Torpor: A Power-Aware HW Scheduler 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 available harvested energy. In this work we propose Torpor, a power-aware HW scheduler which continuously monitors harvesting power and in combination with its software runtime, dynamically activates system functions depending on the available energy. By performing a few key functions in HW, 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 with a sample power-aware dynamic scheduling policy, we can have a 2× or more improvement in execution rates compared to static (power-ignorant) policies. 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.

The problem of designing a system for long-term operation without predictable energy sources nor service degradation is an open challenge. A combination of primary battery and harvesting-based solutions is necessary to provide guarantees on the system's lifetime and still be able to tap into the abundance of available environmental energy [8], [9]. Traditional design methods use harvesting to recharge a secondary battery, as shown in fig. 1A. Only if accurate models of (high-power) periodic energy sources are available, can harvester and secondary battery sizes be minimized.

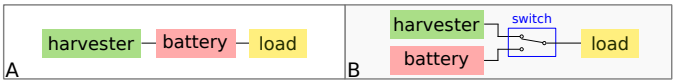

Fig. 1. A) Traditional harvesting-based nodes combine harvested with stored energy. B) Torpor-based nodes switch between battery and harvesting powered operating modes.

In this work, we propose Torpor, shown in fig. 1B, that does not require any assumptions on the source periodicity for long-term operation. It is enough for energy to be produced, even irregularly; its consumption will be opportunistic. Whenever the harvester is not producing any power, the system is powered by primary battery and can thus provide guarantees for low-power, always-on tasks. When the harvester produces enough energy, the system switches to harvesting-powered mode and additional high-power tasks can be supplied from cheap harvested energy. To use harvested energy efficiently, the system needs to monitor its input power and schedule high-power tasks carefully. To this end, we develop a low-overhead HW scheduler that can support different scheduling strategies. In highly variable harvesting scenarios, we show that, compared to static scheduling policies, dynamic ones can double the execution rate of high-power energy-driven tasks and still guarantee continuity of always-on tasks. Furthermore, our system helps designers pre-characterize their task energies, using an external source, before deployment. In summary, the

contributions of this paper 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 and can implement both static and dynamic scheduling policies for energy-driven tasks.
- A sample dynamic scheduling policy that improves task execution rates by over 2× under variable harvesting conditions.
- 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.

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. 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 [12], [13], and low power design [14], [15].

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 [16]. In the latter, tasks are not executed atomically and thus require state retention techniques [6], [17]. 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. However, these systems require high-energy, periodic sources (i.e. the sun) to keep harvesters and storage elements cost effective [18]. Furthermore, these systems need advanced power management techniques [19] with state-of-charge estimation [20] to adapt the system's service [21].

In this work, we propose 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 power-aware scheduling algorithms, which can be executed in hardware with little overhead.

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.

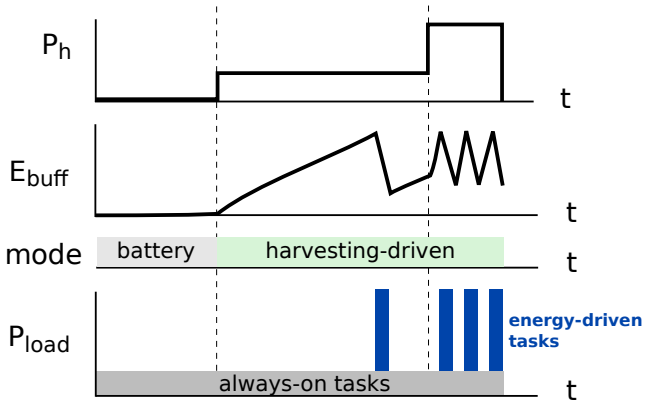

Fig. 2. When there is no harvesting power ($P_h = 0$), Torpor supplies always-on tasks with battery power. As harvesting power becomes available, Torpor uses burst-generation schemes for power-hungry, energy-driven tasks.

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 battery is switched out, thus prolonging its lifetime. To enter this mode, it is only necessary that the harvesting power (P_h) be larger than the load's always-on power. If this condition is met, we can utilize the burst generation schemes presented in [5], [16]. These schemes can work with very little input power and still supply power-hungry, energy-driven tasks. 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_{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 [16], 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_{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. If there is only one possible decision, as is the case in the *single burst* approach, then scheduling only depends on E_{buff}. When multiple choices are available, the scheduler also needs to know the application state, e.g. buffer states and task dependencies, to choose among available tasks. We argue that *static* schedules may still lead to inefficient use of the harvesting power when this exhibits highly variable behavior. To alleviate this, we propose that execution of tasks be activated in a *dynamic* order, according not only to application state and E_{buff}, but also P_h.

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).

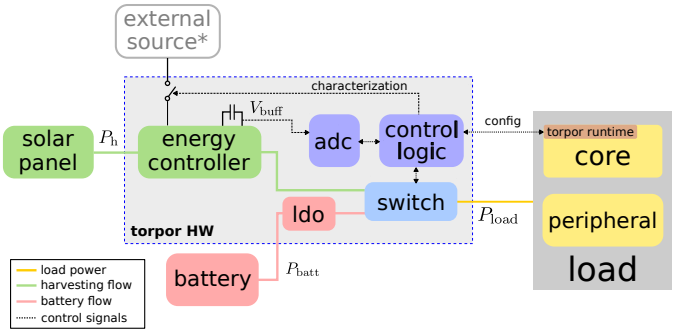

Fig. 3. Top-Level Architecture: An overview of the blocks present in a WSN that uses Torpor. A control logic manages the state of the Load, which is either powered by harvested energy or by battery. (*) The external power source is only used for characterization before WSN deployment.

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 $P_h(t)$ 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.

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 behaviour, 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 triggers the core 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).

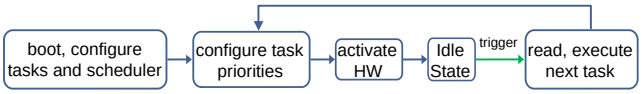

Fig. 4. Simplified state diagram of Torpor's Software-Runtime. Following the initial configuration, the Software-Runtime updates on every iteration the task priorities and waits on its idle state until the HW signals a task to be executed.

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 and a scheduling strategy is selected. Then, the main loop is started. First, the task priorities are calculated according to the data buffer states and scheduling policy. The hardware is configured accordingly and the system then enters its low-power Idle state, where only its always-on functions are active. The hardware is monitoring the V_{buff} and triggers the core when a energy-driven task is to be executed.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). When the transducer produces energy, an ultra low-power boost charger for harvesting application (BQ25505, TI) is used to charge a 330 μF 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 [22], [5], [23], [24] 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_{load} to drop below a threshold. The switch has a low on-resistance of 0.6 Ω 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_{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

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

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_{buff} with a configurable sampling rate, filters those samples and estimates the input power $P(t)$ from their rate of change.

The control logic provides Q 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 $P(t)$ 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.

Note that the abstract task slots of the control logic allow the execution of either one or more tasks per burst. As an example, the Runtime can merge all tasks in a single abstract task and configure the control logic accordingly to implement conventional *single burst* scheduling.

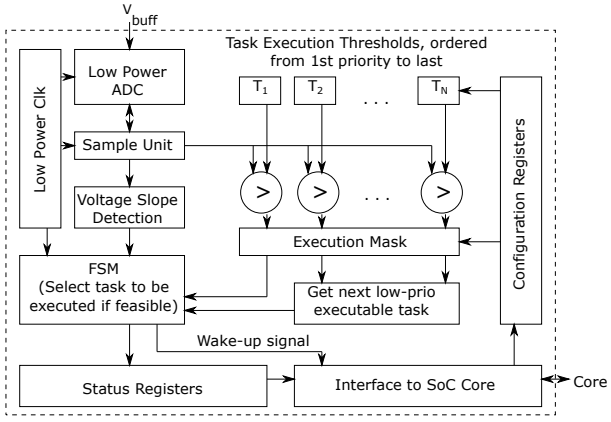

Fig. 5. Simplified block diagram of the HW logic to be integrated.

C. Generic Scheduling Policy Implementation

As just explained, Torpor's control logic allows implementing different scheduling policies by configuring the Runtime accordingly. We explain now in detail how an exemplary *power-aware dynamic* scheduling policy can be implemented with Torpor. This policy will be used in our evaluations later.

To implement a scheduling policy we need to define a *scheduler* as well as a *priority mapping function*. As elaborated in the section III, we focus on applications that are modeled as a task-chain where data is exchanged through FIFO buffers.

With our exemplary power-aware policy we follow a simple strategy to maximize the throughput of the application: 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 mapping favors throughput as it prefers to keeps FIFO fill states low. The rest of the tasks are prioritized according to their energy requirements, with the mapping favoring the more demanding ones.

For the scheduler we select a strategy that follows different policies depending on the available input power $P_h(t)$: If the input power is above a configurable threshold $P_h(t) > P_{h,\text{high}}$, the scheduler aims to execute the task with the highest priority. It waits for V_{buff} to reach $V_{\text{thres,i}}$, the voltage threshold of the task with the currently highest-priority task. 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 mapped to the remaining priority slots by descending order based on their energy requirements. 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 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.

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. To alleviate this effort, Torpor provides an automatic task characterization feature to determine the required values[2].

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_{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. 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

[2]While we generally assume and observe that WSN tasks exhibit little variation, worst-case execution could be artificially enabled during characterization to ensure that worst-case energy consumption figures are obtained, which are required for reliable operation.

to determine whether sufficient energy is available to execute a task. As an additional constraint, during task execution, V_{buff} may not fall below the set output voltage V_{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_{min} is guaranteed to be present on V_{buff} after task execution[3]. By adding some additional margin to V_{min}, overheads and safety margins are taken into account.

Given these constraints, we set the voltage threshold to

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

Finally, the voltage thresholds are stored within Torpor's always-on domain to access them with low energy overhead.

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 power 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. 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. 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 µm^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 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.

[3]We make the reasonable assumption that the energy consumed by the task is independent of V_{buff} by neglecting the voltage-dependent buck-converter efficiency

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

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.

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 input power to our system. The voltage and current at various points of interest were measured using a Rocketlogger [25], 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_{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 $\bar{P}_{\text{load}}$

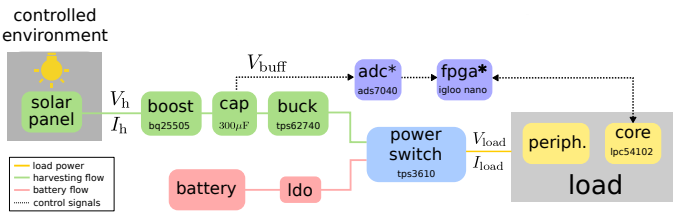

Fig. 6. For repeatable experiments of Torpor's harvesting-driven mode, the solar panel was placed in a controlled environment. The harvesting and load power were recorded for analysis. The ADC and FPGA were externally powered.

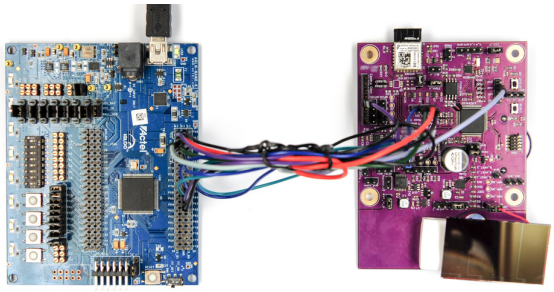

Fig. 7. Photo of FPGA-based Torpor prototype. The blue pcb (left) is a commercial fgpa evaluation kit, the purple-pcb (right) is custom-made and includes all other discrete components used for the evaluation.

C. 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, 7 mW for *medium power* tasks and 90 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 a dynamic scheduler under variable harvesting power. The evaluation took place for time long enough for the system to reach steady state.

1) Constant Harvesting Power (P_h)- Single-burst vs Split execution: 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 II and the voltage of the buffering capacitor for the case of $P_h = 1.31$ mW is depicted in fig. 8.

TABLE II
EVALUATION OF STATIC SCHEDULERS WITH CONSTANT P_h.

	$P_h = 1.3$ [mW]			$P_h = 3.2$ [mW]		
	scheduler		**ratio**	**scheduler**		**ratio**
	single	split		single	split	
exec/min	2.6	6.9	×2.6	31.2	39.1	×1.3
E_{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

2) Variable Harvesting Power (P_h) - Static vs Dynamic execution: 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 periodic (every 57 s) peaks of high available input power (every 3 s). 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 III. 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 dynamic scheduling strategy, as explained in section V-C evaluated and presented in table III. The voltage across the buffering capacitor in Case II is presented in fig. 9.

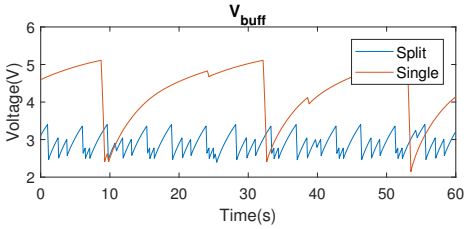

Fig. 8. The voltage on the buffering element when using *single* and *split* schedulers under constant harvesting power ($P_h = 1.3$ mW). The *single* scheduler results in $\bar{P}_{\text{load}} = 0.71$ mW while *split* in $\bar{P}_{\text{load}} = 0.87$ mW.

TABLE III
EVALUATION OF DIFFERENT SCHEDULERS WITH HIGHLY VARIABLE P_h.

	Case I			**Case II**		
	scheduler[a]		**ratio**[b]	**scheduler**[a]		**ratio**[b]
	split	dynamic		split	dynamic	
$\bar{P}_h$ [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_{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.

D. Discussion

The dynamic scheduler improvement in both execution rate and energy efficiency stems mainly from two different effects. The first one is the reduction of the load power consumption. Since the load behaves like a current sink, a lower V_{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. Prolonged stays in high V_{buff} levels are mitigated by static schedulers only when a low-energy task gets its turn for execution. By contrast, dynamic schedulers prioritize low-energy tasks whenever the input power is low. The second effect is the avoidance of E_{buff} saturation when using 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_{buff} when running the low-power task. Dynamic schedulers can mitigate this by matching time periods of low input power to less demanding tasks and time periods of high input power to the most demanding tasks. Since the dynamic scheduler in Case II can avoid saturation of E_{buff}, it manages to harvest more input energy. This is why the improvement ratio of its E_{eff} differs from the improvement ratio of its P_{load}.

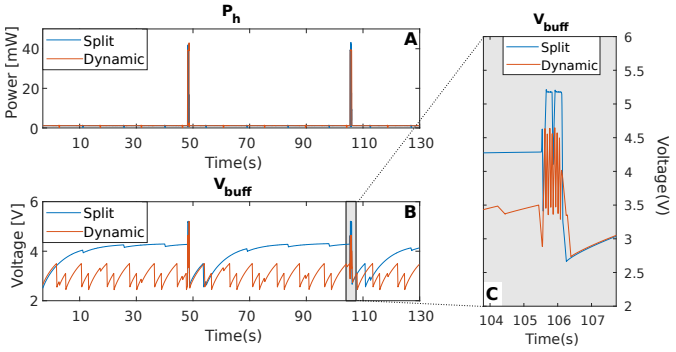

Fig. 9. Traces from Case II experiments of *split* and *dynamic* schedulers. A) Harvesting power B) V_{buff} and C) Zoomed in V_{buff} to show saturation. The dynamic schedulers can reduce the energy losses and avert the saturation effect on short moments of high input power.

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. 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. It can be further doubled by using power-aware, dynamic scheduling when there is enough diversity at the input power levels and available tasks. The energy efficiency is also improved, but its improvement follows a different trend, depending on the amount of energy that the node is using for its always-on tasks as opposed to its energy-driven tasks. 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.

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 power-aware dynamic schedulers can improve the energy efficiency and execution rate of energy-driven tasks by 1.2× and 2.6×, respectively. 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 and support for non-atomic (i.e. interruptible) tasks.

Acknowledgements This research was funded by Swiss National Science Foundation grant 157048: Transient Computing Systems. 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] M. Gorlatova *et al.*, "Networking low-power energy harvesting devices: Measurements and algorithms," *IEEE Transactions on Mobile Computing*, vol. 12, no. 9, pp. 1853–1865, 2013.

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

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

[24] A. Colin *et al.*, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proc. ASPLOS Conf.* ACM, 2018.

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