Energy Harvesting Meets IoT: Fuelling Adoption of Transient Computing in Embedded Systems

Domenico Balsamo[*] **Michele Magno**[†] **Kacper Kubara**[*] **Bogdan Lazarescu**[*] **Geoff V. Merrett**[*]

[*]Centre for IoT and Pervasive Systems
University of Southampton
Southampton, SO17 1BJ, U.K.
{db2a12, kjk1u17, bl10g15, gvm}@ecs.soton.ac.uk

[†]Integrated Systems Laboratory
ETH Zurich
8092 Zurich, Switzerland
michele.magno@iis.ee.ethz.ch

Abstract—**The emerging class of transient computing systems enables computation to be sustained despite power outages due to the variable nature of energy harvesting. However, existing approaches are largely designed for specific architectures, and hence are not broadly applicable across different IoT devices. Emerging platforms based on portable, hardware-independent software should rely on lightweight operating systems (OSs) designed specifically for embedded IoT applications, such as *Arm mbed OS* and *Contiki OS*. To enable the widespread use of transient computing, transient approaches need to be integrated into these operating systems. In this paper, we discuss the challenges of providing software primitives for transient computing to facilitate hardware-independent implementation using standard OS APIs, and present the integration of a state-of-art transient approach, *Hibernus* into *mbed OS*. This OS is chosen due to the large community of developers and the open-source IoT code availability. Transient computing is offered through a modular and layered structure that uses the available *mbed OS APIs*, including different strategies for retaining the system state designed for different types of flash memory. To illustrate the applicability of the proposed design, we implemented *Hibernus* on two *mbed* platforms with different flash memories, which respectively requires 4.7mF and 4.9mF of additional storage.**

Index Terms—**Energy harvesting, Transient computing, Internet of Things, Arm mbed programming framework.**

I. Introduction

Recent research on embedded IoT devices has looked to replace batteries with energy harvesting (EH) sources, in which the energy is scavenged from the device's environment [1]. However, the power from these sources is typically characterized as being spatially and temporally dynamic, and uncontrollable [2], [3]. This depends on factors including the type of harvester, deployment location, time of day, weather, type of activity, and makes systems susceptible to frequent power interruptions and resets [4], [5].

To smooth the temporal dynamics of supply and consumption, IoT devices typically incorporate an additional energy storage. This approach is know as *energy neutral* operation and attempts to balance the long-term energy consumption against the harvested energy [6]. This is typically effective with systems powered by a controllable EH source, in which its availability and output can be determined at design time and controlled at runtime. For these systems, the amount of energy storage is introduced to accommodate the variability of this power source. However, this storage does not necessarily guarantee further application execution with devices powered by an uncontrollable or scarce EH source (most of the natural and renewable EH sources, *e.g.* wind, RFID or thermal). Here, a system can be subjected to long periods of energy scarcity, where it simply stops working once the energy storage available has depleted. Additionally, large energy storage needs more time to charge, poses environmental issues and causes increased size and cost of devices [7], [8].

Transient computing enables computation to be sustained despite power outages, minimizing the need for additional storage [9]. This is achieved by saving the system state when a supply failure is imminent, and, following a supply interruption, allowing execution to continue from where it left off. Various transient approaches exist in literature to facilitate the system state retention. These include software-based approaches, where the state retention is ensured using software strategies, and architectural approaches, which provide hardware support to maintain and save the state using dedicated non-volatile elements, *e.g.* non-volatile processors (NVPs) [10].

However, existing approaches are designed for architectures which use fast non-volatiles memories (NVMs), such as FRAM or MRAM and, therefore, they are currently not applicable to many IoT platforms with flash memory. On the other hand, new lightweight operating systems (OSs), specifically designed for embedded IoT devices and applications, are emerging, which enable portable and hardware-independent software development on IoT devices [11]. Predominant open-source OSs targeting for IoT devices are *Contiki OS* [12], *TinyOS* [13], *Arm mbed OS* [14] and *FreeRTOS* [15]. They typically provide application programming interfaces (APIs) (beyond bare-metal programming) to facilitate large-scale software development. Additionally, most of the IoT platforms that support these OSs use flash memory.

To enable the widespread use of transient computing, transient approaches need to be integrated within these operating systems. In this paper, we discuss the challenges of providing software primitives for transient computing to facilitate hardware independent implementation using standard OSs APIs, and enabling the system state retention with devices that use flash memory. We then redesigned *Hibernus* [16], such that it can be integrated with *mbed OS* to demonstrate the applicability of transient approaches as part of an IoT OS. Specifically, *mbed OS* provides a large number of features (e.g. libraries and tools) to develop and connect IoT platforms based on Arm Cortex-M microcotrollers (MCUs), including security and drivers for sensors and I/O devices.

To incorporate *Hibernus* into *mbed*, a modular and layered structure was designed, which separates platform independent and platform dependent macro layers. This uses the standard and available *mbed OS APIs*. Additionally, different approaches for saving and restoring the system state are proposed that consider flash memories and related parameters. A practical implementation of *Hibernus* on two different MCUs with flash memory (existing systems have used FRAM) is presented to demonstrate the ability for transient computing to operate on *mbed* platforms. An open-source implementation has been released and can be downloaded[1].

The reminder of the paper is organized as follows. Section II presents IoT OSs properties and requirements for transient computing. In Section III, the redesign of *Hibernus* is presented, while its operation on *mbed* platforms is presented in Section IV. Finally, Section V concludes the paper.

II. IoT OS properties and design challenges for transient

In this Section, the IoT OS properties are discussed with the aim of identifying the design challenges for transient computing. An exploration of transient approaches is then presented to demonstrate why *Hibernus* is the most suitable approach.

A. IoT operating systems

Lightweight OSs foster the widespread use of IoT applications, by providing full support for a large number of platforms, including Arm Cortex-M-based platforms. One of the requirements for an OS specifically designed for IoT systems is to fit within memory constrains. This is typically achieved by providing a set of optimized libraries, which enable common IoT functionality and a standard ways to access different components and peripherals. Additionally, driver support for a wide range of standard MCU peripherals such as analog peripherals, interrupts, serial interfaces, ect, is typically included. This enables portability and ease-of-use across multiple platforms. Finally, projects built with these OSs can be compiled and tested with different compilers and tool-chains, such as Arm Compiler 5, GCC and IAR.

In the following, the main design challenges for transient computing are considered:

- Although IoT OSs make available APIs to facilitate the implementation of additional features, specific support (i.e. drivers) for each platform is still required to enable transient computing due to differences in the underlying hardware platforms;
- Most of the platforms that support IoT OSs use flash memory, which typically holds the application image as well as the first-stage boot-loader. However, existing transient approaches have been designed for fast and low-power NVMs; because of this, the impact of using a relatively slow NVM has to be considered. External NVMs, *e.g.* FRAM communicating via SPI with the MCU, can be potentially used for saving and restoring the system state. However, this is not an integrated solution and requires hardware-assisted support;
- Due to the large number of compilers and IDEs typically supported by IoT OSs, the selected transient approach should satisfy the requirement of being suitable across multiple platforms. This limits the applicability of solutions that require modifications at the compiler level or that rely on hardware-assisted solutions (*e.g.* extra hardware).

B. Transient approaches

In this Section, an exploration of the most common transient approaches is briefly presented to clarify, following the design requirements presented in Section II-A, why *Hibernus* was selected as a suitable approach to be implemented with IoT OSs.

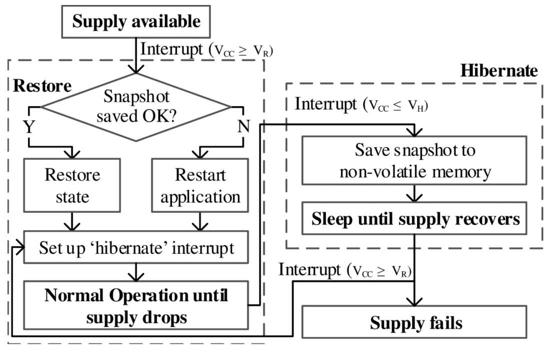
The first software-based approach presented was *Mementos*, which places static trigger points following various strategies. When a trigger point is reached, *Mementos* [17] saves an image of the system state into NVM if the supply voltage is below a static saving threshold voltage. However, it requires compiler-level modifications to enable the placement of these trigger points. A different approach, namely *Quickrecall* [18], relies on a unified memory structure to reduce the time needed for the state retention. However, it can only be applied to fast and low power NVMs, which makes it unsustainable with flash memories.

Hibernus is a platform- and application-agnostic software-based approach that relies on guard bands. A guard band is a voltage threshold that relates to the amount of energy required to hibernate (V_H) the entire system state (snapshot) to NVM (e.g. FRAM) before sleeping. The state is then restored when the voltage rises above a second voltage threshold (V_R). This approach is suitable for IoT OSs because it does not dictate specific hardware constrains or require modifications at the compiler level. Additionally, it can be extended to systems with flash memory but the additional storage to enable the saving operation has to considered. An adaptive version of *Hibernus*, that is *Hibernus*++ [19], enables forward progress in application execution by self-calibrating V_H and V_R but it requires extra hardware.

III. Hibernus redesign

In this Section, we discuss the redesign of *Hibernus* with *mbed*, focusing on the software structure, to demostrate the applicability of transient approaches with IoT OSs (Section III-A). We then present different snapshot strategies for flash memory (Section III-B) and considerations related to the

[1]Available at: https://os.mbed.com/teams/Energy-Driven-Computing-University-of-So/code/Hibernus/



Supply available
Interrupt ($V_{CC} \geq V_R$)
Restore
Snapshot saved OK?
Y
N
Restore state
Restart application
Set up 'hibernate' interrupt
Normal Operation until supply drops
Hibernate
Interrupt ($V_{CC} \leq V_H$)
Save snapshot to non-volatile memory
Sleep until supply recovers
Interrupt ($V_{CC} \geq V_R$)
Supply fails

Fig. 1. Flow-chart illustrating the core algorithm of *Hibernus* [16].

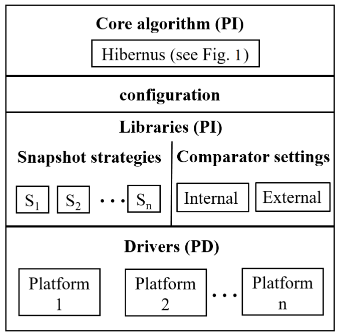


Fig. 2. API for *Hibernus*, based on a layered structure to separate platform independent and platform dependent macro layers.

energy store required to enable the system state retention reliably (Section III-C).

A. Software structure

Fig. 1 shows the operation of *Hibernus*. It moves between two states, active and hibernation, when the supply voltage V_{CC} passes thresholds. It uses hardware interrupts to detect when V_{CC} drops below V_H or rises above V_R. These can be triggered using an internal comparator (if available) or an external comparator.

To integrate *Hibernus* into *mbed*, we designed a modular structure that operates at different layers of abstraction, separating platform independent (PI) and platform dependent (PD) functions. This OS was selected at first because it runs on 32-bit ARM embedded devices, provides a hardware abstraction layer and supports a large number of platforms. However, the structure presented here can be extended to other IoT OSs.

As shown in Fig. 2, the core algorithm can be considered as PI at the level of abstraction presented in Fig. 1. This is because it does not dictate any specific requirement for the type of NVM (and related snapshot strategy) or comparator needed to generate the interrupts. As a result, the algorithm is located in a PI library (`hibernus.h` and `hibernus.cpp`) that only uses standard *mbed APIs* for its methods and does not require any modification by the software designer.

At a lower level of abstraction, a configuration layer defines the type of snapshot strategy and the comparator settings (*i.e.* internal or external). This layer is located into a separate file (`config.h`) that can be accessed by designers to set their parameters (a default configuration is already set for each platform).

The PI libraries are then implemented to enable the selected screenshot strategy and comparator settings. At this level of abstraction, methods are defined that use standard *mbed API* functions as well functions (e.g. `copyRAMtoFlash()`) whose implementation is specific for each platform. A lower device layer is, therefore, added to include the hardware and its software drivers. Specifically, these drivers relate to the platform- an memory-dependent methods to access the flash memory and to access the core registers. This layer is located into a separate PD library (`driver.h` and `driver.cpp`) that only needs to be written once for each platform.

B. Snapshot strategies

Different snapshot strategies are presented specifically designed for flash memories and related parameters.

Flash memories operate by erasing a block of cells (page) before writing. The size of this page can range between a few bytes to a few kilobytes. A given number of pages forms a sector. Erasing a sector is an energy consuming process because it involves generating a voltage pulse using a charge pump and directly impacts on the on the hibernation process. Flash memories are asymmetrical as the reading is faster and more power efficient than the writing. For this reason, the presented strategies aim to optimize the time and energy overhead due to the writing process.

As show in Fig. 3 a), a basic approach was implemented which saves the entire RAM to flash. This approach was originally implemented with *Hibernus* (designed for MCUs with FRAM) and relies on the fact that the size of the NVM is typically many times larger than the RAM. However, with flash memories, the number of sectors dedicated to other purposes other than holding the application image (*e.g.* `.text` segment) can vary. For this reason, different options to reduce the amount of data saved from RAM to flash during hibernation have been considered.

In the basic implementation each snapshot consists of the entire RAM including the unallocated part of the memory, as well as a dedicated segment, containing pointers and flags required for the restore (pointers/flags in Fig. 3 a)). To reduce the number of sectors required in flash, this dedicated segment can be moved into the unallocated part (if available) of the RAM instead (using a static address) before copying the RAM into flash.

A better solution was recently proposed, which consists of saving only the allocated part of the RAM to flash (see Fig. 3 b) [20]. In this case, each snapshot consists of the `.data`, `.bss` and `.heap` segments, as well as the `.stack` and the dedicated section, containing pointers and flags required for the restore. To identify these segments, the saving process needs to track the end of the `.heap` segment and the top of the `.stack`.

Finally, a novel approach is proposed which aims to significantly reduce the number of erase operations, by copying in

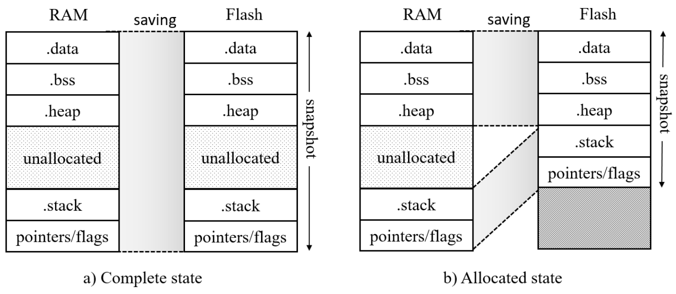


Fig. 3. Snapshot strategies.

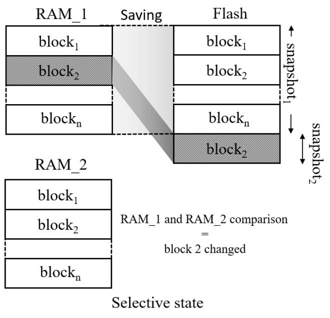


Fig. 4. Snapshot strategy illustrating selective state.

flash only the blocks of the RAM memory that have changed since the last restoring process. As shown in Fig. 4, the RAM memory is divided into two identical segments (RAM-1 and RAM-2). The main application code only acts on the RAM-1 segment, which is divided into different blocks ($block_1$, $block_2$, ... , $block_n$). Similarly, RAM-2 is organized using the same block structure. At each restore, the content from flash is saved to both segments, RAM-1 and RAM-2. In this way, RAM-2 preserves an image, at the restore time, that is not modified during the execution of the main IoT application. At the hibernation time, RAM-1 and RAM-2 are compared and only the blocks that have changed are saved in flash. For example, in Fig. 4 only $block_2$ is saved in flash during the second snapshot.

Other selective policies for efficient state retention with transient systems which exploit the properties of different NVM memories were recently proposed [21]. Due to the modular structure of the presented software, these polices can be added and evaluated as separate modules.

C. Energy Storage

The energy consumed for the state retention process with flash memory, E_σ, depends on the erasing cost and the energy required for saving the RAM and core registers to flash:

$$E_\sigma = n_s E_s + n_\alpha E_\alpha \tag{1}$$

where n_s is the number of sectors to be erased and n_α is the size of the RAM and registers (in bytes). E_s is the energy required to erase one sector (J) and E_α are the energy required to copy each RAM byte to NVM (J/byte).

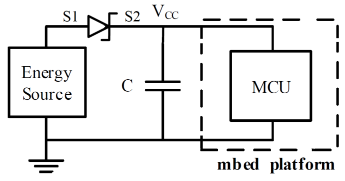


Fig. 5. Test platform.

Hibernus requires enough energy to be stored in the capacitance between the supply rails to save a snapshot. Given a defined V_H, the energy E_δ stored between V_H and systems's minumum opration voltage V_{min} is:

$$E_\delta = \frac{V_{\text{H}}^2 - V_{\text{min}}^2}{2} \cdot C \tag{2}$$

To ensure stability, sufficient C must be present or added so that $E_\sigma > E_\delta$, to enable complete hibernation (even with a sudden loss of supply).

IV. Experimental Validation

In this Section, to illustrate the applicability of the proposed design to platforms with different flash memory, a practical implementation of *Hibernus* on two *mbed* platforms is presented.

The first platform, a Freescale KL05Z, is built on the Arm Cortex M0+, with small RAM capabilities (4KB), 32KB of flash, and holds an internal comparator that can be used to trigger the interrupts for saving and restoring. For this board, due to the small RAM capabilities, flags and pointers are saved into a separate sector in flash. The second one, a NXP LPC11U24, is built on the Arm Cortex M0, with higher RAM capabilities (8KB), and requires an external comparator. For this board, due to the high RAM capabilities, the flags are saved in RAM.

Fig. 5 shows the test platform used for this validation, where the MCU is connected to the output of the energy source through a schottky diode to prevent back-flow of charge to the source. Here, *C* is the required energy storage to enable state retention reliably with flash memory. The value of this capacitance *C* has to be calculated for each board, based on the amount of energy required for hibernation operation. For this purpose, we considered the basic snapshot strategy that is complete state.

Fig. 6 and Fig. 7 show the power and time needed to save the system state with the KL05Z and LPC11U24, respectively. The first platform consumes an average power of 66mW at 3V, for 51.8ms, to erase 5 sectors (each sector is 1KB) and an average power of 50.4mW at 3V, for 66.6ms, to copy the entire RAM plus the pointers/flags segment, corresponding to an energy equal to 6.7mJ. The required capacitance *C* is equal to 4.7mF, for a given V_H equal 2.4V and V_{min} equal to 1.7V (see Eq. 2).

The second one consumes an average power of 66.45mW at 3V, for 98.4ms, to erase 2 sectors (each sector is 4KB) and an average power of 66.33mW at 3V for 41ms, for copying the entire RAM, corresponding to an energy equal to 9.2mJ.

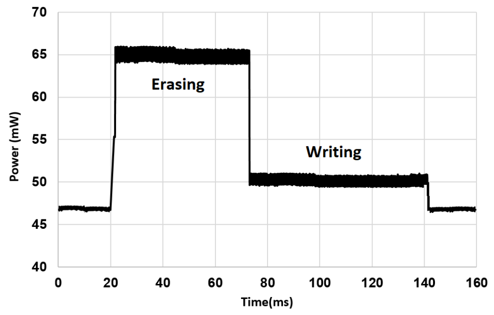


Fig. 6. Power consumption and time required to save the system state including the erasing for the KL05Z.

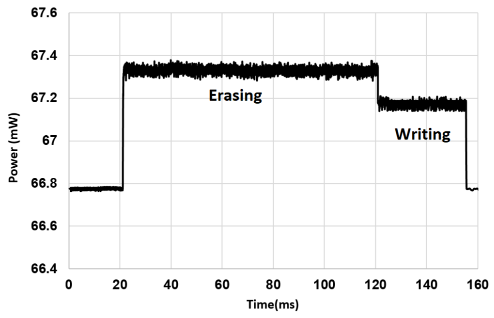


Fig. 7. Power consumption and time required to save the system state including the erasing for the LPC11U24.

The required capacitance C is equal to 4.9mF, for a given V_H equal 3V and V_{min} equal to 2.3V (see Eq. 2).

It is interesting to notice that the time for erasing and writing between the two platforms changes significantly. This depends on the specific flash memory settings (i.e. the ability of erasing multiple consecutive sectors at the same cost), the size and number of sectors to be erased and the size of the RAM memory. Finally, Fig. 8 illustrates the system behaviour with a sinusoidal wave and a simple binary counter as application, demonstrating the suitability of *Hibernus* with *mbed* using the KL05Z. A similar behaviour can be plotted for the LPC11U24 platform. Signals S1 and S2 on this figure refer to the unrectified and rectified supply inputs, respectively. It shows the hibernation, which is divided into two parts; erase and write, main application execution, restore and sleep times.

V. Conclusion

In this paper, we discussed the challenges in integrating transient computing, i.e., *Hibernus* as part of an IoT OS. These relate to the ability of providing software primitives to facilitate hardware-independent implementation of these approaches using standard OSs APIs, and enabling the system state retention with flash memories. This support is offered through a modular structure that uses available OS APIs. We then defined three different strategies for saving the system state designed for flash memory and validated *Hibernus* on two *mbed* platforms.

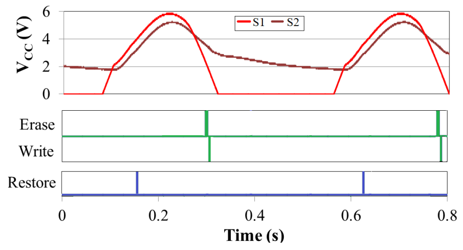


Fig. 8. Experimental results showing basic hibernate/restore operation with the KL05Z.

VI. Acknowledgment

This work was supported by the UK EPSRC Grant EP/P010164/1 and by the Swiss SNF grant 157048. Experimental data can be found at DOI 10.5258/SOTON/D0885.

References

[1] Y. C.-W. *et al.*, *Energy Harvesting in Internet of Things*. Singapore: Springer Singapore, 2018, pp. 35–79.

[2] P. Bogdan *et al.*, "Making the internet-of-things a reality: From smart models, sensing and actuation to energy-efficient architectures," in *Int. Conf. on Hardware/Software Codesign and Syst. Synthesis*, 2016.

[3] H. Jayakumar *et al.*, "Powering the internet of things," in *IEEE/ACM Int. Symp. on Low Power Electron. and Design*, Aug 2014, pp. 375–380.

[4] O. B. Akan *et al.*, "Internet of hybrid energy harvesting things," *IEEE Internet of Things J.*, vol. 5, no. 2, pp. 736–746, April 2018.

[5] M. Magno *et al.*, "Adaptive power control for solar harvesting multimodal wireless smart camera," in *2009 Third ACM/IEEE Int. Conf. on Distributed Smart Cameras*, Aug 2009, pp. 1–7.

[6] A. Kansal *et al.*, "Power management in energy harvesting sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, Sep. 2007.

[7] A. Kerhet *et al.*, "Distributed video surveillance using hardware-friendly sparse large margin classifiers," in *2007 IEEE Conf. on Advanced Video and Signal Based Surveillance*, Sep. 2007, pp. 87–92.

[8] V. Jelii *et al.*, "An energy efficient multimodal wireless video sensor network with ez430rf2500 modules," in *5th Int. Conf. on Pervasive Computing and Applications*, Dec 2010, pp. 161–166.

[9] B. Lucia *et al.*, "Intermittent Computing: Challenges and Opportunities," in *2nd Summit on Advances in Programming Languages*, vol. 71, 2017.

[10] K. Ma *et al.*, "Nonvolatile processor architecture exploration for energy-harvesting applications," *IEEE Micro*, vol. 35, no. 5, pp. 32–40, Sept 2015.

[11] O. Hahm *et al.*, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things J.*, vol. 3, no. 5, pp. 720–734, Oct 2016.

[12] A. Dunkels *et al.*, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *29th IEEE Int. Conf. on Local Computer Networks*, Nov 2004, pp. 455–462.

[13] P. L. *et al.*, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148.

[14] "Arm mbed os," https://www.mbed.com/en/development/mbed-os.

[15] "Freertos," http://www.freertos.org.

[16] D. Balsamo *et al.*, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Syst. Letters*, vol. 7, no. 1, pp. 15–18, March 2015.

[17] B. Ransford *et al.*, "Mementos: System support for long-running computation on rfid-scale devices," in *16th Int. Conf. on Architectural Support for Programming Languages and Operating Syst.*, 2011, pp. 159–170.

[18] H. Jayakumar *et al.*, "Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *27th Int. Conf. on VLSI Design*, Jan 2014, pp. 330–335.

[19] D. Balsamo *et al.*, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Trans. Comput.-Aided Des. Integr. Syst.*, vol. 35, no. 12, pp. 1968–1980, 2016.

[20] N. A. Bhatti and L. Mottola, "Efficient state retention for transiently-powered embedded sensing," in *Int. Conf. on Embedded Wireless Syst. and Networks*, ser. EWSN '16, 2016, pp. 137–148.

[21] T. D. Verykios *et al.*, "Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of nvm technologies," *Sustainable Computing: Informatics and Systems*, 2018.