ARTICLE IN PRESS

Sustainable Computing: Informatics and Systems xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of NVM technologies

Theodoros D. Verykios*, Domenico Balsamo, Geoff V. Merrett

Centre for Internet of Things and Pervasive Systems, School of Electronics and Computer Science, University of Southampton, UK

ARTICLE INFO

Keywords:
Selective state retention
Energy harvesting
Non-Volatile Memory
Transient computing

ABSTRACT

Transiently-powered embedded systems are emerging to enable computation to be sustained during intermittent supply, without the need for large energy buffers such as batteries or supercapacitors. To deal with the intermittent nature of the input source, these systems save the system state (i.e. registers and main memory) to Non-Volatile Memory (NVM) before a power failure, and restore it when the power supply recovers. Existing approaches normally save the entire state of the system upon power failure, but this is both energy and time consuming. In this paper, we analyse existing approaches to identify their inefficiency when used with specific NVM technologies, and propose novel selective policies for efficiently retaining the system state by exploiting properties of different NVM technologies. These policies are based on (1) concatenating multiple images into the available NVM before erasing, and (2) efficiently selecting only the system state that has changed since last saving. The existing and proposed policies are experimentally validated on two embedded platforms featuring different NVM technologies (Flash and FRAM), depending on their characteristics, in order to identify the most energy efficient policy/platform combination. Results show a reduction in energy and time overhead of up to 90.6% for Flash memory using a novel policy, and 86.2% for FRAM, compared to the typical approach of saving the entire system state.

1. Introduction

Batteries have traditionally been used to power embedded systems. However, requirements such as a long lifetime, low cost, and low weight, pose significant challenges to battery-powered systems. In addition, the nature of some applications such as implantable bio-sensors [1,2] and underground WSNs [3] implies limited access and, consequently, maintenance for battery replacement or recharging becomes a challenge. Therefore, the need for embedded systems that can operate without batteries has emerged [4].

Energy harvesting (EH) systems scavenge energy from environmental sources such as light, vibration, motion or temperature to power themselves, instead of relying on batteries [5]. However, factors such as the weather condition, availability of light, or the intensity of vibration can have a significant impact on energy availability. Relying solely on these sources can, therefore, result in the system being unable to sustain computation. The traditional solution to tackle this is the use of energy storage (e.g. a supercapacitor or rechargeable battery) to buffer harvested energy so that the long-term energy consumed equals the harvested energy [6,7]. However, these buffers increase the size, weight and cost of the devices, which makes the realisation of some systems infeasible.

Transiently-powered embedded systems are storage-less systems that enable computation to be sustained, despite the variable and unstable energy harvested from the environment [8]. Due to frequent power interruptions caused by the variable source, transient systems achieve forward progress by retaining their state in Non-Volatile Memory (NVM) upon a power failure. This implies that the main memory, core registers and general-purpose registers are saved before a power outage, and restored when the power is available once again.

Several software-based approaches have recently been proposed for transient computing [9–13]; however, these all save the entire volatile state without considering which parts of the memory need to be saved. Furthermore, they consider the NVM to be somewhat ideal, whereas in practice the characteristics of different NVM technologies can have a significant impact on efficiency. Using a universal policy, without regard for the NVM technology, results in spending a considerable amount of energy for the retention process. Therefore, the active time of the system is significantly reduced, resulting in degraded forward execution progress of the application. Fig. 1 shows the impact of the saving/restoring process on a system with frequent power intermissions. Here, the ratio of active time against the time required to save/restore ($\frac{t_{\text{active}}}{t_{\text{save/restore}}}$) is low while maximising the time spent on useful

* Corresponding author.
E-mail addresses: T.Verykios@soton.ac.uk (T.D. Verykios), D.Balsamo@soton.ac.uk (D. Balsamo), gvm@ecs.soton.ac.uk (G.V. Merrett).

https://doi.org/10.1016/j.suscom.2018.07.003
Received 15 November 2017; Received in revised form 12 March 2018; Accepted 11 July 2018

Please cite this article as: Verykios, T.D., Sustainable Computing: Informatics and Systems (2018), https://doi.org/10.1016/j.suscom.2018.07.003

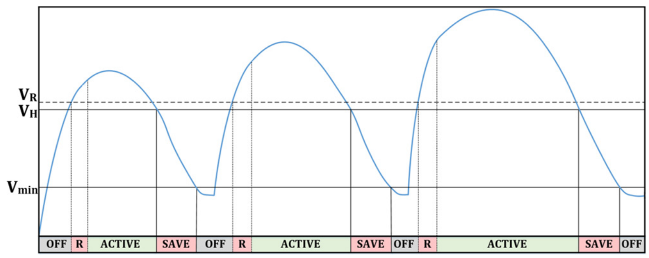


Fig. 1. Typical operation of a transiently-powered system.

computation and, therefore, the forward execution progress, is of vital importance for transiently-powered embedded systems.

In this paper, we propose various novel selective policies for efficient state retention which exploit the characteristics of different NVM technologies and match existing policies with the fundamental properties of each NVM technology, to ensure that state retention is an energy and time efficient operation. Key contributions reported are:

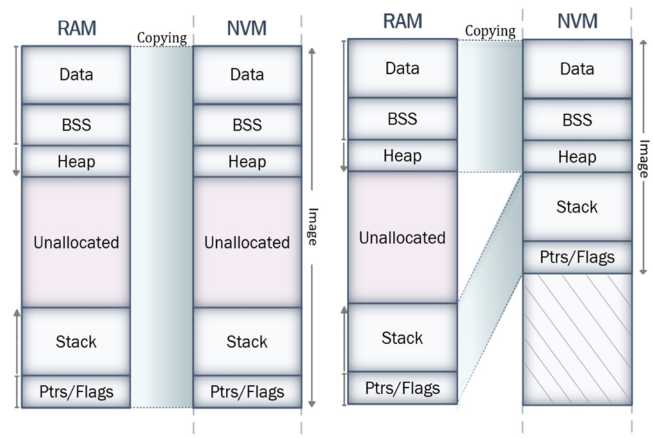
- An exploration and analysis of the inefficiency of existing policies and the effect that the properties of different NVM technologies has on each policy;
- Novel policies for efficiently saving state, which select only memory blocks updated since the last save, and reduce erasing by concatenating multiple images;
- An experimental validation of existing and proposed policies on two platforms from different manufacturers with different NVM technologies (Flash and FRAM) as part of a transiently-powered embedded system, to identify the most energy efficient policy/platform combination.

The remainder of this paper is organized as follows. In Section 2, we discuss the problem and motivate the different policies proposed. Section 3 presents an analysis of current policies for saving and restoring the system state. Novel policies along with their analysis and implementations are then described in Section 4, followed by the experimental design in Section 5. Results are presented in Section 6 and, finally, Section 7 concludes the paper.

2. Related work and motivation

As highlighted in Section 1, various approaches have been proposed to retain system state and enable computation to be sustained upon power failures. An early software-based approach was Mementos [9], which places static trigger-points in strategic locations (e.g. before a function call or inside each loop) at compile time. Mementos saves the core registers, the stack and the global variables (part of .bss and .data segments) in Flash memory, captured through analysis at compile time. Furthermore, Mementos often saves the system state even if the power failure is avoided, which results in wasting time and energy. The policy is not generally applicable as it does not address the saving of the heap segment, which is allocated dynamically at run-time.

Hibernus [11] is an interrupt-driven approach that saves the entire system state (main memory, core and general-purpose registers) in Ferroelectric RAM memory (FRAM) and enters a low-power mode when the supply voltage drops below a specific threshold. Hibernus+ + [12] is an updated version of the same approach, which dynamically adjusts the saving and restoring thresholds, depending on the on-board decoupling capacitance and the available harvested energy. Fig. 2a shows how these approaches can be applied to any system due to their state retention policy which indiscriminately saves and restores the entire RAM memory, including the heap segment and unallocated space. We refer to this policy as *Complete State*, as shown in Fig. 2a. However, these approaches do not consider any intelligent policies to identify the unallocated space, introducing a significant amount of time and energy



(a) *Complete State* [11, 12] (b) Allocated State [14]

Fig. 2. Existing state retention approaches illustrating (a) Complete State and (b) Allocated State.

spent on retaining unnecessary data.

To further reduce the time and energy overhead, QuickRecall [10] proposes a unified memory system, replacing the volatile main memory with FRAM. In this case, only the core and general-purpose registers need to be saved in FRAM. However, FRAM is slower and more power-hungry compared to volatile SRAM, making this approach less attractive for low-power embedded systems [15].

The presented software-based approaches save the entire system state every time, without considering what has changed since the last restore. This leads to a sub-optimal state retention process (Hibernus and Mementos) or inefficient use of NVM (QuickRecall). Recently, Bhatti et al. [14] proposed a selective policy for efficient state retention which dynamically identifies the unallocated space and only saves to Flash memory the parts of the main memory being used by the application. We refer to this policy as Allocated State (see Fig. 2b), while the system state being saved in NVM is referred to as an image.

2.1. Properties of NVM technologies

To address the challenge of efficiently retaining the system state, we consider the relevant parameters of typical and emerging NVM technologies, which have an impact on the saving and restoring process of a transiently-powered embedded system, as summarised in Table 1.

Flash memory operates by erasing a block of memory cells (page) before writing [16]. The size of this page can range between several bytes to a few kilobytes. A given number of pages form a sector, which is a larger block of memory, and can be erased at the same cost of erasing a single page. Erasing is a highly energy consuming process because it involves generating a voltage pulse using a charge pump. This limits its endurance to approximately 10^5 cycles. Moreover, Flash memory is asymmetrical, meaning that reading is faster and more power efficient compared to writing data.

FRAM uses a ferroelectric capacitor as a storage device, offering the benefit of higher power efficiency when compared to Flash memory

Table 1
Comparison of the properties of different NVM technologies.

Prop./type	Flash [16]	FRAM [17]	MRAM [18]	PCM [19]
Read time	70 ns	50 ns	3–20 ns	48 ns
Write time	10 µs	50 ns	3–20 ns	150 ns
Symmetric	No	Yes	Yes	No
Erase	Yes	No	No	No
Endurance	10^5	10^{12}	$> 10^{15}$	10^{12}
Maturity	Established	Production	Production	Testing

[17]. However, this technology has a limited lifetime (~10^{12} cycles), as the ferroelectric material eventually wears out. The read operation with FRAM is destructive because it requires switching the polarisation state to sense the current state. Due to this, the read and write cycles require the same amount of time and energy, thus making FRAM a symmetric memory.

A different symmetric NVM technology is Magnetoresistive RAM (MRAM) that uses a magnetic tunnel junction as a storage device, enabling an unlimited number of read/write cycles [18]. In contrast to FRAM, the read operation is non-destructive, allowing shorter read cycles and improved power efficiency.

Finally, Phase Change Memory (PCM) is an emerging NVM technology which is asymmetric but does not require erasing, offering notably shorter read/write times and significantly higher power efficiency compared to Flash [19]. In addition, erasing is not a prerequisite before writing data which makes PCM an attractive alternative to Flash.

2.2. Motivation

To motivate the need for the work presented in this paper, we implemented the Allocated State policy (Fig. 2b) on two platforms with different NVM technologies (FRAM and Flash). Fig. 3 shows results when different software applications (a Binary Counter and FFT) are executed, where the energy required to save the system state is experimentally measured. The *Allocated State* policy allows for substantial energy savings when used with NVM technologies that do not require erasing (such as FRAM). Fig. 3a and b demonstrate that the cost for saving is proportionally reduced with the size of allocated memory, when compared to saving the entire memory (up to 85% reduction when the Binary Counter application is executed). However, this policy was not validated as part of a transient system and we observe that, when applied on a Flash-based system (as in [14]), it is far less effective as shown in Fig. 3c and d. This is because the overhead due to the erasing process, that is needed before saving the system state, is neglected in [14]. However, this is a typical property of Flash memory, which accounts for up to 94% of the total cost for saving as highlighted in Fig. 3c and d. Moreover, we notice that the Fig. 3a/b and c/d are very similar in terms of percentage overhead as the energy required is a function of time. For this reason, we will consider only the energy overhead as the metric of performance for the rest of this paper.

This example motivates the need to apply novel policies depending on the properties of the NVM being used and ensure that each policy is applied on the appropriate NVM technology. In Section 3, we will analyse the inefficiency in existing state retention techniques, while in Section 4, we will consider the properties of NVM technologies (such as symmetricity, erasing, endurance and efficiency) to explore different selective policies for system state retention, aiming for more efficient saving/restoring mechanisms and extended memory lifetime.

3. Inefficiencies in existing policies

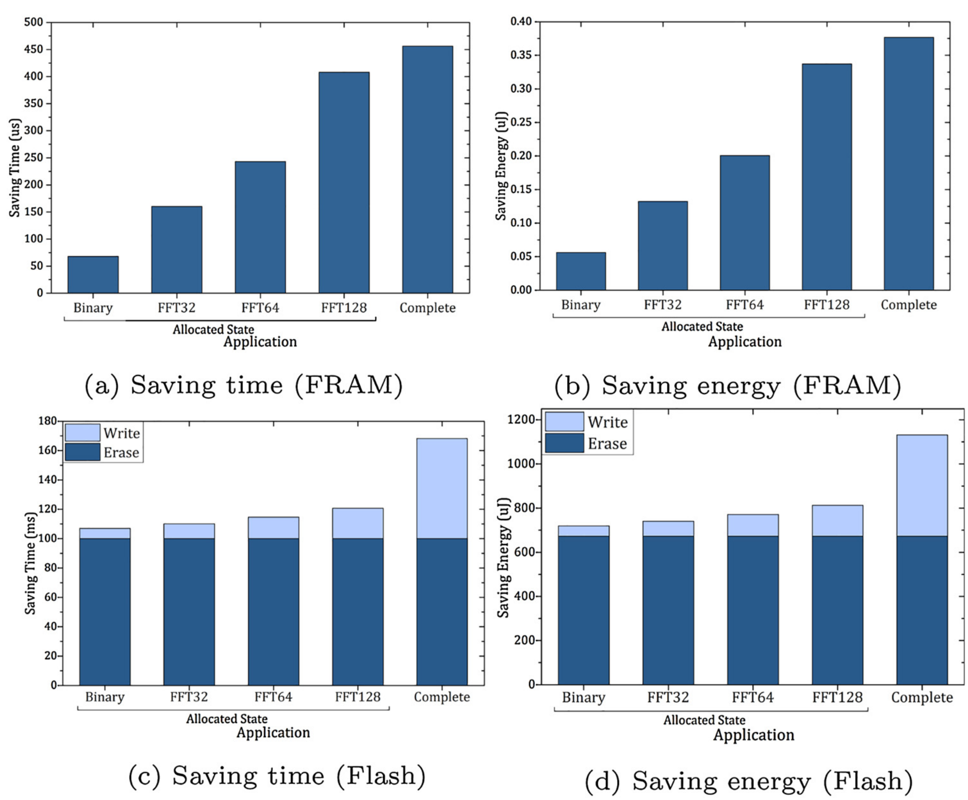
The problem statement and motivation for this work was presented in Section 2, showing that there is a potential for time and energy reductions when saving the system state in transiently-powered embedded systems. In this section, we analyse existing policies to gain insight into the parameters affecting the energy required to save and restore system state. In this way, the factors which play an important role in the saving/restoring process can be determined.

3.1. Complete State

When the *Complete State*, including the entire RAM memory, is saved to a NVM without erase cost, the energy needed to save the state is:

$$E_{Save\_CS} = M \cdot (P_{R_{VM}} \cdot t_{R_{VM}} + P_{W_{NVM}} \cdot t_{W_{NVM}}) \quad (1)$$

where E_{Save_CS} is the total energy required for saving the entire system state, M represents the size of main (volatile) memory in bytes, while $P_{R_{VM}}$ and $t_{R_{VM}}$ refer to the power and time required to read a byte from volatile memory (VM), and $P_{W_{NVM}}$ and $t_{W_{NVM}}$ describe the power and



(a) Saving time (FRAM)
(b) Saving energy (FRAM)
(c) Saving time (Flash)
(d) Saving energy (Flash)

Fig. 3. Experimental results showing the time and energy overhead of Allocated State with FRAM and Flash memories.

Table 2
Typical values for reading/writing data from/to different NVM technologies.

	FRAM [17]	Flash [17]	PCM [19]	SRAM [20]
P_{erase}	–	8 mW	–	–
t_{erase}	–	100 ms	–	–
P_{NVM_R}	2 mW	6 mW	2.5 mW	–
t_{NVM_R}	50 ns	70 ns	48 ns	–
P_{NVM_W}	2 mW	7 mW	3 mW	–
t_{NVM_W}	50 ns	10 µs	150 ns	–
P_{VM_R}	–	–	–	1 mW
t_{VM_R}	–	–	–	10 ns
P_{VM_W}	–	–	–	1 mW
t_{VM_W}	–	–	–	10 ns

time required to save a single byte to NVM. The energy required to restore the system state is given by:

$$E_{\text{Rest\_C S}} = M \cdot (P_{R_{\text{NVM}}} \cdot t_{R_{\text{NVM}}} + P_{W_{\text{VM}}} \cdot t_{W_{\text{VM}}}) \tag{2}$$

In this case, the process is inverted; data is read from NVM and written to Volatile Memory (VM), hence parameters $P_{R_{\text{NVM}}}$, $t_{R_{\text{NVM}}}$ refer to the power and time required to read a byte from NVM, while $P_{W_{\text{VM}}}$ and $t_{W_{\text{VM}}}$ describe the power and time needed to write a byte to the VM. However, when the *Complete State* policy is applied to a system which features a NVM technology with erase cost, Eq. (1) becomes:

$$E_{\text{Save\_C S}} = P_{\text{erase}} \cdot t_{\text{erase}} + M \cdot (P_{R_{\text{VM}}} \cdot t_{R_{\text{VM}}} + P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}}) \tag{3}$$

where P_{erase} and t_{erase} represent the power and time required to erase the NVM.

Table 2 contains typical values of these parameters for three different NVM technologies: FRAM, Flash and PCM. Assuming a platform with a main memory M equal to 4 kB, the energy consumption for saving the system state, when the *Complete State* policy is applied can be estimated as 0.9 µJ (FRAM) or 1.1 mJ (Flash).

Considering Eqs. (1)–(3), to reduce the total amount of energy spent for saving or restoring the system state, the state retention policy needs to reduce the amount of data being saved and restored to/from the NVM, and hence reduce the effective M. For Flash-based systems, the cost for erasing is expected to account for 74% of the total energy overhead (673 µJ). This cost is not a function of M and therefore, to minimise the overhead that the erasing process is contributing, we need to reduce the number of times the NVM needs to be erased.

3.2. Allocated State policy

As described in Section 2, the *Allocated State* policy works by distinguishing the memory segments used by the main application, to reduce the amount of data being saved and restored to/from the NVM on every power failure. The amount of allocated memory is defined as:

$$m = \alpha \cdot M$$

where α is the fraction of the total size M of the main memory being used. For a system featuring a NVM technology without erase cost, the *Allocated State* policy is described by:

$$E_{\text{Save\_A S}} = P_{\text{track}} \cdot t_{\text{track}} + m \cdot (P_{R_{\text{VM}}} \cdot t_{R_{\text{VM}}} + P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}}) \tag{4}$$

where P_{track} and t_{track} refer to the power and time required for tracking the location of the end of the heap segment and the top of the stack segment. In contrast, when a NVM technology with erase cost (such as Flash) is used, Eq. (4) becomes:

$$E_{\text{Save\_A S}} = P_{\text{erase}} \cdot t_{\text{erase}} + P_{\text{track}} \cdot t_{\text{track}} + m \cdot (P_{R_{\text{VM}}} \cdot t_{R_{\text{VM}}} + P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}}) \tag{5}$$

The energy required to restore the system state when using the *Allocated State* policy depends solely on the size of allocated memory space. Therefore, the energy requirement is given by:

$$E_{\text{Rest\_A S}} = m \cdot (P_{R_{\text{NVM}}} \cdot t_{R_{\text{NVM}}} + P_{W_{\text{VM}}} \cdot t_{W_{\text{VM}}}) \tag{6}$$

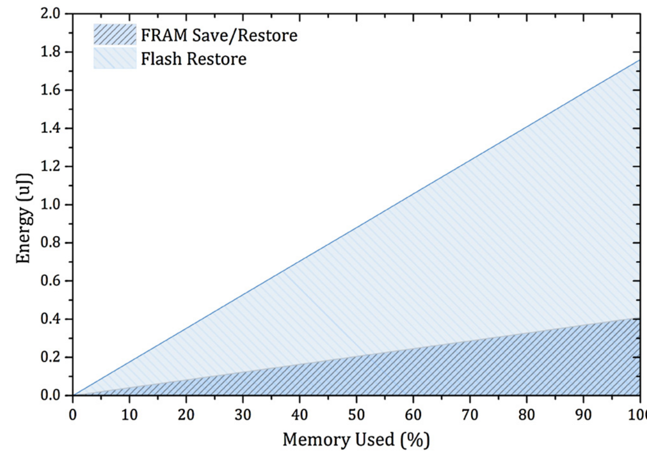


Fig. 4. Modelled energy consumption for restoring the system state using the *Allocated State* policy, applied to a system featuring a symmetric memory without erase cost (Eq. (4)) and a system featuring an asymmetric memory with erase cost (Eq. (6)), using typical parameters of FRAM and Flash from Table 2.

Eqs. (4)–(6) are used with the typical parameter values presented in Table 2 to model the energy consumption of the *Allocated State* policy. However, the cost for tracking the end of the heap segment and the top of the stack ($P_{track} \cdot t_{track}$) is neglected as it typically accounts for only a small proportion of the total energy cost. Figs. 4 and 5 show the relationship between the energy required to save and restore the system state and the percentage of allocated memory when the *Allocated State* policy is applied. Fig. 4 shows the expected effect of this policy both in terms of restoring the system state on two different systems featuring FRAM and Flash memory respectively. For the system with the FRAM memory, the energy overhead is equal for saving and restoring the system state, and it is expected to be reduced by up to 89% when only a small portion of the main memory is used (e.g. 10%), compared to saving the entire system state. The energy overhead for the system featuring Flash memory is more significant as it is a more power-hungry technology. Fig. 5 shows the expected energy requirements for saving the system state when the same policy is applied to a system featuring a NVM with erase cost (i.e. Flash). A breakdown of the components contributing to the total cost for saving the system state is shown in Fig. 5 which highlights that energy savings of up to 24% can be achieved compared to saving the entire system state (with 10% allocated memory). In this case, the erasing process would account for 74–96% of the total cost for saving the system state. As a consequence,

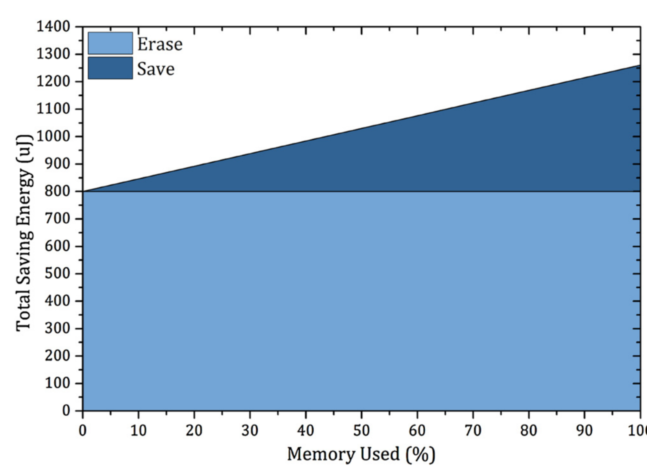


Fig. 5. Modelled energy requirement for saving the system state using the *Allocated State* policy, applied to a system featuring NVM with erase cost (Eq. (5)), using typical parameters of Flash from Table 2.

alternative policies need to be devised to tackle the challenge of efficiently saving the system state in transiently-powered embedded systems.

In the following section, a range of policies are proposed which aim to provide a more energy efficient state retention operation, by exploiting the fundamental properties of different NVM technologies.

4. Proposed policies: selective state retention

In this section, we propose a range of policies based on two principles: (a) for NVM technologies with erase cost, we concatenate multiple images and fill NVM before erasing and (b) for asymmetric read/write memory technologies, we save only data that has changed since the last restore. The presented policies are developed based on the properties of different NVM technologies and the usage of NVM by the main application.

4.1. Multiple Allocated State Images (MASI)

As presented in Section 2, the *Allocated State* policy works efficiently with NVM technologies that do not need erasing (e.g. FRAM, MRAM). However, this policy does not offer significant benefits when applied to Flash memory that requires erasing before writing. For this reason, we propose the *Multiple Allocated State Images (MASI)* policy as shown in Fig. 6, based on concatenating a new image after the previous one, and only erasing when all NVM is filled.

Each image consists of the *.data*, *.bss* and *heap* segments, as well as the *stack* and a 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.

The proposed policy relies on the fact that the size of the NVM is normally multiple times larger compared to the used portion of main memory. For example, some microcontrollers (MCUs) offer up to 32 times more space in their NVM compared to their main memory [21].

To identify the location of the latest image that needs to be restored after a power outage, two variables need to be recorded on every image: (a) the size of the image and (b) a flag indicating whether this image has been previously restored. These variables can be saved at the beginning and the end of the image respectively, as shown in Fig. 6.

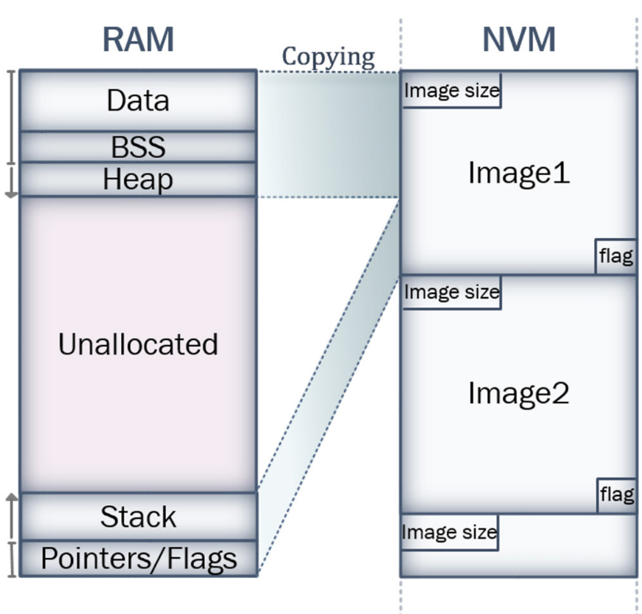


Fig. 6. State retention policy of *Multiple Allocated State Images*: to reduce unnecessary erase operations, multiple allocated images are concatenated into NVM until it is full.

Consequently, during the restore phase, the system first reads the size of the image and then checks whether it has already been restored. If the flag bit is set, it means that a newer image has been stored below and the same procedure will be followed until a cleared flag is detected.

The average energy cost for saving the system state when the presented policy is applied to a system with a NVM technology with erase cost, is described by:

$$E_{\text{Save\_M ASI}} = \frac{P_{\text{erase}} \cdot t_{\text{erase}}}{i} + P_{\text{track}} \cdot t_{\text{track}} + m \cdot (P_{R_{\text{VM}}} \cdot t_{R_{\text{VM}}} + P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}}) \quad (7)$$

where i represents the number of saving iterations the system can perform before erasing the NVM while the term $\frac{P_{\text{erase}} \cdot t_{\text{erase}}}{i}$ describes the average erasing energy.

The energy savings that this policy offers are more evident as the number of images that can be saved in NVM before erasing increases. A smaller amount of allocated memory (m) would result in more images being stored in NVM and, as a consequence, the average energy overhead for saving the system state can be effectively reduced.

The energy required to restore the system state is described by:

$$E_{\text{Res\_M ASI}} = P_{\text{img}} \cdot t_{\text{img}} + m \cdot (P_{R_{\text{NVM}}} \cdot t_{R_{\text{NVM}}} + P_{W_{\text{VM}}} \cdot t_{W_{\text{NVM}}}) \quad (8)$$

where P_{img} and t_{img} refer to the power and time required to locate the latest image that needs to be restored, using one of the methods described earlier.

Fig. 7 shows the modelled energy consumption for saving the system state when the *Multiple Allocated State Images* policy is applied to a system featuring an asymmetric NVM with erase cost such as Flash (using the values from Table 2). This figure is based on the assumption that the cost for tracking the end of the heap segment and the top of the stack is significantly lower compared to the other components of Eq. (7) and can, therefore, be neglected. As the maximum number of iterations (i) before erasing decreases, the average erasing energy increases, as shown by the "steps" in the average erase cost. Compared to saving the entire system state, energy savings of up to 95% can be achieved, when the size of allocated memory (m) accounts for 10% of the total size of main memory (M), according to the model.

A graph showing the energy requirements for restoring the system state can be plotted using Eq. (8). However, parameters P_{img} and t_{img} cannot be estimated accurately as they are also expected to account only for a small portion of the total restore energy, this expression becomes identical to Eq. (6). For this reason, Fig. 5 can be used as an estimate for the restore process when the *Multiple Allocated State Images* policy is applied.

Using this policy, the energy required for saving the system state can

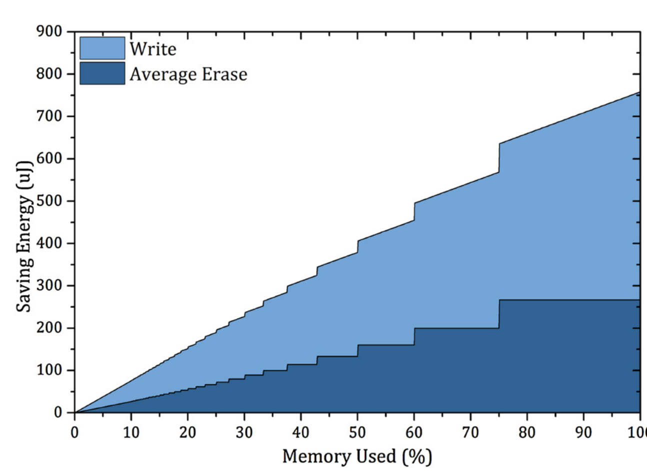


Fig. 7. Modelled energy consumption for saving the system state using the *Multiple Allocated State Images* policy, applied to a system featuring NVM with erase cost (Eq. (7)), using typical parameters of Flash from Table 2.

be effectively reduced depending on the size of allocated state which also dictates the number of saving iterations that can be performed without erasing, tackling the high energy cost of erasing the NVM.

The energy consumption of this policy is modelled using a Flash-based system as an example of a memory with a high erase cost. Even though Flash (which is an asymmetric technology) is currently the only available NVM technology exhibiting this property, this policy may also be beneficial on future symmetric memories which suffer from a high erase cost.

4.2. Block-based policies

The previous policy works efficiently with Flash memory only when the main application is not using a big part of the main memory. When this happens, a small number of images can be saved in the NVM, increasing the frequency that Flash memory needs to be erased. Moreover, when the entire main memory is used, this policy introduces an overhead, due to the time needed to identify the allocated space in the main memory (saving) and to detect the right image to be restored (restoring).

Furthermore, the main memory might contain data that has not been updated since the last restore. In this respect, data already stored in NVM is overwritten with its pre-existing content, resulting in unnecessary write operations. Depending on the NVM technology, these operations can be highly time and energy consuming. To address this challenge, we propose two different selective policies (*Updated Blocks* and *Multiple Updated Blocks*), targeted for asymmetric memory technologies, which aim to reduce the amount of redundant writes.

4.2.1. Updated Blocks

As shown in Fig. 8, this selective policy is based on dividing the main memory into $n + 1$ number of b_n blocks (b_0–b_n) of size s. During the saving process, every block of the main memory is compared with the corresponding block previously saved in NVM. If a memory cell has changed since the last restore operation, the data of the whole block needs to be copied to the corresponding block on the NVM. For example, in Fig. 8, only the highlighted blocks (b_0, b_4, b_{n-1} and b_n) changed since the last restore and, therefore, are the only blocks that are updated in NVM during the saving. As a consequence, the unnecessary write operations are avoided and therefore, the time and energy overhead for saving the system state is reduced. However, this policy does not affect the restore process as the entire state needs to be restored to the main memory.

To get a better insight on the operation of the *Updated Blocks* policy, Eq. (9) describes the maximum energy required to save the system state when this policy is applied to a system with a NVM without erase cost:

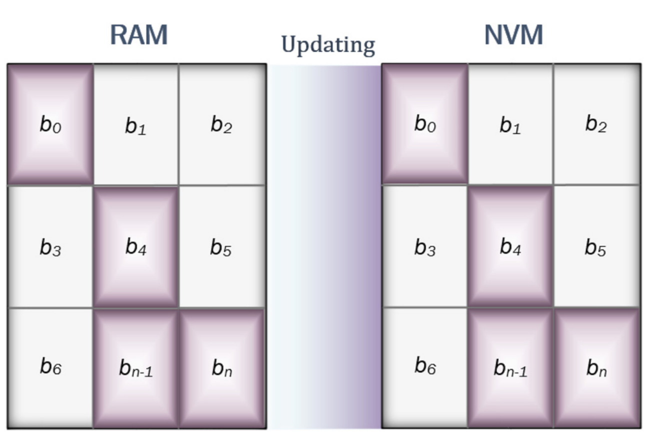


Fig. 8. State retention policy of *Updated Blocks*: current state compared with previously saved image, and only blocks that have changed are updated.

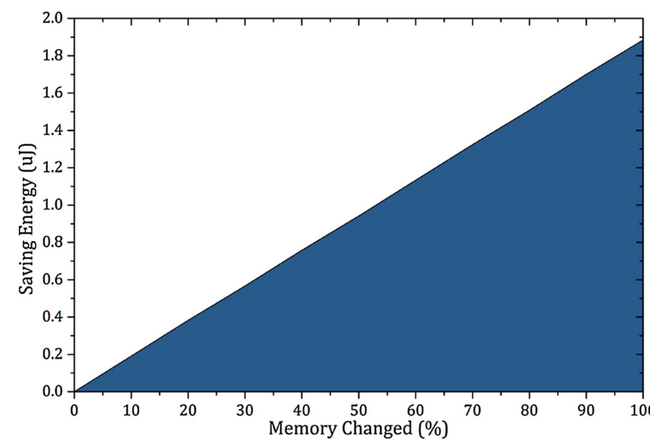


Fig. 9. Modelled energy consumption for saving the system state using the *Updated Blocks* policy, applied to a system featuring an asymmetric NVM without erase cost (Eq. (9)), using typical parameters of PCM from Table 2.

$$E_{\text{Save\_}UB_{\text{MAX}}} = M \cdot P_{\text{comp}} \cdot t_{\text{comp}} + B \cdot s \cdot P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}} \tag{9}$$

where B represents the number of blocks that need to be updated in NVM and s represents the size of each block in bytes. The comparison between the corresponding blocks in VM and NVM includes a memory access for both memories, so that their values can be read and then compared. Consequently, P_{comp} is approximately equal to the sum of $P_{R_{\text{VM}}}$ and $P_{R_{\text{NVM}}}$, while t_{comp} is approximately equal to the sum of $t_{R_{\text{VM}}}$ and $t_{R_{\text{NVM}}}$. Eq. (9) describes the maximum energy required to save the system state as it is considered that all memory cells in VM are compared with their corresponding cells in NVM. In practice, only a fraction of cells will be compared unless there are no memory changes between two consecutive power failures. This happens because once a cell in VM has been updated, the entire block is saved in NVM without comparing the following cells of the block.

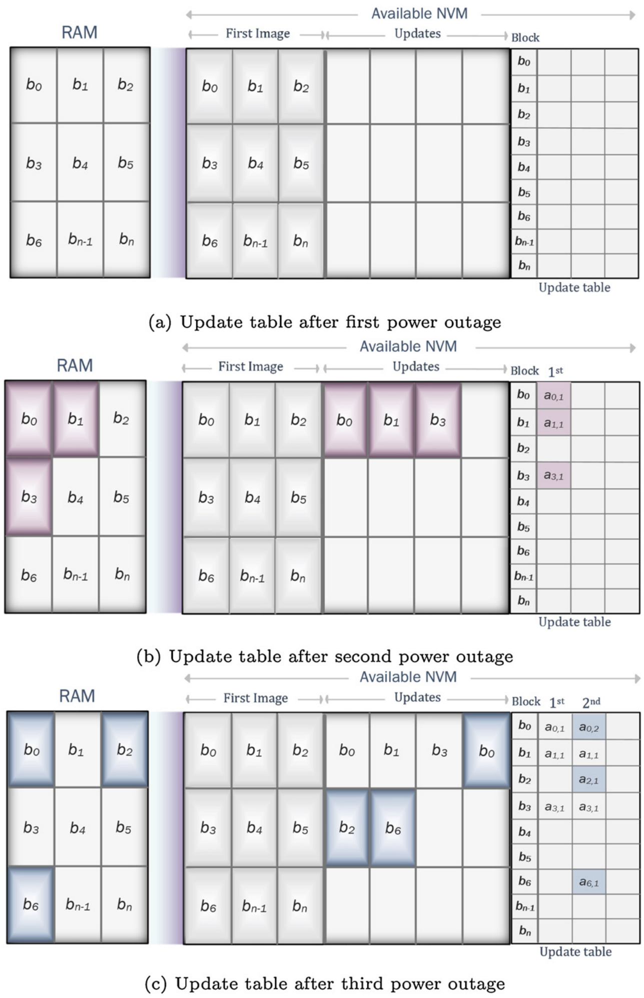
The modelled energy consumption for saving the system state using the *Updated Blocks* policy on a system with an asymmetric NVM without erase cost can be estimated using Eq. (9). Fig. 9 shows the relationship between the expected energy required for saving the system state and the percentage of changed memory, when the *Updated Blocks* policy is applied to a system featuring an asymmetric NVM without erase cost (such as PCM). Here, we conclude that the relationship between the required energy and the number of blocks that need to be updated is of linear nature.

This selective policy is expected to work efficiently with NVM technologies that are asymmetric and do not require erasing, such as PCM. However, it will not have a significant impact with NVM technologies that need erasing (i.e. Flash), due to the high cost of the erasing process. The following policy addresses selectivity with this type of memory.

4.2.2. Multiple Updated Blocks

This policy is based on using the available free space in NVM memory to only save the parts (blocks) of main memory that have changed, using contiguous free space. Similarly to the *Updated Blocks* policy, the main memory is divided into $n+1$ number of b_n blocks (b_0–b_n). As shown in Fig. 10a, the first time a power failure occurs, the available NVM memory is erased and a system state (reference state) is saved. Upon subsequent power outages, each block of the main memory is compared with the corresponding block of the NVM. If a memory cell has changed since the last restore, its updated version is saved in the first available space, without replacing the previous version of the same block in NVM.

As an example, in Fig. 10b, only blocks b_0, b_1 and b_3 changed since the first power outage, while in Fig. 10c, blocks b_0, b_2 and b_6 have changed since the second power outage. If a previously saved block



(a) Update table after first power outage

(b) Update table after second power outage

(c) Update table after third power outage

Fig. 10. State retention policy of *Multiple Updated Blocks*.

changes again (e.g. b_0 in this case), it has to be saved in NVM. A table is created to keep track of the location in NVM of the latest version of each block ($a_{n,m}$, where n is the block number and m is the version). The content of this table is used during the restore process to locate the most recent version of each block to be restored to main memory. This table has a fixed number of rows, equal to the number n of blocks. To avoid erasing the table during every state saving, multiple versions of the table are saved contiguously. To identify the latest table, the area reserved for tables is swept through, until an unwritten cell is located which denotes the end of the most recent table. Once the available space for updates or tables has been filled, the entire NVM is erased and the same process is restarted.

The following equation describes the energy requirements of this policy when applied to a system with NVM with erase cost:

$$E_{\text{Save\_}M\text{UB}_{\text{MAX}}} = \frac{P_{\text{erase}} \cdot t_{\text{erase}}}{i} + M \cdot P_{\text{comp}} \cdot t_{\text{comp}} + B \cdot s \cdot P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}} + P_{\text{table}} \cdot t_{\text{table}} \quad (10)$$

where P_{table} and t_{table} describe the power and time required to create the latest version of the table.

Eq. (10) can be useful for plotting a figure to show the relationship between the energy required for saving the system state and the fraction of memory that has changed since the last restore. However, estimating the overhead for creating the table is difficult while it is expected to introduce only a small overhead compared to the other components of Eq. (10). For this reason, Eq. (11) is a simplified version of this equation which will be used for modelling purposes:

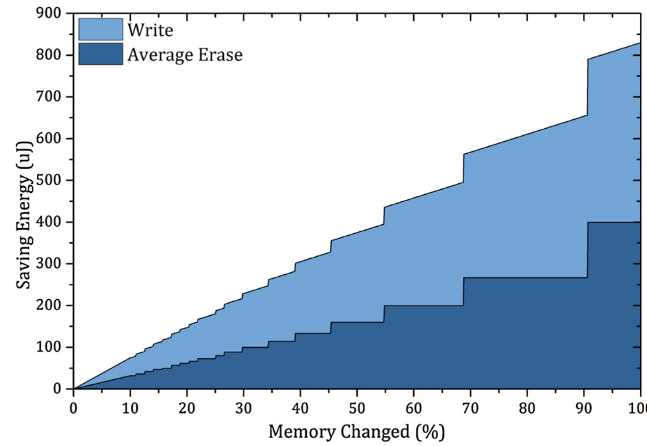


Fig. 11. Modelled energy consumption for saving the system state using the *Multiple Updated Blocks* policy, applied to a system featuring an asymmetric NVM with erase cost (Eq. (11)), using typical parameters of Flash from Table 2.

$$E_{\text{save\_M UB}} = \frac{P_{\text{erase}} \cdot t_{\text{erase}}}{i} + M \cdot P_{\text{comp}} \cdot t_{\text{comp}} + B \cdot s \cdot P_{W_{\text{NVM}}} \cdot t_{W_{\text{NVM}}} \quad (11)$$

Fig. 11 shows the relationship between the expected energy required for saving the system state and the percentage of changed memory, when the *Multiple Updated Blocks* policy is applied to a system featuring an asymmetric NVM with erase cost such as Flash. To plot this graph, a main memory size of 4 kB and a total NVM size of 16 kB are considered. In addition, it is assumed that memory is changing in a contiguous way for simplicity reasons. It is shown that the erasing cost can be significantly reduced compared to erasing on every iteration (Fig. 5) and energy savings of up to 94% can be achieved, when only 10% of the memory has changed. Finally, the instantaneous "steps" observed at the average erase energy are related to the decrease in the maximum number of iterations before erasing the NVM.

When the system state needs to be restored, the latest table needs to be identified so that the most recent version of each block can be located. The following equation can be used to calculate the expected energy requirements for the restore process:

$$E_{\text{Rest\_M UB}} = P_{\text{loc}} \cdot t_{\text{loc}} + M \cdot P_{R_{\text{NVM}}} \cdot t_{R_{\text{NVM}}} + M \cdot P_{W_{\text{VM}}} \cdot t_{W_{\text{VM}}} \quad (12)$$

where the product of P_{loc} and t_{loc} describe the energy required to locate the latest table upon restore.

The Allocated State policy performs well when applied on a system with a symmetric memory without erase cost such as FRAM, as explained in Section 2. We propose *Multiple Allocated State Images* and *Multiple Updated Blocks* which are designed to reduce the cost of erasing. The former can be applied both on symmetric and asymmetric memories with erase cost, while the latter is focused on asymmetric memories, which suffer from high erase cost. Finally, the *Updated Blocks* policy is designed to reduce redundant write operations, offering great potential for systems featuring an asymmetric memory without erase cost.

5. Experimental design

The proposed policies were experimentally validated using two platforms, a Texas Instruments MSP430FR with FRAM memory [22], and a NXP LPC812 /colorredplatform [23] with Flash memory. The MSP430FR with FRAM allows read/write operations at the byte level, whereas the LPC812 with Flash memory is at the page level (64 bytes). The block diagram of the experimental setup is shown in Fig. 12. Here, an energy harvester is used as the energy source of the system. The decoupling capacitance of the system (approximately 16 μF) is adequate for saving/restoring the system state using the MSP430 platform with FRAM memory) while an additional capacitor (C1) is required for

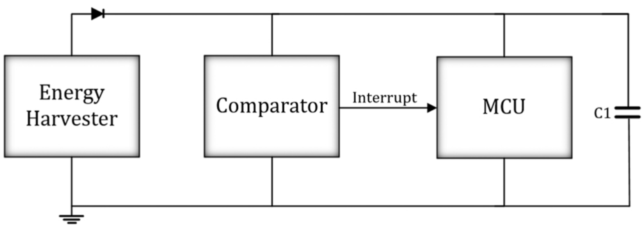


Fig. 12. Schematic of the experimental setup.

the Flash-based platform (LPC812) in order to provide enough energy/time for the state to be saved to Flash. The external low-power comparator used in [12] is used to enable interrupts to be triggered when the supply voltages surpasses a threshold (V_{th}) set by the microcontroller. The microcontroller starts its state retention/restore operation when the corresponding interrupt has been triggered by the comparator.

To implement the *Multiple Allocated State Images* policy, the end of the heap segment and the top of the stack need to be identified. To make this feasible with the available hardware (LPC812), a combination of *malloc()* and *free()* functions is used to locate the end of the heap segment which is preceded by the *.data* and *.bss* segments, as shown in Fig. 2a. When the combination of these functions is executed, a general purpose register contains the address of the end of the heap segment and can, therefore, be copied to NVM. Subsequently, the top of the stack can be obtained in a similar manner, by saving the value of the stack pointer (SP).

In addition, identifying the latest image using the flagging methodology proposed in Section 4.1 is infeasible, as it only works for Flash memories that allow writing of a single byte. However, the minimum writeable size is equal to the page size for this platform (as well as many Flash-based microcontrollers). Consequently, a different method needs to be used as it would not be feasible to update the value of the flag once the image has been restored. In this case, the memory is swept through until an empty page is found, which denotes the latest image. This is done by exploiting the attribute of Flash memory which implies that when a page is erased, its cells are set to logic level "1". Consequently, as images are stored in a continuous way, the first empty page reveals the ending address of the most recent image. Once the latest image has been located, the pointers/flags section is used so that the restore process can be executed. This method, however, introduces a small overhead that gradually increases, depending on the total size of the previous images.

The proposed policies were evaluated using a custom application (uBenchmark), which allows us to define the percentage of allocated memory at compile time. Moreover, it enables to define a portion of the allocated memory where the data is randomly changed. As shown in Fig. 13, three parameters are used to define this allocated space (γ) as well as the boundaries of the randomly changed section (α and β).

For the results presented in Section 6, two different cases were considered. First, to allow plotting of the "Memory Used" (plotted on the x-axis of some results presented), γ is varied in order to change the portion of the allocated memory, while the values of α and β are constant. When "Memory Changed" needs to be plotted, α is fixed to 0 (start of the main memory), γ is equal to the size of the main memory and β is varied to adjust the percentage of the memory being changed.

The proposed selective policies were validated using Hibernus [11]. This transient approach was selected because it is energy efficient and, application and platform agnostic [15]. In the following section, these policies are validated considering symmetricity and erase cost as the main properties of NVM technologies.

6. Experimental results

6.1. Symmetric NVM technologies without erase cost

For symmetric memory technologies (e.g. FRAM), comparing the

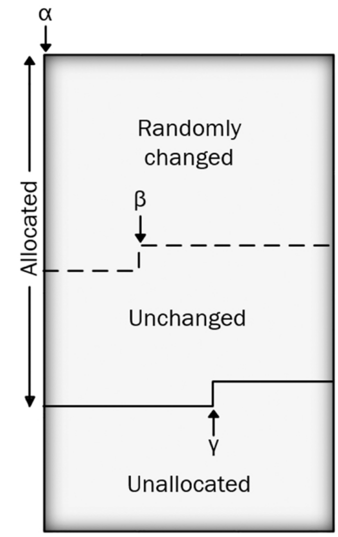


Fig. 13. uBenchmark's configurable parameters.

already saved with the current system state would only have a negative impact on the efficiency of the saving process since the cost of reading from the NVM is equal to writing. Therefore, the *Updated Blocks* approach is unsuitable. In addition, the *Multiple Allocated State Images* and *Multiple Updated Blocks* policies would not offer any benefits as this type of NVM technology does not require erasing before writing. For this reason, only the *Allocated State* policy is experimentally validated.

Fig. 14 shows the amount of energy required for saving the system state while changing the fraction of allocated memory, when the *Allocated State* policy is applied to the MSP430FR with FRAM. Depending on the fraction of memory used by the main application, energy savings of up to 86.2% can be achieved compared to saving the entire system state. As an example, when an application uses a small portion of the main memory such as 10%, the saving/restoring process requires 52nJ of energy to be completed. This is due to the overhead incurred by the tracking of the end of the heap segment and the top of the stack, as described in Section 3.2. This policy is more energy efficient compared to the *Complete State* policy as long as < 88.4% of the memory is being used.

6.2. Asymmetric NVM technologies with erase cost

This section presents the experimental results of various policies when applied to systems featuring asymmetric NVM Technologies *with*

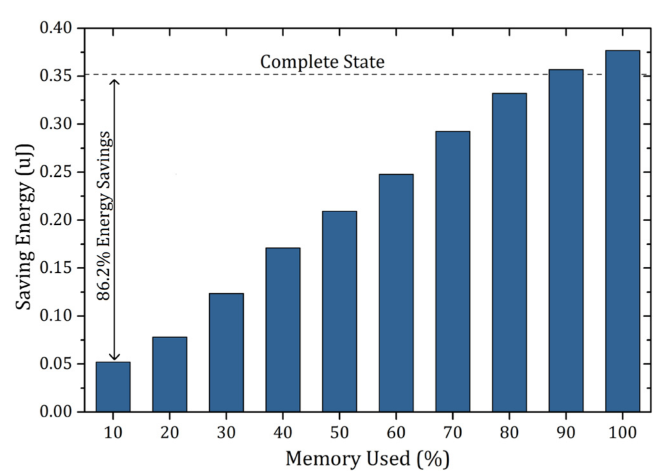


Fig. 14. Experimental results showing the energy required to save the system state, when using the *Allocated State* policy on FRAM.

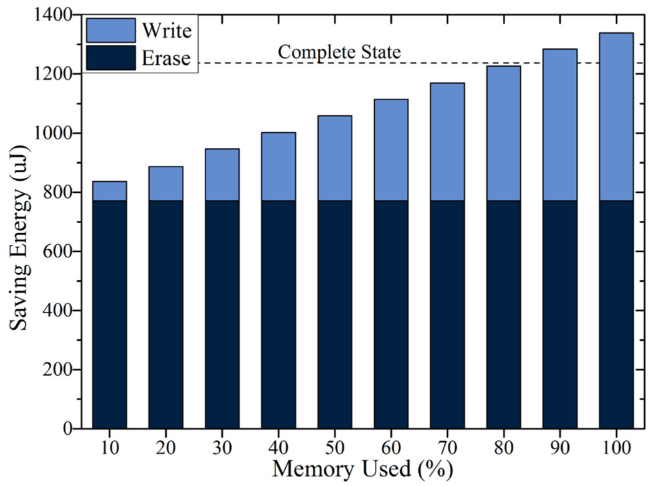
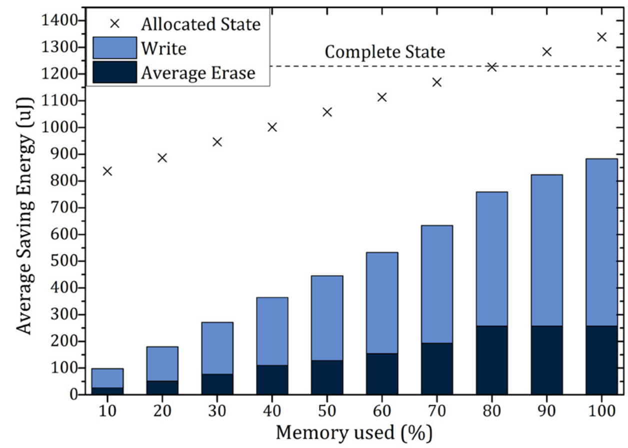
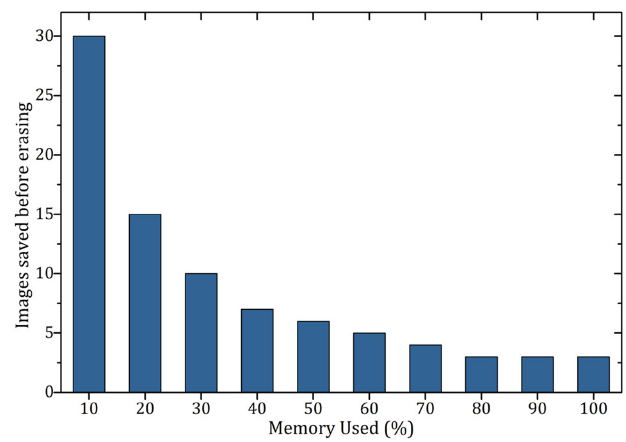


Fig. 15. Experimental results showing the energy required to save the system state, when using the *Allocated State* policy on Flash memory.



(a) Energy requirement for saving the system state



(b) Number of saved images before erasing against percentage of used memory

Fig. 16. Experimental results showing (a) the energy required to save the system state and (b) the number of iterations, when using the *Multiple Allocated State Images* policy on Flash memory.

erase cost and results in a discussion which compares the performance of the applied policies.

6.2.1. Allocated State policy

As shown in Fig. 15, when the *Allocated State* policy is applied to the LPC812 with Flash memory, the energy overhead due to the erasing is dominant (771 μJ).

However, this policy is still working efficiently compared to saving the complete state (shown by the dashed line), when the allocated memory is less than 82.3%. This is due to the overhead of having to track the end of the heap segment and the top of the stack so that only the allocated space is saved in NVM which counteracts with the benefits of this policy. Moreover, when the memory usage is small (e.g. 10%) the energy spent for the saving process alone (i.e. ignoring the erase overhead) is reduced by up to 84%. In this case, the model for the *Allocated State* policy (presented in Section 3.2) is validated as Figs. 5 and 15 show almost identical behaviour with an average error of less than 4%.

6.2.2. MASI

Fig. 16a shows the energy required by the *Multiple Allocated State Images* policy when applied to the same platform. This policy offers significantly higher energy efficiency as the number of saved images without erasing increases, depending on the percentage of allocated memory. This is due to the substantial energy reduction for erasing, considering that the cost of erasing is spread across the number of saving iterations that can be performed with a single erase. For ease of comparison, the X points show the energy requirements of the *Allocated State* policy while the dashed line shows the default *Complete State* policy. When using *Multiple Allocated State Images*, the energy overhead is reduced by 32.3–87.3% compared to *Allocated State*. Fig. 16b shows the maximum number of images that can be saved in NVM before erasing, depending on the memory usage. For this specific platform, the size of main memory is 4 kB, while the available Flash memory is 12 kB, as the .text segment has a static size of 4 kB (total 16 kB). As an example, for an application that has an allocated memory size equal to 10% of the total space, 30 images can be saved before erasing, leading to an average erasing energy overhead of 26.7 μJ.

Comparing the experimental results (Fig. 16a) with the modelled version of this policy (Fig. 7), we observe that there is an average error of approximately 14%. As explained in Section 4.1, the cost for tracking the end of the heap segment and the top of the stack is not included and therefore, it is expected that the modelled energy consumption of this policy would be lower compared to the experimental results.

Fig. 17 shows the energy needed to restore the image as a function of the number of images saved in NVM, for different values of memory usage (10–50%) when the *Multiple Allocated State Images* policy is applied. Here, a energy overhead can be seen that gradually increases with the total size of the memory occupied by the previously saved

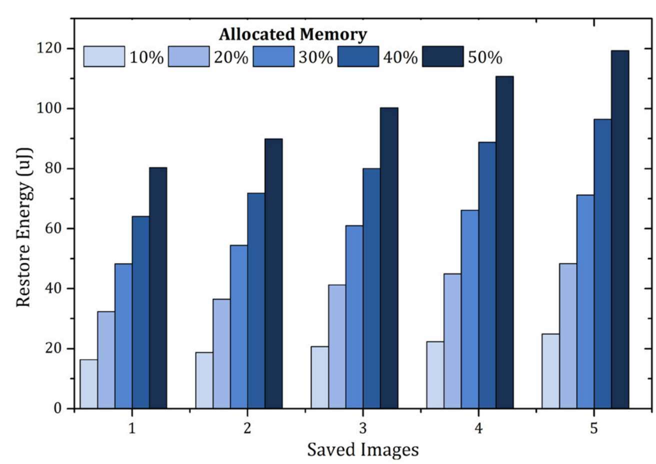
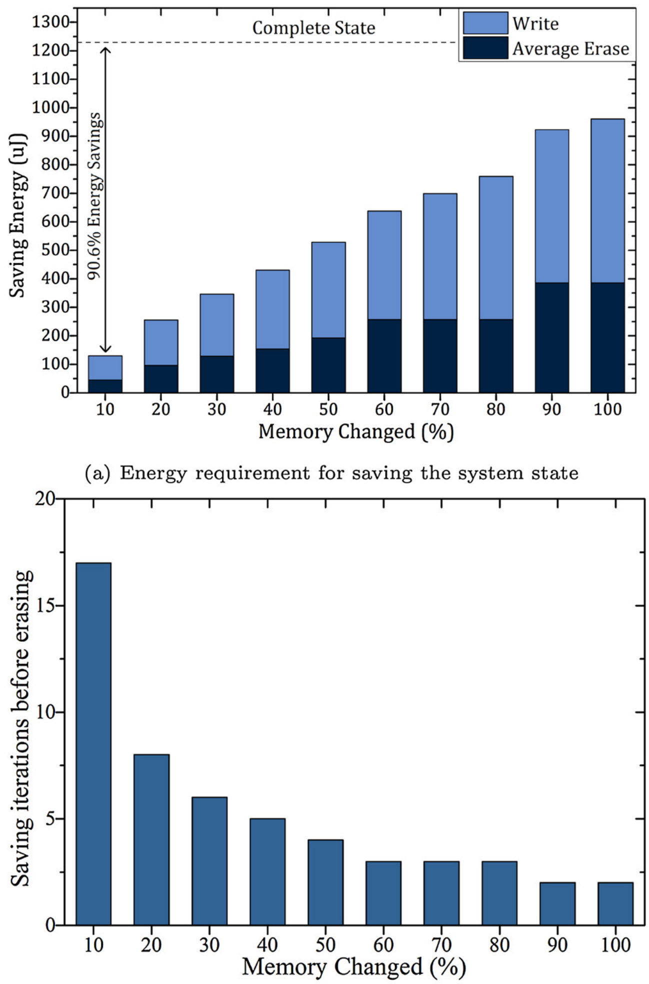


Fig. 17. Experimental results showing the restore energy against the number of stored images in NVM, when using the *Multiple Allocated State Images* policy on Flash memory.



(a) Energy requirement for saving the system state

(b) Number of state retention iterations before erasing

Fig. 18. Experimental results showing (a) the energy required to save the system state and (b) the number of iterations, when using the *Multiple Updated Blocks* policy on Flash memory.

images. This is due to the process of locating the latest image to be restored, as described in Section 4.1.

6.2.3. MUB

Fig. 18a shows the energy overhead when the *Multiple Updated Blocks* policy is applied to this platform. In this case, the block size (b_n) is equal to the page size (64 Bytes), which results in the main memory being divided into 64 blocks. This policy offers significantly higher time saving between 34.9% and 90.2%, when compared to saving the entire system state. Similarly, the energy overhead is reduced between 37.4% and 90.6% when 10% of the main memory is being used. This confirms that *Multiple Updated Blocks* is an energy efficient policy for Flash memory.

Fig. 18b shows the number of state retention iterations that can be performed depending on the percentage of contiguously changed memory. For this platform, assuming a .text segment size of 4 kB, the system is able to save the necessary data up to 17 times before erasing.

6.2.4. Discussion

Comparing the results presented in this section, we conclude that, while the *Allocated State* policy (Section 6.2.1) is the least efficient policy for a platform featuring an asymmetric NVM technology with erase cost such as Flash, it performs better than the *Complete State* policy as long as < 82% of main memory is allocated according to Fig. 15. In addition, it is observed that unless the entire portion of allocated memory has changed between two power intermissions, the *Multiple Updated Blocks* policy (Section 6.2.3) offers higher energy efficiency compared to the *Multiple Allocated State Images* policy (Section 6.2.2). The only exception to this case is when the percentage of allocated memory is < 10%, where saving the allocated memory without considering the memory changes is more effective.

6.3. Asymmetric NVM technologies without erase cost

This section presents the experimental results of various policies when applied to systems featuring asymmetric NVM Technologies *without* erase cost and results in a discussion which compares the performance of the applied policies.

6.3.1. Updated Blocks

The *Updated Blocks* policy is not appropriate for Flash memory, as the cost for erasing is prohibitive. However, for asymmetric memories without an erasing cost (e.g. PCM), it can be advantageous. However, currently there are no systems commercially available featuring a PCM memory to allow experimental evaluation of this policy. To illustrate this using the available hardware, we use Flash but negate the cost of erasing so that the benefits of this policy can be extracted. Fig. 19 shows the energy required to perform a complete system state retention using the *Updated Blocks* policy. This policy is more efficient when compared to saving the entire system state, if < 80.7% of main memory has changed since the last restore, as shown by the dashed line in Fig. 19. As these results have been obtained using a Flash-based system by negating the erase cost, they cannot be considered conclusive for PCM. However, comparing the properties in Table 2, we observe that the saving energy (*y*-axis) would scale by approximately two orders of magnitude, as PCM is more energy efficient compared to Flash (as confirmed by comparing Figs. 9 and 5). Also, the maximum energy savings would be lower, as PCM is a less asymmetric technology (the relative difference between read and write energy is lower), and therefore the effect of the policy is less evident. The overall behaviour, however, would still be observed and these results can be used as a proof of concept, rather than a quantitative result regarding the actual energy savings that this policy could offer to a system featuring a PCM memory.

Fig. 20 shows the average energy required to save a block as a function of the number of updated blocks. As the amount of energy needed to write a block is constant (8.2 µJ), the extra cost comes from

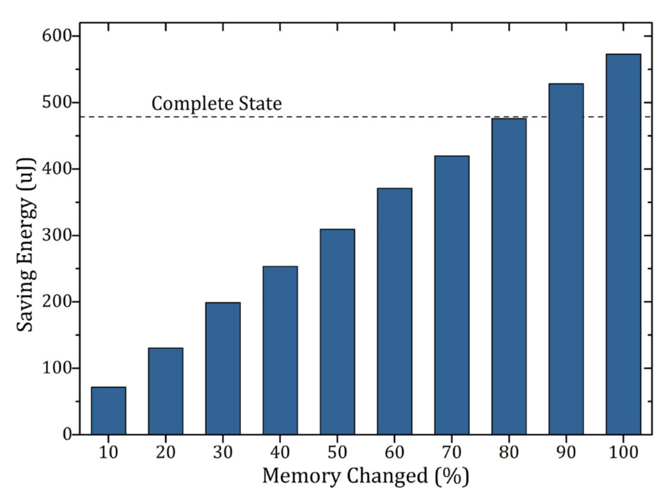


Fig. 19. Experimental results showing the energy required to save the system state, when using the *Updated Blocks* policy on asymmetric NVM without erase cost. Policy validated using the available hardware as a proof of concept (Flash, negating erase cost).

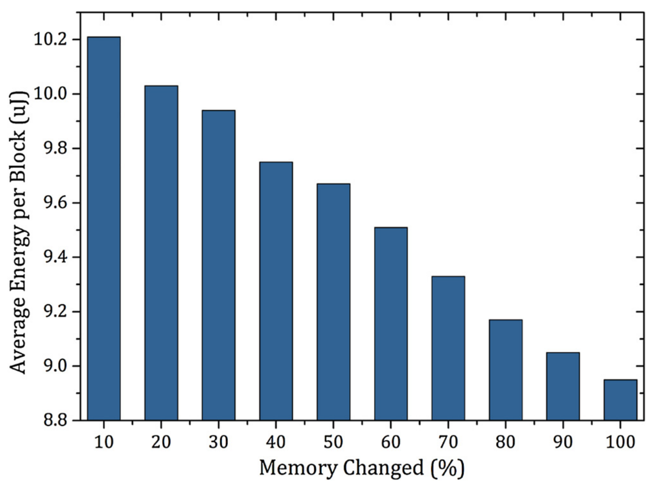


Fig. 20. Experimental results showing the average energy required to save each block against percentage of memory contiguously changed. Policy validated using the available hardware as a proof of concept (Flash, negating erase cost).

the control mechanism needed for reading, comparing and updating each block, as described in Section 4.2. Here, for a smaller number of updated blocks, the effect of the control mechanism is more evident whereas it is reduced when all blocks of the main memory have been updated and saved in NVM. The more blocks that need to be updated, the fewer comparisons need to be made between the current and previously saved system state.

6.3.2. *Allocated State*

As mentioned earlier, the *Allocated State* policy does not offer significant benefits when used with Flash. However, it can be a beneficial policy for asymmetric memories that do not require erasing. To demonstrate the effect of this policy on this type of NVM using the available platform (LPC812), we disregard the energy cost for erasing, focusing on the overhead for writing. Fig. 21 shows the energy required to perform a system state retention using the *Allocated State* policy. When less than 84.7% of the main memory is used by the main application, this policy is more efficient when compared to saving the entire system state, as shown by the dashed line in Fig. 21.

6.3.3. *Discussion*

Comparing Figs. 19 and 21, we observe that unless the entire allocated state has changed since the last power failure, the *Updated Blocks* policy is more energy efficient compared to the *Allocated State* policy. However, the latter should be preferred when the frequency of power failures is low and, therefore, the likelihood that the vast majority of the memory cells have been updated is increased.

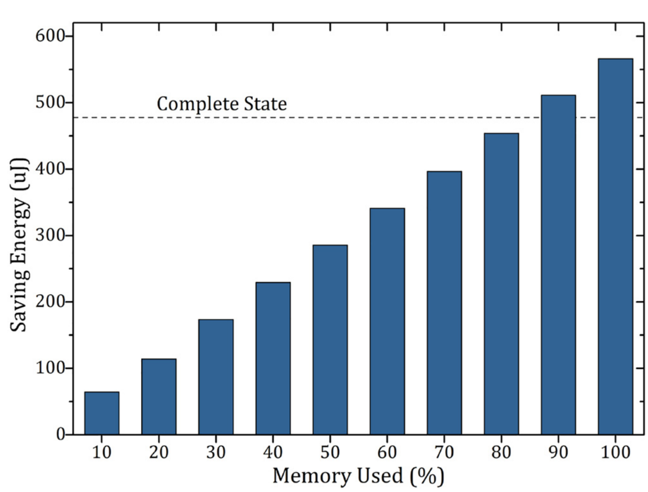


Fig. 21. Experimental results showing the energy required to save the system state using the *Allocated State* policy on asymmetric NVM without erase cost. Policy validated using the available hardware as a proof of concept (Flash, negating erase cost).

Table 3
Comparison of proposed policies with regard to NVM properties (symmetricity, erase cost).

Property / Policy	Allocated State	MASI	Updated Blocks	MUB
Symmetric				
No erase cost	✓			
Erase cost		✓		
Asymmetric				
No erase cost	✓		✓	
Erase cost		✓		✓

6.4. *Summary*

Table 3 summarises the different selective policies and their applicability to different NVM technologies. These have been extracted from the experimental results, considering the properties of the most important parameters of NVM technologies (symmetricity, erasing). For symmetric NVM technologies, only the *Allocated State* (for memories without erase cost) and *Multiple Allocated State Images* (for memories with erase cost) policies should be considered, as the overhead for comparing the VM with the NVM incurred by the other policies is prohibitive. For systems featuring an asymmetric NVM, the way the main memory is used by the application significantly affects the performance of each policy. Table 4 summarises the most suitable policies depending on the memory usage. When the entire allocated memory is updated, the *Allocated State* (asymmetric NVM without erase cost) and *Multiple Allocated State Images* (asymmetric NVM with erase cost) policies offer better energy efficiency, as the cost for comparing VM with its corresponding NVM blocks is eliminated. However, when fewer memory cells have been updated since the last power failure, the *Updated Blocks* (asymmetric NVM without erase cost) and *Multiple Updated Blocks* (asymmetric NVM with erase cost) policies perform better compared to the other policies.

7. Conclusions

In this paper, we have shown the inefficiency of current state retention policies in transiently-powered embedded systems when used on NVM technologies with certain properties. We presented novel selective policies for efficient state retention in order to identify the most energy efficient policy/platform combination. These software-based retention policies are targeted for different NVM technologies, exploiting their properties such as read/write symmetricity and the need for erasing. Unlike existing approaches, the proposed policies are based on the principles of (1) saving only the information that has changed since the last restore (*Updated Blocks* and *Multiple Updated Blocks*), and (2) avoiding the cost of erasing NVM by concatenating multiple images (*Multiple Allocated State Images* and *Multiple Updated Blocks*). The

Table 4
Comparison of proposed policies with regard to memory usage (updates, allocation).

Memory usage	Low allocation	High allocation
Few updates	Upd. Blocks MUB	Upd. Blocks MUB
Entire update	Alloc. State MASI	Alloc. State MASI

existing and proposed policies were experimentally validated on two different and appropriate platforms (featuring Flash and FRAM memory). A comparison between different policies has been made, considering the effect of how the application is using memory. From this, we determine the most energy-efficient policy, based on the characteristics of the NVM technology of the system. Results show that using the appropriate policy/platform combination provides a reduction in the energy overhead of up to 86.2% for FRAM (using the *Allocated State* policy, Fig. 14) and 90.6% for Flash memory (using the *Multiple Updated Blocks* policy, Fig. 18a), compared to the typical approach of saving the entire system state.

Acknowledgements

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under Platform Grant EP/P010164/1 and the PRiME Programme Grant EP/K034448/1. Experimental data used in this paper can be found at doi:10.5258/SOTON/D0590 (https://doi.org/10.5258/SOTON/D0590).

References

[1] J. Olivo, S. Carrara, G. De, Micheli, Energy harvesting and remote powering for implantable biosensors, IEEE Sens. J. 11 (7) (2011) 1573–1586.
[2] P.D. Mitcheson, Energy harvesting for human wearable and implantable bio-sensors, 2010 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), IEEE, 2010, pp. 3432–3436.
[3] S. Kahrobaee, M.C. Vuran, Vibration energy harvesting for wireless underground sensor networks, 2013 IEEE International Conference on Communications (ICC), IEEE, 2013, pp. 1543–1548.
[4] D. Balsamo, A. Das, A.S. Weddell, D. Brunelli, B.M. Al-Hashimi, G.V. Merrett, L. Benini, Graceful performance modulation for power-neutral transient computing systems, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 35 (5) (2016) 738–749.
[5] S. Beeby, N. White, Energy Harvesting for Autonomous Systems, Artech House, 2010.
[6] F. Ongaro, S. Saggini, P. Mattavelli, Li-ion battery-supercapacitor hybrid storage system for a long lifetime, photovoltaic-based wireless sensor network, IEEE Trans. Power Electron. 27 (9) (2012) 3944–3952.
[7] A. Kansal, J. Hsu, S. Zahedi, M.B. Srivastava, Power management in energy harvesting sensor networks, ACM Trans. Embed. Comput. Syst. 6 (4) (2007) 32.
[8] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, V. Narayanan, Architecture exploration for ambient energy harvesting nonvolatile processors, 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2015, pp. 526–537.
[9] B. Ransford, J. Sorber, K. Fu, Mementos: system support for long-running computation on RFID-scale devices, ASPLOS'11, (2011).
[10] H. Jayakumar, A. Raha, V. Raghunathan, Quickrecall: a low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers, 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems (2014).
[11] D. Balsamo, A.S. Weddell, G.V. Merrett, B.M. Al-Hashimi, D. Brunelli, L. Benini, Hibernus: sustaining computation during intermittent supply for energy-harvesting systems, IEEE Embed. Syst. Lett. (2015).
[12] D. Balsamo, A.S. Weddell, A. Das, A.R. Arreola, D. Brunelli, B.M. Al-Hashimi, G.V. Merrett, L. Benini, Hibernus+ +: a self-calibrating and adaptive system for transiently-powered embedded devices, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 35 (12) (2016) 1968–1980.
[13] Y. Liu, J. Yue, H. Li, Q. Zhao, M. Zhao, J. Xue, G. Sun, M.-F. Chang, H. Yang, Data backup optimization for nonvolatile SRAM in energy harvesting sensor nodes, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 36 (Oct. (10)) (2017) 1660–1673.
[14] N.A. Bhatti, L. Mottola, Efficient state retention for transiently-powered embedded sensing, EWSN (2016) 137–148.
[15] A. Rodriguez, D. Balsamo, A. Das, A.S. Weddell, D. Brunelli, B. Al-Hashimi, G.V. Merrett, Approaches to transient computing for energy harvesting systems: a quantitative evaluation, ENSsys 2015 (2015).
[16] P. Pavan, R. Bez, P. Olivo, E. Zanoni, Flash memory cells – an overview, Proc. IEEE 85 (8) (1997) 1248–1271.
[17] Texas Instruments, FRAM – New Generation of Non-Volatile Memory, (2009).
[18] J.S. Meena, S.M. Sze, U. Chand, T.-Y. Tseng, Overview of emerging nonvolatile memory technologies, Nanoscale Res. Lett. 9 (1) (2014) 526.
[19] B.C. Lee, E. Ipek, O. Mutlu, D. Burger, Architecting phase change memory as a scalable dram alternative, ACM SIGARCH Computer Architecture News, vol. 37, ACM, 2009, pp. 2–13.
[20] A.J. Bhavnagarwala, X. Tang, J.D. Meindl, The impact of intrinsic device fluctuations on CMOS SRAM cell stability, IEEE J. Solid State Circuits 36 (4) (2001) 658–665.
[21] Texas Instruments, MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller, (2011) Rev. G.
[22] Texas Instruments, MSP430FR573x Mixed-Signal Microcontrollers, (2016) Rev. K.
[23] NXP Semiconductors, LPC81x User manual, (2014) Rev. 1.6.