# Energy and Power Awareness in Hardware Schedulers for Energy Harvesting IoT SoCs

P Anagnostou*, A Gomez*, PA Hager*, H Fatemi†, J Pineda de Gyvez†, L Thiele*, L Benini*‡
*ETH Zurich †NXP Semiconductors ‡University of Bologna
{panagnos, gomeza, phager, thiele, lbenini}@ethz.ch {firstname.lastname}@nxp.com

***Abstract*—The recent growth of applications in the emerging Internet of Things field is posing new challenges in the long-term deployments of sensing devices. Currently, system designers rely on energy harvesting to reduce battery size and extend system lifetime. While some system functions need constant power supply, others can have their service adapted dynamically to the available harvested energy and harvesting power. Our proposed Torpor is a power-aware hardware scheduler which continuously monitors harvesting power and in combination with its software runtime, dynamically activates system functions depending on the available energy and its rate of change. By performing a few key functions in hardware, Torpor incurs a very low power overhead during continuous monitoring, while the software runtime provides a high degree of flexibility to enable different scheduling policies. We implemented Torpor on a FPGA-based prototype and demonstrated that dynamic scheduling policies which take the harvesting power into account can have a 2× or more improvement in execution rates compared to static (input-power-independent) policies, while dynamic policies that are aware also of the system's power consumption can achieve 1.5× improvement in the execution rates compared to the ones that do not. The power consumption of Torpor's always-on hardware integrated on chip is estimated to be less than 4 µW, making it a very promising power-management add-on for microprocessors used in IoT nodes.**

## I. Introduction

In recent years, energy harvesting has been explored as a promising lifetime extension option for wireless sensor nodes. Compared to battery-only designs, energy harvesting nodes require less energy storage capacity for continuous, long-term operation. Energy harvesting, however, can be subject to great variability [1]. In the most extreme case, this can mean no energy is harvested for long periods of time. To cope with harvesting variability, the system's service is typically reduced as the available energy decreases [2]. Certain applications, however, require an always-on domain for specific tasks which are fundamentally incompatible with service degradation. One example is a sensing system that needs to continuously record data but can defer the processing and transmission of the recorded data for periods of high energy availability.

When an energy source exhibits periodic behavior, e.g. outdoor solar [3] or thermal [4], designers can optimize their design for continuous, energy-neutral operation. This is achieved by dimensioning the battery to supply energy for at most one period since it is the worst-case unavailability of a periodic source. When the energy source has no regularity or cannot be gauged at design time, this methodology inevitably falls back to over-dimensioned, battery-based design.

For this reason, there has been a recent trend towards batteryless, or transient systems [5], [6]. These systems are entirely energy-driven: they can operate only when harvested energy is available —whenever that may be. To use this harvested energy efficiently, task energies must be known and operation must be duty-cycled [7]. This can limit the system's supported tasks as transition costs, between active and sleep states, must be low. While these devices minimize cost (no expensive battery needed), they cannot support always-on tasks unless the environment guarantees energy continuity, which is hardly ever the case [8]. Because of this limitation present in all harvesting-based systems, we divide our application into (i) ultra-low-power guaranteed minimum service and (ii) opportunistic high-power service. A conventional harvester-battery approach, shown in fig. 1A, would require the battery and the power subsystem to be dimensioned for the worst-case conditions. Because both low and high power services have the same source, system parameters will be dominated by high power tasks, leading to excessively high costs. We propose a different topology, shown in shown in fig. 1B, that supplies high power tasks differently from always-on low power tasks. High power tasks are supplied using a harvester and a high-current capacitive energy buffer, while a primary battery is used exclusively to guarantee the continuity of low power tasks.

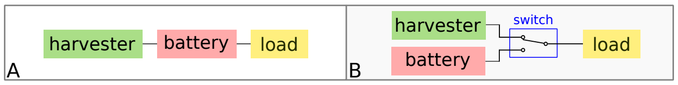

Fig. 1. A) Traditional harvesting-based nodes combine harvested with stored energy. B) Torpor-based nodes switch between battery and harvesting powered operating modes.

In our previous work [9], we introduced Torpor, shown in fig. 1B, that efficiently combines energy harvesting and primary energy storage for long-term operation with service guarantees. Whenever the harvester cannot sustain the load, the system is powered by primary battery and can thus provide guarantees for low-power, always-on tasks. When the harvester produces enough energy to sustain the load, the system switches to harvesting-powered mode and additional high-power tasks can be opportunistically supplied from cheap harvested energy. To use harvested energy efficiently, the system needs to monitor its input power and maintain a balance with the load's power consumption through scheduling.

To this end, we develop a low-overhead HW scheduler that can support different types scheduling strategies, based on different application-specific parameters. In highly variable harvesting scenarios, we show that introducing task energies into a dynamic scheduling policy can double the execution rate of high-power energy-driven tasks compared to a static policy, while still guaranteeing continuity of always-on tasks.

Applications using multiple peripherals can have highly different power consumption depending on the task being executed. As such, an energy oriented scheduling policy might select a task whose power consumption is below the harvested power. In this scenario, the energy buffer will quickly saturate, prevent additional available energy from being harvested and therefore decrease the system's energy efficiency.

In this work, we propose an additional dynamic scheduling policy based on one additional application parameter: the power consumed by the load during each task. When doing so, Torpor is using this additional information so that the load's power consumption can more closely match that of the harvester. By minimizing the time intervals where the harvested power exceeds the energy-driven tasks' power consumption, we minimize the chance of energy buffer saturation and as a result maximize the harvested energy. We show experimentally that, in certain scenarios, this power-oriented dynamic scheduling policy can achieve 1.5 times the execution rate of energy-oriented dynamic policies.

For either energy-oriented or power-oriented scheduling policies to work, the system designer has to know and provide the values of required energy and power for each task. To achieve this, one approach calls for theoretical calculations, which often find their assumed models inadequate to match the reality, while another approach follows a labor-intensive trial-and-error process of practical measurements. Having a procedure that helps designers pre-characterize their tasks' energy and power demands becomes essential for the system's operation and another novel contribution of this work is the Automatic Task Characterization, which delivers this functionality by using an external source, before deployment.

In summary, the contributions of Torpor are the following:

- A system architecture that efficiently combines guaranteed always-on functionality with energy-driven tasks.
- A configurable HW unit that continuously monitors harvested energy availability, exposing to the SW a priorities-mechanism that enables both static and dynamic scheduling policies for energy-driven tasks.
- An automatic procedure to characterize the tasks in terms of their energy and power needs.
- The implementation of an energy-oriented dynamic scheduling policy that improves task execution rates by $2\times$ compared to static policies under variable harvesting conditions.
- The implementation of a power-oriented dynamic scheduling policy that improves task execution rates by $1.5\times$ compared to the energies-oriented under harvesting conditions where the energy buffer may saturate due to short intervals of harvesting power potentially larger than the load's consumption.
- A complete system prototype where Torpor is implemented in a FPGA for verification.
- An extensive experimental evaluation, comparing the performance of static and dynamic scheduling policies.

The remainder of this paper is structured as follows: section II covers existing works in related fields and section III introduces basic concepts of transiently-powered systems that are important for Torpor. After that, section IV presents the high-level architecture of our proposed system, while section V focuses on our physical implementation. On section VI we discuss in detail our evaluation of both the simulated HW unit and the FPGA prototype. Finally, section VII concludes our work.

## II. Related Work

Wireless sensor and actuator systems have been designed to be battery-powered for many decades [10]. In certain specific applications, like implantable devices [11], they will remain battery-powered for many years in the foreseeable future. In many other, typically outdoor scenarios, energy harvesting has been successfully introduced to reduce costs and extend battery lifetimes. Interested readers can find a survey of energy harvester taxonomies and prediction models [12]. Let us assume that a system has a harvester and an initially charged energy storage connected in series. If we break down the energy consumed by the system during its lifetime, there are basically two possibilities:

*1) Storage-dominated energy flow:* If the majority of the load's energy originates from the (initially charged) storage device, it will dominate the system lifetime. This can happen when the harvesting power is significantly smaller than load power. There is so little uncertainty that it is easy, albeit costly, to guarantee long lifetimes at design time. Classical design techniques for these systems include dynamic power management [13], [14], and low power design [15], [16].

*2) Harvesting-dominated energy flow:* If the storage device cannot supply the load by itself, the system can be considered to be transient or completely harvesting-driven. These systems can either be designed for reliable or greedy operation. In the former, the storage can be dimensioned either for a single application iteration [5] or a single task [17]. Other approaches have been proposed to address the inherent energy variability in tasks executed by sensing devices. In [18], for example, a federated approach to energy storage is proposed, separating the energy supply of different components into different capacitors. Capybara [19], is a reconfigurable energy storage architecture that efficiently addresses the variable energy demands of sensing applications. Sometimes the energy storage is so small that tasks cannot be executed atomically, thus requiring state retention techniques [6], [20] to avoid data consistency issues caused by intermittent execution. Some systems manage to increase their efficiency by becoming adaptive to the source and load dynamics and autonomously characterizing their hardware platform [21]. Though transient systems are cost-effective and can operate in an energy efficient manner, they cannot guarantee continuous operation with volatile sources. By contrast, if the storage device is large enough to "filter out" the source's energy variability, continuous operation can be sustained. In fact, given enough information about the environment, even real-time guarantees can be made [22]. However, these systems require a high-energy, periodic source (i.e. the sun) to keep the harvester and storage element cost effective [23]. Interested readers can find an overview of the circuit, architecture and system design considerations in [24]. One way to mitigate the effects of a source's variability is to harvest energy from multiple sources. Multi-harvesting is an effective way to increase the energy availability, but it has multiple trade-offs in terms of efficiency, applicability, and ease of deployment [25]. Even with multi-harvesting, however, these systems need advanced power management techniques [26] with state-of-charge estimation [27] to adapt the system's service [28].

In this work, we introduce Torpor, a hybrid approach that combines the benefits of both: low costs of harvesting-based secondary cell whenever it is available, and a primary cell for a guaranteed, worst-case lifetime. To maximize the energy efficiency during harvesting-driven operation, we propose novel dynamic scheduling algorithms, which can be executed in hardware with little overhead. For the purposes of simplicity, we will focus on sequential *sense-process-transmit* on a single core platform. Using memory buffers, we can use functional-level parallelism, for example, to execute multiple *sense* tasks before the first *process* task is called. This simple execution model is enough for Torpor to demonstrate significant improvement in overall execution rates. The proposed schedulers require application-specific parameters such as the energy and power consumption of each task. Torpor assists developers in experimentally determining these values by using a dedicated power channel for characterization purposes. Those scheduling algorithms will be evaluated against the static schemes proposed in [5] and [17].

## III. Preliminaries

Typically, IoT applications executed in Wireless Sensor Nodes (WSN) can be seen as a sequence of tasks beginning with sensing some environmental information, processing it in a number of steps and finally transmitting the results to a base station. We consider applications with two types of tasks: 1) always-on tasks which have, on average, constant low-power but require continuity and 2) energy-driven tasks which can be high-power but are also deferrable. We consider applications that have always-on tasks running in the background and a chain of $N$ energy-driven tasks with data buffers in-between. As each task may require data produced by the previous task in chain and produce results needed for the next task, these dependencies must always be observed. Each energy-driven task has its own duration and power consumption, and for simplicity, we restrict them to atomic execution.

Supporting applications with always-on and energy-driven tasks requires managing different types of energy sources. Primary batteries can easily guarantee the continuity of always-on tasks for a specified lifetime. Harvesters can supply high-power, energy-driven tasks with cheap energy. Torpor switches between these two sources depending on the availability of harvesting power, as shown in fig. 2.

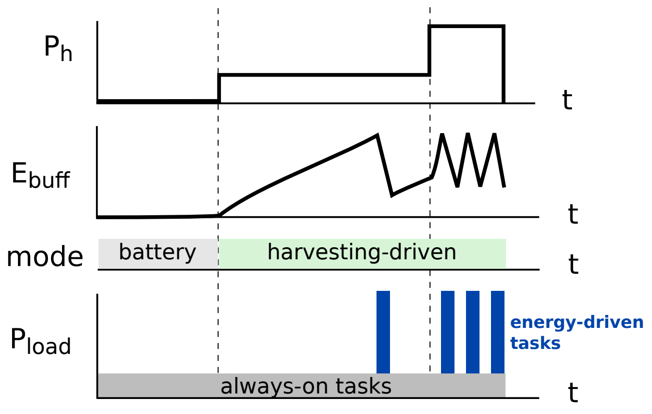

Fig. 2. When there is no harvesting power ($P_h = 0$), Torpor supplies always-on tasks with battery power. As harvesting power becomes available, Torpor uses burst-generation schemes for power-hungry, energy-driven tasks.

### A. Battery-driven mode

The system enters this mode of operation whenever the harvester produces energy at an insufficient rate. Similar to the biological state of *torpor*, the system is in a state of decreased activity, limiting itself to running always-on tasks. Their continuity is guaranteed by the battery, which has been sized to guarantee a specified worst-case lifetime. Since always-on tasks are meant to be low power (in the µW range), multi-year operation can be easily reached with a small primary cell.

### B. Harvesting-driven mode

The execution of energy-driven tasks is supplied, as the name implies, solely by harvesting power. During this time the primary battery is switched out, thus prolonging its lifetime. To enter this mode, it is necessary that the harvesting power ($P_h$) be larger than the load's always-on power ($P_{always\_on}$). If this condition is met, we can utilize the burst generation schemes presented in [5], [17]. Otherwise, the harvested power cannot be utilized by the load wihout additional circuitry to manage a secondary battery for storing energy harvested in these low power levels. These additional costs outweigh any benefits from operating in the $P_H < P_{always\_on}$ region. Burst-generation schemes can work with low input power, in the $\mu$W range, and still supply power-hungry energy-driven tasks in the hundreds of mW range. This is achieved by accumulating the energy necessary for only one activation. The energy is buffered in a small capacitor, and its consumption due to a load activation is referred to as an energy burst.

Since there are many possible scheduling policies to determine when the load should start a burst and how big it should be, any scheduler implemented in HW should be configurable to support this.

### C. Burst Scheduling Policies

The simplest scheduling policy is to accumulate enough energy to execute all energy-driven tasks together in a single energy burst of size $E_{\text{buff}} = E_{\text{app}}$. This policy will be referred to as *single burst*, similar to the approach presented in [5]. Single burst schedules are simple and could be implemented by a single comparator reading the capacitor voltage $V_{\text{buff}}$. But waiting always for a high capacitor voltage can be inefficient since the load discharge power will increase as the voltage increases.

Another approach is to *split* the application into multiple bursts. As such, each individual task is executed as soon as there is enough energy to execute it. As first proposed in [17], the order in which the individual tasks are executed is static and defined at compile time. While *split* execution requires less buffered energy than *single burst*, they are both static scheduling techniques.

Independent of the scheduling policy, the buffering capacitor charges or discharges, with rate:

$$\frac{dE_{\text{buff}}(t)}{dt} = P_h(t) - P_{\text{load}}(S_i) - \bar{P}_{\text{Torpor}} \qquad (1)$$

where $P_{\text{load}}$ is power consumed by the load, $S_i$ is the system state and $\bar{P}_{\text{Torpor}}$ is the average power consumed by Torpor. The system state reflects what the load is executing, whether always-on functions only or also energy-driven tasks. The state is updated by the scheduler, when it makes a decision to trigger a new energy-driven task, or when a task is completed.

We are referring to static scheduling policies when the order of energy-driven task execution is always the same. If there is only one possible decision, as is the case in the *single burst* approach, then the only variable will be *when* the application is executed, which depends only on $E_{\text{buff}}$. This approach was first introduced in [5]. When an application requires multiple bursts or activations, the scheduler also needs to know the application state, e.g. buffer states and task dependencies, to determine which task to execute next. For example, a FIFO scheduler will execute a chain of tasks one by one, always in the same order. This approach was first proposed in [17].

We argue that *static* schedules may still lead to inefficient use of the harvesting power when this exhibits highly variable behavior.

*Dynamic* scheduling policies can execute a chain of tasks in a different order, depending on one or more parameters. As we have seen, the harvesting power determines the rate of increase of $E_{\text{buff}}(t)$ and consequently the time the system will need in order to accumulate enough energy to execute its next energy-driven task. When the harvesting power is low, the accumulated energy may remain relatively flat for prolonged periods or even be gradually lost due to leakage and the load's base power consumption. This phenomenon which will have an important effect on system metrics such as execution rate and overall energy efficiency.

For this reason, input-power-awareness is an essential parameter for any dynamic scheduling policy. On the application side, a dynamic scheduling policy needs to be able to re-arrange the order of execution (otherwise it will be a static schedule). To do this, the system has to be able to store the result data tokens of each executed task in memory buffers and ensure that tasks are executed only as long as their required input data tokens are available and there is enough space in their output data token buffer. When there are many (possibly all) tasks available, different policies can prioritize the execution of tasks based on either the largest energy requirement, the highest power consumption, the task's position in the task-chain or any other metric. In our previous work [9], we demonstrated how an energy-oriented policy improved the energy efficiency and application execution rate. We will now introduce a novel power-oriented policy which will improve system behavior even further. When each task's power is available as information, Torpor is able to avoid the energy buffer's saturation in a much wider harvesting power range and a much broader set of possible task-sets.

## IV. System Architecture

### A. Top-Level Architecture

The top-level system architecture of a proposed Wireless Sensor Node (WSN) with Torpor is depicted in fig. 3. Two power sources, a battery and a harvester, (e.g. a solar cell or an electro-mechanical generator), are available to supply the node's *load*. The load contains the node's system-on-chip (SoC) to run the application as well as external peripherals to interact with the outside world (e.g. sensors, radio).

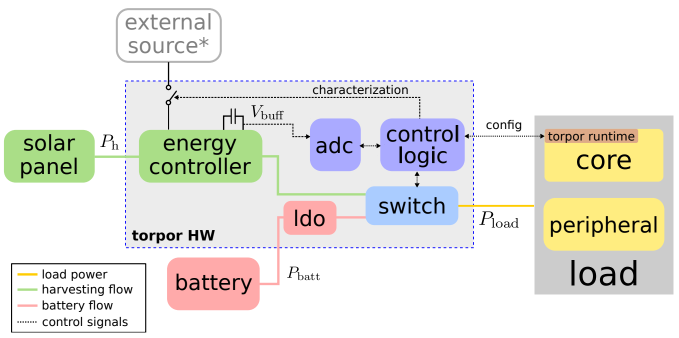

Fig. 3. Top-Level Architecture: An overview of the blocks present in a WSN that uses Torpor. A control logic manages the state of the Load, which is either powered by harvested energy or by battery. (*) The external power source is only used for characterization before WSN deployment.

### B. Torpor Functions

As proposed earlier, Torpor distinguishes between periods of no harvested energy, where always-on tasks are guaranteed to run with a battery, and periods of energy harvesting, where all tasks (always-on and energy-driven tasks) can be supplied with harvesting power. To support this operation, Torpor must provide the following functions:

- A switch-over mechanism between the battery-supply and harvester power source.

- A mechanism which ensures that energy-driven tasks run reliably. This means buffering sufficient energy to complete a task after starting it.
- The scheduling capability to decide what, if any, energy-driven task should be launched based on the buffered energy $E_{\text{buff}}(t)$ the input power $P_h(t)$, according to priorities that reflect the task energy and power demands and the task availability due to the software application state.
- The required always-on monitoring capabilities to estimate $E_{\text{buff}}(t)$ and $P_h(t)$.
- An easy-to-use interface to the application developer that can characterize the task energies and powers.

### *C. Torpor Architecture*

Torpor should support various applications and be configurable. Additionally the implementation of Torpor must consume very little power and introduce negligible energy overheads - ideally, the added power cost should be much smaller than what is consumed by the always-on tasks. For this reason, Torpor's implementation is split in a *Torpor-HW* part and a *Torpor-Software-Runtime*, running on the SoC's processor. This split makes the solution configurable, extensible and user-friendly without inflating the hardware and keeps the power consumption of Torpor's HW low.

*1) Torpor-Software-Runtime:* The runtime allows the user to specify tasks and make use of a scheduler with a priority mapping function. This function, depending on the application state, specifies which tasks are executable and assigns an execution priority. The runtime then abstracts this scheduling information (scheduler strategy, task executability and priority) and passes it to the HW.

The actual scheduling decision is then done by the control logic in Torpor-HW while the processor can go to Idle state. Every time the HW makes a decision, it wakes the processor's core up and notifies the Runtime, which then launches the energy-driven task and updates the abstracted scheduling information in HW.

*2) Torpor-HW:* The hardware part of Torpor does not only perform the scheduling decision, but also provides the required hardware components to supply energy to the load in a controlled manner. Moreover, it provides the monitoring of $E_{\text{buff}}(t)$ and $P_h(t)$ to enable *dynamic power-aware* scheduling. The Torpor-HW (fig. 3) consists of the following blocks:

- An *energy controller*, that uses harvesting power to buffer energy and delivers it at the load's operating voltage.
- A *power switch and a battery* to ensure that the load has enough power to perform its always-on tasks even when there is not enough harvested energy available.
- An *ADC* that measures $V_{\text{buff}}(t)$and estimates $E_{\text{buff}}(t)$ and $P_h(t)$.
- A *characterization power switch* used by Torpor to switch over to an external power source. This is used before the deployment of the WSN to automatically measure the required energy of the application tasks (see section V-D).
- The *Torpor control logic* that performs the actual scheduling decision based on the ADC values and decides when and what energy-driven task is executed.

## V. Torpor Implementation

To achieve maximal design density and cost efficiency, it would be desirable to integrate the entire Torpor-HW on the WSN SoC. However, the Torpor-HW contains several power-electronic blocks which are not easy to co-integrate in advanced technology nodes. Therefore we propose to keep most of the Torpor-HW external and implement it with discrete components – except the ADC and control logic (purple in fig. 3), which we suggest to be integrated in the SoC. This avoids I/O power dissipation for the communication between the control logic and runtime running on the SoC. Additionally, the proposed external part of Torpor is mainly powered by harvesting power, so its consumption is less critical.

To verify Torpor in a real-world scenario we implemented a complete system prototype as a proof of concept. We did not fabricate the SoC with integrated Torpor logic, but we built the system with discrete components and emulated the control logic on a low-power FPGA. Our prototype is based on a solar-powered WSN equipped with a microcontroller (LPC54102, NXP) and the control logic implemented on a low-power FPGA (IGLOO nano, Microsemi).

### *A. Torpor-Software-Runtime*

The Runtime running in the LPC core provides the main execution loop of the system, which is illustrated in fig. 4. After the initial boot, the application's tasks are declared with the API provided by Runtime: a maximum of 8 tasks in a chain are assumed, where each task is represented by a function that may require result item(s) from the previous task and provide a number of result items to the next task. The transfer of the result items is done using FIFO circular buffers provided by the API with configurable length. The energy required for each task to execute is communicated by setting the minimum buffered voltage threshold needed. A scheduling policy is selected, by providing a function that assigns priorities to the task according to their position in the chain, their energy thresholds, power needs (if available), the data buffer states and a scheduling strategy. The available tasks for execution are marked accordingly. Then, the main loop is started. First, the task priorities are calculated for the first time. The hardware is configured accordingly (with the desired ADC sampling and filtering rates, the input power threshold and integration settings) and the system then enters its low-power Idle state, where only its always-on functions are active. The hardware is monitoring the $V_{\text{buff}}$ and takes the system out of its Idle state when a energy-driven task is to be executed, by using an interrupt signal. The Runtime then reads the chosen task and executes it, updates the priorities, passes them to the hardware, signals its activation and goes again to its Idle state.

### *B. Torpor-HW*

The *Energy Controller* consists of a boost converter, an energy storage element and a buck converter (see fig. 3).

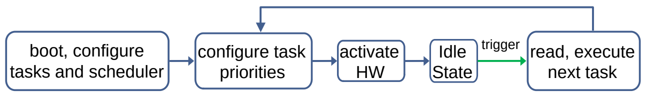
boot, configure tasks and scheduler
configure task priorities
activate HW
Idle State
trigger
read, execute next task

Fig. 4. Simplified state diagram of Torpor's Software-Runtime. Following the initial configuration, the Software-Runtime updates on every iteration the task priorities and waits on its idle state until the HW signals a task to be executed.

When the transducer produces energy, an ultra low-power boost charger for harvesting application (BQ25505, TI) is used to charge a $330\,\mu$F ceramic capacitor[1]. An ultra low-power step-down buck converter (TPS62740, TI) delivers the energy stored in the capacitor to the load at the required voltage (1.8 V). This step-up and step-down topology is commonly found in micro-energy harvesting systems [29], [5], [30] since decoupling allows simultaneous maximum power point tracking (MPPT) on the transducer side and minimum power point tracking on the load.

The *Battery Switch* (TPS3610, TI) automatically switches the load's power source to the *Battery* when low harvesting power input causes $V_{\text{load}}$ to drop below a threshold. The switch has a low on-resistance of $0.6\,\Omega$ and thus introduces a negligible loss.

The *Torpor control logic* and the *ADC* as depicted in fig. 5 are to be integrated in the node's SoC. This logic is always-on, monitors $V_{\text{buff}}$ and signals to the core when to execute which task. In our prototype, an external ultra low-power ADC component (ADS7040, TI) is used and the control logic is implemented in a low power FPGA (IGLOO nano, Microsemi). The FPGA connects to the ADC and the SoC over two separate Serial Peripheral Interfaces (SPI). Over the SPI interface the SoC core can set Torpor's configuration registers and read the status registers. An IRQ line allows Torpor to trigger the core to execute an energy-driven task.

As part of the control logic a block called *Sampling Unit* manages the interfacing with the ADC. It allows to sample $V_{\text{buff}}$ with a configurable sampling rate, filters those samples and estimates the input power $P_h(t)$ from their rate of change.

The control logic provides 8 abstract task slots, which are ordered in descending priority and configured by the Runtime. For each task slot, the corresponding *task energy execution threshold* ($V_{\text{thres,i}}$) is stored in a configuration register. The thresholds are expressed in voltage to be directly comparable to the ADC samples. A bit-mask (*Execution Mask*), stored in a configuration register, marks which slots are active. The mask is set by the Runtime depending on the application state to indicate which energy-driven tasks are currently available for execution.

This information, i.e. the threshold comparison results masked by the execution mask and the $P_h(t)$ estimate, is fed to the control logic's *Finite State Machine* (FSM). The FSM checks if the highest-priority task can be executed or alternatively, when the input power $P_h(t)$ drops below a configurable threshold, if any other lower-priority task could be launched instead. When the FSM comes to a decision, it writes the slot ID of the task to launch in a status register and sends an interrupt to the microcontroller.

[1]The size of the capacitor was chosen to provide sufficient energy storage for the most energy-intensive energy-driven task we evaluated.

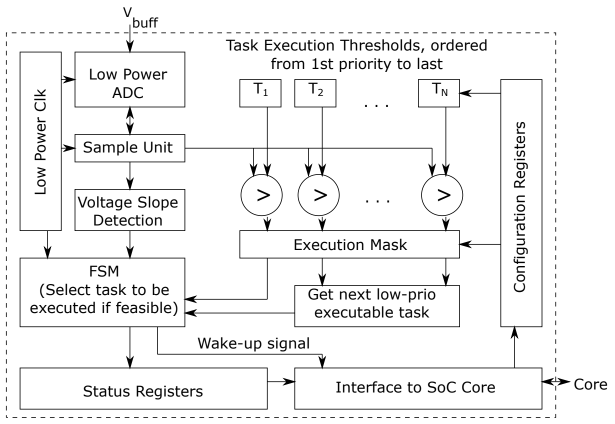

Fig. 5. Simplified block diagram of the HW logic to be integrated.

## C. Scheduling Policy Implementation

As explained, Torpor's control logic allows implementing different scheduling policies by configuring the Runtime accordingly. To implement a scheduling policy we need to define a *priority mapping function*, that determines when the HW is going to notify the Runtime to execute a task. As long as the $P_h(t)$ is above the set threshold, the HW will wait until the highest priority task can be executed, otherwise it will wait until *any* task can be executed, pick the most prioritized among them and report it back to the Runtime. Tasks that must not be executed due to software dependencies are masked out using the appropriate register.

The abstract task slots of the control logic allow the execution of either one or more tasks per burst. In order to implement conventional static *single burst* scheduling, the Runtime can merge all tasks in a single abstract task and configure the control logic accordingly to simply notify the Runtime when this one abstract task can be executed.

In order to implement the static *split* scheduling, the Runtime will keep all tasks declared separately. The task priorities in this case are not relevant, as each time the Runtime will always mark only one task (the next in chain) as executable and mask-out the rest. This way the Runtime dictates completely the order of execution (static) and the HW will wake the system's core up each time the appropriate execution thresholds are reached.

On our first proposed dynamic scheduling policy (*Energy-oriented*) we follow a simple strategy to maximize the throughput of the application and favor the largest in energy tasks. We program the priority mapping function to assign the highest priority to the task closest to the end of the chain, while considering the eligibility of the task depending on the FIFO fill states. This decision favors throughput as it prefers to keeps FIFO fill states low. The Runtime will assign this task to the first (highest priority) abstract task slot $T_1$ in the HW control logic The rest of the tasks are prioritized according to their energy requirements, with the mapping favoring the more demanding ones. This way, when the input power is low $P_h(t) \leq P_{h,\text{high}}$, the scheduler tries to launch any task possible to minimize energy losses by using the most of the available energy for whatever it can be used. To do so, the scheduler first checks which tasks can be run (enough energy and mask bit) and then launches the one with highest priority.

Our second proposed dynamic scheduling policy will be the *Power-oriented* one. Here, we program the priority mapping function to assign the highest priority to the task with the highest power demands and this is the one placed on the $T_1$ slot of the HW control logic. This way, periods of high input power will be matched to our system's most power-demanding functions. The rest of the tasks are prioritized from the *lowest* to the *highest* powers, so that when the input power is low, low-power tasks are favored whenever possible. Again, only tasks that can be executed based on their software dependencies are left unmasked.

### D. Task Characterization

To ensure that energy-driven tasks run reliably, Torpor checks if there is sufficient buffered energy to execute a task before scheduling this task for execution. In order to do this, all task energies need to be known. This information could be obtained through manual trial-and-error engineering or by complex energy estimations, one being time-consuming and the other potentially inaccurate or overly pessimistic. As an example, a sample CNN application[31] developed for our same microcontroller is composed of more than 5000 tasks, which is beyond the reach of manual labor. To alleviate this effort, Torpor provides an automatic task characterization feature to determine the required values. While we generally assume and observe that the energy consumption of individual tasks such as sensing and transmitting exhibit little variation, we do note that the absolute energy consumption per task depends heavily on the peripherals being used. These values are essential for the minimization of the storage element, such that task atomicity can be guaranteed. In processing tasks, there can be some variability depending on the type of application, but in most cases a worst-case execution can be artificially enabled during characterization to ensure that worst-case energy consumption figures are obtained, which are required for reliable operation.

During automatic task characterization, each task is characterized sequentially. To do so, as depicted in fig. 3, an external power source is connected to the energy controller through a controllable power switch. First, a large test capacitor is charged to its maximum, $V_\text{max}$, then the input power is cut off and the task to be characterized is executed. When the task is completed, the remaining voltage across the buffering capacitor $V_\text{meas,i}$ is measured.

Repeating the procedure with smaller capacitors will reach the minimum functional capacity and each time the procedure is carried out a pre-configured number of times to average the estimate.

While we can estimate the consumed energy by the task with

$$\tilde{E}_\text{task,i} = \frac{1}{2} C \cdot V_\text{max}^2 - \frac{1}{2} C \cdot V_\text{meas,i}^2 \quad , \tag{2}$$

ultimately, the HW logic needs the corresponding voltage threshold $V_\text{thres,i}$ to compare to the current $V_\text{buff}(t)$ in order to determine whether sufficient energy is available to execute a task. As an additional constraint, during task execution, $V_\text{buff}$ may not fall below the set output voltage $V_\text{load}$ of the buck converter in the energy controller. Otherwise, the battery switch will jump in and the load will start draining the battery. To prevent this, we compute the threshold in a way such that a minimal residual voltage $V_\text{min}$ is guaranteed to be present on $V_\text{buff}$ after task execution.

Given these constraints, we set the voltage threshold to

$$V_\text{thres,i} = \sqrt{V_\text{max}^2 - V_\text{meas,i}^2 + V_\text{min}^2} \quad . \tag{3}$$

The voltage-dependent buck-converter efficiency makes the energy consumed by the task largest when the execution begins on higher $V_\text{buff}$, which makes the threshold calculated by this procedure a worst-case estimation. If desired, by adding some additional margin to $V_\text{min}$, higher safety margins can be taken into account. Finally, the voltage thresholds are stored within Torpor's always-on domain to access them with low energy overhead. The task's execution duration $T_\text{task,i}$ is at the same time being measured by a hardware timer on the LPC processor, appropriately scaled. This enables the calculation of an estimated average task power consumption

$$\tilde{P}_\text{task,i} = \frac{\tilde{E}_\text{task,i}}{T_\text{task,i}} \quad , \tag{4}$$

which gives the scheduling policies one more metric to potentially base their decisions upon. As it will be shown in section VI, this can be a vital factor in scenarios where the input power may surpass for short intervals the power consumed by the system during the execution of some of its tasks.

## VI. Evaluation

With Torpor we propose to add additional hardware components to the WSN in order to increase its overall energy efficiency. In this section, we will evaluate what can be gained with Torpor in terms of power efficiency and what the overhead is for the added functionality.

First, we will evaluate the power consumption of the always-on parts of Torpor. For this evaluation, we will investigate the final target implementation with the control logic and ADC co-integrated on the SoC as proposed in section V. The overhead introduced by the Torpor software runtime will be also addressed, both in terms of its size and time overhead. Afterwards, we will experimentally verify and evaluate Torpor on our FPGA-based prototype and quantify the achieved gains. The node will be powered with a solar panel placed in a controlled lighting environment for reproducible experiments.

## A. Power and Software Overhead

*1) Estimation of Torpor's Power Consumption:* For this estimation, we consider only the parts of Torpor that can be powered from the battery. These are the integrated parts (control logic, ADC) and the power switch. The power consumption of the booster and buck converter were omitted as they are powered only by harvested energy and their overhead will be considered when computing the achieved gains.

The implemented Torpor hardware control logic was synthesized in a 22 nm FDX technology (0.65 V, TT, 25 C) and its power consumption was estimated in PrimeTime using activity vectors derived from ModelSim. The logic requires 1700 $\mu m^2$ of area and consumes 1.57-1.96 µW depending on the activity. The operating frequency used for the idle state was 32 KHz while the configuration and sample fetching was simulated with clock bursts of 2.4 MHz. The lower bound indicates the idle state while the upper bound indicates the maximum momentary consumption. The power consumption is dominated by leakage and could be further optimized with using special low leakage gates, which have longer channel transistor and/or thicker gate oxide. Commercially available components (TPS3610, ADC7040) were measured in order to deduce the power required by the switch and the ADC.

The estimation results are summarized in table I. It shows that for the proposed partially integrated solution the power consumption of Torpor (<4 µW) can be neglected compared to the power of the load (600 µW-90 mW) present in our WSN.

As the ADC and the control logic are meant to be co-integrated on the SoC and their consumption is negligible, we power their discrete implementation on our concept prototype (ADC7040, FPGA) with an external power supply.

TABLE I
TORPOR POWER ESTIMATION FOR THE PROPOSED INTEGRATED SOLUTION

| Module | Consumption |
|---|---|
| Torpor Logic | 1.57 - 1.96 µW |
| ADC | 0.2 - 1 µW |
| Power switch | 800 nW |
| Total | under 4 µW |

*2) Software Overhead:* To estimate the SW overhead in terms of non-volatile memory footprint, an example application with 3 tasks was written and compiled once using the Torpor runtime software and once without. The difference in binary size was approximately 5kB with standard compiler optimization, out of which the 2.7kB are needed for the automatic task characterization routine, that does not however have to be in the final binary of the application, once the tasks are defined and characterized.

To estimate the SW overhead in terms of execution time overhead, it was measured that per task execution that was approximately 430us with a CPU clock of 12MHz and 1MHz SPI clock. For the tasks used in our evaluation that will be described below, this execution time overhead amounted for the 0.1-1% of the execution time. The actual instructions needed could be executed much faster with higher clock-rates but a large part of the measured overhead is attributed to SPI read/write commands that are configured to be blocking. The SPI communication was necessary on this prototype as discrete components were used, but in a fully integrated torpor solution it will not, so this overhead is expected to be lower.

## B. Experimental Setup

To evaluate the benefits of Torpor, the implemented setup was configured to execute a number of synthetic applications under different input power conditions using different execution strategies (fig. 6).

A solar panel was placed inside a solar test-bed, an isolated environment where the illuminance levels can be set. The solar panel was connected to the input of our booster circuit, providing a controlled and reproducible way of specifying the available input power to our system. The voltage and current at various points of interest were measured using a Rocketlogger [32], an open-source measurement device meant for the characterization of harvesting powered IoT devices. Rocketlogger not only supports the measurement of multiple voltage and current channels but also has a wide range and high accuracy. This enabled the calculation of several figures of merit, the main being:

- The energy efficiency factor $E_{\text{eff}}$, indicating the percentage of the harvested energy was used by the load while in its active or idle state
- The execution rate, indicating the number of application executions per minute
- The average power consumed by the load $\bar{P}_{\text{load}}$

In IoT applications that rely *primarily* on volatile energy sources, only opportunistic execution can be achieved. The lifetime is mainly determined by the choice of battery and the always-on tasks. The energy efficiency and execution rate are crucial figures of merit to show how well a platform will perform in given harvesting conditions. Achieving higher efficiency and execution rate than the target of the application may often lead to redesigning with a smaller harvester and reducing the cost.

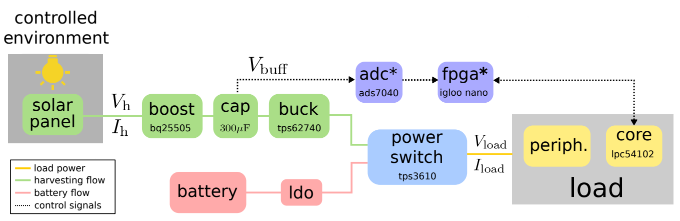

Fig. 6. For repeatable experiments of Torpor's harvesting-driven mode, the solar panel was placed in a controlled environment. The harvesting and load power were recorded for analysis. The ADC and FPGA were externally powered.

## C. Validation of Task Characterization

To evaluate the task characterization process, all voltage execution thresholds for the scheduler evaluation were derived

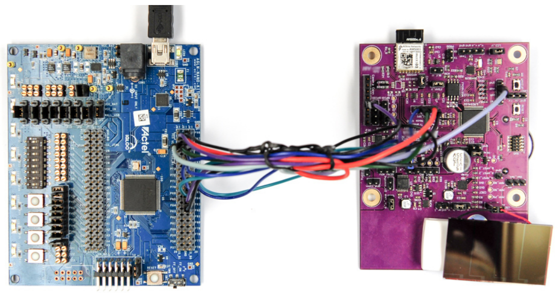

Fig. 7. Photo of FPGA-based Torpor prototype. The blue PCB (left) is a commercial FPGA evaluation kit, the purple-PCB (right) is custom-made and includes all other discrete components used for the evaluation.

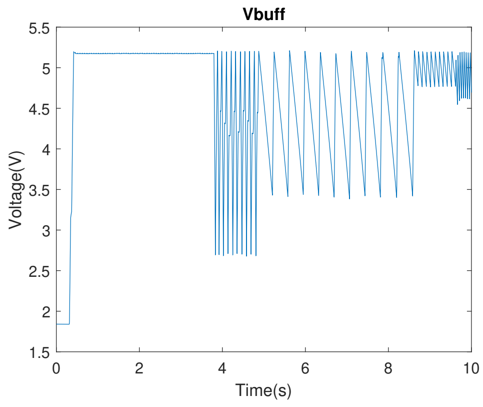

Fig. 8. System starting up with the characterization process for a chain of 3 tasks, followed by normal execution.

using this method. An example run of the process, used to characterize the task-set for the Energy-oriented and Power-oriented dynamic schedulers comparison, can be seen in fig. 8. One after the other, each task is executed a pre-configured number of times from maximum voltage and the average threshold is calculated. The experiment was repeated to test its repeatability and the thresholds were found to be within 2 LSBs of difference between each run. The results can be found in table II. The characterized power drawn at $V_{\text{buff}}$ is always more than the one drawn from $V_{\text{load}}$, due to the voltage-dependent buck controller's efficiency. The voltage threshold estimation is scaled so that the interval [0...255] is mapped to [0...5.32]V of $V_{\text{buff}}$. The task power estimation is scaled arbitrarily so that the expected task power numbers can fit to be represented in the [0...255] range with sufficient accuracy. Only the relative power between tasks matters and the ratio between the estimated task powers is matching the ratio of the $P_{\text{load}}$ while the tasks are being executed.

### D. Validation of Battery- and Harvesting- driven modes

Low or zero harvesting input power, should lead the system to fall back to Battery-driven mode without losing $V_{\text{load}}$, pausing energy-driven tasks. This scenario was tested and in fig. 9. It can be seen that Torpor manages to retain its state and as soon as there is sufficient $P_h$, switches back to Harvesting-driven mode, picking up the task execution from where it left off.

TABLE II
EXECUTION VOLTAGE THRESHOLDS AND POWER ESTIMATIONS BY THE TASK CHARACTERIZATION PROCESS

| Task | Sense | Process | Transmit |
|---|---|---|---|
| Task Duration (s) | 0.035 | 0.333 | 0.088 |
| $P_{\text{load}}$ (mW) | 77 | 5.1 | 5.1 |
| $E_{\text{load}}$ (mJ) | 2.695 | 1.7 | 0.449 |
| $V_{\text{thres,est}}$ [0...255] | 235 | 213 | 144 |
| $P_{\text{task,est}}$ [0...255] | 221 | 17 | 14 |
| $E_{\text{task,est}}$ (mJ) | 3.025 | 2.351 | 0.665 |

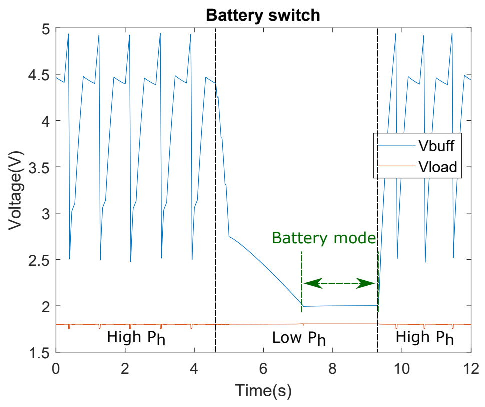

Fig. 9. System falling back to battery mode, pausing energy-driven tasks and recovering when enough harvesting power is available.

### E. Scheduler Evaluation

In order to evaluate different Torpor scheduling policies, synthetic applications will be executed under several input power conditions. Applications are composed of a low power always-on task, and two types of energy-driven tasks: *medium power* and *high power*. The power consumption is approximately 600 µW for always-on tasks, 5 mW for *medium power* tasks and 77 mW when executing additional *high power* tasks. These power levels represent sense and transmit tasks typically found in WSN applications. Static schedulers are evaluated under constant harvesting power and then contrasted with the dynamic *energy-oriented dynamic* scheduler under variable harvesting power. Finally, the *energy-oriented dynamic* scheduler is compared to a *power-oriented dynamic* scheduler under similar variable harvesting power scenario but different application. The evaluation took place for time long enough for the system to reach steady state.

*1) Static Scheduling Evaluation:* We consider two corner cases of static schedulers. *Single* executes all energy-driven in a single energy burst, while *split* schedules one burst per task. For this experiment, a synthetic application with five *medium power* tasks was chosen. Two constant harvesting levels were tested: 1.3 mW and 3.2 mW. The application execution rate per minute and the energy efficiency are presented in table III and the voltage of the buffering capacitor for the case of $P_h = 1.31\,\text{mW}$ is depicted in fig. 10.

TABLE III
EVALUATION OF STATIC SCHEDULERS WITH CONSTANT $P_h$.

| | $P_h = 1.3\,[\text{mW}]$ scheduler single | split | ratio | $P_h = 3.2\,[\text{mW}]$ scheduler single | split | ratio |
|---|---|---|---|---|---|---|
| **exec/min** | 2.6 | 6.9 | ×2.6 | 31.2 | 39.1 | ×1.3 |
| $E_{\text{eff}}$ [%] | 55.2 | 67.0 | ×1.2 | 58.0 | 66.7 | ×1.2 |
| $\bar{P}_{\text{load}}$ [mW] | 0.71 | 0.87 | ×1.2 | 1.84 | 2.16 | ×1.2 |

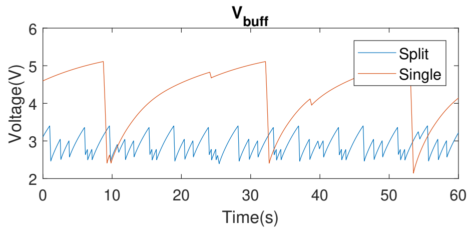

Fig. 10. The voltage on the buffering element when using *single* and *split* schedulers under constant harvesting power ($P_h = 1.3\,\text{mW}$). The *single* scheduler results in $\bar{P}_{\text{load}} = 0.71\,\text{mW}$ while *split* in $\bar{P}_{\text{load}} = 0.87\,\text{mW}$.

*2) Task Energy-Oriented Scheduling:* To evaluate the benefits of dynamic scheduling strategies, the input power was variable but periodic, alternating between low and high energy availability. In the first evaluated case (Case I), the illuminance was set to provide a low base of available input power with peaks of high available input power (lasting 3 s and repeating every minute). In the second case (Case II), the peaks were shorter (0.5s) and provided very high available input power. The average $\bar{P}_h$ for each case is reported in table IV. Two synthetic applications consisting of *medium* and *high power* tasks were chosen, with 5 tasks for Case I and 3 tasks for Case II. The *energy-oriented* dynamic scheduling strategy, as explained in section V-C, is evaluated and presented in table IV. The voltage across the buffering capacitor in Case II is presented in fig. 11.

TABLE IV
EVALUATION OF DIFFERENT SCHEDULERS WITH HIGHLY VARIABLE $P_h$.

| | Case I scheduler[a] split | dynamic | ratio[b] | Case II scheduler[a] split | dynamic | ratio[b] |
|---|---|---|---|---|---|---|
| $\bar{P}_h$ [mW] | 1.33 | 1.32 | ×1.0 | 1.15 | 1.29 | 1.1 |
| **exec/min** | 3.0 | 3.7 | ×1.3 | 2.8 | 6.2 | ×2.2 |
| $E_{\text{eff}}$ [%] | 60.6 | 66.4 | ×1.1 | 61.2 | 69.7 | ×1.1 |
| $\bar{P}_{\text{load}}$ [mW] | 0.80 | 0.88 | ×1.1 | 0.70 | 0.90 | ×1.3 |

[a] Both schedulers run one energy-driven task per burst. *Split* has a static order while *dynamic* depends on harvesting power and application state.
[b] Ratio refers to the dynamic scheduler evaluation metric over the split scheduler evaluation metric.

*3) Task Power-Oriented Scheduling:* In this experiment we consider a similar scenario of low available input power for 59s and very high peaks for 1s, repeating periodically, but this time considering a *high power* Sense task, followed by a long *medium power* Process task and a short *medium power* Transmit task. The schedulers evaluated are the *energy-oriented* dynamic scheduler and a *power-oriented* dynamic scheduler which makes use of the task power estimations obtained by the task characterization process (table V). This scheduler prioritizes the highest power executable task when the input power is above a threshold and prioritizes the lowest power tasks when the input power is below that (fig. 12, fig. 13).

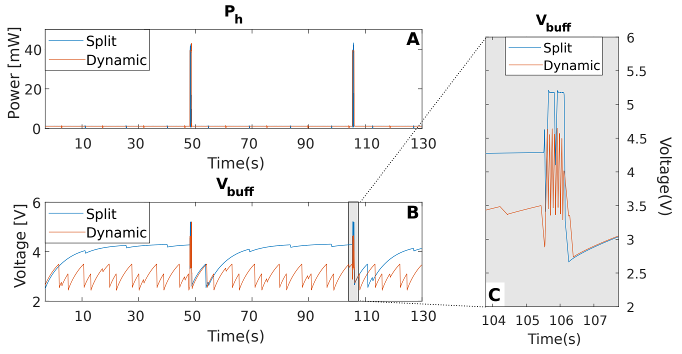

Fig. 11. Traces from Case II experiments of *split* and *dynamic* schedulers. A) Harvesting power B) $V_{\text{buff}}$ and C) Zoomed in $V_{\text{buff}}$ to show saturation. The dynamic schedulers can reduce the energy losses and avert the saturation effect on short moments of high input power.

TABLE V
EVALUATION OF DYNAMIC SCHEDULERS WITH HIGHLY VARIABLE $P_h$.

| | scheduler E-oriented | P-oriented | ratio |
|---|---|---|---|
| $\bar{P}_h$ [mW] | 1.34 | 1.97 | ×1.5 |
| **exec/min** | 8.5 | 13.2 | ×1.5 |
| $E_{\text{eff}}$ [%] | 63.6 | 65.5 | ×1.0 |
| $\bar{P}_{\text{load}}$ [mW] | 0.85 | 1.29 | ×1.5 |

## F. Discussion

The results show that the execution rate of energy-driven tasks can be easily doubled when using *split* scheduling as opposed to *single burst* scheduling. This is due to the reduction of the load power consumption. Since the load behaves like a current sink, a lower $V_{\text{buff}}$ also reduces the power dissipation from the buffering capacitor. The saved energy is then translated into useful load energy and therefore leads to more application executions.

Dynamic schedulers behave exactly as static *split* schedulers when harvesting power is *constant*, or when all energy-driven tasks are of equal power and energy. In these cases, both dynamic and static schedulers have the same choice of available tasks, so the dynamic schedulers cannot differentiate their behaviour and influence the $P_{\text{load}}$ in any way different than the static split scheduler. The benefits of using dynamic

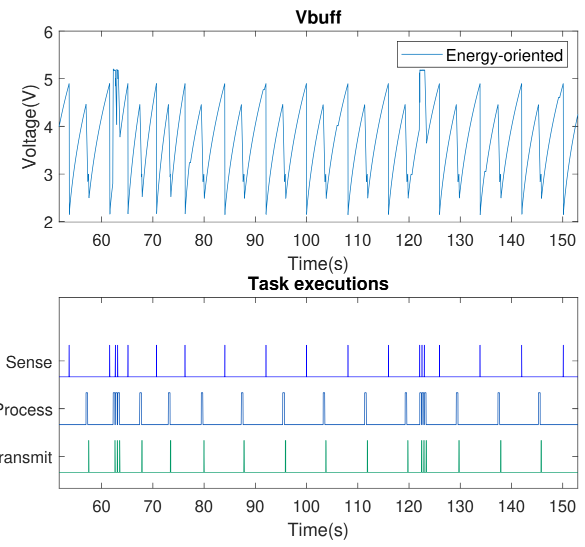

Fig. 12. A time interval of the voltage on the buffering element when using the *energy-oriented* scheduler under highly variable harvesting power and the order of execution of its tasks. Energy buffer saturation is not avoided.

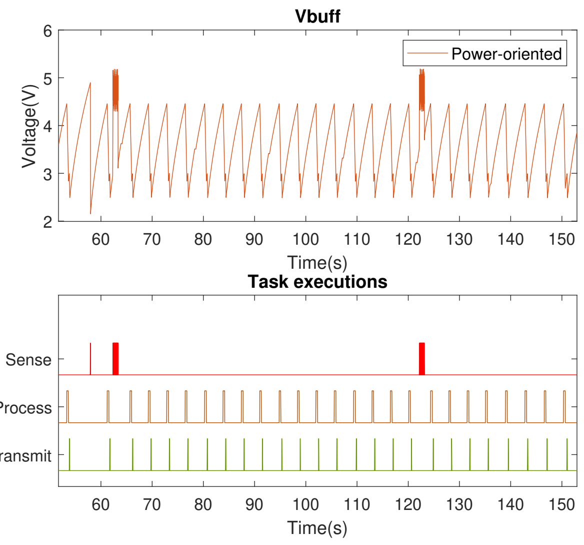

Fig. 13. A time interval of the voltage on the buffering element when using the *power-oriented* scheduler under highly variable harvesting power and the order of execution of its tasks. Energy buffer saturation is avoided.

scheduling become visible when there is enough diversity at the input power levels and available tasks.

Compared to the (static) split scheduler, the performance improvement of the (dynamic) energy-oriented scheduler stems from two effects. The first is again the reduction of the load power consumption; when the input power is low, static schedulers will execute lower-energy tasks or wait until enough energy is accumulated for higher-energy tasks depending on which task gets its turn for executed. By contrast, dynamic schedulers prioritize low-energy tasks whenever the input power is low and by doing so limit prolonged stays in high $V_{\text{buff}}$ levels. The improvement in energy efficiency is positively correlated to the execution rate, but differs depending on the amount of energy that the node is using for its always-on tasks as opposed to its energy-driven tasks. The second effect is the avoidance of $E_{\text{buff}}$ saturation and is visible in Case II of the comparison between static and dynamic schedulers. If, for example, an application consists of a low-power and high-power task and the harvesting power falls in between, a static scheduler can saturate $E_{\text{buff}}$ when running the low-power task. Dynamic schedulers can mitigate this by matching time periods of low $P_h$ to less demanding tasks and time periods of high $P_h$ to the most demanding tasks. Since the dynamic scheduler in Case II can avoid saturation of $E_{\text{buff}}$, it manages to harvest more input energy. This is why the improvement ratio of its $E_{\text{eff}}$ differs from that one of its $P_{\text{load}}$. Since the $E_{\text{buff}}$ will not be saturated, the total harvested energy $E_{\text{in}}$ can be increased. This means that the total energy consumed by the load may increase, without necessarily increasing the metric of energy efficiency. Overall, the increased energy efficiency results in the load receiving a larger portion of the harvested energy. Thanks to the decoupled nature of the energy management system, the application circuit is guaranteed to operate at its most efficient point, regardless of the transducer's current and voltage. In this way, the system's energy efficiency is directly correlated to the application execution rate.

On the comparison between the two dynamic schedulers however, the energy-oriented scheduler can no longer avoid saturation, because the high-power task is the first in the task-chain (Sense) and this conflicts with its policy to prioritize tasks close to the end of the chain whenever the input power is high. In such cases, it is crucial for the scheduler to know which task has the highest power needs and this is what the power-oriented scheduler utilizes to perform better. The power-oriented scheduler, clearly defers the execution of its highest-power task until the input power is high and during these short time intervals this task monopolizes the execution time, making the most out of the available harvesting energy.

It should be noted that even though the harvesting power was, on average, less than 4 mW for all experiments, the load was able to execute power-hungry tasks of up to 90 mW. This is thanks to the burst-generation scheme that efficiently reduces the average power of energy-driven tasks to match the harvesting power, while still guaranteeing always-on tasks.

The software overhead may be negligible or quite relevant as cost, depending on the size and duration of the application's tasks. It will also greatly vary if Torpor is implemented on a different MCU as part of an ASIC. No effort has been put to optimize either in size or speed at this prototyping phase. A direct optimization would be to revise the code so that it is not stalling when using the SPI and a further optimization would be not to re-write all the task priorities when they have not been changed. With SEMU is implemented as an ASIC peripheral, another more optimal internal communication scheme would be used instead of SPI anyway.

## VII. Conclusions and Future Work

In this work, we have presented Torpor, a power-aware hardware scheduler that enables IoT nodes to efficiently execute part of their applications using irregular harvesting power, while still guaranteeing their always-on required functionality with a battery. Torpor supports both static and dynamic scheduling policies and is highly configurable, offering an interface to the application designer and an automatic task characterization procedure. As demonstrated by a complete system prototype based on discrete components, Torpor's input-power-aware dynamic schedulers can improve the energy efficiency and execution rate of energy-driven tasks by 1.2× and 2.6×, respectively. Using dynamic schedulers aware not only of the input power but also of the tasks' power, may improve the harvested energy by 1.5× resulting into the improvement of the execution rate by the same factor. The power overhead introduced by Torpor's hardware when integrated in a SoC is estimated to be under 4 µW, making it suitable for a wide range of IoT applications.

This work opens up the potential of future extensions in various directions, such as more sophisticated dynamic schedulers. One main aspect to take into account is applications that include non-atomic (i.e. interruptible) tasks. Non-atomic tasks can be handled as atomic but not vice-versa. In the case of non-atomic tasks, depending on the cost of saving the task state, pausing execution and resuming, additional improvement could be achieved. Another aspect that could be considered is look-ahead schedulers that would expect to make a more efficient task-scheduling decision in the immediate future, as in [33]. Such an approach would imply predicting the input power in the future and depends on the predictability of the energy source, but its benefits versus its risks could be explored.

**Acknowledgements** This research was co-funded by Swiss National Science Foundation grant 157048: Transient Computing Systems and a grant by NXP Semiconductors. The authors would like to thank L. Sigrist, M. Coray, F. Amstutz, S. Schweizer and A. Blanco for their contributions.

## References

[1] N. A. Bhatti *et al.*, "Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences," *ACM Transactions on Sensor Networks (TOSN)*, vol. 12, no. 3, p. 24, 2016.

[2] C. Moser *et al.*, "Adaptive power management for environmentally powered systems," *IEEE Transactions on Computers*, 2010.

[3] V. Raghunathan *et al.*, "Design considerations for solar energy harvesting wireless embedded systems," in *Proc. IPSN Symp.* IEEE Press, 2005.

[4] E. E. Lawrence *et al.*, "A study of heat sink performance in air and soil for use in a thermoelectric energy harvesting device," in *Proc. ICT Conf.* IEEE, 2002, pp. 446–449.

[5] S. Naderiparizi *et al.*, "WISPCam: A battery-free RFID camera," in *RFID (RFID), 2015 IEEE International Conference on*. IEEE, 2015.

[6] H. Jayakumar *et al.*, "QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers," in *Proc. Conf. on VLSI Design*. IEEE, 2014.

[7] Y. Wang *et al.*, "Storage-less and converter-less photovoltaic energy harvesting with maximum power point tracking for internet of things," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 2, pp. 173–186, 2016.

[8] S. Sudevalayam *et al.*, "Energy harvesting sensor nodes: Survey and implications," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 3, pp. 443–461, 2011.

[9] P. Anagnostou *et al.*, "Torpor: A power-aware HW scheduler for energy harvesting IoT SoCs," in *Proc. PATMOS Symposium*, 2018, pp. 1–8.

[10] A. Wang *et al.*, "Energy-scalable protocols for battery-operated microsensor networks," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 29, no. 3, pp. 223–237, 2001.

[11] V. S. Mallela *et al.*, "Trends in cardiac pacemaker batteries," *Indian pacing and electrophysiology journal*, vol. 4, no. 4, p. 201, 2004.

[12] A. S. Weddell *et al.*, "A survey of multi-source energy harvesting systems," in *Proc. DATE Conf.* EDA Consortium.

[13] L. Benini *et al.*, "Battery-driven dynamic power management," *IEEE Design & Test of Computers*, vol. 18, no. 2, pp. 53–60, 2001.

[14] A. Sinha *et al.*, "Dynamic power management in wireless sensor networks," *IEEE Design & Test of Computers*, 2001.

[15] T. Pering *et al.*, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proceedings of the 1998 international symposium on Low power electronics and design*. ACM, 1998, pp. 76–81.

[16] A. Gomez *et al.*, "Reducing energy consumption in microcontroller-based platforms with low design margin co-processors," in *Proc. DATE Conf.* EDA Consortium, 2015, pp. 269–272.

[17] ——, "Dynamic energy burst scaling for transiently powered systems," in *Proc. DATE Conf.* EDA Consortium, 2016.

[18] J. Hester *et al.*, "Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2015, pp. 5–16.

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

[20] P. A. Hager *et al.*, "A scan-chain based state retention methodology for IoT processors operating on intermittent energy," in *Proc. DATE Conf.* EDA Consortium, 2017.

[21] D. Balsamo *et al.*, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[22] C. Moser *et al.*, "Real-time scheduling for energy harvesting sensor nodes," *Real-Time Systems*, vol. 37, no. 3, pp. 233–260, 2007.

[23] B. Buchli *et al.*, "Towards enabling uninterrupted long-term operation of solar energy harvesting embedded systems," in *European Conference on Wireless Sensor Networks*. Springer, 2014, pp. 66–83.

[24] C. Lu *et al.*, "Efficient design of micro-scale energy harvesting systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 3, pp. 254–266, 2011.

[25] F. K. Shaikh *et al.*, "Energy harvesting in wireless sensor networks: A comprehensive review," *Renewable and Sustainable Energy Reviews*, vol. 55, pp. 1041 – 1054, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364032115012629

[26] A. Kansal *et al.*, "Power management in energy harvesting sensor networks," *ACM Trans. on Embedd. Comp. Sys.*, vol. 6, no. 4, 2007.

[27] M. Charkhgard *et al.*, "State-of-charge estimation for lithium-ion batteries using neural networks and EKF," *IEEE transactions on industrial electronics*, vol. 57, no. 12, pp. 4178–4187, 2010.

[28] C. M. Vigorito *et al.*, "Adaptive control of duty cycling in energy-harvesting wireless sensor networks," in *Proc. SECON Conf.* IEEE, 2007.

[29] M. Thielen *et al.*, "Human body heat for powering wearable devices: From thermal energy to application," *Energy conversion and management*, vol. 131, pp. 44–54, 2017.

[30] A. Gomez *et al.*, "Efficient, long-term logging of rich data sensors using transient sensor nodes," *ACM Trans. on Embedd. Comp. Sys.*, 2017.

[31] ——, "Thermal image-based CNN's for ultra-low power people recognition," in *Proceedings of the Computing Frontiers Conference*. ACM, 2018.

[32] L. Sigrist *et al.*, "Measurement and validation of energy harvesting IoT devices," in *Proc. DATE Conf.* EDA Consortium, 2017, pp. 1159–1164.

[33] C. Bergonzini *et al.*, "Comparison of energy intake prediction algorithms for systems powered by photovoltaic harvesters," *Microelectronics Journal*, vol. 41, no. 11, pp. 766–777, 2010.