Journal of Systems Architecture 115 (2021) 102013

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

A survey on program-state retention for transiently-powered systems

Saad Ahmed [a,*], Naveed Anwar Bhatti [c], Martina Brachmann [b], Muhammad Hamad Alizai [a]

[a] LUMS, Pakistan
[b] RISE Research Institutes of Sweden, Sweden
[c] Air University, Pakistan

ARTICLE INFO

Keywords:
Transiently-powered computers
Intermittent computing
Checkpointing
Program-state retention

ABSTRACT

Low-power small-scale embedded sensing systems employing batteries generally impose high maintenance costs. To enable maintenance-free operation, they are powered from energy harvested from the environment thus making them batteryless. However, due to high variance of ambient energy, these batteryless embedded devices are unable to harvest enough energy from the environment required for continuous device operation thus hampering application progress and causing frequent loss of volatile program-state. Therefore, these batteryless devices have to employ state retention mechanisms to save the volatile program-state to non-volatile storage before interruption. These batteryless embedded sensing devices are known as *transiently-powered systems* (TPS). In this article, we survey existing literature to identify strategies and techniques used by each existing literature to decide *what* amount of volatile program-state needs to be saved and *when* to save it. We list the challenges in retaining program-state across periods of energy unavailability and how existing state-of-the-art solutions tackle them. We also describe different memory models and discuss factors governing the choice of each model for TPS deployment.

1. Introduction

In recent years, the vision of "smart dust" [1] has driven the development of tiny, embedded sensing devices that run autonomously for decades. Equipped with programmable microcontroller units (MCUs), these sensing devices are envisioned to be used in large-scale applications such as wearables [2], implants [2], small satellites [3], and wireless robotic materials [4,5]. The monitoring of existing infrastructure [6] and the environment using insects carrying the tiny devices has also been recently proposed [7,8].

To enable these applications, the tiny sensing devices require a continuous and perpetual energy source. Batteries are commonly used to power such embedded devices [9,10]. However, recent advances in nanotechnology and microelectronics have made it possible for these devices to shrink in size [11]; small enough to fit on the head of a honey bee [12]. Battery designs have not scaled in the same way, thus making it hard for them to fit in such devices despite recent developments [13].

Furthermore, the use of batteries nullifies the initial vision of "smart dust" of being invisible and maintenance-free. Modern-day batteries have only limited number of power cycles, i.e., batteries have one power cycle and rechargeable batteries have a few thousand. As a result, batteries from a high number of devices have to be frequently replaced, preventing a perpetual operation of the devices and resulting in high maintenance costs.

One solution to enable the maintenance-free operation of these devices is to liberate them from batteries by powering them from harvested energy [14–17], e.g., from light, vibrations, and thermal sources. Miniaturized mechanical systems have enabled the design of tiny energy harvesters at the scale of nanometers that fit into the form factor of these tiny, embedded sensing devices [17], thus making them batteryless. However, the smaller an energy harvesting unit is, the smaller is the harvested energy. Furthermore, the energy that can be harvested from the environment is generally erratic and exhibits high spatial and temporal variations [17].

To cope with the small amounts of harvested energy and to smooth the fluctuations of the feeble incoming energy, devices powered by harvested energy employ an energy buffer. This is typically a capacitor offering unlimited power cycles as opposed to batteries. The form factor of this buffer must be small enough to not only fit in the device footprint but to also allow fast recharge, thus limiting its storage capacity (typically $<100\mu$F). As a result, one power cycle of this buffer cannot supply enough energy needed to complete most of the programs running on MCU.

* Corresponding author.
E-mail addresses: saad.ahmed@lums.edu.pk (S. Ahmed), naveed.bhatti@mail.au.edu.pk (N.A. Bhatti), martina.brachmann@ri.se (M. Brachmann), hamad.alizai@lums.edu.pk (M.H. Alizai).

https://doi.org/10.1016/j.sysarc.2021.102013
Received 20 January 2020; Received in revised form 30 November 2020; Accepted 6 January 2021
Available online 11 January 2021

Intermittent Program Execution. Intuitively, the notion of energy is transformed from "limited but continuous" in traditional battery-powered devices to "unrestricted but intermittent" in these maintenance-free devices. We call these tiny, energy harvesting-based, batteryless, embedded devices *transiently powered systems (TPSs)* and refer to the computing paradigm as *intermittent computing* A TPS is active when the energy buffer is full and it is inactive, i.e., it turns off, when the energy buffer is empty and is recharging. Program execution can only occur during active periods and the device loses its volatile state when the energy falls below the minimum energy that is required by the TPS to operate.

The TPS state comprises three sub-states, i.e., program-, peripheral- and timer-state, which have to be saved onto non-volatile memory (NVM) shortly before the device becomes inactive. Forward progress of the application running on the TPS can only be ensured by retaining all three sub-states at the same time. When the TPS switches back to an active period, it restores the saved system state and continues execution. However, retaining each sub-state has its challenges that are orthogonal to each other.

- The program execution can be interrupted at any time during the operation of a TPS. At the time of interruption, size of volatile program-state is different for different applications and depends on the program-point at which energy failure occurs. Therefore, it requires sophisticated techniques to determine volatile program-state that needs to be retained across periods of energy unavailability.
- Peripheral devices require reconfiguration after each reboot. Simply capturing the current state of these peripheral devices can result in incorrect program execution, thus giving rise to new challenges.
- TPSs collect time-sensitive data and require time stamping of the collected data. However, transient energy supply can cause stale values to be processed by the application, thus producing incorrect results.

We discuss these challenges in detail in Section 2.1. Due to their unique set of requirements, existing literature devise different strategies to address each of the above-mentioned challenges [18–26].

Scope of the Survey. To keep the discussion focused, we survey existing literature to find solutions that address challenges associated with program-state retention, thus enabling energy-efficient TPS operations. This research space has seen rapid and significant progress over the past few years [18–22,27,28,28–37]. However, a classification and a common taxonomy in the realm of program-state retention solutions is still missing.

To this end, this paper encapsulates existing state-of-the-art solutions that answer the following research question: *How to ensure the successful execution of a program on a TPS under intermittent energy supply?* We build a taxonomy of solutions for program-state retention based on how they ensure forward progress of the programs running on TPSs. More precisely, this paper distinguishes between two key aspects of program-state retention. First, we discuss literature proposing strategies for reducing the energy required to retain the program-state before the TPS switches to an inactive period. Second, we focus on techniques that, based on compile-time analysis, identify program points where saving the program-state would require minimal energy.

Our Contribution. There exist well-organized surveys on energy harvesting and wireless energy transfer techniques within the sensor network domain [17,38,39]. There are also detailed surveys available for large-scale distributed systems regarding various checkpointing techniques [40,41]. In contrast, we present a survey on TPSs, highlighting different checkpointing techniques and the underlying memory technologies employed by these tiny, batteryless, embedded sensing devices that have entirely different constraints than the devices used in the distributed and parallel computing domain.

A recent article surveyed several intermittent computing approaches [42] to briefly define challenges and future research directions in the TPS domain. In contrast, this paper gives a detailed survey of all existing state-of-the-art intermittent computing approaches and draws on an up-to-date taxonomy of program-state retention for TPSs. This paper covers all existing techniques, which answer the above-mentioned research question while discussing the trade-offs of different memory models on efficient program-state retention.

Organization. The rest of the paper is structured as follows. Section 2 gives the bigger picture of TPS operation while defining basic terminology and challenges faced by TPSs in retaining overall system state. Section 3 explains the criteria of classification for existing solutions for retaining program-state for TPSs. Sections 5 and 6 classify the strategies employed by different intermittent computing solutions to ensure energy efficiency while performing program-state retention. Section 4 discusses different memory models and their trade-offs before we conclude the paper in Section 7.

2. TPS dynamics: A bigger picture

To understand the fundamental TPS operation, we consider a simple *sense-compute-send* application widely used in these maintenance-free devices e.g., fitness trackers [43]. The device senses value from the sensor (peripheral device) and perform computations on it (MCU) before sending the processed values to the user.

2.1. TPS fundamentals

Due to variable energy supply, TPSs switch between active and inactive periods many times during their entire life. Each inactive period is followed by an active period forming an energy cycle of these TPSs.

TPS Energy Cycle. Fig. 1 shows a typical energy cycle of these TPSs. The energy cycle starts when a TPS is switched off and is harvesting energy from the environment to charge its buffer. As the capacitor gets fully charged, the voltage level for the capacitor reaches V_{on}– voltage at which the MCU can start executing program instructions– and the TPS is turned on indicating the start of an active period.

Sense: As application execution starts, it accesses a sensor to collect environmental information. After reading the sensor value, the data is usually time-stamped to ensure its freshness and accuracy of computations performed in the later stages.

Compute: Since sending raw data over the radio consumes a lot of energy, learning algorithms are deployed to extract meaningful information. This involves performing complex computations which consume both time and energy.

Send: After performing application-specific analytics, TPS sends the final result over the network to either the base station or the cloud server using anyone of the heterogeneous network protocols available. Successful data delivery marks completion of one application iteration.

These operations must be performed in a continuous manner in order to give correct results. However, with stringent computational capability and increasing complexity of applications, the energy buffer is exhausted faster than it could be replenished [44,45]. As a result, *sense-compute-send* operation suffer frequent interruptions which causes loss of volatile data while wasting precious energy spent on performing repeated computations. As a result, a single iteration of the application can take from few hundreds to tens of thousands of energy cycles to complete one application iteration depending upon the complexity of application, choice of algorithm, and size of the energy buffer.

To ensure forward progress of the application, TPS needs to save its volatile data onto NVM before energy buffer depletion. Therefore, it has to stop performing computations before it reaches a threshold voltage V_{th} where the energy left in the buffer is only sufficient for the device to successfully save the state onto NVM. As the volatile state is saved, the capacitor voltage reaches V_{off} and the device is turned off

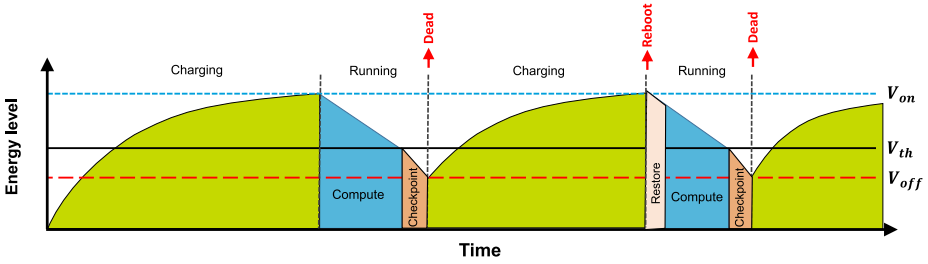

Fig. 1. Intermittent execution. *An intermittently-powered device executes its program in bursts as energy is available. The slope of the energy level follow standard capacitor charge/discharge pattern, however, when the system is saving the state its slope gets slightly steeper compared to running the application, as writing to the NVM is generally the most energy expensive operation.*

marking the end of active period (start of inactive period). While the device is turned off, it harvests energy from the environment to charge its energy buffer and it is turned on whenever the energy buffer gets fully charged.

Challenges. Various techniques have been proposed in main-stream computing to efficiently retain system state across failures [46,47]. However, these techniques assume continuous energy supply which makes them inapplicable in intermittent computing domain. Frequent power interruptions during the *sense-compute-send* operation give rise to a unique set of challenges required to be addressed for successful state retention of a TPS.

- *Peripheral state retention:* TPSs can lose power in the middle of a *sensing* operation e.g., during data acquisition from sensors or data transmission. Peripheral devices require reconfiguration after each power failure and a simple snapshot of peripheral state can result in either program liveness or safety issue [25]. MCUs either need to maintain an operational log [48] or require a device context to be saved in NVM to ensure peripheral state retention [49]. However, this still does not guarantee correct program execution as peripheral devices can also operate asynchronously [48,49].
- *Program State Retention:* While performing *computations*, TPSs require integrated system support to retain the state of the program running on the MCU. This system support helps these TPSs decide, *when* and *what* amount of program-state is required to be saved for successful *program-state retention*. However, deciding *when* and *what* portion of program state to be saved is an ongoing research problem [19–21,29] involving either modified compilation techniques or requiring programmer's effort.
- *Persistent Time Keeping:* In order to *send* timestamped data, keeping the persistent notion of time on these TPSs is a challenging task as the traditional timekeeping options, i.e., real-time clocks (RTCs), stop working at each power blackout. The time interval between two sensed values can be of seconds, minutes or even hours, if an energy interruption occurs between them [50]. This can render a sensed value stale and their is no way to know the freshness of the sensor reading. This makes applications requiring real-time or monotonic-time stamps impossible to operate on TPS.

  TPSs usually perform tasks in a synchronized manner in the network. With incorrect timing information, these devices can get unsynchronized with other devices after each inactive period. Thus, data transmission and reception in an unsynchronized, duty-cycled network becomes challenging as both senders and receivers spend much of their time synchronizing themselves. There are solutions proposing languages and runtime to ensure persistent time across reboots, however, it is still an active research area [34,51].

Fig. 2 categorizes the challenges in retaining the TPS's state. Catering all these challenges ensure correct execution of programs on TPS. However, to facilitate a detailed discussion, this paper focuses only on classifying different software-based strategies for program-state retention.

2.2. Program-state retention

The program-state in TPS comprise all global variables, system stack, dynamic data structures, general- and special-purpose registers (GPRs and SPRs). Global variables are placed in `.data`, `.bss` segments of the main memory whereas all dynamic data structures are placed on the `heap` segment. All function call frames are pushed to the `stack` memory segment. Since the main memory is volatile, MCU has to save its current program-state onto NVM just before switching to the inactive period so that it can be revived at the next active period. *Checkpoint* of a TPS comprise of program-state residing in volatile memory which can vary depending on the underlying memory model. In case of volatile systems, for example, MCU registers, `.data`, `.bss`, `stack` and `heap` memory segments reside in the volatile memory. A checkpoint represents the program-state running on TPSs at any point in time and we call, the process of saving the current program-state onto NVM (the checkpoint), as *Checkpointing*. All memory segments contain different parts of the program-state and missing anyone will result in a faulty checkpoint. This checkpointed state is *restored* at the start of the next active period and requires a significant amount of energy due to expensive NVM write operations as shown in Fig. 1.

Checkpointing must be performed during the active period to ensure its correctness. Therefore, it must be triggered at a particular voltage V_{th}–the threshold voltage indicating energy that is only sufficient for the checkpointing process — and no further program execution can be done afterwards. Any execution of a program beyond this voltage level will result in an inconsistent and faulty checkpointed program-state.

The value of V_{th} is directly proportional to the capacitance of the capacitor, the size of the checkpoint, as well as the memory technology used for NVM. Smaller capacitor have to set a higher V_{th} while larger capacitor can set V_{th} close to V_{off}, as shown in 3. Similarly, larger the program-state, more will be the energy that is required to finish checkpointing before V_{off}. Flash/EEPROM is the typical NVM employed by these TPSs because of their fast read operations, while write operations are computationally expensive and energy hungry as they have to erase the content of EEPROM before writing new data [52].

Recently, FRAM is also being employed as main memory [35] as well as a secondary memory in these TPSs [19,20]. However, employing FRAM as main/secondary memory increases the overall energy consumption of the system [53]. We discuss the future of FRAM as main or secondary memory in detail in Section 4.

With promising application scenarios of these embedded sensing devices, there is an extensive amount of existing literature proposing various solutions to address the challenges faced by TPSs in retaining program-state. In the next section, we define the rationale for classifying each solution in the domain of intermittent computing.

3. Intermittent program's state retention: Taxonomy of solutions

In order to retain program-state across reboots, TPSs have to spend a major chunk of the energy buffer on saving the state onto NVM and reviving it; leaving a very small amount for program execution. Therefore, TPSs aim to maximize the energy spent on program execution

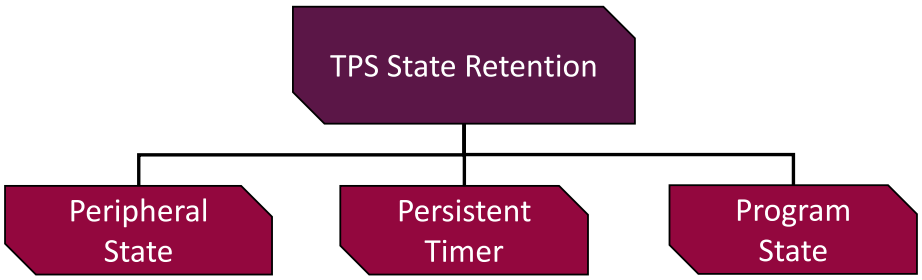

Fig. 2. System State Challenges for TPS.

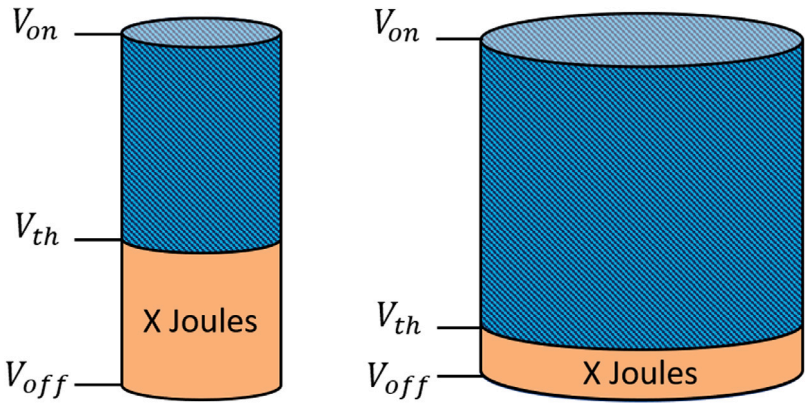

Fig. 3. Capacitance vs. Voltage Threshold (V_{th}). *Assuming checkpointing process consumes X joules of energy then smaller capacitor will need to set higher V_{th} as compared to larger capacitor to supply required amount of energy.*

while ensuring forward progress of the application. However, these two goals are in tension with each other.

Energy needed to perform checkpointing depends on the underlying memory model of TPS. There are three prevalent memory models in TPSs; volatile, non-volatile, and mixed volatility systems. On one hand, executing program using a volatile main memory allows faster access to the memory and consumes lesser energy. On the other hand, it requires program-state to be checkpointed onto NVM which is an energy hungry operation thus demanding an energy efficient mapping of program-state in order to achieve the optimal energy consumption.

Larger the amount of energy reserved, lesser is the energy available for the device to run program. A naive way of *checkpointing* program-state is to save the entire main-memory onto NVM. The problem is that the threshold voltage would have to be set quite high and the frequency of interruptions will, in turn, be increased. With the increasing size of volatile memory in TPS [53], this approach is only going to increase the checkpointing cost. Therefore, a naive solution is neither energy-efficient nor scalable considering the tight energy budget of TPSs. This demands smart strategies to maximize the energy available for program execution by reducing the energy spent on checkpointing.

To trigger checkpointing, TPSs need to constantly poll the energy buffer to detect V_{th} thus consuming additional energy. One can easily set a pessimistic value of V_{th} to avoid polling. However, this only wastes the available energy that could have been used to make progress over program execution. Thus, these devices have to strike a trade-off between high value of V_{th} versus frequent probing of energy buffer in order to conserve energy.

Based on these challenges, existing state-of-the-art solutions can be classified into three-layer topology each with its own taxonomy, as shown in Fig. 4. First layer discusses pros and cons of different memory models which can be used with TPS. Second layer deals with strategies proposed for balancing the trade-off between high energy buffer probing rate versus high value of V_{th}. Third discusses the strategies proposed by existing literature to reduce the checkpoint size for these devices and then classifies them.

Memory Model. Traditional memory models use volatile main memory (RAM) that enables energy-efficient access and retrieval of data. However, it increases the volatile state of the program thus increasing the energy required for checkpointing; demanding an efficient mechanism to avoid overshadowing of the energy savings during program execution. Recently, a new idea is emerging which employs non-volatile main memories (NVRAM) to gain persistence in case of power failures. This approach significantly reduces the checkpoint size as the device only needs to save its internal registers. However, the energy consumption of these NVRAMs is around 10x higher than the RAM and much of the energy which is saved by reducing checkpoint size is now spent on additional NVM energy consumption [53]. Researchers have also proposed a middle ground by employing mixed-volatile main-memory, where the frequently accessed data is stored in volatile RAM while the rest is placed in NVRAM. This bring frequent accesses at a cheaper cost, however, NVRAM have to suffer from idempotence violations which can occur due to frequent interruptions [24]. We discuss this in detail in Section 5.

Triggering Strategy. While it is essential to keep the checkpoint size small, it is equally important to trigger the checkpointing process at the right time in order to successfully complete the process. Ideally, it should be triggered at the very last moment when energy in the buffer is just enough to perform a successful checkpoint. This will result in the most efficient utilization of energy buffer by TPSs. However, following are the challenges faced by these systems in achieving energy-efficiency.

- Checkpoint size changes because of updates in the global variable segment and variation in stack size during code execution. Therefore, the energy required to successfully complete the checkpointing procedure also changes. This changes the code point at which triggering a checkpoint would result in minimum energy consumption.
- Energy availability in the environment is unknown in the program at the time of triggering decision. Therefore, the system can end

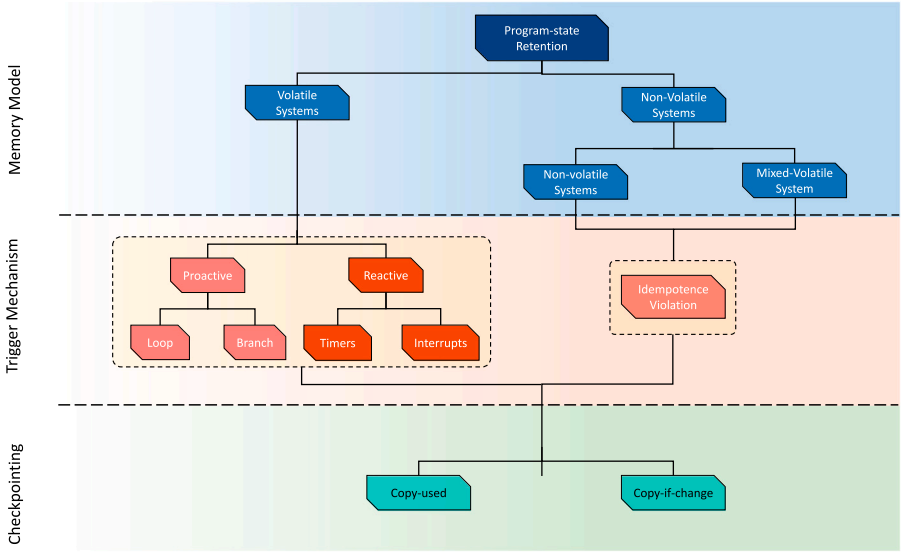

Fig. 4. Three-layered topological taxonomy of program-state retention solutions for TPS.

up performing unnecessary checkpoints, even in a case when the energy available in the environment was sufficient to complete the execution of a program.

Checkpointing Strategy. A naive way of checkpointing is not energy-efficient because of following reasons:

- A major portion of RAM contains non-valid/un-used memory locations. Therefore, saving the entire RAM as part of checkpoint wastes energy by copying the unused memory location onto NVM.
- Saving the entire main-memory is not always required. Computing what has changed from the previous checkpoint can create an up-to-date checkpoint without copying unmodified memory locations thus saving energy.

Therefore, TPSs require specialized methods to enable energy conservation at the time of checkpointing. It is important here to note that the focus of this taxonomy are the techniques that enable correct restoration of program-state across reboots. There are systems which use task-based semantics thus minimizing the need to perform checkpointing [22,33,54,55]. We will discuss them in the upcoming sections but they are not part of this taxonomy.

Based on these challenges, the plethora of solutions proposing techniques to cater the above mentioned challenges for all three layers of the TPS taxonomy; Memory Models, Triggering Mechanisms and Checkpointing solutions. Fig. 4 represents our taxonomy for the proposed solutions in the existing literature.

4. TPS memory models

The transition from battery-powered to batteryless computing has brought many changes in the design, architecture and memory models of embedded computers which were previously taken for granted. In this section, we describe the modifications in memory models and discuss the trade-offs in employing each one of them.

The choice of TPS architecture to be deployed in any real-world environment is driven by the type of application and ambient energy available in the environment. Program execution and checkpointing are the two main sources of energy consumption during the life-cycle of each TPS. Therefore, goal of each TPS solution is to minimize the cost of checkpointing thus maximizing the time the device spends on program execution. However, these two goals are orthogonal to each other. Balancing these goals not only increases the system's life but it also increases throughput of the TPS's application.

Reduction in execution energy requires TPSs to achieve energy-efficient execution of program. Traditional memory models use volatile main memory in these TPSs which allows energy-efficient access and retrieval of data. However, it increases the volatile state of the program and the energy saved during program execution is spent at the time of checkpoint. Therefore, researchers have proposed different hardware modifications in the traditional memory models for TPSs using both volatile RAM and NVRAM. We next discuss common memory models employed in the intermittent computing domain.

4.1. Memory models

There are three main categories of memory models prevalent in the TPS's domain, as shown in Fig. 4. In the first category, a TPS is formed by employing a volatile main memory (RAM) and non-volatile external storage typically based on widely-used flash technology along with integrated energy harvesting methodology; thus making a ***volatile system***. The use of RAM allows these systems to execute program at higher CPU frequencies [42], while consuming lower energy per cycle. However, efficient energy consumption comes at the cost of the increased size of the volatile system state. Such systems need to save a snapshot of system state onto flash before power blackout; an expensive operation energy-wise as flash requires large sectors of data to be erased before writing. In recent years, however, integrated circuit manufacturers have been considering non-volatile main memory (NVRAM) as a strong contender for embedded and non-volatile storage, allowing read/write speed as that of volatile main memory. With this aim, many memory technologies have been proposed in the literature, e.g., MRAM [56], PRAM [57], FRAM [58] etc., which allow persistent program state across reboots. The second category of TPS platforms, ***non-volatile systems***, leverage persistence of NVRAM and employ it as main memory thus trading-off higher checkpointing energy and with increased cost of program execution [32,35]. The final category aims to combine the best of both worlds [27] by combining volatile and non-volatile RAM in single chip. This allows a TPS to dynamically configure itself to the energy-efficient program mapping at run-time. However, such systems still need to save CPU state and deal with idempotence violations that may arise by re-execution of code. Following sections discuss these systems in detail.

Table 1
Micro-measurements for non-volatile and volatile systems: Here non-volatile system uses an FRAM whereas Volatile Systems employ SRAM as main memory.

Memory model	Frequency (MHz)	Access time (μs)	Current consumption (mA)	Cycle energy (nJ)
Non-volatile system	16	0.125	1.77	0.66
	8	0.125	1.12	0.42
Volatile system	16	0.0625	1.5	0.28
	8	0.125	0.89	0.33

4.2. Volatile systems

The first architecture of TPS (Fig. 4), consisting of volatile RAM, is the main choice for each application designer when it comes to reducing the execution energy. It has 2x lesser energy consumption and can work on higher MCU frequency values than non-volatile main memories [42]. This allows the programmer to execute the program at much higher MCU frequencies and with low current consumption compared to NVRAM. However, RAM is volatile in nature and loses Its content at the time of power loss. As a result, these systems have to take frequent checkpoints (RAM state + CPU registers) to keep making progress on the task. Larger the size of a checkpoint, higher is the energy required to save it on the NVM.

Flash is the most commonly used NVM used by the system deploying volatile RAM as the main memory. Its memory is divided into sectors which are further divided into pages. Each write operation requires the memory sector to be erased before writing, which further increases the energy cost of a checkpoint. Therefore, checkpointing makes a significant part of the energy consumed by the device.

4.3. Non-volatile systems

Fully non-volatile Systems. Recent research efforts propose the use of NVRAM; the third category of TPC architecture. It eradicates the need to save program-state residing in main memory allowing the device to spend entire energy on program execution and saving CPU state, i.e., internal registers . However, access latency and current consumption for NVRAM are still higher when compared with RAM e.g., for FRAM used in MSP430FR5969, access latency is 3x and current consumption is almost 2x [53] more than that for SRAM. This implies a higher per-cycle energy consumption for the non-volatile system making them energy-inefficient(see Table 1). Not only that, with persistent main memory, re-execution of the same code can cause data inconsistency issues that are absent in continuous execution. As a result, such systems require special system and language support to avoid inconsistency [22,33,54]. However, catering such scenarios adds an additional energy overhead on the energy buffer. Therefore, systems using NVRAM have to adopt solutions that require lesser energy to avoid data inconsistency issues than what RAM based systems require for checkpointing, to be feasible for deployment. Otherwise, it is better to use volatile RAM due to its energy-efficient program execution and support for higher MCU frequencies.

Mixed-Volatile Systems. To get the best of both worlds, a smart solution is to employ both these memories i.e., unified main memory containing both NVRAM and volatile RAM; the third category of TPS architecture. With important data residing in NVRAM, mixed-volatility systems significantly reduce the checkpoint size.

Jayakumar et al. [27] proposes a hybrid approach to find the optimal mapping of code section in either NVRAM or volatile RAM. The proposed approach works at the granularity of function and maps each function to either on of the memory and, based on where the optimal energy consumption would be achieved. This allows the proposed approach to get benefited from NVRAM's reliability as well as RAM's energy efficiency during code execution. The approach, however, has a trade-off between data transfer cost and execution cost of code.

Each code section has to be transferred to the memory region it is mapped. Larger the code section to be transferred, higher is the energy cost of migration. Furthermore, frequently accessed data that is placed on volatile RAM for fast retrieval has to be checkpointed at the time of energy failure or otherwise it has to be placed in NVRAM thus bearing a high energy cost. Therefore, solutions employing mixed-volatility systems have to make an informed decision about data placement to make efficient utilization of the energy budget.

5. Trigger mechanism

In the TPSs' domain, there are two ways of deciding whether to trigger checkpointing or not, as shown in Fig. 4. We call these triggering mechanisms as Proactive and Reactive checkpointing.

Proactive. TPS inserts special function calls in the program which probes the energy buffer to decide whether it is the right time to checkpoint or not; We call these function calls as trigger calls. Note that energy failure can occur before or after these trigger calls thereby wasting the computations performed from the last checkpoint.

Reactive. To overcome wasted computations/energy in proactive approach, reactive mechanisms employ strategies to trigger checkpointing only when the voltage reaches V_{th}. They either employ hardware or software-based techniques to trigger checkpointing at V_{th}. This allows these strategies to overcome wasted computations/energy of a proactive approach.

5.1. Proactive

Different solutions propose different candidate code points for placing these trigger calls in the TPS's program. We can, therefore, categorize proactive triggering mechanism in the following three subcategories based on the choice of the code point, as shown in Fig. 4.

Loops. A major chunk of computation in any program is performed in loops. Therefore, loops are an important place to insert trigger calls.

Ransford et al. [18] statically place trigger calls at the end of loop iterations. This is a location where a program can checkpoint the computations performed in a single iteration of loop execution. This can save a TPS from re-executing the same iteration again in the next active period. However, the computation performed in the last iteration is always lost. Furthermore, this approach does not ensure correct execution when the code is unable to reach the first trigger call on the given energy budget. This issue was addressed by HarvOS [21] by placing two trigger calls; one inside the loop to decide if energy to execute next iteration is available or not; second at the end of the loop to check if there is sufficient energy to go to the next trigger call or not. HarvOS needs to check the energy buffer at the second trigger call as the decision to reach this trigger call was done inside the loop. Since the loop exited, more energy would have been spent than the last trigger call expected to make it necessary to read the energy buffer.

Branch. Since TPS's MCU is unable to speculate the next instruction after branch, it cannot give surety whether it would successfully reach the trigger call placed at some other code point in the program. Therefore, branch instructions are an important code point to make checkpoint triggering decision.

Bhatti et al. [21] tackle this issue by placing trigger calls after each branching construct. If there exists a point inside the scope of the branching construct where the stack size is minimum, an additional trigger call is placed at that point. This rule forces the solution to probe energy buffer to find an exact value of remaining energy as it

is unaware of the energy cost of the basic block(s) executed in that construct. Function-return statements are special branch instructions where one may expect the stack to store less data, which would then reduce the size and energy cost of saving system state to NVM. Therefore, it is also a favorable option to place a trigger call.

Based on this motivation, Ransford et al. [18] place trigger calls after each function return. However, as discussed earlier, this approach does not ensure correct execution when the code is unable to reach the first trigger call on the given energy budget.

To ensure correctness, Bhatti et al. [21] assume every function as an in-line function. The authors estimate the maximum number of cycles (C_{use}), that are available for performing computations, by analyzing the memory allocation pattern of the program to find maximum checkpoint size (worst case memory allocation). It, then, places two trigger calls in the code; the first trigger call is inserted at the point which is at most $\frac{C_{use}}{2}$ cycles away from the previous trigger call; second trigger call is placed after each function return. Hardware and software interrupt is another way of branching and adds another dimension in challenges as they can come anytime (non-deterministic interrupts) and their execution length is unknown.

To handle interrupts, Bhatti et al. [21] treat them in the same way as function calls and additionally place two trigger calls. The first trigger call is placed right at the very first and the secondone is placed at the very last instruction. This is required as ISR has no information on where the execution code was interrupted so it has to read the energy buffer to check the remaining energy. The second trigger is placed in the caller's code and helps deal with the corner case where the remaining energy was exactly equal to the next trigger call (in the callee's code).

There are solutions [22,33,54] which provide APIs to the programmer to write the program as a set of atomic tasks. Each task can be considered as a function definition. As code execution crosses the function boundary, triggering decision is made. In this way, triggering mechanism is integrated with language semantic thus allowing the programmer to know the location of checkpoint in program execution. Ideally, the distance between two trigger calls must be strictly less than the energy budget of the device. This can only be ensured through accurate analysis of the application code which demands accurate modeling of the TPS energy consumption [30,36]. This is an open research area and discussing such solutions and strategies is out of the scope of this survey.

Idempotence Violation. Each write-after-read (WAR) dependency between two program instructions causes data inconsistencies with the modern memory model of TPSs [53]. These instructions must execute within same energy cycle in order to avoid data corruption. For this purpose, Mathew Hicks [35] proposes to separate such instructions with trigger calls. These trigger calls act as checkpoint calls as they do not poll the energy buffer to perform checkpointing. However, such an approach can end up placing excessive trigger calls in the same idempotent code section. As a result, it suffers from wasted computation as it is unable to know if the execution will reach the next trigger call or not.

5.1.1. Discussion

As discussed earlier, the proactive checkpointing technique is prone to energy wastage as computations performed after the last checkpoint till energy failure do not become part of any checkpoint. These computations are performed again in the next active cycles.

One way to address this issue is to increase the number of trigger calls placed in the program. This will decrease the number of program instructions between two consecutive trigger calls thereby decreasing the wasted computations. However, such an approach will not only increase the program size but it will also increase the runtime overhead. TPSs have limited memory and energy budget to execute the program. With such an approach, it will make it more difficult to execute programs on these resource-constrained devices. This demands each proactive approach to strike a tradeoff between the runtime overhead and the wasted computations to ensure efficient utilization of the given energy buffer.

5.2. Reactive

The reactive triggering mechanism can be divided into two subcategories, as shown in Fig. 4. The first category uses additional hardware to decide whether to make the triggering decision. At the time of checkpoint, the TPS generates a hardware interrupt to checkpoint system state onto NVM. This category of triggering mechanism consumes additional energy as it employs hardware support. To mitigate this additional cost, the second category focuses on software timers to trigger checkpointing.

Interrupts. Hardware interrupts are an easy way to trigger checkpointing eradicating the need to poll the energy buffer at each trigger call [19,20,32]. Existing state-of-the-art solutions modify the design of a TPS to integrate voltage comparators. As soon as the current goes below the threshold voltage V_{th}, an interrupt is generated and the current device state is checkpoint onto NVM. The value for V_{th} is set at the time of TPS deployment and is estimated by measuring the energy required to save registers, MCU, and peripheral state. With NVRAM [32], the threshold voltage does not need to be high, as it only saves registers, peripheral state, and a checkpoint flag. Contrarily [32], Balsamo et al. [19] employ volatile RAM and save entire system state (complete RAM along with MCU registers) onto NVM thus requiring higher V_{th}. One main limitation of such an approach is an off-line characterization of the device to find the threshold voltage V_{th} for checkpointing and restoring the system state. They addressed this limitation by proposing an adaptive approach to self-calibrate the checkpointing threshold (V_{th}), depending upon the dynamics of energy source and power consumption of the system [20]. This calibration strategy makes the overall system transparent and portable across multiple systems by adapting the voltage threshold at run-time, considering system power consumption, decoupling capacitance and energy source behavior.

Unlike Woude et al. [35], Mathew Hicks [31] does not take checkpoint at every idempotency violation. An interrupt is generated to trigger checkpointing when any one of the three buffers overflow. This allows the solution to stretch the execution of an idempotent section past its natural limits and reduces the number of checkpoints taken by approach [35]. However, there can be a scenario when no single idempotent section is big enough to make the buffer overflow. If an energy failure occurs before a buffer overflow, all changes in the state would be lost and the device would restart program execution. In such a case, the device can get stuck in a live-lock problem.

Timers. Interrupt based reactive triggering approaches subtract a significant amount of energy from the energy buffer and is suitable only when the checkpoint size of the device is too small.

Ransford et al. [18] propose a timer-based approach to periodically trigger checkpointing after a fixed interval of time. As soon as the timer expires, the system saves the checkpoint. The timer is set at the time of deployment and, ideally, it should be equal to the time of capacitor discharge. However, the rate at which the energy is consumed for each program depends on its complexity of computations which is different for different programs. Therefore, it is difficult for a programmer to set an accurate value at the time of deployment.

To overcome a live-lock problem, Mathew Hicks [31] also use watchdog timers to minimize the gap between two checkpoints to avoid the overhead of re-executing the idempotent section thus requiring programmer's effort.

5.2.1. Discussion

It must be noted that an interrupt based TPS solution employs hardware support to trigger checkpointing either by using an external comparator or by using hardware buffers. In both cases, additional energy is consumed which reduces the amount of energy available for program execution.

Timer-based approach eradicates this additional energy consumption by executing trigger calls after fixed time intervals. However,

finding the right time interval for a particular program requires repeated execution under energy profiles [18]. Precisely, the timer value should be set equal to the time it was taken by energy buffer to reach V_{th}. This demands a manual configuration of timer value and has to be repeated once the TPS program changes. Therefore, reactive triggering mechanism needs to balance these trade-offs to reach an energy-efficient solution.

6. Checkpointing strategy

Based on the strategy to determine checkpoint size, existing literature can be divided into two main categories; namely copy-used and copy-if-change as shown in Fig. 4.

Copy-used. This category encapsulates all those solutions which exploit the inherent division of program into different memory regions. Solutions falling under this category focus on tracking the change in size of the used memory regions and save only those regions at the time of checkpoint.

Copy-if-change. This category identifies modified memory locations in the current program state from the checkpointed state. To identify changes in the state, each solution performs runtime tracking of writes to the main-memory. Note that solutions falling under this category save complete RAM state at the time of first checkpoint. This allows these solutions to create a one-to-one mapping of current program state with the checkpointed state thus helping them while updating the checkpoint.

6.1. Copy-used

An intermittent program can use entire RAM during its execution. Therefore, some approaches consider entire RAM as part of the checkpoint [19,20]. This results in a large checkpoint size thus requiring high checkpointing energy. Actually, an intermittent program uses predefined memory segments during its execution; leaving a large chunk of memory unused.

Ransford et al. [18] identify used memory regions by finding address ranges occupied by each code segment, i.e., stack and global variables, at the time of interruption. This allows the system to save only the occupied memory regions of RAM while checkpointing and not the entire RAM. The size of memory in-use depends on the program-point of interruption. Additionally, they do not track `heap` segment of memory thus limiting the programmers by not allowing dynamic data structures. Modern IoT platforms, i.e., ARM-based platforms [59], encourage developers to use dynamic data structures for higher programming flexibility which enables more complex IoT applications [60]. Therefore, this approach limits the applicability of the proposed approach to a wide set of IoT applications by not allowing `heap`.

Bhatti et al. [29] propose two different flavors of copy-used checkpointing strategies that cover all memory segments and can be integrated with any TPS; namely, Split and Heap Tracker.

- **Split** It makes use of existing segregation of RAM in different segments namely `.bss`, `.data`, `heap` and `stack`. The `.bss`, `.data`, and `heap` segments are contiguous in the address space whereas system `stack` grows from the last address in RAM. This approach is, however, unable to identify memory fragmentation that may exist in heap due to `free()` function.
- **Heap tracker** To cater for fragmentation in the heap, this approach keeps track of the allocation to heap using wrapper functions for `malloc()` and `free()`. By dividing the memory into blocks, HeapTracker increments (decrements) the count of the block in which the address is allocated (de-allocated) when `malloc()` (`free()`) is executed.

While deciding the threshold voltageV_{th}, these approaches take into account the energy required to erase the NVM before re-programming. After each successful checkpointing operation, the device is turned off to gather energy from the environment.

Traditionally, program data reside in the volatile main memory for fast access [18–20,29]. Recently, new systems [53] have emerged that are equipped with both volatile and non-volatile main memory thus allowing persistent program data. Therefore, copying only the volatile state at each power interruption in such systems can result in an output which was not possible in a continuous execution. This can cause an idempotence violation which occurs when global variables are located in the NVM and instructions accessing such variables are executed redundantly in the two consecutive active periods.

To solve idempotence violation, a solution exists which provides special function calls to the programmer thus helping them divide the code into different re-executable chunks [22]. Checkpointing takes place at each function call similar to the one used by Ransford et al. [18] for the volatile state. For non-volatile state, data versioning is performed and the current version of non-volatile variables are copied onto stack thus making them a part of the checkpoint. At each restore, the non-volatile variables are written back again. However, the used memory region estimated in such an approach becomes very high due to non-volatile variable versioning. Furthermore, a lot of unnecessary checkpoints can take place because of poor placement of the function calls by the programmer.

There are solutions that embed the sense of used memory location in the way the programmer writes the program [33,54] thereby reducing the amount of used memory region. These solutions provide language semantics to allow the programmer to write a program in a set of modular tasks than can be run independently. The programmer can share data between two tasks only through specified data channels created in NVM. It eradicates the need to checkpoint global state as necessary data already resides in the NVM. Since multiple tasks can share the same variables, this approach [54] suffers from NVM wastage as multiple channels between different tasks may contain the same data.

One way to avoid NVM wastage is to use memory pointer for locations shared between task pairs but this can, again, give rise to the idempotence violation problem. Maeng et al. [33] extend the existing language support [22,54] for intermittent programs by enabling the programmer to define variables that are shared between tasks and are named as Task Shared (TS) variables. These variables are declared in the global scope and reside on NVM while other variables are treated as local variables. Among all TS variables, the authors find all possible WAR dependencies that may exist among these variables using the control flow graph of tasks. For each WAR dependency, it creates a private copy of the variable in a privatized buffer located in volatile memory. It, then, redirects every reference to such variables towards the privatized buffer. In this way, a single copy of each TS variable is accessed and updated during the entire execution of the program. If an energy interruption occurs, all changes in the private buffer are lost thus avoiding idempotence violation. This solution, however, incurs a significant run-time overhead due to the frequent accesses to the memory locations and function calls. Therefore, its feasibility on the cost benefit spectrum needs to be investigated.

Some solutions assume main memory as completely non-volatile [32,35]. This makes, previously volatile state, non-volatile thus significantly reducing the checkpoint size as the TPS only needs to save the MCU registers. However, this gives rise to new set of challenges. On one hand, executing a program with non-volatile main memory limits the capability of the device to execute on higher frequencies [42]. On the other hand, it consumes more power to execute program instructions thus draining the energy buffer quickly. We discuss the trade-off between having a volatile and NVM in detail in Section 4.

6.1.1. Discussion

Copy-used solutions reduce the size of checkpoint by only saving used memory regions of the main-memory while bearing minimal computational overhead. This reduces the amount of energy required at the time of checkpoint. They perform coarse grained analysis of the program to estimate checkpoint size. A large chunk of these used memory regions remain unchanged from one checkpoint to another. Therefore, copying used regions would result in saving redundant memory locations to the NVM thus wasting energy.

While language support reduces used memory regions by allowing programmer to make changes only in the predefined memory regions [22,33,54], employing such support gives rise to new set of challenges. First, it demands the user to learn new language thus hampering its wide scale adoption. Furthermore, it increases the runtime overhead which puts a burden on the constrained energy budget. Even with these challenges resolved, there is still a possibility of having redundant data in the used memory regions [54].

In a nutshell, copy-used approaches have to strike a trade-off between computational overhead to compute the checkpoint size and its reduction.

6.2. Copy-if-change

Checkpointing solutions in the this category remove all unchanged memory locations from the checkpoint size at the time of saving the state. These solutions exactly identify changed memory locations at the time of checkpoint which saves checkpointing energy but at the cost of performing runtime tracking; which is an overhead.

Aouda et al. [61] proposed a technique which makes incremental changes to the checkpoint program-state in the NVM. This approach divides the program-state into the same size blocks and maintains a hash of each block. As the program state changes, the hash of particular block changes. This approach compares previously computed hash value with the new one to identify changes in content. It then copies this changed block into a newly allocated block and updates the hash value. Garbage collection is performed frequently to collect free blocks. Therefore, this approach maintains two checkpoints at a time; active and scratch the image of the checkpoint. At the time of energy failure, all changes are made to the scratch image and both images switch their roles once the checkpoint is successfully updated.

As read operations are cheaper than writes [52], there are approaches which read the entire RAM state and compare it with the checkpointed state [29]. In this way, these approaches identify changes in the RAM state which have to be replicated in the checkpointed state in order to update it. To make their approach efficient, they divide the entire RAM into blocks defined by the smallest writable units of NVM. At each checkpoint call, they only update the corresponding chunk of memory in the NVM. However, if nothing changes from the previous checkpoint, this approach still needs to sweep the entire RAM which wastes energy.

Ahmed et al. [62] observed that there are very few instructions in code that can modify the program's state e.g. assignment, increment, and decrement operations. Therefore, the actual change in the current program-state from the checkpointed state is very small when compared to the one estimated by copy-used approaches [18–20,29].

To exactly identify changes, the authors propose a solution that exposes a function call, `record()`, and an in-memory data structure, *modification record*, required for book-keeping of all the changes made to the state. It uses a pre-compiler, ANTLR [63], to place `record()` calls before any state modifying statement. The `record()` call takes the address of the variable and its size as arguments and saves it in the *modification record* data structure. At the time of checkpoint, it only copies the current value of saved addresses onto the checkpointed state in NVM thus reducing checkpoint size. Unlike Bhatti et al. [29], this approach does not read checkpointed state in NVM to compute changes in current state.

Variables in a program have different lifespan depending upon their scope of declaration. Thus the proposed solution has two different schemes for tracking changes in state.

- **Global Context**: To track changes in the global context, it places `record()` calls before every global variable modification.
- **Stack tracking**: As local changes are located in system stack, it introduces an additional pointer to stack namely stack tracker (ST) to be used in combination with stack pointer (SP) and base pointer (BP) to track changes in local state. The stack region between ST and SP is always checkpointed at each checkpoint call as it contains changes to local variables.

This approach is memory-efficient as it uses a bit array to represent modifications made in the entire memory. In this way, the amount of memory consumed by this approach is $\frac{1}{8}$th of the total memory of the device. The gain in performance stems from its ability to exactly identify what has changed from the previous checkpoint thus ignoring empty or unused locations. Furthermore, the authors also propose different optimizations for placement of record calls helping them reduce their run-time overhead. In case of page programmable memories, however, such an approach can result in redundant writes to NVM in an extreme case where only few bytes of a page can cause a full page write. Sliper et al. [64] and Verykios et al. [65] propose an approach for handling such extreme cases by introducing light-weight memory management techniques.

Some systems perform liveness analysis of global variables to estimate changes in the program-state [66]. They calculate the span of code over which a variable may be used without being re-written and assume a hybrid memory model. If a variable's live range crosses a trigger call, they relocate it in the NVM. To keep a track of all writes to a non-volatile variable, it inserts a function call before every potential write to a non-volatile variable. It takes the address of the variable as an argument and logs the value before the write executes. This code instrumentation helps the system in tracking all changes in the non-volatile variables. These variable values are later restored after each power interruption thus avoiding idempotence violation.

Matthew Hicks [31] propose a hardware approach to identify changes in program state and propose a custom-built TPS which uses three additional volatile memory buffers to keep track of the memory addresses accessed: read-first, write-first, and write-back buffers. The read-first buffer holds read dominated addresses, while the write-first buffer holds write dominated addresses. During program execution, it checks if a current instruction is going to "write" at a memory address. If yes, it signals an idempotence violation if the address is already in the read-first buffer. It simply adds the violating address and its value in the write-back buffer to delay the checkpointing. The checkpoint size is dependent on the size of the write-back buffer. Larger the size, greater will be the checkpointing and code re-execution cost of the program. However, it is still much smaller than saving the entire RAM [19]. In the case of the hybrid memory model, it uses a similar approach used by Ahmed et al. [62] but with hardware support.

6.2.1. Discussion

Copy-if-change strategies provide the least amount of checkpoint size to be saved at the time of checkpoint thus requiring lower energy than the copy-used category. However, runtime tracking performed by copy-if-change approaches have a non-zero computational overhead. It is either because of special function calls inserted in the program or due to specialized hardware employed by the TPS. The additional energy consumed by the system while tracking changes which is subtracted from the energy budget available for program execution.

The energy consumed by runtime tracking is directly proportional to the memory access patterns of the application and the time interval between two checkpoints. More widespread and frequent accesses of memory can result in a larger change in program-state thereby requiring more energy than normal to save the checkpoint. This will

also increase the amount of energy spent in tracking those changes. Therefore, a copy-if-change solution is only feasible if the additional energy overhead caused by runtime tracking is always greater than the amount of energy saved due to checkpoint size reduction.

Solution belonging to this category have to answer this key question in order to ensure energy-efficient checkpointing of program-state.

7. Conclusion

After extensively surveying the existing state-of-art literature, we present a three-layered topological taxonomy of program-state retention solutions for TPS. First layer describes how different memory models effect the decision of what checkpoint size to save and when to trigger it. We discuss that volatile systems are a viable and an energy-efficient option for reasonably bigger capacitor size. For extremely small capacitors size, fully non-volatile system perform better. However, mixed volatility systems are the solution which can adapt to any ambient energy conditions to extract maximum performance from the TPS.

The second and third layer discuss two main challenges in the transiently-powered domain. First one is to devise an energy-efficient checkpointing strategy while second is to delay the checkpointing as much as possible so that the device can avoid excessive checkpointing. We classify existing state-of-the-art solutions proposing energy-efficient checkpointing into two categories: First employs a coarse-grained approach to find the used memory regions of the program while the second uses a fine-grained approach to exactly the find the changed memory locations from one checkpoint to another. This helps the device to reduce the checkpoint size, however, this fine-grained analysis comes at a cost of computational overhead. The second category covers triggering mechanisms for the checkpoint which can either be proactive or reactive in nature. Solving both these challenges allow the device to spend more time on program execution than checkpointing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M.D. Scott, B.E. Boser, K.S. Pister, An ultralow-energy ADC for smart dust, IEEE J. Solid-State Circuits (2003) 1123–1129.

[2] G.M. Calvagna, G. Torrisi, C. Giuffrida, S. Patanè, Pacemaker, implantable cardioverter defibrillator, CRT, CRT-D, psychological difficulties and quality of life, Int. J. Cardiol. (2014) 378–380.

[3] International space station, 2018, URL https://www.nasa.gov/mission_pages/station/research/experiments/2374.html.

[4] N. Correll, P. Dutta, R. Han, K.S.J. Pister, New directions: Wireless robotic materials, 2017, ArXiv abs/1708.04677.

[5] D. Piumwardane, C. Pérez-Penichet, C. Rohner, T. Voigt, Backscatter communication for wireless robotic materials, in: Proceedings of the International Conference on Embedded Wireless Systems and Networks, EWSN, 2019.

[6] J. Shah, B. Mishra, IoT enabled environmental monitoring system for smart cities, in: Internet of Things and Applications (IOTA), International Conference on, IEEE, 2016, pp. 383–388.

[7] D. Daly, P.P. Mercier, M. Bhardwaj, A.L. Stone, Z.N. Aldworth, T.L. Daniel, J. Voldman, J.G. Hildebrand, A.P. Chandrakasan, A pulsed UWB receiver SoC for insect motion control, IEEE J. Solid-State Circuits 45 (1) (2010).

[8] V. Iyer, R. Nandakumar, A. Wang, S.B. Fuller, S. Gollakota, Living IoT: A flying wireless platform on live insects, in: International Conference on Mobile Computing and Networking, MobiCom, 2019, pp. 1–15.

[9] D. Rakhmatov, S. Vrudhula, Energy management for battery-powered embedded systems, ACM Trans. Embedded Comput. Syst. (2003) 277–324.

[10] T. Simunic, L. Benini, G. De Micheli, Energy-efficient design of battery-powered embedded systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (2001) 15–28.

[11] Y. Shen, X. Meng, Q. Cheng, S. Rumley, N. Abrams, A. Gazman, E. Manzhosov, M.S. Glick, K. Bergman, Silicon photonics for extreme scale systems, J. Lightwave Technol. (2019) 245–259.

[12] CSIRO, Global initiative for honey bee health, 2019, URL https://research.csiro.au/gihh/about/.

[13] D.C. Daly, P.P. Mercier, M. Bhardwaj, A.L. Stone, Z.N. Aldworth, T.L. Daniel, J. Voldman, J.G. Hildebrand, A.P. Chandrakasan, A pulsed UWB receiver SoC for insect motion control, IEEE J. Solid-State Circuits (2009) 153–166.

[14] K. Lin, J. Yu, J. Hsu, S. Zahedi, D. Lee, J. Friedman, A. Kansal, V. Raghunathan, M. Srivastava, Heliomote: enabling long-lived sensor networks through solar energy harvesting, in: Proceedings of the 3rd International ACM Conference on Embedded Networked Sensor Systems, SenSys, ACM, p. 309.

[15] S. Priya, C.-T. Chen, D. Fye, J. Zahnd, Piezoelectric windmill: a novel solution to remote sensing, Japan. J. Appl. Phys. (2004) L104.

[16] S. Meninger, J.O. Mur-Miranda, R. Amirtharajah, A. Chandrakasan, J.H. Lang, Vibration-to-electric energy conversion, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (2001) 64–76.

[17] N.A. Bhatti, M.H. Alizai, A.A. Syed, L. Mottola, Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences, ACM Trans. Sensor Netw. (2016) 40.

[18] B. Ransford, J. Sorber, K. Fu, Mementos: System support for long-running computation on RFID-scale devices, in: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2011, pp. 159–170.

[19] 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 Embedded Syst. Lett. (2014) 15–18.

[20] 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. (2016) 1968–1980.

[21] N.A. Bhatti, L. Mottola, Harvos: Efficient code instrumentation for transiently-powered embedded sensing, in: 2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN, IEEE, 2017, pp. 209–220.

[22] B. Lucia, B. Ransford, A simpler, safer programming and execution model for intermittent systems, ACM SIGPLAN Not. (2015) 575–585.

[23] J. Hester, L. Sitanayah, J. Sorber, Tragedy of the Coulombs: Federating energy storage for tiny, intermittently-powered sensors, in: Proceedings of the 13th International Conference on Embedded Networked Sensor Systems, SenSys, ACM, 2015, pp. 5–16.

[24] B. Ransford, B. Lucia, Nonvolatile memory is a broken time machine, in: Proceedings of the Workshop on Memory Systems Performance and Correctness, ACM, 2014, p. 5.

[25] A. Branco, L. Mottola, M.H. Alizai, J.H. Siddiqui, Intermittent asynchronous peripheral operations, in: Proceedings of the 17th Conference on Embedded Networked Sensor Systems, ACM, 2019, pp. 55–67.

[26] S. Ahmed, Q. ul Ain, J.H. Siddiqui, L. Mottola, M.H. Alizai, Intermittent computing with dynamic voltage and frequency scaling, in: Proceedings of 2020 International Conference on Embedded Wireless Systems and Networks, EWSN, 2020.

[27] H. Jayakumar, A. Raha, J.R. Stevens, V. Raghunathan, Energy-aware memory mapping for hybrid FRAM-sram MCUs in intermittently-powered IoT devices, ACM Trans. Embedded Comput. Syst. (2017) 1–23.

[28] S. Ahmed, H. Khan, J.H. Siddiqui, J.Á. Bitsch, M.H. Alizai, Incremental checkpointing for interruptible computations, in: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems, SenSys, 2016, pp. 350–351.

[29] N. Bhatti, L. Mottola, Efficient state retention for transiently-powered embedded sensing, in: Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks, EWSN, 2016, pp. 137–148.

[30] A. Colin, B. Lucia, Termination checking and task decomposition for task-based intermittent programs, in: Proceedings of the 27th International Conference on Compiler Construction, CC, ACM, 2018, pp. 116–127.

[31] M. Hicks, Clank: Architectural support for intermittent computation, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 228–240.

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

[33] K. Maeng, A. Colin, B. Lucia, Alpaca: intermittent execution without checkpoints, in: Proceedings of the ACM on Programming Languages, OOPSLA, ACM, 2017, pp. 1–30.

[34] J. Hester, K. Storer, J. Sorber, Timely execution on intermittently powered batteryless sensors, in: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–13.

[35] J. Van Der Woude, M. Hicks, Intermittent computation without hardware support or programmer intervention, in: Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2016, p. 17.

[36] S. Ahmed, A. Bakar, N.A. Bhatti, M.H. Alizai, J.H. Siddiqui, L. Mottola, The betrayal of constant power× time: finding the missing joules of transiently-powered computers, in: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES, ACM, 2019, pp. 97–109.

[37] S. Ahmed, M.H. Alizai, J.H. Siddiqui, N.A. Bhatti, L. Mottola, Towards smaller checkpoints for better intermittent computing, in: 17th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN, 2018, pp. 132–133.
[38] W.K. Seah, Z.A. Eu, H.-P. Tan, Wireless sensor networks powered by ambient energy harvesting (WSN-HEAP)-Survey and challenges, in: 2009 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, Ieee, 2009, pp. 1–5.
[39] F.K. Shaikh, S. Zeadally, Energy harvesting in wireless sensor networks: A comprehensive review, Renew. Sustain. Energy Rev. 55 (2016) 1041–1054.
[40] M. Treaster, A survey of fault-tolerance and fault-recovery techniques in parallel systems, 2005, arXiv preprint cs/0501002.
[41] S. Kalaiselvi, V. Rajaraman, A survey of checkpointing algorithms for parallel and distributed computers, Sadhana (2000) 489–510.
[42] B. Lucia, V. Balaji, A. Colin, K. Maeng, E. Ruppel, Intermittent computing: Challenges and opportunities, in: LIPIcs-Leibniz International Proceedings in Informatics, 2017.
[43] A. Rodriguez, D. Balsamo, Z. Luo, S.P. Beeby, G.V. Merrett, A.S. Weddell, Intermittently-powered energy harvesting step counter for fitness tracking, in: 2017 IEEE Sensors Applications Symposium, SAS, IEEE, 2017, pp. 1–6.
[44] M. Habibzadeh, M. Hassanalieragh, A. Ishikawa, T. Soyata, G. Sharma, Hybrid solar-wind energy harvesting for embedded applications: Supercapacitor-based system architectures and design tradeoffs, IEEE Circuits Syst. Mag. (2017) 29–63.
[45] T. Instruments, ULP meets energy harvesting, 2019, URL https://bit.ly/2GkBkOg.
[46] M.-T. Chiu, Y.-P. You, CLPKM: A checkpoint-based preemptive multitasking framework for OpenCL kernels, J. Syst. Archit. (2019) 53–62.
[47] B. Cai, K. Li, SLO-aware colocation: Harvesting transient resources from latency-critical services, J. Syst. Archit. 101663.
[48] A. Rodriguez Arreola, D. Balsamo, G. Merrett, A. Weddell, RESTOP: Retaining external peripheral state in intermittently-powered sensor systems, Sensors (2018) 172.
[49] G. Berthou, T. Delizy, K. Marquet, T. Risset, G. Salagnac, Sytare: a lightweight kernel for NVRAM-based transiently-powered systems, IEEE Trans. Comput. 68 (9) (2018) 1390–1403.
[50] J. Hester, K. Storer, J. Sorber, L. Sitanayah, Towards a language and runtime for intermittently powered devices, in: Workshop on Hilariously Low-Power Computing, HLPC, 2016.
[51] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W.P. Burleson, J. Sorber, Persistent clocks for batteryless sensing devices, ACM Trans. Embedded Comput. Syst. (2016) 77.
[52] L.V. Cargnini, L. Torres, R.M. Brum, S. Senni, G. Sassatelli, Embedded memory hierarchy exploration based on magnetic random access memory, Multidisciplinary Digital Publishing Institute, 2014, pp. 214–230,
[53] T. Instruments, MSP430FR573x mixed-signal microcontrollers, 2014, http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf.
[54] A. Colin, B. Lucia, Chain: tasks and channels for reliable intermittent programs, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2016, pp. 514–530.
[55] A. Gomez, L. Sigrist, M. Magno, L. Benini, L. Thiele, Dynamic energy burst scaling for transiently powered systems, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, 2016, pp. 349–354.
[56] S. Senni, L. Torres, G. Sassatelli, A. Gamatie, Non-volatile processor based on MRAM for ultra-low-power IoT devices, ACM J. Emerg. Technol. Comput. Syst. (2016) 1–23.
[57] K. Kim, S. Lee, Memory technology in the future, Microelectron. Eng. (2007) 1976–1981.
[58] E. Philofsky, FRAM-the ultimate memory, in: Proceedings of Nonvolatile Memory Technology Conference, 1996, pp. 99–104.
[59] D. Balsamo, A. Elboreini, B. Al-Hashimi, G. Merrett, Exploring ARM mbed support for transient computing in energy harvesting IoT systems, in: Proceedings of the 7th IEEE International Workshop on Advances in Sensors and Interfaces, IWASI, 2017, pp. 115–120.
[60] C. Leech, Y.P. Raykov, E. Ozer, G.V. Merrett, Real-time room occupancy estimation with Bayesian machine learning using a single PIR sensor and microcontroller, in: Sensors Applications Symposium (SAS), 2017 IEEE, IEEE, 2017, pp. 1–6.
[61] F.A. Aouda, K. Marquet, G. Salagnac, Incremental checkpointing of program state to NVRAM for transiently-powered systems, in: 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC, IEEE, 2014, pp. 1–4.
[62] S. Ahmed, N.A. Bhatti, M.H. Alizai, J.H. Siddiqui, L. Mottola, Efficient intermittent computing with differential checkpointing, in: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES, ACM, 2019, pp. 70–81.
[63] T. Parr, The Definitive ANTLR 4 Reference, 2013, https://goo.gl/RR1s.
[64] S.T. Sliper, D. Balsamo, N. Nikoleris, W. Wang, A.S. Weddell, G.V. Merrett, Efficient state retention through paged memory management for reactive transient computing, in: Proceedings of the 56th Annual Design Automation Conference, DAC, 2019, pp. 1–6.
[65] T.D. Verykios, D. Balsamo, G.V. Merrett, Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of NVM technologies, Sustain. Comput.: Inf. Syst. (2019) 167–178.
[66] K. Maeng, B. Lucia, Adaptive dynamic checkpointing for safe efficient intermittent computing, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2018, pp. 129–144.

Saad Ahmed is a Ph.D. candidate in the Department of Computer Science, School of Science and Engineering, LUMS Pakistan. His research interests include batteryless-IoT devices, embedded systems and edge computing. He has published full papers in premier venues in networked systems domain such as IPSN, EWSN and LCTES with posters in SenSys. He also worked as a visiting researcher at the Communications and Distributed Systems(ComSys) Lab, RWTH Aachen Germany for two consecutive years i.e., June 2016 and July 2017. He also served as a reviewer for ICPADS and ComNets in 2019. He won the intermittent computing hackathon held at the Doctoral school organized by the IDEA league and his paper has been nominated as the Best Paper Candidate in EWSN 2020.

Dr. Naveed Anwar Bhatti is an Assistant Professor at Air University (Pakistan). He completed his Ph.D. at Politecnico di Milano (Italy) in 2018. Later, he joined RISE (Sweden) as an ERCIM postdoctoral fellow for one and a half years. His primary research area is Cyber–Physical Systems (CPS) with a focus on transiently-powered embedded systems. Out of this research, he managed to publish papers in IPSN, EWSN, ICC, LCTES, and TOSN which are considered as flagship events in the field of networked embedded systems. He also won the best Ph.D. presentation award at IPSN 2016. He was also in the technical program committee (TPC) of ICPADS 2019, AlgoSensor 2019, ISIoT 2019 and IoTDI 2019.

Martina Brachmann is an ERCIM Alain Bensoussan postdoctoral researcher working at the Networked Embedded Systems (NES) group of Thiemo Voigt at RISE Research Institutes of Sweden in Stockholm, Sweden. Martina's research area is the Internet of Things with main focus on communication in low-power wireless networks. Over the past years, she has been working on different aspects and layers in the communication stack, from security over medium access control to physical layer considerations. Martina completed her Ph.D. at the Technische Universität Dresden, Germany, under guidance of Prof. Dr. Silvia Santini in 2018. From May to August 2015, she joined Prof. Dr. Olaf Landsiedel's research group in Chalmers University of Technology, Sweden, as a research scholar. She received her Master of Science in Information and Communication Technology from the Brandenburg University of Technology Cottbus (BTU), Germany, in 2012.

Dr. Alizai has over 10 years of experience as a researcher, software engineer, and technical lead both in industrial and academic settings. He has authored book, book chapters and published numerous scientific papers, while abroad and indigenously from Pakistan, several of them in top flight ACM SIG sponsored venues such as ACM SenSys, IPSN, BuildSYS, CoNEXT. He is experienced in leading innovative research projects in pervasive computing technologies such as Internet of things, sensor and delay tolerant networks, ICT4D, and mobile computing. He was employed as a software engineer in several European Union projects and has a wealth of experience in teaching/training cutting edge technologies and courses in theoretical and practical computer sciences at grad, post grad and professional level. He is also a visiting researcher at his alma mater: ComSys, RWTH Aachen, Germany.