Peripheral State Persistence For Transiently-Powered Systems

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac
Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France.
e-mail: firstname.lastname@inria.fr

***Abstract*—Recently has emerged the concept of *transiently-powered systems*: tiny battery-less embedded systems which harvest energy from their environment. To retain information despite frequent and unpredictable power failures, a transiently-powered system can use non-volatile memory to store checkpoints of program state. However current checkpointing techniques only consider the state of computation (processor, memory) and disregard peripheral state completely. This paper presents a software framework that allows for the use of non-trivial peripherals such as analog-to-digital converters, serial interfaces or radio devices, in transiently-powered systems.**

I. Introduction

With technology improving, ubiquitous computing gradually becomes a reality as more and more "things" turn into "smart things". The smallest of these systems, e.g. RFID tags, e-health devices, smart cards, provide useful services in various domains. These architectures typically include a slow processor (in the MHz range), little memory space (a few kilobytes), and peripheral devices. But supplying power to these systems is a problem because their small form factor and/or low price forbid the use of a battery.

Energy harvesting is a promising way of solving this issue. Many technologies exist or are emerging to harvest power from the environment, be it from RF radiation, temperature gradient, mechanical energy, etc. This led to the appearance of so-called *transiently-powered systems* (TPS), a class of battery-less embedded systems which offer interesting challenges in terms of software programming. Because energy harvesting yields very low power levels, a TPS typically buffers energy into a capacitor and uses it in short bursts of activity (dozens of milliseconds). For the software programmer, this means that a power shortage may happen at any time and that program logic should resist to frequent, unwanted reboots.

The recent appearance of non-volatile RAM (NVRAM) could provide a solution to keep program state across power outages. Some recent works have proposed to save the state of the system before each power shortage, a technique referred to as *checkpointing*. But these works focus on saving the state of the *computational part* of the system (CPU, execution stack, global variables), they do not handle cases where the application uses peripherals (e.g. LEDs, ADCs, or a RF transceiver) which also have some internal state to preserve.

The main contribution of this paper is a new checkpointing technique for transiently-powered systems with non-volatile memory that makes it possible to use of peripherals devices even in presence of power failures. The paper is organized as follows: Section II presents the context, the problem in more detail and the related works, Section III presents our approach. The implementation is presented in Section IV and the results in Section V. Finally, conclusions are presented in Section VI.

II. Problem statement and related works

This Section presents the problem addressed in this paper: how to enable TPS platforms equipped with NVRAM to use peripherals.

A. Transiently-Powered Systems

Most embedded systems, from smartphones to tiny wireless sensor nodes, rely on battery power. The combination of battery capacity and average power draw determines the system operational lifetime, from a few days for a smartphone up to a few years for a wireless sensor network. However there are also some situations where using a battery is impossible [1]. In particular if the system, e.g. smart card, is to be manufactured in large quantities, then including a battery would dramatically impact the unit cost. In such cases, the platform must harvest energy from its environment and/or from external sources, *e.g.* solar power, piezoelectricity, thermal gradients, or electromagnetic fields [2].

The last decade has seen a growing interest in making such battery-less systems programmable with software. Industrial examples are RFID and smart cards, but there are many innovative platforms. For instance, Intel's Wireless Identification and Sensing Platform [3] bridges the gap between RFID systems and traditional sensor networks. More recently, researchers have presented arguments in favor of more and more miniaturization, and tackled the problem of miniaturizing the whole platform: the M^3 [4] is a $1.0mm^3$, general purpose sensor node able to harvest energy from different sources.

One common characteristic of these systems is that they must cope with an unreliable power supply. Even when the energy source is active, the harvested power level is typically low [2] compared to what the system consumes in active mode. Storing energy in an energy buffer, e.g. a capacitor, is thus necessary just to allow for any useful work to be done. For instance, contactless credit cards must perform the *whole* transaction within a few hundreds of milliseconds, i.e. within one *life-cycle* of the device. If the transaction to be processed is longer, then it is simply unfeasible. Software development for such platforms is thus often done in an ad-hoc fashion, resulting in high engineering costs. Traditional

thread-based programming models and operating systems like FreeRTOS [5], Contiki [6] or RIOT [7] are unsuitable for TPS management because they do not support unexpected power failures. Just booting such an operating system takes longer than the typical duration of a TPS life-cycle.

Hence it is important to provide a new paradigm separating the application layer from low level operating systems issues, so as to facilitate implementing non-trivial applications on transiently-powered systems.

B. NVRAM for Transiently-Powered Systems

Recently, there have been significant advances in the field of non-volatile memory. Several technologies are emerging which promise to blur and eventually remove the distinction between slow/non-volatile "storage" and fast/volatile "memory" [8].

In a transiently-powered system, merely replacing RAM with NVRAM has undesirable side-effects. Because power failures are frequent, they can occur while a (non-volatile) data structure is being modified. When the platform reboots, the program will restart with inconsistent data, causing the so-called *broken time machine* problem [9]. A possible solution would be to have the processor itself non-volatile [10]. For instance, Bartling et al. [11] designed such a non-volatile micro-controller. This kind of approach is interesting in terms of architecture but has a major limitation in terms of software programming.

Indeed storing a program data structure in NVRAM does make it persistent, but also means that each access might be slow, energy-expensive or suffer from NVRAM performance issues depending on the memory technology used. In contrast, storing data in RAM gives good execution performance, but brings back the problem of volatility. For this reason, most non-volatile architectures actually use a combination of both RAM and NVRAM [2], [11], and defer the problem of volatility to the software developer.

C. Program State Checkpointing

The classical solution to this problem is known as *checkpointing* [12], [13]. The power shortage is detected in advance and before the failure occurs, all necessary volatile data is saved to NVRAM. When the platform reboots, data is restored to RAM and the program resumes execution from where it was interrupted.

Assuming that the system is powered from a capacitor, anticipating power failures means detecting when a certain voltage threshold is crossed. To that end, some systems [12], [14] interrupt execution on a regular basis to measure capacitor voltage through an ADC. A less flexible but more efficient approach [13], [15] is to employ a hardware voltage comparator to trigger an interrupt on the desired voltage level.

When the energy level becomes too low, program execution is halted and the system saves program state to NVRAM. To protect against the broken time machine problem in the checkpointing logic itself, most systems adopt a double buffering approach. Instead of always overwriting a single *checkpoint image*, using two distinct images makes the system more robust to crashes. If a checkpointing operation fails (for instance because it was triggered too late) then only one image gets corrupted, and the the other one can be used to recover a previous valid state without losing all progress. Only after a checkpoint is successfully saved, pointers to the two images are swapped in an atomic fashion. This guarantees that there is at least one valid checkpoint image in the system at all times.

The first paper to study checkpointing in the context of transiently-powered systems is Mementos by Ransford et al [12]. But Mementos targets Flash memory which is very power hungry, resulting in poor energy efficiency overall. In the following years, more sophisticated checkpointing mechanisms were proposed [13], [15], [14], [16] specifically for NVRAM.

However all these contributions focus on computational tasks and disregard interactions with hardware peripherals. Even when they are not purely computational, the programs use only very simple (stateless and/or passive) devices. For example in Mementos [12], communication is done passively using backscatter on the RFID signal which powers the device. In a more recent paper by Cassens et al. [17], peripherals are considered but restored in a default state after wake-up or reboot of the platform.

D. Problem Statement

The aforementioned checkpointing-oriented works focus on computations and typically save CPU registers, execution stack, global variables but ignore peripheral states. Non-trivial peripherals such as ADC or radio transceivers have an internal state machine controlling their execution which is usually accessible through the *device driver*. If their state is not protected, the broken time machine problem might occur in the driver code execution.

We explain in this paper how to make hardware peripherals *persistent* (i.e. solve *peripheral state volatility* problem) and *consistent* across reboots so that the application does not notice power failures (i.e. ensure *peripheral access atomicity*).

III. Checkpointing for peripherals

Even when there is no strict separation betweem the two, embedded bare-metal programs typically consist of two main parts: application logic and device drivers. Each driver is a collection of low-level functions providing easier access to a hardware peripheral. On the other hand, the application is a higher-level program which makes use of this *driver API* to perform a specific operation.

Our approach is to interpose an additional layer between the application code and the driver code. This so-called *kernel code* encapsulates every function of the driver API into a wrapper. This change should be as transparent as possible to the application developer: each wrapper has the same signature and provides the same service as the original function. However we make the assumption that the application always uses device drivers and never communicates with hardware directly.

The kernel code will be in charge of saving (and restoring) the state of the drivers. This information is stored in NVRAM in a data structure called *device context* which ensures peripheral state non-volatility. But the kernel will commit to NVRAM the changes of the device context *only* when the peripheral has reached a stable status, this will solve the peripheral access atomicity problem. In the following, we explain these two ideas in more details.

A. Addressing the Peripheral State Non-volatility Problem

Restoring the state of a hardware device typically requires non-trivial operations like configuring some I/O pins, respecting certain timing constraints etc. Rather than a completely automated mechanism, our approach relies on the driver code itself to perform this task. The idea is to ask each driver developer to write an additional restoration function in their driver. This function is responsible for initializing the hardware and then bringing it back to the required state as described by the *device context* data structure. Adding this feature in each driver does require some additional work, but thanks to all other existing functions within the driver, the actual effort required is reasonable.

Also, we require the driver developer to change each driver function so as to explicitly update the device context. The idea is to have a driver-specific data structure in non-volatile memory which mirrors the state of the peripheral closely enough so that this state can be restored.

On the kernel side, the checkpointing mechanism saves these device contexts automatically. At next boot, the kernel loads each device context to RAM and then calls the corresponding restoration function. This ensures that each peripheral is brought back in the correct state.

B. Addressing the Peripheral Access Atomicity Problem

In a TPS application, execution can be interrupted at any point by the checkpointing procedure, at next boot it will be restored transparently and will resume execution exactly where it was left. However for the code handling peripherals it is different: resuming execution inside a driver function would result in incoherent state of the peripheral. Driver calls must be somehow protected and this is the motivation for our kernel layer. Each driver function is only exposed to application through a kernel wrapper. To distinguish from the original driver function call, we refer to the wrapper as a *system call* (or syscall), by analogy with the homonym concept in classical operating systems.

The contract between the application and the kernel is that a syscall will be executed entirely within one life-cycle. If a power failure occurs in the middle of a driver function, then at next boot the function will be re-executed from the beginning (instead of just resuming from where it was interrupted). This contract implies that checkpointing driver state and application state are handled differently.

In order to implement the separation between application checkpointing and driver code checkpointing, we need to isolate local variables as well as the control flow of the driver code. To this end we use a separate execution stack for executing driver calls. This so-called *kernel stack* is never included in the checkpoint image, which guarantees that any partial progress inside a driver function is volatile and will be lost upon power failure. Driver functions need not be modified to benefit from this feature, which is implemented transparently in the kernel layer. The syscall wrapper is responsible for switching stacks and for forwarding the function parameters to the driver function. Also, it saves a copy of these arguments in the checkpoint image together with the syscall address.

If a syscall is interrupted by a power failure, then the checkpointing operation will be triggered. It will save to non-volatile memory the application state and all information required to retry the call. At next boot, the kernel reloads application state to RAM, restores device states from the device contexts, and checks whether a syscall has been interrupted. In that case, the syscall arguments are repopulated from the checkpoint image and the syscall is invoked afresh. If no syscall has been interrupted, then execution resumes directly to application code as usual.

When a syscall returns successfully, the kernel wrapper clears the saved syscall address and arguments from the checkpoint image, modifies (i.e. commits) the device context to NVRAM, and switches back execution to the main stack.

There are some technical issues that must be handled carefully (for instance when a driver calls another driver), the interested reader will find the complete technical description in our research report [18].

IV. The Sytare implementation

We now describe our implementation software, denoted *Sytare* [18]. We start by describing an example of TPS application using Sytare, then we describe the hardware platform on which the mechanism described in Section III has been implemented.

A. Example of execution with Sytare

The simple scenario described by Fig. 1 highlights the checkpointing mechanism implemented in Sytare, allowing consistency between application and peripheral state. Initially, the application state is *App_0* and peripheral *A* has state *A_0*. The user application requests access to peripheral *A*, which has to be done through Sytare API (i.e. *kernel* column).

The *Last Checkpoint Image* on the right represents the state that we restored at last boot. This state will also be restored if a power loss occurs and we do not have enough time to checkpoint our current state. The *Next Checkpoint Image* represents the checkpoint image currently being built during execution. In these checkpoint images, the *App* column indicates the state of the application to restore, and the *Driver* column indicates the state of driver *A* to be restored. The *current driver call* column indicates the saving of the driver call in execution as explained at the end of Section III-B.

The sequence diagram on the left describes a call to the driver function `drvA_fn()` through the kernel wrapper `syt_drvA_fn()`. After the successful return of syscall, a

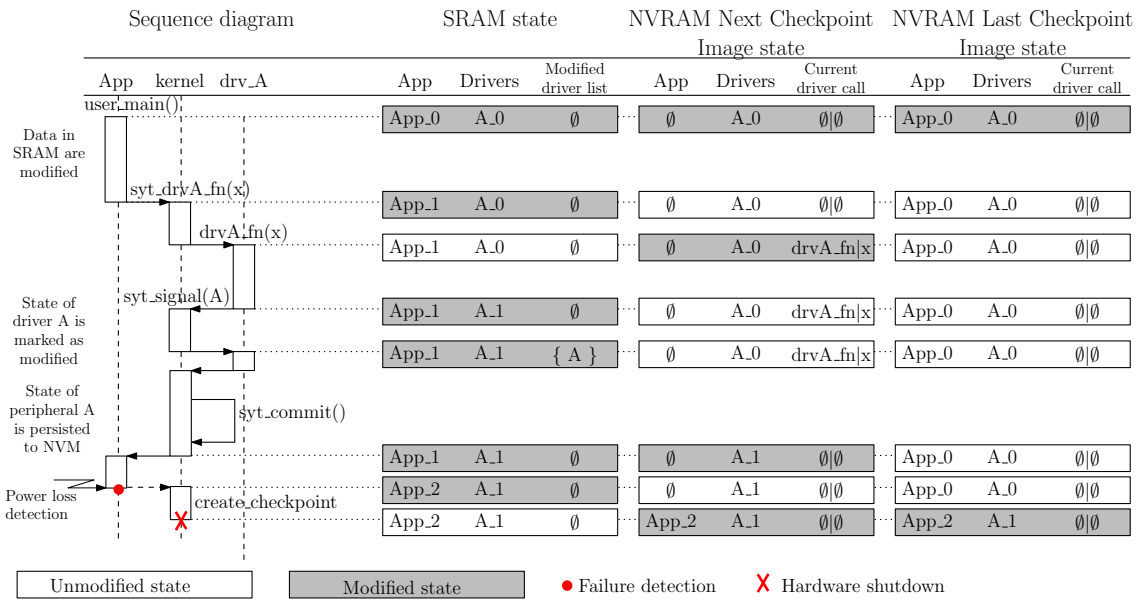

Fig. 1. Sequence diagram of a simple syscall with SRAM and NVRAM kernel data structures content. A power loss is detected while running user application after the syscall `syt_drvA_fn()` returned.

power loss is detected and a checkpointing occurs before hardware shutdown. Just before shutdown, the kernel makes the "Last" image pointer point to the "Next" image and the state of the program that is saved in NVRAM is: application in state *App_2*, driver *A* in state *A_1*, no driver call interrupted.

The `syt_signal()` primitive is used to notify the kernel that peripheral "A" had its state changed. The `syt_commit()` primitive persists the new state of the peripheral into the "Next" image and removes driver *A* from the modified driver list. This mechanism is useful when multiple drivers are implemented on top of others. The kernel uses this information to avoid committing *all* device contexts after each syscall.

B. Hardware Platform

Our prototype is implemented on the Texas Instruments MSP-EXP430FR5739 FRAM Experimenter's board[1]. The MSP430FR5739 micro-controller includes 16kB of embedded NVRAM (ferroelectric memory) and 1kB of classical SRAM. To the best of our knowledge, no other NVRAM-based micro-controller is commercially available today, except for other sibling chips from Texas Instruments.

On this architecture, RAM and NVRAM offers very different performance. FRAM supports a maximum operating frequency of 8MHz, while the rest of the platform (CPU, RAM, and most peripherals) can run up to 24MHz. In our experiments we set the clock frequency to 24MHz.

As for peripherals, we use several on-chip peripherals (clock generation system, analog-to-digital converter, SPI bus controller, etc) as well as an external CC2500 radio transceiver[2]. The RF transceiver is accessed via SPI and thus illustrates a complex scenario with nested drivers: the radio driver is implemented in terms of SPI primitives.

[1]http://www.ti.com/lit/ug/slau343b/slau343b.pdf
[2]http://www.ti.com/lit/ds/swrs040c/swrs040c.pdf

C. Power Supply and Power Shortage Detection

Power supply is implemented with a function generator for reproducibility. We use a square signal generator directly connected to the *Vcc* and *Ground* pins of the target board and configure the signal with various frequencies and duty cycles, i.e. various ratios between the duration of "On power" and "Off power" durations. "Off power" mode depicts the state of the platform when the supply voltage is below the required value to power the micro-controller and the peripherals. We have not yet investigated a real power outage scenario, this is definitely one of the future works to identify the differences between the simulated power supply and a real power outage scenario. +++

The FR5739 micro-controller offers a voltage comparison unit (Comparator_D module) which we use to implement power failure detection. We added a voltage divider montage composed of two resistors of 1MΩ each connected to the Comparator_D module input pin in order to monitor the input voltage variation indicating the power shortage. Such a voltage divider is quite common is this type of measurement (see Fig. 3 in [15]), it consumes approximately 1.6μA and allows us to trigger checkpointing with a theoretic voltage monitoring precision of 53mV. Referring to the device data-sheet the minimum execution voltage of the platform is 2.0V, so we set the checkpoint trigger threshold to 2.063V. When this threshold is reached the module raises an interrupt flag and the checkpointing routine is launched.

V. Performance Evaluation

The contribution of Sytare is to enable the use of formerly written applications to run in a transient power environment.

Evaluating the *performance* of Sytare consists in evaluating the overhead implied by the use of Sytare. The execution time overhead highly depends on the application, and scenario of power shortages.

We did not address performance comparison with other checkpointing mechanisms such as [13], [15], [14], [16], [17] because none of these works provide a safe mechanism for using peripherals with non-trivial internal state in the context of possible power outage.

A. Benchmark Applications

We use 4 benchmark applications that have various levels of interaction with peripheral devices:

- **RSA** This purely computational application encrypts a 128 bits data buffer with the RSA algorithm. It does not use any peripherals but has a significant memory footprint, it allows us to study the impact of Sytare on memory footprint.
- **Diode counter** The program mostly executes `nop` instructions. It slowly counts from 0 to `max_integer` and displays the value of the counter on the platform's diodes. This application allows us to study the impact of adding persistence to very simple access peripherals (i.e. I/O port).
- **Sense and aggregate** This application senses the temperature 10 times using the ADC and stores the values in an array. A delay of 5 milliseconds is observed between each measurement using a software delay.
- **WSN** This benchmark is meant to illustrate a typical Wireless Sensor Network (WSN) application. It senses the temperature and sends the value through the CC2500 RF chip. This application demonstrates the use of Sytare with complex peripherals and nested (Radio chip accessed through SPI) peripheral calls.

B. Performance Metrics

For a given application program, we define T_{wired} as the time it takes to run the complete application under continuous power, without Sytare. The "starting point" is the instant when power is turned on. The duration we measure includes hardware boot time as well as program initialization. The "finish point" is defined as the instant when the program reaches some arbitrary position in the code. Each application has been setup in order to last a few hundred milliseconds.

For each application, we compare the "wired" version against the Sytare version. The latter is run under intermittent power . If we denote T_{on} the duration of the powered period for a given lifetime, we run each experiment with a given T_{on} value and configure the power supply to repeatedly turn on for this duration at each life-cycle. We define a life-cycle as a contiguous period that starts when the supplied voltage is sufficient to power the micro-controller and ends when the supplied voltage decreases below the minimum condition for the micro-controller to operate. During each of these life-cycles, the platform boots, then the Sytare kernel restores execution and peripheral states and then the application runs until power runs out.

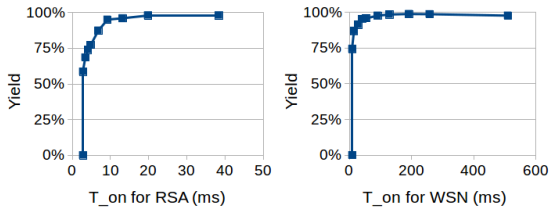

Fig. 2. Yield $Y(T_{on})$ measured for RSA and WSN applications

For a given T_{on}, we define $T_{transient}(T_{on})$ as the time it needs for the system to reach the "finish point" defined above. When measuring this duration we exclude all "Off time" periods as they do not contribute any information to the experiment, but we do include the boot time (hardware and software) and the cost of the checkpointing operations.

C. Sytare Impact on Application Performance

To assess the impact of Sytare on performance, we are interested in the execution time overhead induced by Sytare. For a certain value of T_{on}, we define the *effective yield* Y as the following ratio:

$$Y(T_{on}) = \frac{T_{wired}}{T_{transient}(T_{on})} \quad (1)$$

For very small values of T_{on}, the platform will never have a chance to boot successfully and so it will never finish executing the application. In other words $T_{transient}$ would be "infinite" and the effective yield will be zero. We denote T_{on}^{min} which is the minimal T_{on} that allows application completion (for smaller T_{on}, the program does not have enough time to perform a complete checkpointing).

On the other hand, when the T_{on} duration approaches T_{wired} then the application will be able to run to completion in just one life-cycle with little kernel interaction. But the effective yield never reaches 100% because of the overhead arising from the syscall wrappers. We denote by Y^{max} the yield obtained for very large T_{on}. It basically indicates the overhead of syscalls for each application.

Figures 2 and 3 illustrate the the performance of the Sytare prototype on our benchmarks. These results show that the impact of Sytare on the overall execution time is small.

	RSA	LEDs	Sense	WSN
T_{on}^{min}	2.79ms	2.79ms	2.90ms	9.40ms
Y^{max}	0.98	0.99	0.97	0.99

Fig. 3. Performance of Sytare implementation

D. Evaluation of Syscall Overhead

In order to compare Sytare to other TPS run-time systems, we measured the *additional* impact incurred by Sytare syscalls. These results are presented on Fig. 4 for a variety of device driver calls.

For simple driver functions the additional cost is significant, sometimes as much as +137% for radio-sleep. On the other hand, the radio-wake-up operation itself already takes a long time, so the overhead is much more reasonable. These results show that the relative overhead depends mostly on the complexity of the accessed peripheral itself. But the absolute overhead is about $30\mu s$ per syscall, which is reasonable.

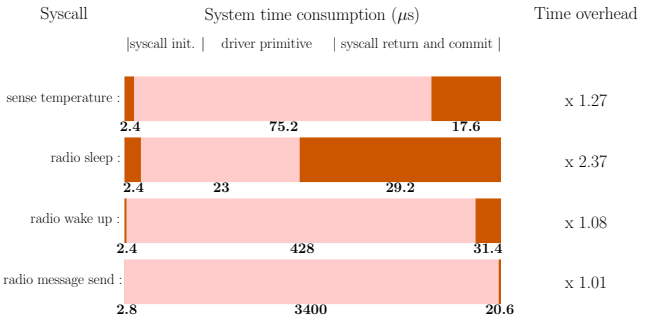

Fig. 4. Kernel temporal impact on drivers primitives calls

E. Memory Overhead

The RAM overhead of Sytare is mostly imputable to the need of a separate kernel stack. The RAM footprint of drivers is increased approximately by the size of the mirrored configuration. For example the most complex application (WSN) used 44 additional bytes in RAM compared to the "wired" version. All kernel variables and checkpoint images are located in NVRAM so they do not occupy precious RAM space.

VI. Conclusion and Perspectives

This paper presents Sytare, a software framework for transiently-powered systems which allows the programmer to use peripherals in their applications. As Sytare provides a safe access to peripherals, programmers are no longer limited to execute the program within one life-cycle. In addition to the application to smart cards and RFID devices, this work might open a great amount of new opportunities for IoT devices, where harvesting will be the main power supply.

Numerous perspectives are open. First, of course, we need to support more peripherals and provide a methodology for platform designer to integrate their device in Sytare. We are also working on the integration of Sytare in a embedded OS such as Contiki or RIOT.

Acknowledgment

This work is partially sponsored by Région Rhône Alpes ADR 16-005688-01, by INRIA ADT program and by Insa/Spie IoT Chair.

References

[1] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, "Powering the internet of things," in *ISPLED'14: International Symposium on Low Power Electronics and Design*, 2014.

[2] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *HPCA'15: High Performance Computer Architecture*. IEEE, 2015, pp. 526–537.

[3] M. Buettner, R. Prasad, A. Sample, D. Yeager, B. Greenstein, J. R. Smith, and D. Wetherall, "RFID sensor networks with the intel WISP," in *Sensys 2008: 6th ACM Conference on Embedded Network Sensor Systems*. ACM, 2008, pp. 393–394.

[4] Y. Lee, S. Bang, I. Lee, Y. Kim, G. Kim, M. H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw, "A modular 1 mm^3 die-stacked sensing platform with low power I2C inter-die communication and multi-modal energy harvesting," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, 2013.

[5] F. Guan, L. Peng, L. Perneel, and M. Timmerman, "Open source freertos as a case study in real-time operating system evolution," *Journal of Systems and Software*, vol. 118, 2016.

[6] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki: a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*. IEEE, 2004.

[7] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, Apr. 2013.

[8] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *HotOS 2011: 13th USENIX conference on Hot topics in Operating Systems*, 2011.

[9] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *MSPC 2014: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2014.

[10] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M. Chang, S. John, Y. Xie, J. Shu, and H. Yang, "Ambient energy harvesting nonvolatile processors: from circuit to system," in *DAC 2015: 52nd Annual Design Automation Conference*, 2015, pp. 150:1–150:6.

[11] S. Bartling, S. Khanna, M. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams, "An 8MHz 75ua/MHz zero-leakage non-volatile logic-based cortex-M0 MCU SoC exhibiting 100-percent digital state retention at VDD=0V with <400ns wakeup and sleep transitions," in *ISSCC 2013 : IEEE International Solid-State Circuits Conference*, 2013, pp. 432–433.

[12] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on rfid-scale devices," in *ASPLOS 2011: 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[13] H. Jayakumar, A. Raha, and V. Raghunathan, "QuickRecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *VLSID 2014: 27th International IEEE Conference on VLSI Design and 13th International Conference on Embedded Systems*, 2014, pp. 330–335.

[14] F. Ait Aoudia, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to nvram for transiently-powered systems," in *ReCoSoC 2014: 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2014.

[15] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, p. 15–18, Mar 2015.

[16] N. Bhatti and L. Mottola, "Efficient state retention for transiently-powered embedded sensing," in *EWSN'16: 13th ACM International Conference on Embedded Wireless Systems and Networks*, 2016.

[17] B. Cassens, A. Martens, and R. Kapitza, "The neverending runtime: Using new technologies for ultra-low power applications with an unlimited runtime." in *EWSN*, 2016, pp. 325–330.

[18] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence For Transiently Powered Systems," INRIA, Research Report 9018, Feb. 2017.