.

SIREN is based on MSPSim and it simulates at instruction level, running an unmodified target platform firmware against different energy traces. To better support the experiments conducted to evaluate our proposal, and to support all the features needed by our scheduler, we extended SIREN, as presented in the next Section.

8.2 EXTENDING SIREN SIMULATOR

SIREN is based on the MSPSim [9] instruction level simulator and both of them are written in Java. Figure 8.2 shows an overview of the simulation environment, and how it integrates in the workflow for our proposed solution, described in Chapter 7.

The simulator takes as input the firmware binary file and the representation of the energy trace. Once the simulation is complete, the developer can understand the performance of the system from two

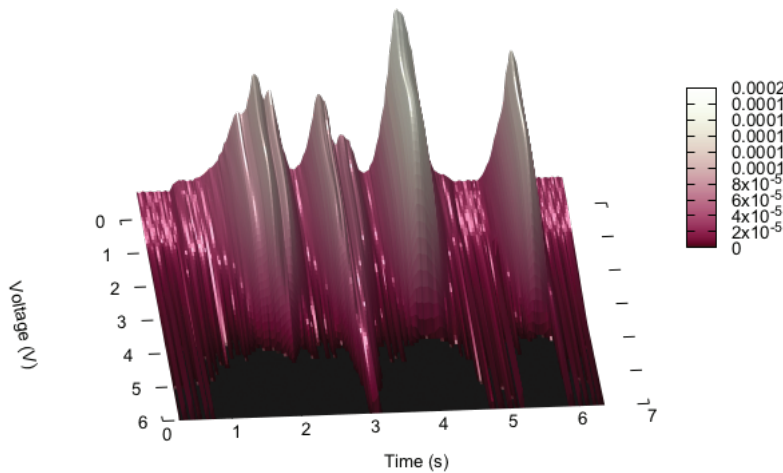


Figure 8.1: Figure taken from Furlong et al. [11]. A solar IV surface generated from an IXYS solar cell exposed to a lightbox. An IV surface captures all possible harvesting scenarios. Each possible harvesting current (I) for the supply voltage (V) over time are shown.

outputs: a log of the simulation, and the trace of the capacitor voltage throughout the simulation.

In this Section we provide an overview on SIREN main components, and on our improvements to the simulator. In particular, we focus on the classes shown in Figure 8.3. CPU class abstracts the core simulator that implements an MSPSim instruction level simulator within SIREN. Its features are enriched by the rest of the classes shown in Figure 8.3 that allow an energy aware simulation.

The simulator decompiles the binary file and starts the execution machine code instructions. A simplified version of this execution loop is presented by the sequence diagram in Figure 8.4.

In SIREN, the execution of each machine code instruction is speculative. First the CPU executes it, then it checks if the number of clock cycles spent for its execution fits within a power cycle. If so, the execution proceeds with the next instruction, otherwise the board resets due to a power failure. This execution model keeps memory and registers consistent with what would happen in a real execution. In fact, in case of a power failure the results of an instruction are lost, since both volatile memory and registers are zeroed by the reset, unless the instruction writes these results to NVM. In that case, each machine code instruction that alters the NVM writes one byte at a time, and this behavior is consistent with the charging pump mechanism, described in Chapter 2, that guarantees the completion of an NVM word write in presence of a power failure.

Our first extension to the original SIREN is the implementation of a simulation of TARDIS mechanism [28]. To simplify the device reset

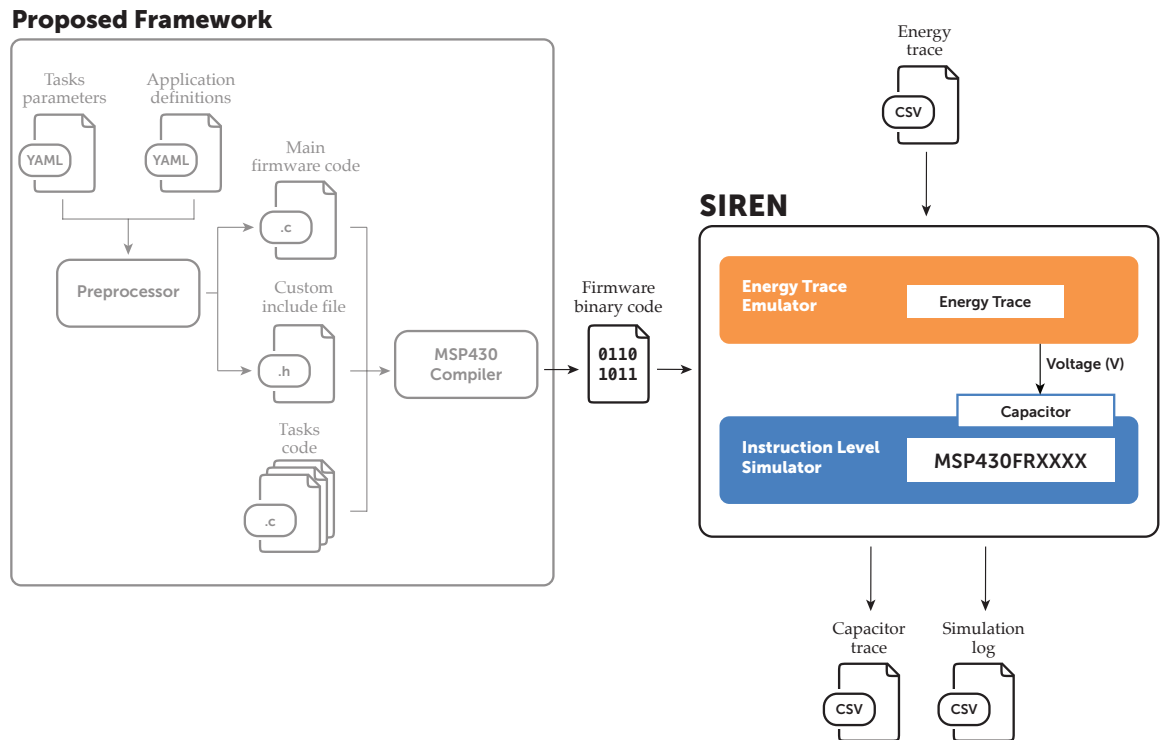


Figure 8.2: Overview of SIREN. SIREN receives as input the binary code produced at the end of the process described in Chapter 7, together with the energy trace. It produces a trace of the capacitor voltage throughout the simulation, and the log of the simulation to understand applications performance.

procedure during the simulation, we refactored SIREN code to extract from the CPU class the methods related to this activity, creating a `ResetManager`. When a power failure happens, the `ResetManager` persists the duration of the off time interval on a specific register that does not lose its state on reset. The persisted value has a resolution that mimics the one offered by a TARDIS. The scheduler can access this value, similarly to what would happen with the aforementioned technique.

We said that the capacitor simulator checks if the execution of the current instruction is compatible with the current energy budget. Let us discuss the details of this process that simulates a buffering capacitor of arbitrary capacity, that stores the energy coming from an energy trace.

8.2.1 Capacitor Simulator

As we discussed so far, SIREN capacitor simulator checks, after each machine code instruction, if the execution of such instruction is compatible with the buffered energy. This can not be done by simply checking if the energy spent to execute a given number of clock cycles

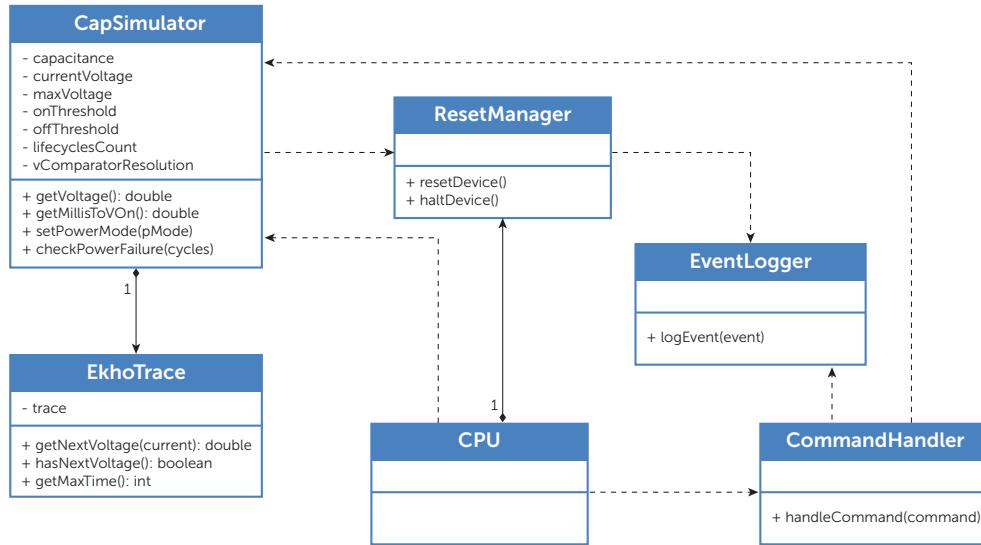


Figure 8.3: SIREN UML class diagram. CPU class abstracts the core simulator that implements a MSPSim instruction level simulator within SIREN. Its features are enriched by the rest of the classes to allow an energy aware simulation.

is lower than the energy that is currently buffered, as during these cycles the capacitor receives energy from the harvester. In fact, the capacitor is placed as a decoupling capacitance in parallel with the harvester, as shown for instance in Figure 4.1 from Chapter 4.

Depending on the harvester voltage, the capacitor either discharges or not. If the harvester’s voltage is higher than the current value for the capacitor, then the capacitor charges as in an RC circuit where the device has an equivalent resistance that depends both on the power state of the device, and on its load. Otherwise, if the harvester has a voltage that is lower than the capacitor’s one, the buffer discharges through the RC circuit, with a minimum value that is limited by the harvester’s voltage.

In our simulation the energy trace has a resolution of 1ms, meaning that the harvester’s voltage does not change within a millisecond. Algorithm 8.1 describes how the simulator updates the capacitor at each machine code instruction. When the capacitor discharges, the spent energy is subtracted from the buffered value and the voltage is updated following Equations 8.1 and 8.2, where E_s is the energy spent to execute the clock cycles, and V_h is the harvester’s voltage.

$$E = \left(\frac{1}{2}CV^2\right) - E_s; \quad (8.1)$$

$$V = \max\left\{\sqrt{2\frac{E}{C}}; V_h\right\} \quad (8.2)$$

On the other hand, when the capacitor charges, its new voltage is computed following Equation 8.3, where, again V_h is the harvester’s voltage.

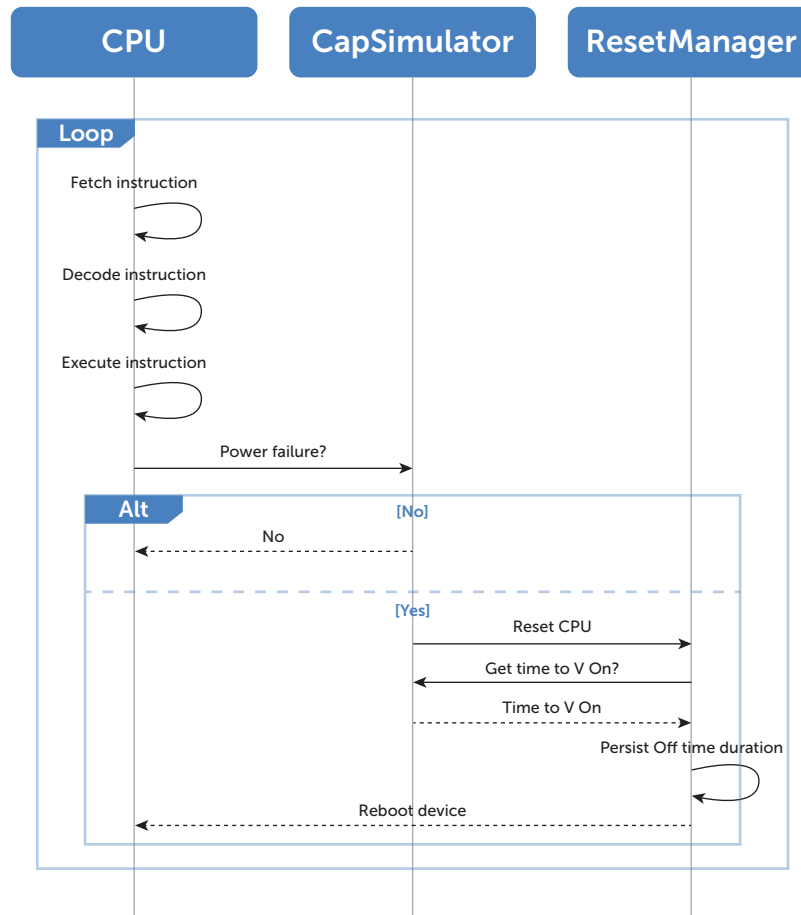


Figure 8.4: Sequence diagram for SIREN main execution loop

$$V = V_h + (V - V_h) e^{-\frac{t}{RC}} \quad (8.3)$$

During this process, the simulator updates a log that allows to trace the capacitor voltage throughout the simulation.

SIREN provides a debug interface between firmware and simulator through a C instruction called `siren_command`. This instruction allows the firmware to send a formatted string to the simulator. As we describe in the next Section, we extended this interface to support commands specific for our proposed solution.

8.2.2 Extended SIREN Commands

The original SIREN, supports `siren_command`, a custom C function that writes a formatted string to a section of memory outside the bounds of the MSP430's memory. The formatted string corresponds to a command and its arguments. In particular it follows the syntax "`<COMMAND>:<ARGS>`", where `<COMMAND>` is the command name, while `<ARGS>` is a substring that holds command's arguments and its format depends on the specific command. The `CommandHandler` class, shown

Algorithm 8.1. Capacitor update**Input Variables:**

- `clockCycles`: number of clock cycles to execute the last machine code instruction;
- `energyPerCC`: energy spent to execute 1 clock cycle;
- `cpuFrequency`: current CPU frequency.
- `current`: load current.

Global Variables:

- `voltage`: current capacitor voltage;
- `millisecondFraction`: microseconds left for the current ekho voltage step of 1 millisecond;
- `vOff`: off threshold.

Body:

```

1  microseconds = [clockCycles/cpuFrequency] * 1000000
2  timeLeft = microseconds
3  while timeLeft ≥ 0 do
4    if millisecondFraction == 0 then
5      millisecondFraction = 1000
6      if ekhoTrace.hasNext() then
7        vSupply = ekhoTrace.getNextVoltage(current)
8      else
9        closeSimulation()           ▷ The trace is over, finish simulation
10     resistance = getResistance(current)
11     time = min(millisecondFraction, timeLeft)
12     execCC = [ time * 1e-6 * frequency ]
13     if vSupply < voltage then      ▷ The capacitor discharges
14       spentEnergy = energyPerCC * execCC
15       consumeEnergyAndUpdateVoltage(spentEnergy) ▷ Equations 8.1 and 8.2
16       if voltage ≤ vOff then
17         resetManager.resetDevice()
18     else
19       chargeCapacitor(time)       ▷ Equation 8.3
20     millisecondFraction -= time
21     timeLeft -= time

```

in the UML diagram in Figure 8.3, is responsible for the execution of the correspondent command.

It is important to notice that this function does not impact the energy buffer, as the simulator intercepts its context and does not account for the time and energy spent for its execution.

The current implementation of SIREN only includes the `PRINTF` command, that prints to simulator's standard output the `<ARGS>` string. To better support our proposed solution, and include the functions that we need for our scheduler implementation, we extend the `siren_command` infrastructure, with the following commands.

SET_VON:<on threshold> it allows to set the activation threshold, in accordance with the scheduler logic presented in Chapter 6. This command can be also seen as an implementation of the hardware technique proposed by Gomez et al. [12]. They describe an hardware-based approach in which an energy management unit

builds up charge to a predefined energy level, generating energy bursts predictably, even under variable harvesting conditions.

LOG_EVENT:`<event description>` it adds an event to the simulation log, each event is composed by a description, a timestamp, and the current voltages of both the capacitor and the harvester.

RESET:`<string>` it allows to ask for a device reset from the firmware. It produces a log event whose description is the argument string. This command can be used for debug reasons, for instance to test memory consistency across injected reset points.

In general, prior to our extension and refactoring activity, the communication between firmware code and simulator was one way, as the `siren_command` function only allowed to pass log messages. Thank to our work, the `CommanHandler` is able to alter the simulation and interact with memory and CPU, based on the received command, resulting in a more powerful simulation environment.

8.3 EVALUATION SCENARIO

In this Section we describe the scenario that we consider for the evaluation of our proposed solution. Following this scenario, we deploy a set of applications to address real problems that batteryless, energy harvesting systems can solve. The scenario has two applications deployed on the same board. The first one implements a sense-compute-operate loop, the second is in charge of compressing and sending data to a data sink; both of them encompass multiple tasks. This scenario is a variant of the one used for the evaluation of Mayfly [16].

Exercise Recognition

A wrist-worn wearable device equipped with an accelerometer can be used to track exercises. By discarding the batteries, the wearable is easier to wear, and does not have to be taken off to charge, therefore we increase the device effectiveness.

The activity recognition application samples a sliding window, filters it, extracts features, classifies and eventually operates some actuator depending on the classification output.

Accordingly to Hester et al. [16] activity recognition with intermittent devices has been undertaken successfully [7, 30], and provides an interesting application space for energy harvesting.

To implement this scenario, we deploy on the board the following applications.

Application 1 it is the main application and performs sensing, filtering, classification and actuation thanks to the following tasks:

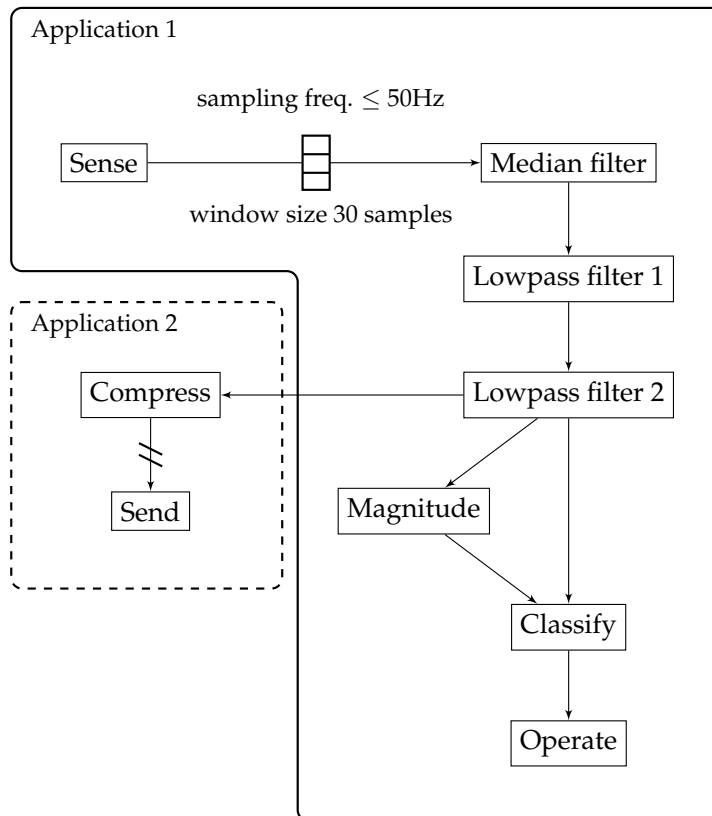


Figure 8.5: Dependency graph for the evaluation scenario. This graph presents the dependencies between tasks, following the graphical conventions described in Section 5.3.

Sense reads data from the accelerator with a minimum inter sampling delay of $1/50$ s;

Filters a median and two low pass filters, they work on a 30 samples size window and filter out noise;

Magnitude extracts a feature from the window;

Classify performs the classification based on the filtered window and the extracted feature;

Operate switches on a led depending on the result of the classification task.

Application 2 an accessory application with a minimum requested throughput lower than Application 1, it is deployed for logging purposes and it is composed by the following tasks:

Compress re-samples the filtered data to compress them;

Send sends the compressed data to a data sink.

The execution flow of each task does not depend on the input data. Figure 8.5 presents the dependency graph for the described scenario.

Following the steps described in Chapter 7, we build a firmware that includes the code for the aforementioned applications and the dynamic

scheduler. The board is powered by replaying different energy traces recorded from a real harvester. The execution is simulated with the extended version of SIREN described in Section 8.2.

8.4 EVALUATION BASELINE

Among the task-based solutions, described in Chapter 3, we select Mayfly [16] as a baseline. Its system is based on a dependency graph similar to ours. Moreover, like our solution, it supports time related requirements.

To produce a correct comparison, we added to Mayfly the ability to support multiple applications and the dependencies types described in Section 5.3. In particular, the extended Mayfly static scheduler receives the dependency graph at compile time and selects a static schedule compliant with the dependencies. Moreover, the baseline static scheduler is able to address throughput requirements, selecting at compile time a schedule that interleaves applications in order to meet the deadlines derived from the requested minimum throughputs. Though, given its nature, a static solution can not react to the unpredictability of the real life energy sources.

For instance, let us suppose that two applications are deployed on our system: application A_1 with $X_{\min}(A_1) = 2$ iterations/s, and application A_2 with $X_{\min}(A_2) = 1$ iteration/s. To satisfy these requests, the baseline solution must produce a static schedule in which A_1 is executed twice the time than A_2 . Still, since it can not adjust the schedule, the workload, or the activation threshold at runtime, it can not prioritize the execution of A_1 in case the energy is not enough to run both applications with the required throughputs. For this reason, we expect that the baseline exhibits worst performance in terms of throughput for A_1 compared to our solution, especially when powered with energy that are not able to provide enough energy.

We execute this statically scheduled firmware on the same extended version of SIREN, described in Section 8.2 against the same energy traces.

8.5 OUTPUTS AND METRICS

As shown in Figure 8.2, the extended version of SIREN produces the following outputs:

- a trace of the capacitor voltage during the simulation;
- a timestamped log of the simulation.

Thanks to the simulation log, we are able to reconstruct all the decision that the scheduler takes during the simulation, in terms of tasks selection and application activation or deactivation.

With the aforementioned outputs we can extract the following metrics.

- Average values and variances of applications throughput.
- A measure of scheduler correctness, defined as the sum of time intervals during which the minimum throughput of an application is satisfied, over the complete simulation timespan. As mentioned earlier, the main application is the one with the highest X_{\min} that, as stated in Section 6.2, we assume is the one implementing the most relevant functions of the deployed solution. Other applications are activated only once the throughput requirements of the main one are satisfied. Given that, in our evaluation we measure the correctness value only for the main application, because it is the only application that is always active.
- Given a generic application application A , the difference between its measured average throughput $X_{\text{avg}}(A)$ and its minimum desired throughput $X_{\min}(A)$

$$\Delta_t(A) = X_{\text{avg}}(A) - X_{\min}(A)$$

- Wasted energy per application $E_w(A)$, defined as follows:

$$E_w(A) = \begin{cases} 0, & \text{if } \Delta_t(A) \leq 0 \\ \frac{E_{\text{tot}}(A)}{I(A)} \Delta_t(A), & \text{otherwise} \end{cases} \quad (8.4)$$

where:

- $E_{\text{tot}}(A)$ is the total energy spent by application A ;
- $I(A)$ is the number of complete iterations of application A during the simulation.

This metric intuitively represents the amount of energy that the system consumes to execute unrequested iterations of an application, obtaining a throughput that exceeds developer's request, instead of running additional workload.

- The fairness, computed with Equation 6.6 from Section 6.8.

With these metrics, we want to demonstrate that the proposed dynamic scheduling solution produces a scheduler correctness value comparable to the static one, minimizing wasted energy and reacting to power failures to reach the requested minimum throughputs.

8.6 EVALUATION RESULTS

In the following Sections, we present the results of the evaluation experiments on the scenario described in Section 8.3, whose dependency graph is depicted in Figure 8.5.

The simulator runs with the following input parameters:

Capacitance 47 μ F;

Maximum capacitor voltage 5V;

X_{min}(Application 1) 3 iterations/s;

X_{min}(Application 2) 1 iterations/s.

Capacitor parameters are obtained from the simulation of Hibernus [2].

In particular, in Section 8.6.1 we run a simulation powering the board with a stable solar energy trace; in Section 8.6.2 the trace is derived from a radio frequency harvester; in Section 8.6.3 we prove the reactive nature of the proposed scheduler, running a simulation with a solar power trace that exhibits sudden drops in provided energy. The traces are derived from the evaluation of EPIC [1].

Section 8.6.4 presents a discussion on the results of different selections of the parameter Γ , described in Section 6.7, and its interaction with the fairness index.

In Section 8.6.5, we introduce the concept of scheduler stability and we show the results of an experiment conducted to prove this feature.

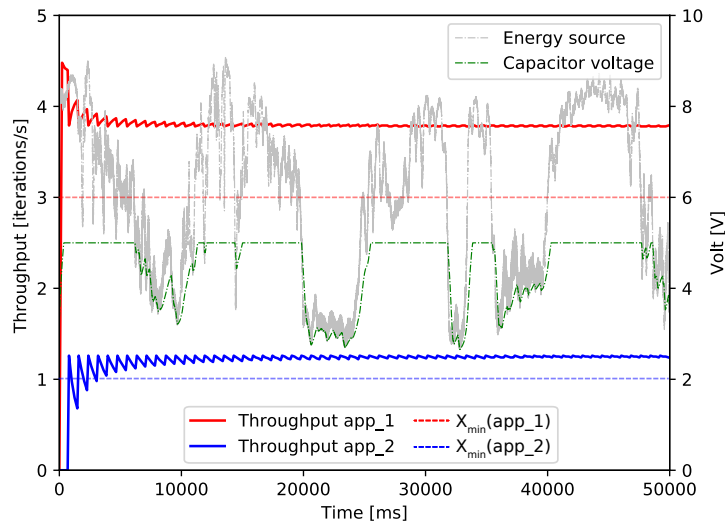
8.6.1 *Stable Energy Source*

With a stable source that keeps the board running with no power failures, we prove that the scheduler satisfies the requested throughput for both **Application 1** and **Application 2**. To prove our claim we run a simulation against a trace collected from a solar harvester. Figure 8.6 shows the results of the experiment.

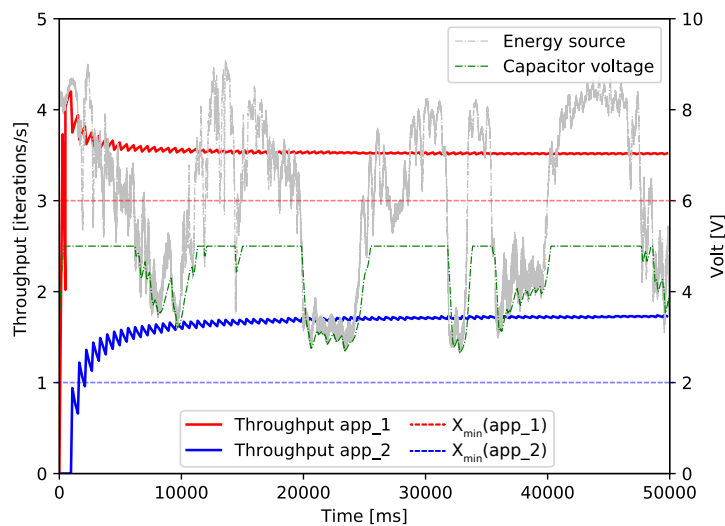
With this trace the capacitor voltage is always higher than 1.88V: the minimum value required to sustain computation. In particular, the voltage is in the interval [5V; 2.68V], where 5V is the maximum value supported by the capacitor, and 2.68V is the minimum value reached during the simulation, after 32897ms from the start.

In this steady computation scenario, both the static scheduler and the dynamic one are able to satisfy the requests in terms of minimum throughput: 3 iterations/s for **Application 1** and 1 iteration/s for **Application 2**.

The proposed dynamic solution after 268ms presents a drop of **Application 1** average throughput, from 3.73 iterations/s to 2.02 iterations/s. This fluctuation is caused by the initial activation phase



(a) Static scheduler



(b) Proposed dynamic scheduler

Figure 8.6: Comparison between static and dynamic scheduler with a stable solar energy source.

of **Application 2**, given **Application 1** over performance. In fact, 3.73 iterations/s is higher than the requested value of 3 iterations/s. This sudden throughput fluctuation lasts only one application iteration ($\approx 400\text{ms}$) and the value is back to an average of 4.03 iterations/s at 496ms. This second value, higher than the requested one, is progressively lowered by the interleaving with the now active second application, decreasing to a stable value of 3.52 iterations/s. With this particular trace, the second application is never removed after the initial activation. In the following tests we show different experiments with different traces, where secondary applications are deactivated to address under performing main applications.

Due to the model described in Section 6.6.1, **Application 2** reaches a higher throughput compared to the one reached with the static

Table 8.1: Mean throughput, variance, correctness and fairness obtained when powered by a stable source with no power failures. The table shows the metrics for the main application **Application 1**.

APPROACH	$\chi_{\text{avg}}(\text{APP. 1})$	VARIANCE	CORRECTNESS	FAIRNESS
Static scheduling	3.80	0.01	100%	0.99
Dynamic scheduling	3.14	0.02	99.5%	0.96

approach. In fact, the scheduler keeps selecting tasks from the second application within the slack time accumulated by the first application.

As we said before, we define *scheduler correctness* the time interval during which the minimum throughput is satisfied, divided by the total simulation time. Due to the aforementioned short fluctuation of the main application throughput, the correctness of the dynamic scheduler for the main application is slightly affected and it is equal to 99.5%, while the static one is 100%.

Table 8.1 summarizes the results. In that Table, and in the following ones as well, we focus on the metrics for the application with the highest minimum requested throughput: **Application 1**. As described in Chapter 6, we assume that the higher is the throughput request, the higher is the importance of the application, and the scheduler prioritize its execution. Nevertheless, all the charts show the throughput obtained during the simulation for all the application deployed on the device.

For what concerns threshold management, in this experiment the dynamic scheduler changes V_{on} only once. In particular, it happens at millisecond 268 when the second application is activated and, in accordance to what stated in Chapter 4, it is done to match the higher energy requirement of its tasks.

8.6.2 Underpowered Execution

In case of continuous power failures, thanks to its dynamic nature, the scheduler can maximize the throughput of the main application (i.e. **Application 1**), preventing the execution of the second one. To prove this claim we run a simulation powering the board with a highly unstable source recorded from a radio frequency harvester.

Figure 8.7 presents the results of the simulation.

With such an unstable energy source, neither the static approach, nor the proposed one successfully reach the requested throughput for **Application 1** (3 iterations/s), because of the continuous power failures and inactivity intervals during capacitor charge.

Still, by selecting a Γ value that leads to the starvation of the second application, the dynamic version can obtain an average throughput over 43% higher than the one obtained with the static solution. The

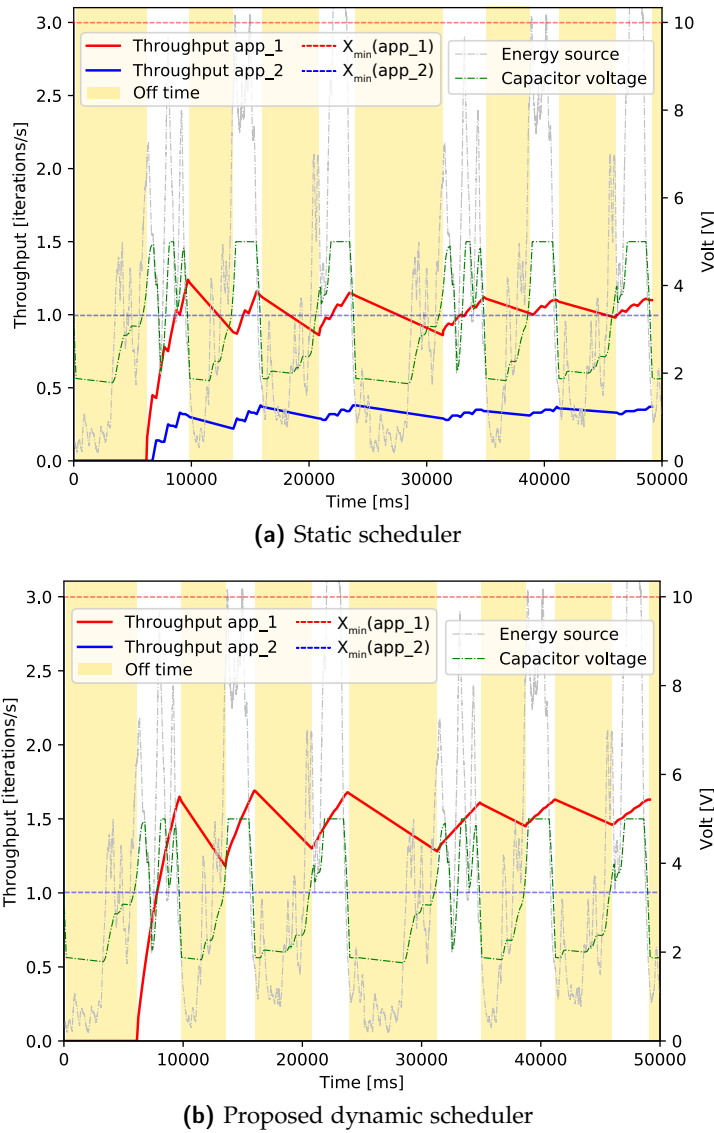


Figure 8.7: Comparison between static and dynamic scheduler with a highly unstable RF energy source.

static approach can not respond to energy shortage by dynamically changing the workload at runtime.

In particular, with a selection of $\Gamma \geq 0.6$ the second application is never activated due to the delay accumulated by the main one. Any selection of the parameter higher than this value simply increases this delay, but does not produce an increase in terms of throughput, whose growth is bounded by the occurrence of continuous power failures. On the contrary, a selection of $\Gamma < 0.6$, reduces enough the weight of previous delays to allow the activation of the second application. Starting from $\Gamma = 0$ up to $\Gamma = 0.5$, the second application is enabled at different times and its activation is delayed the more we increase Γ . In particular, with $\Gamma = 0$ the second application executes its first iteration after 7307ms, while with $\Gamma = 0.5$ after 16016ms.

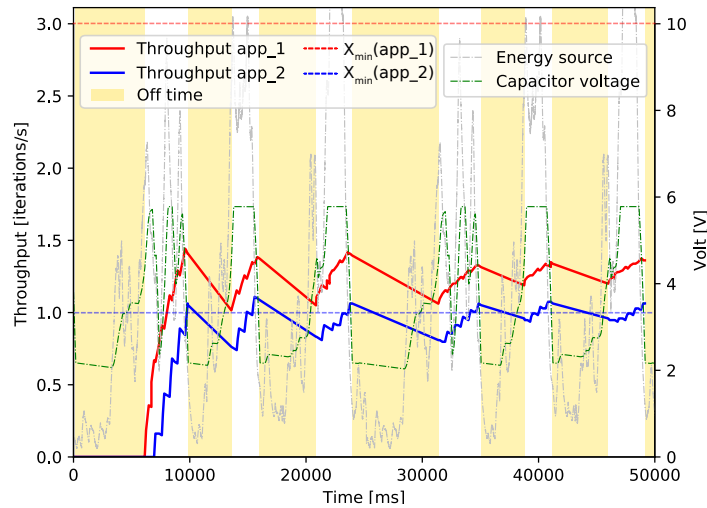
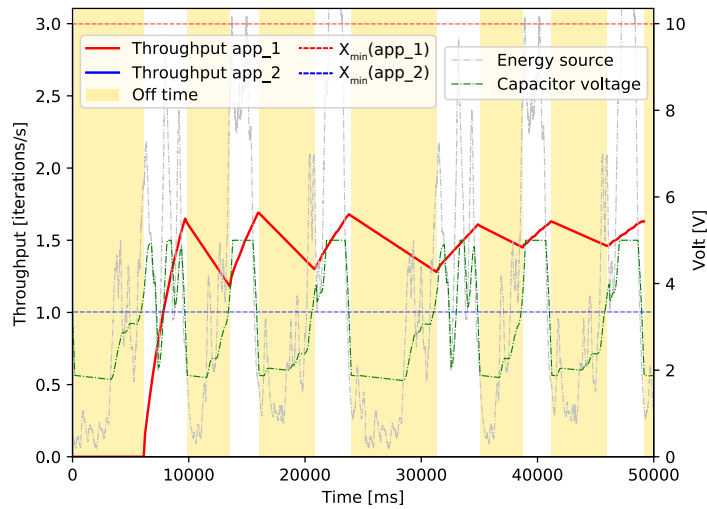
(a) Dynamic scheduler with lowest Γ (b) Dynamic scheduler with Γ selection that maximizes first application throughput

Figure 8.8: Different selections of Γ parameter can lead to different performance with the same energy source.

As we stated in Section 6.7, Γ can be seen as an intuitive measure of the source instability: a higher Γ means that the source provided energy oscillates at a high frequency. By lowering Γ to its minimum value, that corresponds to a source that is less prone to sudden drops, we activate the second application as early as possible, therefore granting it a longer execution timespan. This maximizes its average throughput and increases fairness, at the expense of the main application throughput performance. Still, even with a Γ selection equal to 0, the average throughput reached by the main application (1.05 iterations/s) is slightly higher than the one reached with the static scheduler (0.99 iterations/s).

Figure 8.8 highlights the differences in terms of throughput between a selection of $\Gamma > 0.6$ and 0. A more in depth analysis of the role of

Table 8.2: Mean throughput and fairness with design-time, runtime with Γ that maximizes App 1 throughput, and with lowest Γ parameter.

APPROACH	$X_{\text{avg}}(\text{APP. 1})$	VARIANCE	FAIRNESS
Static scheduling	0.99	0.03	0.99
Dynamic with $\Gamma > 0.6$	1.42	0.08	0.50
Dynamic with $\Gamma = 0$	1.05	0.04	0.92

the Γ parameter, and in particular its relationship with the scheduler fairness, is presented in Section 8.6.4.

With this energy source, the correctness is 0% for both the static and dynamic scheduler, given that neither is able to guarantee the minimum throughput for **Application 1**.

Table 8.2 summarizes the results.

By looking at chart from Figure 8.7b, and in particular by considering the voltage value at the intersection between the end of a yellow area and the capacitor voltage green dashed line, we can see that the activity interval, progressively starts at higher capacitor voltages. This happens because, in accordance with the logic presented in Chapter 4, the scheduler progressively increases the V_{on} because:

- the throughput of **Application 1** does not match the requested one;
- **Application 1** is the only one running and there are no other applications to deactivate.

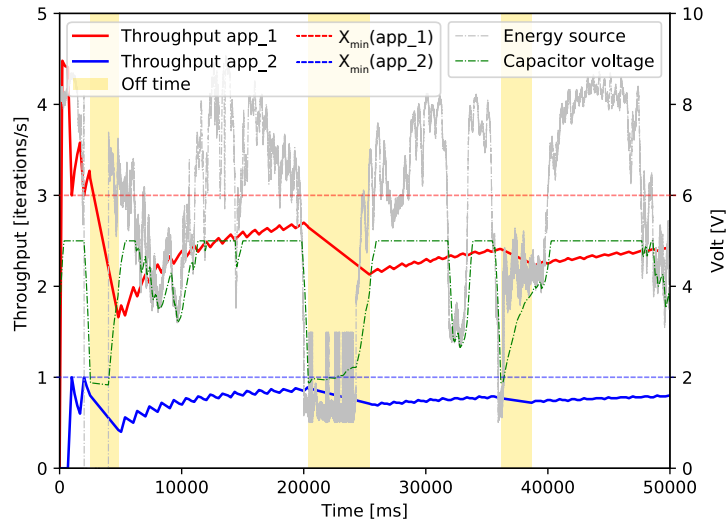
8.6.3 Fairly Stable Source With Energy Failures

Even with a stable source we can not guarantee the absence of power failures. In presence of source instability, we highlight the importance of a dynamic approach, showing the resilient behavior of our proposed scheduler that successfully reacts to sudden drops.

To sustain this claim, we run a simulation with a trace recorded from solar harvester that exhibits the aforementioned pattern. Figure 8.9 presents the results of this simulation.

In presence of power failures the throughput of the applications is severely affected. For instance, the first power failure causes a drop in the performance of the main application, from 3.27 iterations/s to 1.66 iterations/s in the static environment, from 4.11 iterations/s to 2.07 iterations/s with the dynamic scheduler.

With a static scheduler, the second application is active since the beginning of the experiment, completing its first iteration at millisecond 1000. With the proposed approach **Application 2** is activated after 1179ms from the beginning and it completes its first iteration at millisecond 2433. This causes the difference between the aforementioned



(a) Static scheduler

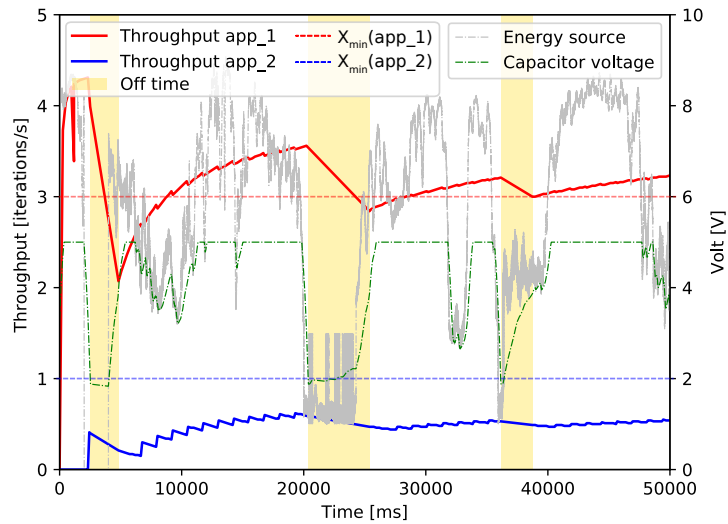
(b) Proposed dynamic scheduler with $\Gamma = 0.4$ and fairness = 0.89

Figure 8.9: Comparison between static and dynamic scheduler with a fairly stable source with sudden drops in provided energy.

throughputs of the first application: the absence of a second application stealing execution time during the initial milliseconds, before the first power failure, allows the main application to reach a higher average throughput.

Figure 8.10 shows a detail of the initial 5000 milliseconds of the simulation results presented in Figure 8.9b. In particular, with the proposed scheduler:

- at millisecond 951 the first application registers an average throughput of 4.21 iterations/s, that triggers, in accordance with the evaluation of the cumulative slack, the activation of the second application;

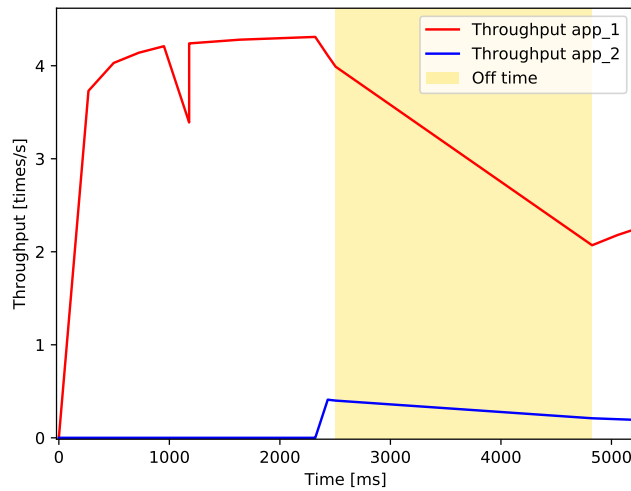


Figure 8.10: Detail of the initial 5000 milliseconds of the simulation results shown in Figure 8.9b

- the activation phase of **Application 2** causes a drop of the first application throughput to 3.39 iterations/s;
- the selected Γ ($\Gamma = 0.4$) produces a slack value high enough to support the execution of the second task of the second application at millisecond 2433, right before the first power failure that happens at millisecond 2504.

Once again, the activation of the second application causes an increase of V_{on} because the scheduler must match the higher energy demands of its tasks.

Going back to the chart shown in Figure 8.9b that presents the result of the complete simulation, we see that the board resumes computation, after the power failure, at millisecond 4825. The new average throughput for **Application 1** is 2.07 iterations/s, this causes a stop in **Application 2** execution and a reset of the activation threshold to its initial value, in accordance with the logic of the proposed scheduler. The throughput steadily increases reaching the value of 2.7 iterations/s at 6655ms. The static scheduler can not change the rate of execution of the second application, therefore at the same time in the experiment (4825ms) the throughput of the first application reaches a lower value of 2.02 iterations/s.

In the dynamic scheduler simulation, the execution of the second application is resumed at millisecond 6768, with the throughput of the other application being 2.7 iterations/s, slightly below the requested value. This behavior is caused by the slack mechanism. In fact, even if the average throughput is below the request, locally the application is running ahead of its time. In particular its last iteration completed in 229 milliseconds, lower than the period of 333 milliseconds obtained from the minimum requested throughput of 3 iterations/s. These 104 milliseconds contribute to the increase of the slack that is com-

puted taking into account the Γ factor as shown in Equation 6.5 from Chapter 6.

After the initial power failure, as soon as the slack is back to a positive value, big enough to accommodate its iteration, the second application is restored, speculating that the throughput, even if locally below the request, will soon reach the intended value. This slack mechanism is detailed in Section 6.3.

The more the slack increases, the more iterations of the second application are executed, for instance:

- in the interval of 2000ms between millisecond 5055 and 7055, there is one execution of the second application after 11 iterations of the first one;
- in the next interval of 2000ms between millisecond 7055 and 9055, there is one execution of the second application after 5 iterations of the first one.

After the second power failure, the same pattern repeats, as the second application is delayed until the first application accumulates enough slack. In particular, the first iteration of the second application, after the second power failure, is performed at time 27674ms.

This activation deactivation mechanism contributes to the reactivity of the proposed scheduler. The throughput of the first application is restored up to the requested value even in presence of power failures. In particular **Application 1** is back to a value equal or higher than 3 iterations/s:

- after 3811ms from the first power failure;
- after 2744ms from the second power failure.

The length of the inactivity interval after the third power failure, does not cause a decrease below the request.

On the contrary, as we can see from the chart in Figure 8.9a, the static approach can not cope with the first failure and the throughput of the main application stays below the requested value throughout the rest of the experiment.

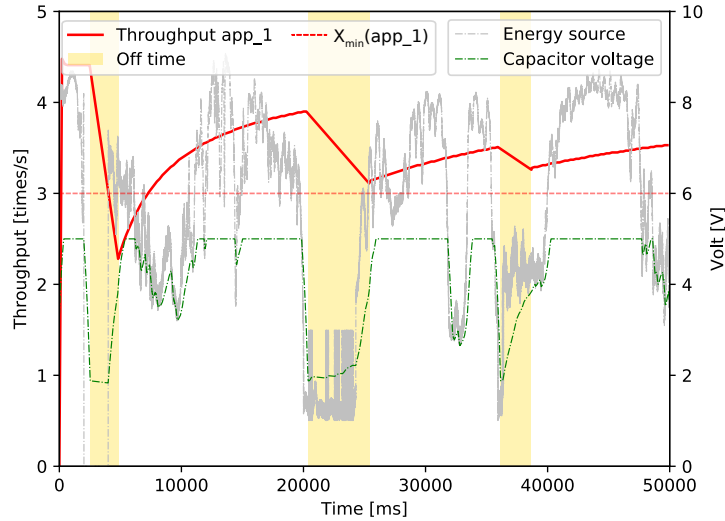
On average, the dynamic approach leads to a throughput for the main application more than 32% higher than the static one. The correctness of the proposed scheduler is more than 80% higher than the one obtained with the design time scheduling.

Table 8.3 shows a summary of average throughput, variance, correctness and fairness of the two approaches.

To improve the correctness of the static approach, it is possible to remove the second application, obtaining the result shown in Figure 8.11. This leads to a suboptimal energy utilization with an average wasted energy E_w value for **Application 1** of 115 μ J per iteration, while the average wasted energy with the dynamic scheduler is 51 μ J per iteration.

Table 8.3: Average throughput, correctness and fairness with design-time and dynamic runtime scheduling, against an energy source with sudden drops.

APPROACH	$X_{avg}(APP. 1)$	VARIANCE	CORRECTNESS	FAIRNESS
Static scheduling	2.41	0.18	5.3%	0.98
Dynamic scheduling	3.19	0.12	89.7%	0.89

**Figure 8.11:** Static scheduling with a single application leads to wasted energy

In fact, in the absence of a second application, the static scheduler can only run new iterations of the first one, even if its throughput is higher than the requests, resulting in a waste of computation: we spend energy to produce outputs at an unnecessarily high rate, instead of using that energy budget to run useful secondary computation. Moreover, the correctness improvement is only of 5.7%. Table 8.4 summarizes the comparison between the static approach with one application, and the dynamic one.

Tuning the Γ factor it is possible to increase the correctness of the dynamic scheduler, delaying the activation of the second application, at the expenses of an increase in wasted energy. In fact, a higher parameter causes an increase of the correctness and of the wasted energy, a lower parameter lowers the wasted energy and the correctness.

Table 8.4: Comparison between static scheduling with a single application, and runtime scheduler with multi-tenancy.

APPROACH	$X_{avg}(APP. 1)$	CORRECTNESS	E_w
Static with single app	3.46	95.4%	115 μ J/iteration
Dynamic with two app	3.19	89.7%	51 μ J/iteration

8.6.4 *Gamma and Fairness Interaction*

As presented in Section 6.3, each application except the first one, in decreasing requested throughput order, recursively runs within the slack time of the previous one. As pointed out in Equation 6.5 from Section 6.3, Γ being a parameter between 0 and 1, affects the value of the updated slack, changing the weight of older values. With $\Gamma = 1$ the slack value accumulated so far, and the current advance or delay in application's execution, equally contributes to the computation the new slack. Decreasing $\Gamma \in [0; 1]$ we decrease the number of iterations needed to fill the delay caused by an off time interval, as we progressively reduce the importance of old slack values.

As shown in previous results, different selections of Γ can lead to different performance and fairness.

The fairness of the scheduler is obviously affected by this parameter, intuitively a high Γ easily leads toward secondary application starvation, decreasing fairness. Given the unpredictability of energy sources it is unpractical to capture the link between Γ and fairness index with a closed equation. In fact, the highest fairness is reached when the parameter selection leads to an interleave compliant with the ratio between requested throughputs.

To substantiate this claim we run different experiments varying Γ with different sources and scenarios.

In particular:

- Figure 8.12a presents the result obtained varying Γ from 0 to 1, with the energy trace and the scenario from Section 8.6.3;
- Figure 8.12b presents the result obtained varying Γ from 0 to 1, with the energy trace from Section 8.6.3, changing the requested throughput for **Application 2** from 1 iteration/s to 2 iterations/s, by comparing the results of this experiment with the ones of the previous scenario, we can see how Γ and fairness interact with throughput requests;
- Figure 8.12c presents the result obtained varying Γ from 0 to 1, with the energy trace and the scenario from Section 8.6.2;

From these experiments we see that the fairness is higher for lower Γ values, though its maximum value corresponds to different parameter selections, depending on the energy source and the scenario.

In fact, in accordance with Equation 6.6 on fairness and Equation 6.5 on slack update, we see that:

- in all the experiments by increasing Γ , we increase the throughput of the main application, decreasing the throughput of the second one, in fact the activation of this last application is delayed the more we increase the parameter;

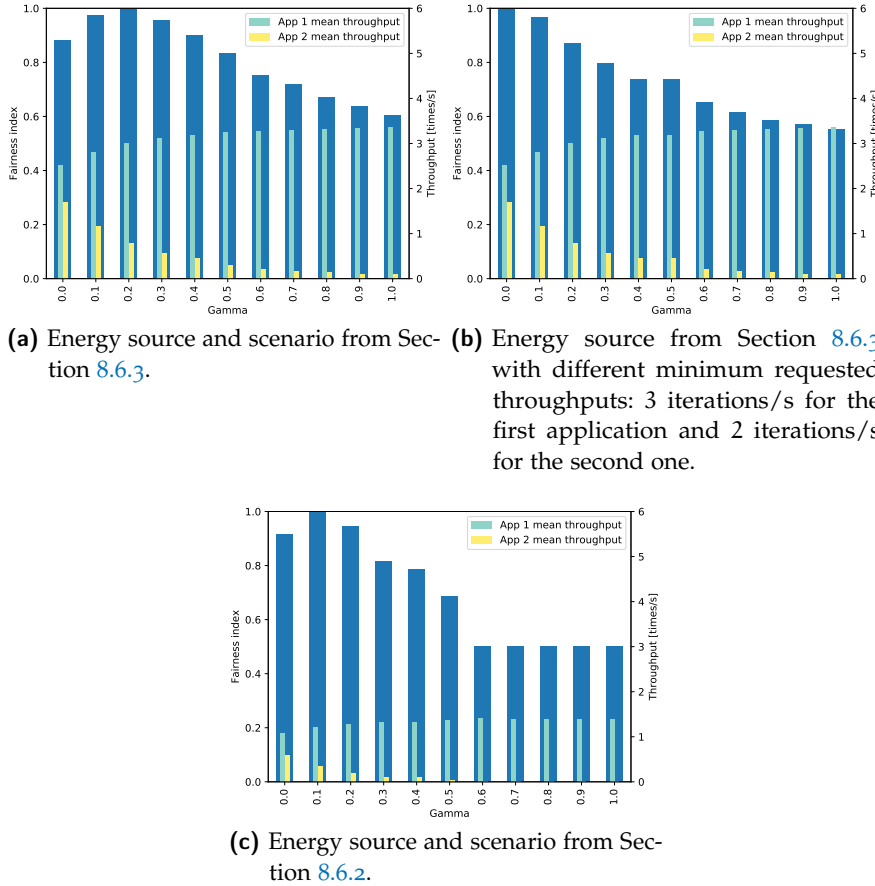


Figure 8.12: Interaction between Γ selection, throughputs and fairness.

- in both the experiments shown in Figure 8.12a and Figure 8.12c the highest fairness is reached when the throughput of **Application 1** is approximately 3 times the throughput of **Application 2**, but given the different energy sources, this corresponds to different Γ values;
- in the experiment shown in Figure 8.12b, the highest fairness is reached when the ratio between the throughput of **Application 1** and **Application 2** is closer to 3/2, that is the ratio between the requested values.

The experiment conducted varying the Γ on radio frequency source, whose results are presented in Figure 8.12c, shows that with Γ values higher than 0.6 the fairness does not change. In fact, from that value on, the second application is never activated and only **Application 1** is running at full speed. Still, given the continuous power failures, its throughput does not increase, therefore the fairness does not change.

The combined interaction between Γ and slack, gives the developer the opportunity to change the scheduler behavior. Different selections alter the policy, leading to different results, tailoring the framework to the specific deployment scenario.

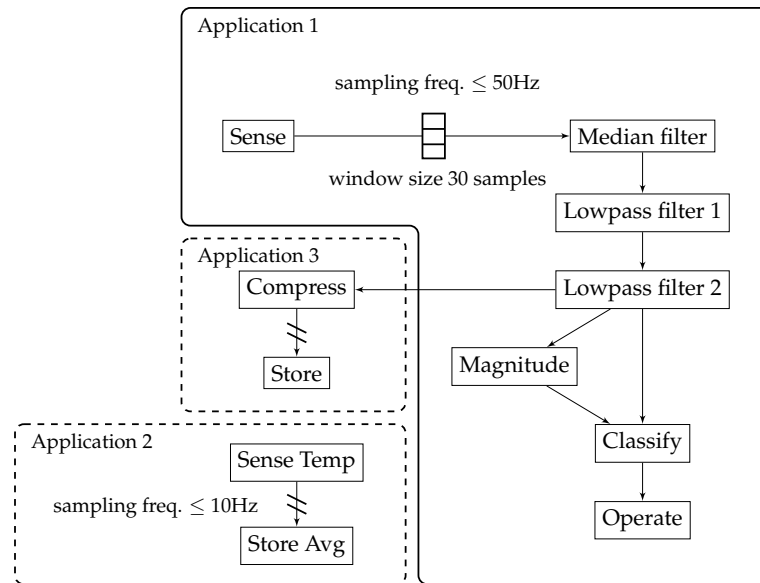


Figure 8.13: Dependency graph of the modified scenario, implemented to measure scheduler stability

8.6.5 Scheduler Stability

As we said in Chapter 6, the main goal of the dynamic scheduler is to satisfy the minimum throughput of the applications, considering the requests of one application at a time in decreasing X_{\min} order.

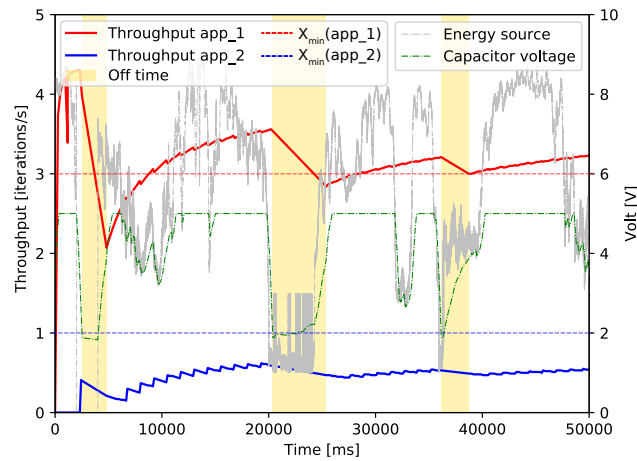
The performance of an application with a given throughput request, should not be affected by the nature and number of the applications with a lower throughput request, but only by the nature of the energy source and the selection of the tuning parameter Γ . We summarize this behavior with the term *stability*. We say that a scheduler is *stable* when background applications do not affect the behavior of the main application.

The concept of slack presented in Section 6.3 is a consequence to this principle: secondary applications must completely run within the slack time of the main one.

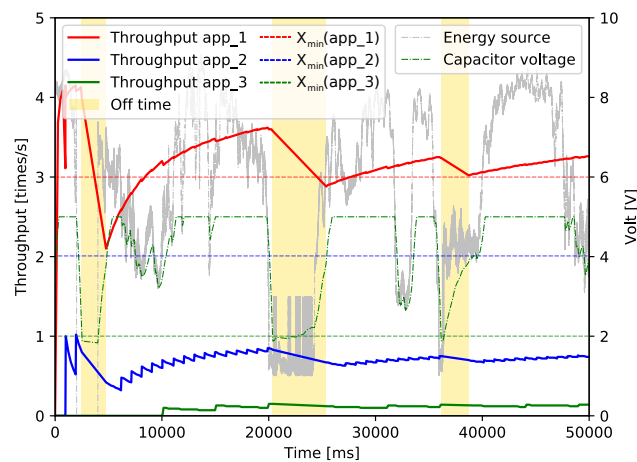
To prove that our implementation is stable, we change the number and nature of secondary applications, while we keep the one with the highest X_{\min} as described in the scenario presented in Section 8.3.

The dependency graph for this modified scenario is depicted in Figure 8.13

In particular, we change a task from Send to Store, to reduce its impact, given the high cost of operations related to radio equipment, so that we can accommodate a third application that performs sensing activities related to a temperature sensor.



(a) Dynamic scheduler with scenario from Figure 8.5



(b) Dynamic scheduler with scenario from Figure 8.13

Figure 8.14: Comparison between different secondary applications with the same main application and energy trace.

The applications are deployed with the following requested throughputs:

- **Application 1** request is set to 3 iterations/s, same as in the original scenario;
- **Application 2** request is set to 2 iterations/s;
- **Application 3** request is set to 1 iteration/s.

We run the simulation of both the original scenario and the new one, against the same energy source. The results shown in Figure 8.14 prove the stability of the proposed solution.

In particular, Figure 8.15 shows a detailed comparison between the main application throughput curves for the original and the modified scenario.

While the scheduler stability guarantees a similar behavior for the main application, we see different throughput for the other applica-

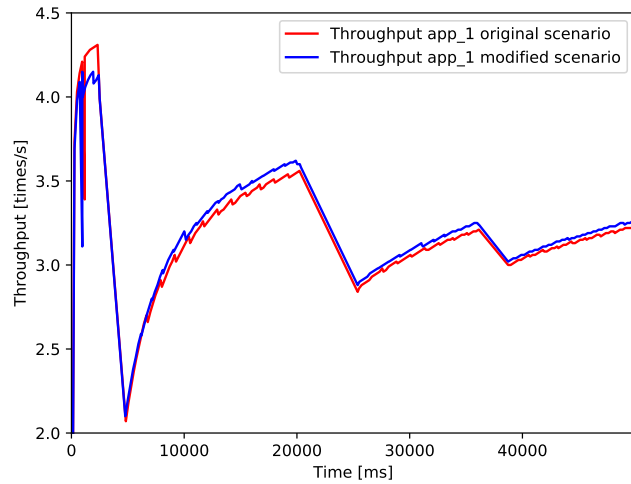


Figure 8.15: Detailed comparison between the throughput of **Application 1** with different secondary applications. The original scenario is described in Section 8.3, while the modified one is in Section 8.6.5

tions. In fact, in Figure 8.14b we see that the second application of this new scenario is enabled earlier. This happens because, independently from inputs, its tasks individually require less clock cycles than the one from the original second application. In particular, the storage of a single average temperature value requires far fewer clock cycles than the access to the antenna to perform send operations. For this reason a complete iteration of the new second application fits earlier in the slack of the main one.

CONCLUSION AND FUTURE WORKS

Our proposed solution performs as intended when the board is powered with a constantly high energy source, showing a correctness of 99.5%. In the worst case scenario, with a highly unstable energy source and an underpowered board, the dynamic scheduler guarantees an average throughput that is between 1.05 and 1.42 iterations/s, depending on the Γ selection, while the static one is 0.99 iterations/s.

With a stable energy source, in presence of power failures, the dynamic scheduler is able to guarantee an increase of over 32% on average throughput of the main application, and a correctness over 80% higher than the static one.

The Γ parameter provides to the developer an extra knob to tune the scheduler performance, trading correctness with wasted energy and changing scheduler fairness.

Thanks to the proved scheduler stability, the developer can change the firmware modifying the background applications, without worrying about the performance of the core logic represented by the main application.

In the introduction to this document, we said that batteryless devices are the most promising solution to finally enable the “smart dust” vision. But, paraphrasing a famous quote, with great powers come great challenges: those presented in Chapter 2. Batteryless systems bring to Transiently Powered Computation (TPC), and the issues coming from power failures pose a severe threat to their large-scale adoption.

In Chapter 3, we described the most promising state of the art solutions to address the TPC challenges. All of them tackle in different ways the issue of data consistency. Yet, they lack the ability to truly react at runtime to the unpredictability of energy sources, or to capture the peculiar features of this class of devices that we presented in Chapter 4, being unable to alter at runtime the voltage parameters that rule the activation and deactivation of computation.

To fill this gap, we proposed our task-based solution: a dynamic scheduler that, for the first time, focuses on throughput requirements, instead of trying to obtain the maximum energy efficiency. Our solution, leveraging on a new programming abstraction and on multi-tenancy, is able to dynamically adapt both the workload. Moreover, it

adapts at runtime the voltage threshold to the specific energy scenario at runtime.

In Chapter 5, we introduced this new programming abstraction to match the requirements of our new system, starting from tasks and their transactional semantic, moving on to the new concept of applications and minimum requested throughput, and finally presenting a rich set of data dependency semantics among tasks, to match different common scenarios.

In Chapter 6, we described the logic of our scheduler and its internals, presenting the set of properties that guide its functioning. We described the concept of applications priority and how it is connected to the minimum throughput requirement; we showed how our dynamic scheduler is able to change the workload at runtime to adapt to variations in energy provisioning; we detailed the algorithms that guide the reactivity of our solution that is able to take countermeasures, in case of under performing or over performing applications. Finally, we presented the parameter Γ and its tuning: an important feature of our solution, that provides an extra knob to the developer to tailor the performance of the system to the specific scenario, deciding whether to lean toward a fairer scheduler, or toward the maximization of the main application's performance.

In Chapter 7 we presented the workflow that allows the developer to obtain a firmware from the description of tasks and applications, describing the implementation of our system.

Finally, in Chapter 8, we presented an evaluation of our system, comparing its performance to a static solution, similar to the one implemented in Mayfly [16]. To conduct this evaluation we gave our contribution to an existing instruction level simulator: SIREN [11], enriching its features and partially restructuring its codebase.

9.0.1 *Future Works*

Task-based systems require an effort to the developer, who has to restructure the code to embrace this abstraction. For this reason we tried to propose a solution that increases as little as possible developer's effort. Still, the performance of the system depend on Γ : a parameter that with our current implementation, must be selected at compile time and can not be refined at runtime. Moreover, its impact on the system performance is not intuitive and varies with different energy sources.

As an improvement to our solution, we would like to hide the complexity of Γ selection and offer a parameter that relates more intuitively to the expected performance and informs an automatic selection of Γ . In addition, we would like to introduce a refinement process that fixes its value at runtime.

In Section 4.2, we described some peculiar features of our target devices, in particular how the time and energy needed to execute a clock cycle varies at different voltage levels. We developed the management of the activation threshold in light of these results. Still, our solution does not explicitly select an activation threshold to match a given speed or energy consumption, yet it applies a refinement process that should converge to the optimal selection. Moreover, we do not consider clock frequency as a tunable parameter, as these devices currently do not implement DVFS. Further investigations on the relationship between voltage and frequency could increase the performance of our system. In fact, this would offer to the scheduler the ability to change at runtime the device's frequency, as an additional knob to satisfy the requirements.

In Section 6.5, we said that, if the task selected by the scheduler requires more energy than the one that is currently stored in the energy buffer, then the task is skipped to avoid power failures. Approximate computing is a set of techniques that allow to produce code that tunes the result accuracy, based on specific requirements. In particular, some of these techniques, like loop perforation, can decrease this accuracy in ways that reduce the number of executed instructions, hence the energy consumption for a given task. Allowing to propose different approximation levels, therefore different energy consumption for a given task, could allow us to execute a task even when the buffer does not hold enough energy for its original version. In general, our system could be improved by offering the possibility to provide more than one version of a task, each one characterized by different energy consumption.

In conclusion, batteryless execution is highly dependent on the runtime energy conditions. Different classes of energy sources have different features, so a possible solution would be to specifically adapt the firmware and its logic to the specific scenario. Though, this would produce a rigid solution that potentially reacts badly to deviations from the expected behavior of the source. To improve this, one could relax the specifications and produce a more general solution, yet without the ability to dynamically adapt at runtime, it can not truly react to unexpected changes in the energy intake. These observations guided the development of our proposal: a dynamic task-based scheduling framework that is able to react at runtime to the unpredictability of energy sources. Thanks to this framework we proposed a shift in perspective: from energy, to throughput in the intermittent computing scenario.

BIBLIOGRAPHY

- [1] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. “The Betrayal of Constant Power \times Time: Finding the Missing Joules of Transiently-Powered Computers.” In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 97–109. ISBN: 9781450367240. DOI: [10.1145/3316482.3326348](https://doi.org/10.1145/3316482.3326348) (cit. on pp. [1](#), [5](#), [49](#), [51–54](#), [66](#), [88](#), [95](#), [134](#)).
- [2] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. “Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices.” In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 35.12 (Nov. 2016), pp. 1968–1980. ISSN: 0278-0070. DOI: [10.1109/TCAD.2016.2547919](https://doi.org/10.1109/TCAD.2016.2547919) (cit. on pp. [3](#), [5](#), [6](#), [17](#), [31](#), [54](#), [56](#), [134](#)).
- [3] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. “Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems.” In: *IEEE Embedded Systems Letters* 7.1 (2015), pp. 15–18. ISSN: 1943-0663. DOI: [10.1109/LES.2014.2371494](https://doi.org/10.1109/LES.2014.2371494) (cit. on pp. [3](#), [30](#), [49](#), [53](#)).
- [4] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. “Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences.” In: *ACM Trans. Sen. Netw.* 12.3 (Aug. 2016). ISSN: 1550-4859. DOI: [10.1145/2915918](https://doi.org/10.1145/2915918) (cit. on pp. [1](#), [14](#), [17](#), [49](#)).
- [5] N. A. Bhatti and L. Mottola. “HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing.” In: *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '17. Pittsburgh, Pennsylvania: ACM, 2017, pp. 209–219. ISBN: 978-1-4503-4890-4. DOI: [10.1145/3055031.3055082](https://doi.org/10.1145/3055031.3055082) (cit. on pp. [26](#), [27](#), [49](#)).
- [6] M. Buettner, B. Greenstein, and D. Wetherall. “Dewdrop: An Energy-Aware Runtime for Computational RFID.” In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 197–210 (cit. on pp. [5](#), [6](#), [48](#), [55](#), [58](#)).

- [7] Michael Buettner, Richa Prasad, Matthai Philipose, and David Wetherall. "Recognizing daily activities with RFID-based sensors." In: *Proceedings of the 11th international conference on Ubiquitous computing - Ubicomp '09* (2009). DOI: [10.1145/1620545.1620553](https://doi.org/10.1145/1620545.1620553) (cit. on p. 130).
- [8] E. W. Dijkstra. "The Humble Programmer." In: *Commun. ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: [10.1145/355604.361591](https://doi.org/10.1145/355604.361591) (cit. on p. 33).
- [9] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, and T. Voigt. "Mspsim—an extensible simulator for msp430-equipped sensor boards." In: *Proceedings of the European Conference on Wireless Sensor Networks*. EWSN (cit. on p. 124).
- [10] A. Fuggetta, G. P. Picco, and G. Vigna. "Understanding code mobility." In: *IEEE Transactions on Software Engineering* 24.5 (1998), pp. 342–361. ISSN: 2326-3881. DOI: [10.1109/32.685258](https://doi.org/10.1109/32.685258) (cit. on p. 17).
- [11] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. "Realistic Simulation for Tiny Batteryless Sensors." In: *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems - ENSys'16* (2016). DOI: [10.1145/2996884.2996889](https://doi.org/10.1145/2996884.2996889) (cit. on pp. 25, 113, 124, 125, 150).
- [12] A. Gomez, L. Sigrist, M. Magno, L. Benini, and L. Thiele. "Dynamic energy burst scaling for transiently powered systems." In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2016, pp. 349–354 (cit. on p. 129).
- [13] J. Hester, T. Scott, and J. Sorber. "Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors." In: *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. SenSys '14. Memphis, Tennessee: Association for Computing Machinery, 2014, pp. 330–331. ISBN: 9781450331432. DOI: [10.1145/2668332.2668382](https://doi.org/10.1145/2668332.2668382) (cit. on p. 124).
- [14] J. Hester and J. Sorber. "Flicker." In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems - SenSys '17* (2017). DOI: [10.1145/3131672.3131674](https://doi.org/10.1145/3131672.3131674) (cit. on pp. 13, 47, 58).
- [15] J. Hester and J. Sorber. "The Future of Sensing is Batteryless, Intermittent, and Awesome." In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. SenSys '17. Delft, Netherlands: ACM, 2017, 21:1–21:6. ISBN: 978-1-4503-5459-2. DOI: [10.1145/3131672.3131699](https://doi.org/10.1145/3131672.3131699) (cit. on pp. 1, 13).
- [16] 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 - SenSys '17* (2017). DOI: [10.1145/3131672.3131673](https://doi.org/10.1145/3131672.3131673) (cit. on pp. 1, 3, 4, 8, 20, 40, 67, 86, 130, 132, 150).

- [17] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burlison, and J. Sorber. “Persistent Clocks for Batteryless Sensing Devices.” In: *ACM Transactions on Embedded Computing Systems* 15.4 (2016), pp. 1–28. ISSN: 1539-9087. DOI: [10.1145/2903140](https://doi.org/10.1145/2903140) (cit. on pp. 4, 20, 40, 43, 90, 99).
- [18] Michael Jackson. “The world and the machine.” In: *Proceedings of the 17th international conference on Software engineering - ICSE '95* (1995). DOI: [10.1145/225014.225041](https://doi.org/10.1145/225014.225041) (cit. on p. 75).
- [19] Raj Jain, Dah Ming Chiu, and Hawe WR. “A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems.” In: *CoRR* cs.NI/9809099 (Jan. 1998) (cit. on p. 108).
- [20] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. “Quick-Recall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers.” In: *J. Emerg. Technol. Comput. Syst.* 12.1 (Aug. 2015). ISSN: 1550-4832. DOI: [10.1145/2700249](https://doi.org/10.1145/2700249) (cit. on pp. 3, 31).
- [21] L. Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.” In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. ISSN: 2326-3814. DOI: [10.1109/TC.1979.1675439](https://doi.org/10.1109/TC.1979.1675439) (cit. on p. 16).
- [22] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. “A modular 1mm³ die-stacked sensing platform with optical communication and multi-modal energy harvesting.” In: *2012 IEEE International Solid-State Circuits Conference*. 2012, pp. 402–404. DOI: [10.1109/ISSCC.2012.6177065](https://doi.org/10.1109/ISSCC.2012.6177065) (cit. on p. 1).
- [23] P. Levis, N. Lee, M. Welsh, and D. Culler. “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications.” In: *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. SenSys '03. Los Angeles, California, USA: Association for Computing Machinery, 2003, pp. 126–137. ISBN: 1581137079. DOI: [10.1145/958491.958506](https://doi.org/10.1145/958491.958506) (cit. on p. 124).
- [24] B. Lucia and B. Ransford. “A Simpler, Safer Programming and Execution Model for Intermittent Systems.” In: *SIGPLAN Not.* 50.6 (June 2015), pp. 575–585. ISSN: 0362-1340. DOI: [10.1145/2813885.2737978](https://doi.org/10.1145/2813885.2737978) (cit. on pp. 3, 35, 36).
- [25] Kiwan Maeng, Alexei Colin, and Brandon Lucia. “Alpaca: Intermittent Execution Without Checkpoints.” In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 96:1–96:30. ISSN: 2475-1421. DOI: [10.1145/3133920](https://doi.org/10.1145/3133920) (cit. on pp. 3, 8, 21, 35, 37, 84).

- [26] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. “On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper).” In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 203–207. ISBN: 9781450367240. DOI: [10.1145/3316482.3326346](https://doi.org/10.1145/3316482.3326346) (cit. on p. 121).
- [27] *MSP430FR6989 Datasheet*. <https://www.ti.com/lit/ds/symlink/msp430fr69891.pdf>. Texas Instruments (cit. on pp. 5, 17, 23, 47, 123).
- [28] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burseson, and K. Fu. “TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks.” In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 221–236. ISBN: 978-931971-95-9 (cit. on pp. 4, 20, 121, 125).
- [29] B. Ransford, J. Sorber, and K. Fu. “Mementos: System Support for Long-running Computation on RFID-scale Devices.” In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 159–170. ISBN: 978-1-4503-0266-1. DOI: [10.1145/1950365.1950386](https://doi.org/10.1145/1950365.1950386) (cit. on pp. 1, 3, 24, 47, 51).
- [30] A. Rodriguez, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddell. “Intermittently-powered energy harvesting step counter for fitness tracking.” In: *2017 IEEE Sensors Applications Symposium (SAS)*. 2017, pp. 1–6. DOI: [10.1109/SAS.2017.7894114](https://doi.org/10.1109/SAS.2017.7894114) (cit. on p. 130).
- [31] A. P. Sample and J. R. Smith. “The Wireless Identification and Sensing Platform.” In: *Wirelessly Powered Sensor Networks and Computational RFID*. Ed. by Joshua R. Smith. New York, NY: Springer New York, 2013, pp. 33–56. ISBN: 978-1-4419-6166-2. DOI: [10.1007/978-1-4419-6166-2_3](https://doi.org/10.1007/978-1-4419-6166-2_3) (cit. on pp. 13, 48, 56).
- [32] E. Sazonov, H. Li, D. Curry, and P. Pillay. “Self-Powered Sensors for Monitoring of Highway Bridges.” In: *IEEE Sensors Journal* 9.11 (2009), pp. 1422–1429. ISSN: 2379-9153. DOI: [10.1109/JSEN.2009.2019333](https://doi.org/10.1109/JSEN.2009.2019333) (cit. on p. 1).
- [33] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms, 2nd Edition*. Pearson, 2007 (cit. on p. 16).
- [34] J. Van Der Woude and M. Hicks. “Intermittent Computation Without Hardware Support or Programmer Intervention.” In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. Savannah, GA, USA: USENIX

- Association, 2016, pp. 17–32. ISBN: 978-1-931971-33-1 (cit. on pp. 3, 28).
- [35] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. “InK: Reactive Kernel for Tiny Batteryless Sensors.” In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems. SenSys '18*. Shenzhen, China: Association for Computing Machinery, 2018, pp. 41–53. ISBN: 9781450359528. DOI: [10.1145/3274783.3274837](https://doi.org/10.1145/3274783.3274837) (cit. on pp. 3, 42, 63).