Energy-Efficient Memory Tracing for State Retention in Transient Computing Systems

Theodoros D. Verykios[1*], Domenico Balsamo[2], Geoff V. Merrett[1]
[1]School of Electronics and Computer Science, University of Southampton, Southampton, UK.
[2]MicroSystems Research Group, School of Engineering, Newcastle University, Newcastle, UK.
[*]T.Verykios@soton.ac.uk

***Abstract*—Transient computing systems, also known as intermittent computing systems, are batteryless systems powered by energy harvesting (EH) sources that do not require large energy storage for system operations. Instead, they rely on retaining their state, i.e. a snapshot, in non-volatile memory (NVM) in the event of a power outage and restoring it when the power recovers. In this paper, we first discuss the limitations of state-of-the-art techniques that attempt to minimize the amount of system state saved to NVM. Therefore, we propose a novel energy-efficient system-level approach for state retention through memory tracing based on a custom hardware module named *MeTra* that traces changes in the main (volatile) memory between power outages. *MeTra* allows the voltage threshold that activates the state retention process to be dynamically adjusted according to the energy requirement of each snapshot. Thus, a great proportion of the energy harvested can be spent on useful operations. Experimental results show that the system's active time can be extended up to 17x for Flash-based systems and 92.2% for FRAM-based systems, compared to saving the entire system state, with an area overhead of as little as 2.48%.**

***Index Terms*—Transient computing, state retention, memory tracing, energy harvesting, energy efficiency.**

I. Introduction

Energy harvesting (EH) can help operate sensor systems by harnessing electrical energy from environmental sources, making them self-powered [1]. However, the intermittent and varying nature of these sources makes EH unreliable for performing sensing, processing and transmitting tasks. Therefore, self-powered systems usually require large energy storage to flatten the mismatch between EH output and systems' consumption. However, this storage increases their volume, weight and cost and requires a long time to charge, delaying system startup, which is not ideal for smart electronics [2].

Transient computing, also known as intermittent computing, allows operations to span across power outages, thereby minimizing the size of the required energy storage [3]. To ensure the forward progress of system operations, a snapshot of the system state is saved in non-volatile memory (NVM) in the event of a power outage and restored when power recovers. This means that the contents of the core and general purpose registers and main (volatile) memory, i.e. RAM, are saved to and restored from NVM, e.g. Flash or FRAM.

The system state can be saved periodically, which can result in significant energy overhead for redundant checkpoints. Alternatively, snapshots can be saved on-demand when the system voltage, V_{sys}, reaches a hibernation threshold, V_h, indicating that a power outage is imminent ($V_{sys} \leq V_h$) [4], [5]. Therefore, the content needed to resume the operation from a specific point in time is stored in NVM, and forward execution can be achieved when power recovers.

979-8-3503-3694-8/23/$31.00 ©2023 IEEE

Software-based approaches have mainly been proposed to implement transient computing, e.g. Mementos [3], HarvOS [6], Hibernus [7], Hibernus++ [8], and QuickRecall [9]. However, these "blindly" save the entire system state without distinguishing between useful and redundant content in RAM, or use NVM inefficiently. The term *useful* here refers to data that has changed between two power outages, whilst *redundant* is content that has not changed. Saving the entire state each time reduces the system's active time, which in turn leads to a reduction in energy efficiency. Different techniques have been proposed to address this challenge and reduce the amount of RAM content saved in NVM [10], [11]. However, most of them rely on the main processor (CPU) to trace changes in RAM, meaning that the CPU cannot be fully used for useful operations. Also, even though some of these can trace changes in RAM, they cannot dynamically adjust V_h.

In this paper, we propose a novel energy-efficient system-level approach to retaining the system state based on a custom module called *MeTra* that traces changes in RAM between power outages while minimizing the amount of state that needs to be saved. *MeTra* allows V_h to be adjusted at run-time, based on the useful data to be saved in NVM. As a result, the system's active time increases, and the number of snapshots decreases, thereby increasing overall energy efficiency.

The novel contributions of this paper are:

- The analysis of the state retention process in transient computing systems, with particular attention to energy efficiency aspects (Sec. III);
- The design and implementation of *MeTra*, our hardware module, to minimize redundant RAM when saving a snapshot of the system state to NVM (Sec. IV);
- The integration of *MeTra* with a low-power CPU architecture (ARM Cortex-M0) typically used in sensor system applications (Sec. IV), and evaluation of energy efficiency gains and system-level benefits (Sec. V).

II. Related work and Motivation

In this section, we discuss different techniques to reduce the amount of RAM state saved in NVM, and thus provide the motivation for our approach.

Several software-based techniques for minimising the state saved in NVM have been proposed in the literature. Bhatti *et al.* [12] presented a selective state retention policy that identifies unallocated space in RAM (also called free space) and saves only the allocated, i.e. used by the main application, parts to Flash. However, current sensor system applications typically use most of the available RAM; therefore, unallocated space can represent only a small portion. Also, most of the data in RAM will remain unchanged between power cycles, but this is not captured by this technique, meaning that unchanged RAM will be saved to the NVM, even if it already exists from a previous snapshot. Finally, this technique has only been applied to Flash-based sensor systems, offering little benefit due to the high erasure cost.

Verykios *et al.* [13] have proposed a number of software techniques, such as *Multiple Allocated State Images, Updated Blocks* and *Multiple Updated Blocks*, to address the above challenges, which exploit the individual characteristics of different types of NVM technologies (e.g. symmetry and erasure requirements). However, in these techniques, identifying unallocated RAM or changed RAM between EH cycles does not occur until a preset V_h is reached, meaning that the system cannot efficiently use excess energy. Sliper *et al.* [14] proposed *ManagedState*, a page-based memory manager for tracing allocated RAM and changed data at run-time, and thus dynamically adjusting V_h. However, this manager relies on the CPU to trace changes in RAM using a specific set of APIs. Also, the granularity in tracing changes is limited by the size of the page, which can result in redundant data being saved to NVM depending on the locality property of the application.

Pala *et al.* [15] proposed a first attempt based on a hardware technique. Their backup controller, named *Freezer*, can save a reduced snapshot of the system state in the event of a power outage by monitoring RAM accesses during program execution and then committing the changes to NVM. However, this controller was only simulated in isolation with no measurable system-level benefits. In addition, *Freezer* does not implement any dynamic threshold voltage adjustment for V_h, which means that excess energy is not used efficiently, despite the reduced snapshot size. The authors also speculate that the frequency between power outages can be up to 100Hz; however, this is not realistic for typical EH sources [6]. Finally, the tracing mechanism offers granularity of up to 256 bytes, which is a large block size, given that a typical low-power sensor system contains 4-32KB of RAM[1].

Limitations of existing approaches: While we provide some examples of software-based techniques and only a hardware-based approach, the issues mentioned are valid field-wise. Therefore, we believe that the ideal solution would need to: (a) offer an adjustable V_h without significantly sacrificing the system application active time, (b) provide fine-grained RAM tracing to minimise the number of redundant NVM writes, and (c) enable comparable performance across all application workloads, regardless of RAM spatial and temporal locality.

Hence, we propose an energy-efficient system-level module for memory tracing (*MeTra*), which provides fine-grained RAM tracing and dynamically adjusts V_h, minimizing the CPU time for state retention and maximizing the system application active time. Details regarding each part of *MeTra* and how this is integrated within a low-power CPU architecture (ARM Cortex-M0) are provided in the following sections.

III. Analysis of State Retention Process

This section discusses the retention process in transient systems, emphasizing aspects that affect system energy efficiency, and justifying the need for a hardware solution to trace changes in RAM for more efficient state retention.

Depending on the control strategy, the system state can be saved periodically with different checkpoint granularity, e.g. after completing a task, or saved on-demand, i.e. when V_h is reached [16]. In the former case, applications can be organised into tasks (sensing, processing, and transmitting), each of which must be completed within one EH cycle such that a snapshot can be saved in NVM for transition to the next task. However, this strategy requires that the energy storage, C_{store}, is sized for the task that consumes the most energy to ensure high reliability in task execution (application dependent), or it can result in significant system energy overhead for redundant checkpoints, e.g. when the application is divided into micro-tasks. Conversely, tasks do not need to be completed in one EH cycle with on-demand schemes since the system state is saved before a power outage occurs, i.e. when $V_{\text{sys}} \leq V_h$. This allows to divide the execution of the task into consecutive EH cycles, minimizing C_{store}, whose size depends only on the energy needed to save the system state. This strategy is particularly suitable for tasks involving extended processing operations. At the same time, it is not ideal for tasks that cannot be split among EH cycles, such as sensing or transmission.

We focus on tasks involving extensive processing operations (e.g. Fast Fourier Transform (FFT) or Advanced Encryption Standard (AES)), i.e. those that may be spread over consecutive EH cycles and potentially require a number of state retention operations. The energy, E_{task}, required to complete one iteration of each task with a stable supply can be described by $E_{task} = P_{task} \cdot t_{task}$, where P_{task} is the average power consumption while running the task, and t_{task} the time required to complete it. If power outages occur before the task is completed, the system state must be retained in NVM. Hence, the total energy per task becomes $E_{total} = E_{task} + E_{snap}$, where E_{snap} is the total energy overhead associated with saving and restoring system state a number of times, s, equal to the number of power outages, defined as

$$E_{snap} = \sum_{i=0}^{i=s} m_i \cdot E_{save} + E_{res} \quad (1)$$

Here, m_i is the amount of RAM changed and thus saved - from 0 (no updates in RAM) to 1 (entire RAM updated) per

[1]STM32G081xB - Arm Cortex-M0+ 32-bit Microcontroller datasheet. Available at https://www.st.com/resource/en/datasheet/stm32g081rb.pdf

each snapshot i, and E_{save} and E_{res} are, the energy required to save and restore the system state, respectively.

On the one hand, for transient systems that save the entire system state, m_i always equals 1, the total size of the RAM. Therefore, the total cost depends solely on s. On the other hand, the aforementioned techniques (see Sec. II) can ideally reduce m_i but cannot use the excess energy, E_{exc}, for task execution. Consequently, using a hardware module to trace changes in RAM can provide a solution to minimize m_i for every snapshot i and use E_{exc} efficiently, i.e. for task execution. Here, E_{exc} can be defined as

$$E_{exc} = \sum_{i=0}^{i=s} (1 - m_i) \cdot E_{save} \tag{2}$$

When considering hardware solutions such as a memory tracing module, additional overheads that contribute to the total energy must also be considered. These are (a) the energy, E_{trace}, used to power the module and trace changes in RAM while running tasks, and (b) the energy, E_{acq}, to read from the module the RAM that has changed and needs to be saved to NVM at snapshot time. Thus, E_{total} can be rewritten as

$$E_{total} = \underbrace{P_{task} \cdot t_{task} + E_{trace}}_{\text{Energy during active time}} + \underbrace{\sum_{i=0}^{i=s} (m_i \cdot E_{save} + E_{res} + E_{acq})}_{E_{snap}} \tag{3}$$

Since the state retention process must be application-independent, attempting to optimize P_{task} and t_{task} is beyond the scope of this work. Also, E_{res} cannot be improved, as the entire system state must be restored after a power outage. Thus, the justification for designing a memory tracing module lies in the trade-off between E_{exc} and E_{trace}.

Here, E_{exc} directly depends on m_i, which in turn relates to the ability of a memory tracing module to adjust V_h dynamically and hence extend the active time for the task. This feature is not currently available in today's state-of-the-art solutions. In addition, m_i also depends on the tracing granularity of the memory module. For example, tracing changes in RAM at a single-byte level significantly reduces the amount of redundant data when saving state to NVM. Conversely, high tracing granularity comes at the cost of increased area (in terms of hardware) and energy to trace the changes in RAM at a finer resolution. This results in a trade-off between tracing granularity and energy/area, which must be considered at design time and has not been explored in other works.

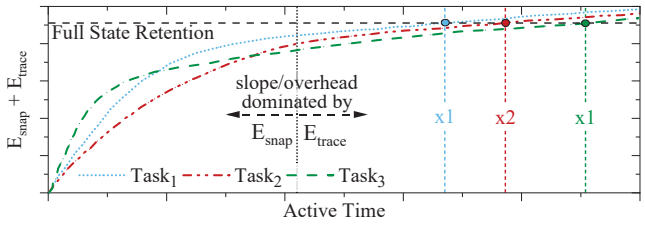

Fig. 1: Illustration of overheads (E_{snap} + E_{trace}) vs active time using a memory tracing approach for processing tasks.

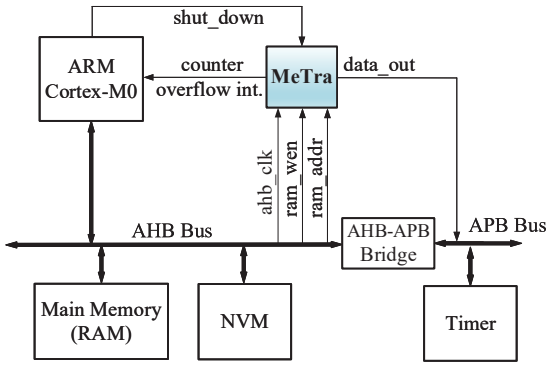

Fig. 2: Block diagram including the ARM Cortex-M0 subsystem with *MeTra*.

Fig. 1 shows a conceptual illustration of how typical tasks ($Task_1$, $Task_2$ and $Task_3$) involving processing operations use RAM, and how this affects saving a snapshot using memory tracing. These tasks may heavily change data in RAM at the beginning, i.e. when they first initialize their variables, but then the changes are less frequent. This behaviour is captured by the change in profile slope in Fig. 1, indicating that RAM changes flatten out over time. This is mainly due to reusing the same RAM locations until the execution of the tasks is completed (spatial and temporal locality). Consequently, using a memory tracing module, while E_{trace} increases linearly with the time the system is active, E_{snap} only increases initially and then flattens out. Here, the vertical dotted black line indicates the limits within which the contribution between E_{snap} and E_{trace} dominates overheads as a function of time; the longer the time the system is active, the more overhead is due to E_{trace} and vice versa. Finally, assuming that the time the system is active is extended significantly, a task can potentially change a more significant portion of the RAM. Therefore, the overhead for RAM tracing may outweigh the benefits of using a memory tracing module over saving the entire system state to NVM. This is captured in Fig. 1 via the cross-over points x_1, x_2 and x_3, showing that these can differ for different tasks depending on how RAM is used.

IV. MeTra Design

This section describes our custom hardware memory tracing module *MeTra* while also taking into account the energy efficiency aspects discussed in Sec. III.

Fig. 2 shows a system-level diagram of an ARM Cortex-M0 architecture (typically used for sensor system applications) incorporating *MeTra*, and Fig. 3 illustrates the general hardware design of *MeTra* based on three main parts: a trace memory, two bit-shift modules and a dedicated counter. *MeTra* operates by "sniffing" RAM base addresses when data is being written and storing this information in the trace memory. In the case of ARM architecture, this is accomplished by accessing the advanced high-performance bus (AHB) used to

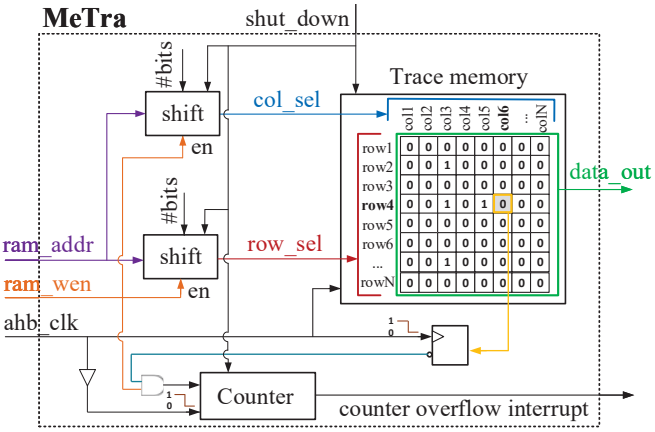

Fig. 3: General hardware design of *MeTra*.

transfer data between the CPU and RAM so that *MeTra* can acquire RAM accesses for write operations and the base RAM address of the location being written. Thus, the AHB address line (`ram_addr`), together with the AHB clock (`ahb_clk`), which synchronises read/write operations, and the control signal (`ram_wen`), which distinguishes RAM writes from other operations, are routed to *MeTra*.

RAM is divided into blocks of equal size, and each block is mapped to a bit of trace memory (tracing granularity in Sec. III). When a RAM write (change in RAM) occurs, *MeTra* locates the corresponding cell in the trace memory and tags the bit-cell from `0` to `1`, indicating that the content in that RAM block has changed. As shown in Fig. 3, the trace memory is a bit addressable, two-dimensional array which requires a set of coordinates to identify the bit to be tagged, in the form of row (`row`) and column (`col`), derived from `ram_addr`. *MeTra* uses two bit-shift modules to convert the RAM base address to these coordinates via `row_sel` and `col_sel`. The CPU can read the trace memory using the advanced peripheral bus (APB) before saving a snapshot, such that only changed blocks of RAM previously mapped into this memory (`data_out`) are saved to NVM at snapshot time.

MeTra uses a counter to count the number of RAM blocks that consequently need to be saved to NVM at snapshot time. At the rising edge of the clock, `row_sel` and `col_sel` are obtained, and the current value of the corresponding bit-cell in the trace memory is first saved in a buffer at the falling edge of the clock, before being updated to `1`. If the buffer value is `0`, it means that the RAM block has not been changed previously, so the counter should take this into account. Therefore, the counter is incremented only when (a) a write operation is enabled (`ram_wen` is `1`) and (b) the bit was not previously changed (buffer value is `0`). This counter is then used to adjust V_h and ensure that changed RAM blocks are reliably saved to NVM at snapshot time.

The value of V_h is stored in the RAM, and therefore to adjust this value *MeTra* uses the counter overflow interrupt to notify the CPU that V_h must be adjusted (this operation typically takes only a few CPU cycles to be processed). Every time a counter overflow happens, meaning that it reaches its limit (i.e. value 0x1F for a 5-bit counter), the overflow interrupt is issued. The maximum value of this counter, c_{max}, is based on the size of the RAM block, the total size of the RAM and the intended number of $V_{h,i}$ steps. The value of $V_{h,i}$, at each counter overflow interrupt, i, is updated based on

$$i \cdot c_{max} \cdot E_{block} = C_{store} \cdot \frac{{V_{h,i}}^2 - {V_{sys,min}}^2}{2} \quad (4)$$

where E_{block} is the energy required for saving a single RAM block to NVM, while $V_{sys,min}$ is the minimum system voltage. E_{block} can be obtained experimentally depending on the NVM technology and the size of the RAM block. Eq. 4 can be implemented in the form of a look-up table that can be used at run-time to update V_h. When deciding the size of the RAM block, which in turn affects the size of the trace memory and the counter, it is necessary to take into account the CPU architecture. For example, a 32-bit microcontroller (MCU) would require data to be written to RAM in four-byte blocks, and thus minimum trace granularity would be one bit to be mapped to a four-byte RAM block. Lower tracing granularity (larger block size mapped to a single bit), which would minimize hardware area and energy costs, can be effective with an application that uses RAM locations consecutively (spatial locality). Otherwise, higher granularity would be required.

MeTra also includes a shutdown option (`shut_down`), therefore, allowing the CPU to shut down *MeTra* when the overhead E_{snap}+E_{trace} outweighs the benefits of using *MeTra* over saving the entire system state (discussed in Sec. III). Fig. 4 shows the execution flow between *MeTra* and the CPU for each EH cycle (excluding the system state restore process, which does not involve *MeTra*). This also includes shutting down *MeTra* when the CPU issues an interrupt, after detecting that a crosspoint has been reached. Crosspoint checking is done on the CPU whenever *MeTra* sends the counter overflow interrupt. This interrupt is therefore used to: (a) adjust V_h, and (b) calculate the energy overhead (E_{snap}+E_{trace}) based on the

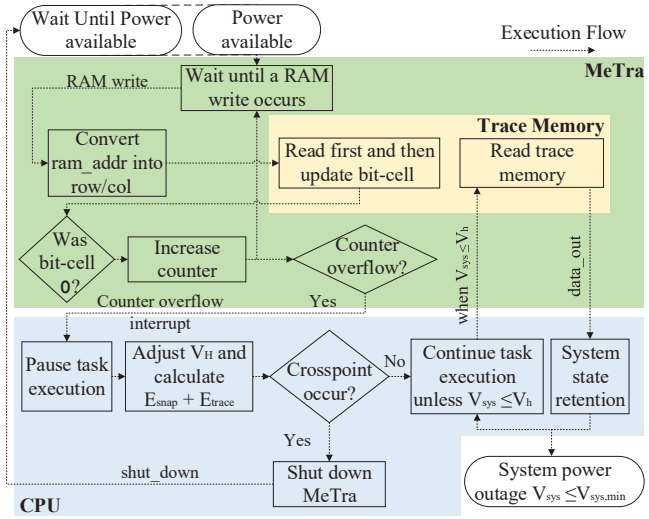

Fig. 4: Execution flow between *MeTra* and ARM Cortex-M0.

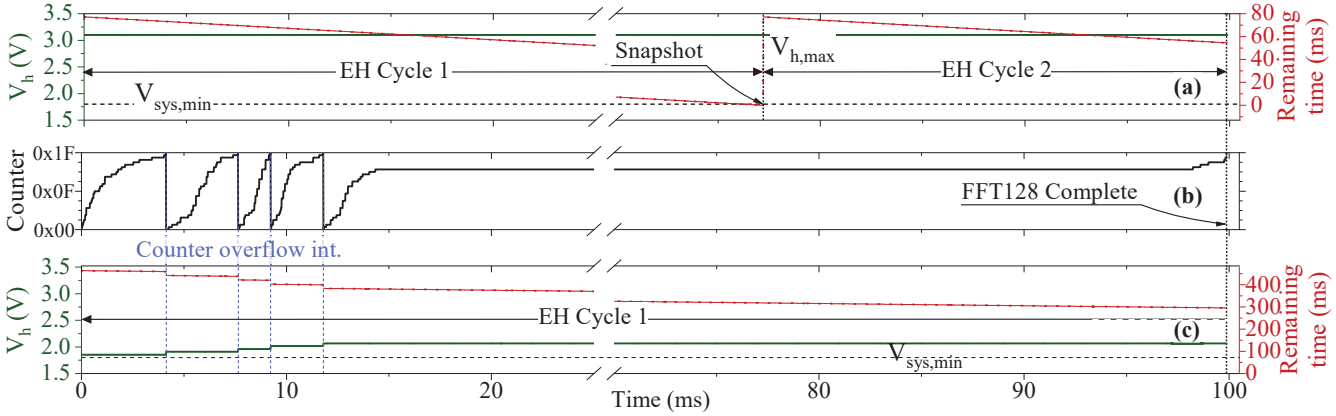

Fig. 5: Behaviour of *MeTra* when running FFT128, with RAM block size of 4 bytes, f_{source} of 2Hz and Flash as NVM.

current value of V_h as well as the active time measured by a timer (see Fig. 2), to determine if the overhead is greater than retaining the entire state. If so, the CPU can use an interrupt (`shut_down`) to shut down *MeTra* immediately afterwards until the next EH cycle. Otherwise, upon reaching V_h, the CPU reads the trace memory to determine which RAM blocks need to be saved in NVM, thus starting the state retention process.

V. Evaluation

This section analyzes the performance and benefits of using *MeTra* considering different NVM types (Flash or FRAM), RAM block sizes (4, 8, 16 and 32 bytes with total RAM of 4KB), and processing tasks performed on the ARM Cortex-M0. These are as follows: (1) Advanced Encryption Standard (AES128), a 128-bit encryption algorithm generally used to protect classified information; (2) Cyclic Redundancy Check (CRC32), a 32-bit checksum generation algorithm typically used for error detection in communication; and (3) Fast Fourier Transform (FFT128), a 128-bit time-domain to frequency-domain transform algorithm used in signal processing.

The energy (time) required to complete each task (profiled on an ARM Cortex-M0), E_{task} and t_{task}, is 73μJ (100ms) for FFT128, 60μJ (82ms) for AES128 and 47μJ (64ms) for CRC32, where the maximum voltage $V_{sys,max}$ is 3.2V, and minimum voltage $V_{sys,min}$ is 1.8V. We emulate an EH cycle with a variable frequency synthesised source, where the amplitude is 3.2V, and the interruption frequency, f_{source}, can vary from 2Hz to 20Hz for testing purposes. The active time spent executing the task before saving a snapshot for each EH cycle depends on f_{source} and $V_{h,}$; therefore with a constant value of f_{source}, the active time depends exclusively on V_h.

We first performed the tasks without using *MeTra* to obtain the maximum value of $V_{h,max}$ to save the entire system state to NVM, considering both Flash and FRAM. Here, $V_{h,max}$ is 3.1V for Flash using a C_{store} of 100μF to ensure the entire system state is reliably saved with a profiled E_{snap} of 328μJ. In contrast, $V_{h,max}$ is 2.6V for FRAM using a C_{store} of 10μF, and E_{snap} of 17μJ. We can state that retaining the entire system state requires a significant amount of energy compared to E_{task} for all tasks with both types of NVM, reducing the active time and the system's overall energy efficiency. We then profiled our tasks using *MeTra* to evaluate the benefits in terms of additional active time per EH cycle based on adjusting V_h.

Fig. 5 shows the FFT128 as an example, first performing without *MeTra* and then with *MeTra*, with a constant f_{source} of 2Hz and using Flash as NVM. In Fig. 5(a) (without *MeTra*), V_h is constant over time ($V_{h,max}$ = 3.1V, green line), and the remaining active time (outstanding time until the end of the EH cycle, red line) decreases linearly until it reaches 0, when a snapshot is saved. To complete the task, one snapshot is required. Here, we do not show the time between a power outage and power recovery. We then illustrate the case with *MeTra* in Fig. 5(b), where, whenever the counter overflow interrupt is issued (with a 4-byte RAM block size, a 5-bit counter is required with maximum value 0x1F), the value of V_h is adjusted using Eq. 4, as well as the remaining active time until a snapshot is saved, if needed (Fig. 5(c)). Contrary to the case without *MeTra*, the FFT128 completes within one EH cycle without requiring snapshots and with a remaining active time of 300ms after completion. Over time, the number of RAM blocks changed increases in the first part but then remains constant. This behaviour is because the FFT128 uses the same RAM blocks after the first 14ms, which are then periodically updated over time. A similar behaviour can be observed with FRAM and different values of f_{source}.

Fig. 6 shows the total active time (y-axis) profiled per EH cycle as a function of f_{source} (x-axis) without *MeTra* (marked as "default") and with *MeTra* while performing FFT128, CRC32 and AES128, for both FRAM (top) and Flash (bottom). Here, the active time per EH cycle is extended between 25.8% and 92.2% for the FRAM-based system, while the system featuring Flash shows an increase that ranges from 5x to 17x. The reason for this significant increase is the reduction of V_h from 3.1V (fixed $V_{h,max}$) to a maximum of 2.065V for the Flash and 1.975V for FRAM. Therefore, thanks to *MeTra*, the active time per EH cycle has significantly increased. As a result, the system may reduce the number of snapshots required during task execution or complete the task without saving any snapshots. Thus, the total energy required

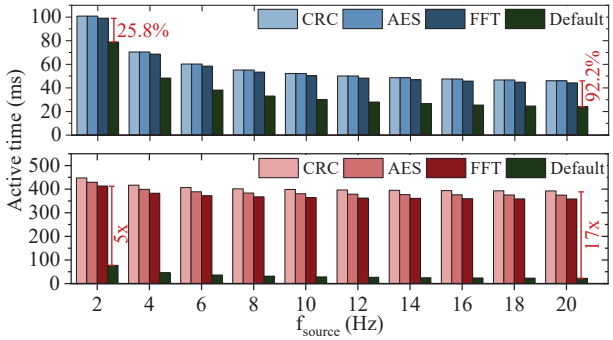

Fig. 6: Total active time per EH cycle with varying f_{source} from 2Hz to 20Hz, using FRAM (top) and Flash (bottom).

to retain the state until the task is completed is significantly reduced. This is shown in Fig. 7, where E_{snap} is profiled with and without *MeTra* while performing AES128, CRC32 and FFT128, with f_{source} equal to 20Hz, which is the worst case for *MeTra*. Here, E_{snap} is significantly lower when using *MeTra* for two main reasons: reduced number of snapshots and reduced energy $m_i \cdot E_{save}$ per each snapshot. In particular, the energy spent for saving the state is reduced between 15x-20x, compared to not using *MeTra*, across all benchmarks.

Finally, Table I shows the power consumption and hardware area (and relative overheads) of *MeTra* with different block sizes with a total RAM of 4KB. It can be observed that the area overhead, compared to the total design area of the ARM Cortex-M0 (0.15mm^2), ranges between 2.48%-12.21%, while the additional power consumption of *MeTra* ranges between 16.1μW-19.3μW; an overhead of 2.20%-2.63%, when compared to the power consumption of the ARM-Cortex-M0 (730μW power consumption at 8MHz, without peripherals).

VI. Conclusion

This paper proposes a custom hardware module, *MeTra*, that traces changes in the main memory between power outages for efficient system state retention. *MeTra* allows the state retention process to be activated dynamically by adjusting V_h) at run-time based on the energy requirement of each snapshot. This maximises the system's active time and the energy available to perform tasks. *MeTra* was integrated to a low-power embedded architecture (ARM Cortex-M0), and its functionality was verified using various benchmarks. Experimental results show that active time within an EH cycle can be

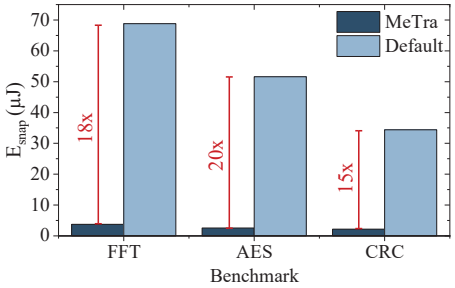

Fig. 7: E_{snap} while performing FFT128, AES128 and CRC32 with $f_{source} = 20$Hz (worst case) and using FRAM as NVM.

RAM Block	Power (μW)	Area (μm^2)
4	19.3 (2.63%)	18329 (12.21%)
8	18.4 (2.51%)	9364 (6.24%)
16	17.1 (2.33%)	5509 (3.67%)
32	16.1 (2.20%)	3716 (2.48%)

TABLE I: Power and area overheads for *MeTra* with different RAM block sizes.

extended up to 17x (Flash-based) and 92.2% (FRAM-based), compared to saving the entire system state. *MeTra* incurs an area overhead of 2.48%-12.21% (assuming a RAM of 4kB), while its power consumption is 19.3μW (2.63% overhead) for the highest tracing granularity (4-byte RAM block).

Acknowledgment

This work was supported by the UK EPSRC grants EP/W022877/1, EP/P010164/1 and EP/K034448/1. Experimental data can be found at DOI:10.5258/SOTON/D2596 (https://doi.org/10.5258/SOTON/D2596).

References

[1] T. Becker *et al.*, "Energy harvesting for a green internet of things," PSMA, NJ, USA, White Paper, 2021.

[2] D. Balsamo *et al.*, "A control flow for transiently powered energy harvesting sensor systems," *IEEE Sensors Journal*, vol. 20, no. 18, pp. 10 687–10 695, 2020.

[3] B. Ransford *et al.*, "Mementos: System support for long-running computation on RFID-scale devices," in *ASPLOS XVI*. New York, USA: Association for Computing Machinery, 2011, p. 159–170.

[4] A. Rodriguez Arreola *et al.*, "Approaches to transient computing for energy harvesting systems: A quantitative evaluation," in *ENSsys'15*. New York, USA: Association for Computing Machinery, 2015, p. 3–8.

[5] U. Senkans *et al.*, "Applications of energy-driven computing: A transiently-powered wireless cycle computer," in *Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ser. ENSsys'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–7.

[6] N. A. Bhatti and L. Mottola, "HarvOS: Efficient code instrumentation for transiently-powered embedded sensing," in *IPSN'17*. New York, USA: Association for Computing Machinery, 2017, p. 209–219.

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

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

[9] H. Jayakumar *et al.*, "Quickrecall: A HW/SW approach for computing across power cycles in transiently powered computers," *J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 1, aug 2015.

[10] T. D. Verykios *et al.*, "Exploring energy efficient state retention in transiently-powered computing systems," *Proceedings of the IDEA League Doctoral School on Transiently Powered Computing*, 2017.

[11] S. Ahmed *et al.*, "A survey on program-state retention for transiently-powered systems," *Journal of Systems Architecture*, vol. 115, 2021.

[12] N. A. Bhatti and L. Mottola, "Efficient state retention for transiently-powered embedded sensing." in *EWSN'16*, 2016, pp. 137–148.

[13] T. D. Verykios *et al.*, "Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of nvm technologies," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 167 – 178, 2019.

[14] S. T. Sliper *et al.*, "Efficient state retention through paged memory management for reactive transient computing," in *DAC'19*. New York, USA: ACM, 2019, p. 6.

[15] D. Pala *et al.*, "Freezer: A specialized nvm backup controller for intermittently powered systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1559–1572, 2021.

[16] A. Rodriguez Arreola *et al.*, "Federated time persistency in intermittently powered IoT systems," *Journal of Systems Architecture*, vol. 130, p. 102667, 2022.