DOI: 10.1109/TSUSC.2025.3621509
This is the author's accepted manuscript. © 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.

Enhancing State Retention with Energy-Efficient Memory Tracing in Intermittent Systems

Osama Bin Tariq[1*], Theodoros D. Verykios[2], Geoff V. Merrett[2], Domenico Balsamo[1]
[1]MicroSystems Research Group, School of Engineering, Newcastle University, Newcastle, UK.
[2]School of Electronics and Computer Science, University of Southampton, Southampton, UK.
[*]osama.bin-tariq@newcastle.ac.uk

***Abstract*—Intermittent systems powered by harvested energy frequently encounter power outages, requiring efficient mechanisms to save and restore their internal state, to ensure computational progress. In these systems, minimising the overhead of state retention, comprising CPU core registers and main volatile memory contents (a snapshot), is essential to maximise computational progress within the constraints of limited energy availability. This paper introduces a hardware module, *MeTra* (MEmory TRAcing), designed to enhance energy-efficient state retention in intermittent systems. This is achieved by tracing and selectively saving the modified parts of the main volatile memory (RAM) to non-volatile memory (NVM), i.e. FRAM. Additionally, *MeTra* dynamically adjusts the voltage threshold that initiates state saving, optimizing energy usage for each snapshot and enabling the system to dedicate more harvested energy to useful computations. *MeTra* was integrated with an Arm Cortex-M1 processor on an FPGA and evaluated using benchmarks including matrix multiplication, array sorting, and advanced encryption standard (AES), demonstrating its effectiveness in reducing state-saving time and improving energy efficiency by selectively saving modified RAM regions instead of the entire memory. Experimental results demonstrate that snapshot time can be reduced by 48.34% to 57.56% in FRAM-based systems, leading to an improvement in energy efficiency of 65.24% to 77.76%. These gains are achieved by selectively saving 5.66% to 19.82% of RAM, using *MeTra*, compared to saving entire RAM.**

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

I. Introduction

The rise of intermittent systems has opened up a range of applications, extending its reach from extreme environments such as satellites to advanced innovations in medical implants and wearable devices and sensing systems [1]–[3]. Intermittent systems can harvest energy from a range of sources, including solar power, radio-frequency signals, microbial fuel cells, and structural vibrations, storing it in dedicated energy storage units like small capacitors or supercapacitors [4]–[7].

Examples of such systems include WISP [5], Flicker [8], Camaroptera [4], and Empire [7]. The energy storage typically holds a limited amount of energy used to power a microcontroller unit (MCU), which includes a main processor (CPU), main volatile memory (RAM) and non-volatile memory (NVM), as well as various sensors and other peripherals. This presents a dual challenge: first, the energy consumed by these components often depletes the energy storage before the application can be fully executed, and second, the variability and unreliability of energy harvesting (EH) sources results in an intermittent power supply, requiring the system to remain shutdown during the charging phase. As a result, the most efficient operation of intermittent systems involves cycles of charging, progressing the application, and shutting down, a process known as intermittent computing.

Different studies on intermittent computing highlight the challenge of ensuring computational progress while retaining the internal state of the MCU during power outages [9]–[12]. Power outages reset volatile components within the MCU, including core registers and main volatile memory, i.e., RAM. Without state retention, computation cannot resume after an interruption. Software-based approaches for state retention have been widely explored in the literature, including methods such as Mementos [13], HarvOS [14], Hibernus [15], Hibernus++ [10], and QuickRecall [16]. However, these approaches generally save the entire internal state, including core registers and all of RAM, without differentiating between "valuable" and "redundant" data. This leads to inefficient utilization of NVM. In this context, "valuable" data refers to the contents of RAM that have been modified between power outages, whereas "redundant" data refers to unchanged content.

Saving the entire internal state during each power outage reduces active time and decreases energy efficiency. To address this, various strategies for selective state retention have been proposed to minimize the amount of RAM saved in NVM [17], [18]. However, most of these strategies are software-based and rely on the CPU to trace changes in RAM, which limits the CPU's availability for application progress. Moreover, although these approaches can trace changes in RAM, they lack the ability to dynamically adjust the voltage threshold that initiates state saving. We provide a detailed discussion on the current state of research regarding selective state retention strategies in Sec. II.

In this paper, we present a novel system-level approach for energy-efficient state retention using a custom hardware module, *MeTra* (MEmory TRAcing). This module efficiently traces changes in RAM between power outages, reducing the amount of internal state that must be saved before an upcoming power outage. *MeTra* allows dynamic adjustment of the voltage threshold that initiates state saving based on the amount of valuable RAM that needs to be saved in NVM. As a result, the system's active time for running the application is extended, leading to improved overall energy efficiency.

MeTra was designed, implemented, and seamlessly integrated with the system, which features an Arm Cortex-M1 CPU [19] that serves as the soft processor, along with RAM and NVM, using a Field-Programmable Gate Array (FPGA). This is the first work to demonstrate a memory tracing approach combined with a dynamic voltage threshold mechanism for efficient state retention, avoiding redundant memory saving, extending computation time, and validating

the system in hardware to provide practical insights into its feasibility and energy behaviour.

The novel contributions of this paper are:

- An analysis of state retention in intermittent systems, focusing on memory tracing trade-offs to optimize energy consumption (Sec. III);
- Development of *MeTra*, a memory tracing module that traces CPU-RAM data writes, saving only modified RAM regions and CPU registers before power loss. This extends the active time by dynamically adjusting the voltage threshold for state retention before shutdown (Sec. IV);
- Implementation of *MeTra* as an IP block integrated with a low-power CPU, the Arm Cortex-M1, on an FPGA (Sec. IV), and its evaluation using three benchmarks—matrix multiplication, array sorting, and AES—to analyze state-saving time reduction and energy efficiency improvement (Sec. V).

II. Background and Motivation

In this section, we review existing techniques for selective state retention designed to address the challenges of intermittent computing, while also outlining the motivation for our proposed approach.

A comprehensive survey of state retention techniques is provided by Singla and Sarangi in [20]. Depending on the control strategy, the internal state of the MCU can be periodically saved with varying granularity, such as after task completion (checkpoint), or saved on demand when a voltage threshold is crossed. In the former scenario, applications can be divided into smaller tasks, with each completed within a single power cycle, allowing a snapshot to be retained in NVM before proceeding to the next task [14], [21]. However, this requires sizing the energy storage to accommodate the most energy-intensive task to ensure reliable execution, rendering the approach application-dependent.

An example of this strategy is HarvOS [14], employing automated code instrumentation to insert checkpoints to save the internal state at strategically optimal locations within the main application. Its adaptive checkpoint placement dynamically adjusts the checkpointing frequency based on the available energy, increasing frequency during low-energy conditions and reducing it when energy is abundant. Through compile-time analysis, HarvOS identifies critical code sections to optimize checkpointing, while its energy-aware execution strategy ensures effective utilization of harvested energy. These features enable it to reduce checkpointing overhead and enhance computational progress under intermittent power conditions. However, the system does not optimize energy usage during runtime and incurs redundancy by saving state repeatedly whenever a checkpoint for system state retention is triggered.

Majid *et al.* [21] proposed Coala, an adaptive task-based execution model that dynamically adjusts task sizes based on available energy, to ensure forward progress in EH devices by coalescing or splitting tasks at runtime. While their approach addresses many limitations of static task-based checkpointing systems, Coala-based systems require the use of the Coala API and significant software adjustments, such as converting code into tasks, sequencing control flow between tasks, and annotating RAM accesses to task-shared data.

In contrast, on-demand schemes, i.e., saving internal state when the voltage threshold is reached, eliminate the need to complete tasks within a single power cycle, as this state is saved before an outage occurs [22]–[25]. This allows task execution to extend across multiple cycles, making it well-suited for tasks involving extensive processing operations.

Within on-demand schemes, various software-based techniques have been introduced to minimise the amount of internal state saved in NVM. For instance, Bhatti and Mottola [22], proposed selective state retention which is achieved by identifying unused RAM (free space) and saving only the allocated portions to NVM, i.e., flash memory. However, in modern sensor applications, most of the RAM is typically used by the main application, leaving little free space to optimise. Furthermore, this approach does not consider data in RAM that remains unmodified across power cycles, leading to redundant writes to NVM as previously saved data is unnecessarily overwritten. The authors also incorporate heap tracing, even though dynamic memory allocation is generally discouraged in embedded systems due to the risk of memory fragmentation and allocation failures [26], and is explicitly prohibited by coding standards like MISRA C [27]. Finally, the technique has been applied exclusively to Flash-based systems, where its benefits are constrained by the high erase cost.

Another on-demand approach, named *ManagedState*, is presented by Sliper *et al.* [23], which is a page-based memory manager that traces RAM and monitors modified data during runtime. However, this approach relies heavily on the CPU to trace changes in RAM using a specific set of APIs. Furthermore, the page size limits the granularity of tracing, which can result in redundant data being saved to NVM, mainly when the application's data exhibits temporal and spatial locality.

In a different on-demand strategy, Verykios *et al.* [24] proposed a suite of software techniques, including *Multiple Allocated State Images*, *Updated Blocks*, and *Multiple Updated Blocks*, designed to enhance the efficiency of state retention by leveraging the unique properties of different NVM technologies, such as symmetry and erasure requirements. These methods provide more tailored and optimised solutions for state retention management. However, a significant limitation lies in the delayed identification of unallocated or modified RAM, which only occurs once a preset voltage threshold that initiates state saving is reached. This prevents the system from making full use of any excess energy available prior to reaching that threshold.

Pala *et al.* [25] introduced an initial hardware-based approach with their backup controller, *Freezer*, which captures a reduced snapshot of the internal state during power outages by monitoring RAM accesses and committing the changes to NVM. However, this solution was only simulated in isolation and did not demonstrate measurable system-level benefits. Moreover, *Freezer* lacks dynamic threshold voltage adjustment, resulting in inefficient use of excess energy despite the reduced snapshot size.

We believe that an effective solution should: (a) provide an adjustable voltage threshold to initiate state saving without

TABLE I: Definitions of energy and power parameters for task execution and state saving.

Notation	Description
E_{task}	Energy needed to complete one task iteration
C_{store}	Capacitor that stores energy during operation and powers the system to save its state during power interruptions
P_{task}	Average power consumption during task execution
t_{task}	Time required for task completion
V_{th}	Voltage threshold that initiates internal state saving
$V_{th}[i]$	Voltage threshold after i^{th} overflow interrupt
$V_{sys,min}$	Minimum voltage required to maintain regular system operation
$V_{sys,max}$	Maximum voltage the system can handle safely during regular operation
E_{total}	Total energy per task, when energy for snapshots for saving internal state is also taken into account
E_{snap}	Energy overhead for saving and restoring the internal state
N	Number of power outages
E_{block}	Energy needed to save one block of RAM. Number of bytes in one block are defined by granularity setting
E_{save}	Energy needed for saving the internal state
E_{res}	Energy needed for restoring the internal state
E_{trace}	Energy needed to power the memory tracing module and trace changes in RAM during task execution
E_{acq}	Energy required by the CPU to acquire RAM changes from the tracing module at the snapshot time
E_{excess}	Excess energy saved from not saving the entire RAM (e.g., 75% excess if only 25% RAM is saved)
α_i	Fraction of RAM changes, ranging from 0 (no RAM update) to 1 (full RAM update) for each snapshot i
K_{max}	The maximum counter value for counting memory blocks that have changed, as observed by *MeTra*

significantly compromising the system's active time, (b) enable fine-grained RAM tracing to minimize redundant writes to NVM, and (c) maintain consistent performance across different application workloads, regardless of the spatial and temporal locality of the RAM.

To address these challenges, we propose *MeTra*. This energy-efficient, system-level memory tracing module offers fine-grained RAM tracing and dynamically adjusts the voltage threshold to initiate state saving. Our proposed solution minimizes CPU time spent on state retention while maximizing the active time of the application. The subsequent sections detail each component of *MeTra* and its integration with a low-power CPU, specifically the Arm Cortex-M1.

III. Analysis of State Retention Process

This section first discusses the importance of a hardware-based approach to tracing RAM changes for efficient state retention. It then investigates the state retention process and its impact on energy efficiency. Our focus lies on tasks involving substantial processing operations, e.g., matrix multiplication, array sorting, and AES, which may span multiple EH cycles and necessitate multiple state retentions. Table I provides a summary of the mathematical terms and definitions for energy and power parameters referenced throughout the paper.

The energy, E_{task}, needed to complete one iteration of each task with a stable power supply and assuming no interruptions can be expressed as

$$E_{task} = P_{task} \cdot t_{task}, \tag{1}$$

where P_{task} represents the average power consumption during task execution, and t_{task} denotes the time required for completion. If power outages occur before task completion, the snapshot of the internal state must be retained in NVM. Thus, the total energy per task is given by $E_{total} = E_{task} + E_{snap}$, where E_{snap} represents the energy overhead for saving and restoring the snapshot of internal state N times, with N denoting the number of power outages.

In the scenario where only modified RAM regions are saved within a single EH cycle, the fraction of RAM changes per snapshot, denoted as α, ranges from 0 (no updates) to 1 (full RAM update). Given that E_{save} and E_{res} represent the energy required for state saving and restoring, respectively, the energy per snapshot, E_{snap}, can be expressed as:

$$E_{snap} = \sum_{i=0}^{i=N} (\alpha_i \cdot E_{save} + E_{res}). \tag{2}$$

For intermittent systems that retain the entire internal state, α_i consistently equals 1, corresponding to the total RAM size. Hence, the overall cost is solely reliant on the number of power outages, N.

Note, however, that Eq. 1 and Eq. 2 do not take into account the overheads incurred by either software methods, such as additional CPU cycles needed to trace changes in the memory, or hardware overheads, such as the energy needed to power the memory tracing module. In hardware-based approaches, the overheads contributing to the total energy consumption include (a) the energy needed to power the memory tracing module that traces changes in RAM during task execution, E_{trace}, and (b) the energy required by the CPU to acquire the information from the memory tracing module about memory locations that have changed in RAM, E_{acq}.

Thus, taking into account the energy overheads related to powering the memory tracing module and reading data from the module to the CPU, Eq. 1 and Eq. 2 can be updated accordingly to calculate the total energy, E_{total}, as:

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

Given that the state retention process should be application-independent, efforts to optimize P_{task} and t_{task} are beyond the scope of this study. Similarly, the improvement of energy needed for restoring the internal state from NVM to RAM and core registers, E_{res}, is not considered, as restoring the entire internal state is essential after a power outage. Consequently, the rationale for developing a memory tracing module is to utilize the excess energy saved by avoiding full RAM

saves during power outages (E_{excess}) while accounting for the energy required to operate the tracing module (E_{trace}).

These equations represent an idealised energy model to illustrate the conceptual relationship between system parameters. In practice, factors like conversion inefficiencies, leakage, and environmental effects also influence energy use. While not explicitly modelled, these are implicitly reflected in the empirical evaluation in Sec. V-F.

Methodologies described in Sec. II aim to reduce the fraction of RAM changes that need to be saved in NVM (α_i), but cannot leverage excess energy, E_{excess}, for computational progress, as they lack the ability to adjust the voltage threshold initiating state saving dynamically, V_{th}. Consequently, the adoption of a hardware module for monitoring RAM changes combined with a dynamically adjustable V_{th}, offers a viable approach to utilize the fraction of RAM changes, α_i, for each snapshot i and utilize excess energy, E_{excess}, efficiently, i.e., for task execution. E_{excess} can be defined as

$$E_{excess} = \sum_{i=0}^{i=N} (1 - \alpha_i) \cdot E_{save}, \tag{4}$$

where E_{excess} is directly influenced by α_i, which enables dynamic adjustment of V_{th}, thereby extending a task's active time. This capability, absent in current state-of-the-art solutions, underscores the significance of such a module. Furthermore, α_i is dependent on the tracing granularity of the memory-tracer hardware module. For instance, tracing RAM changes at a single-byte level substantially reduces redundant data when saving state to NVM. But finer tracing granularity means increased hardware area and energy consumption for finer RAM change tracing. Thus, a trade-off exists between tracing granularity and energy/area, which needs careful consideration during design, an aspect overlooked in existing literature.

During initialisation, tasks induce substantial changes, followed by a reduction in change frequency. This behaviour is further discussed in Sec. V-D, where the reduction in RAM usage over the application's runtime is discussed. Such behaviour primarily arises from the reutilization of the same RAM locations until task execution concludes, demonstrating spatial and temporal locality.

With the integration of a memory tracing module, the energy expenditure, E_{trace}, increases linearly with the time the system is in an active state, while the energy for snapshots, E_{snap}, increases initially before stabilising. The longer the system is active, the more overhead is due to E_{trace}, and vice versa. In the long run, if the system remains active for an extended period and, depending on the application in use, a task consumes a significant portion of the RAM. Consequently, the overhead of RAM tracing may outweigh its benefits compared to simply saving the entire internal state to NVM, as discussed in Sec. V-C.

In intermittently-powered systems, the key objective is to retain the system state reliably during power outages. This must be achieved under constraints such as limited energy, minimal hardware resources, and unpredictable energy availability. The goal is to minimize the overhead of state-saving so that the available energy can be used efficiently for computational progress, while ensuring consistency upon recovery. This should be done with minimal architectural changes and adaptability to varying RAM sizes and application behaviours. The following section introduces our solution, *MeTra*, which addresses these challenges through selective, energy-efficient memory tracing.

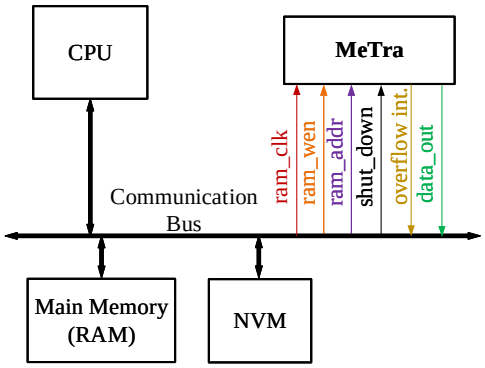


Fig. 1: System block diagram, showing how *MeTra* interfaces with the system bus.

IV. *MeTra* Design and Implementation

In this section, we begin by outlining the principle behind retaining the internal state. Next, we introduce our design and the internal functionality of *MeTra*, explaining how it facilitates selective state retention. Finally, we go into the details of the FPGA implementation of the system and the integration of *MeTra* as an IP component.

MeTra is designed for compact, intermittently-powered embedded systems that typically have limited RAM. However, it scales naturally to systems with larger RAM by proportionally adjusting the trace memory size, which depends on total RAM and tracing granularity. Notably, *MeTra* requires no changes to the CPU architecture, as it connects to the system bus as an independent peripheral, ensuring compatibility with a broad range of microarchitectures.

A. Principle of State Retention

1) The system uses *MeTra* to trace changes in RAM, monitoring which memory blocks have been modified during execution;
2) To maximize active time for system applications, V_{th} is dynamically adjusted based on the number of memory blocks modified. By tailoring V_{th} to the workload, the system can delay state-saving operations and extend its active runtime;
3) When the supply voltage drops below V_{th}, the system initiates saving the internal state. This process ensures that both core registers and modified RAM content are securely saved in NVM (snapshot). After the snapshot is complete, the system transitions into low-power mode or shuts down entirely;
4) Once power is restored, the system resumes operation by restoring the saved internal state. The snapshot is copied back from NVM to RAM, and the core registers

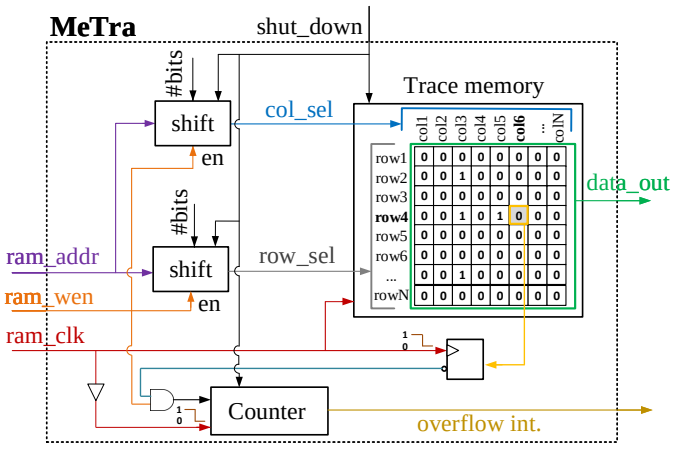


Fig. 2: *MeTra* internal architecture.

are reloaded, enabling the system to continue execution from the exact point where it left off.

B. Design of MEmory TRAcing unit (MeTra)

Fig. 1 illustrates the system-level block diagram of the FPGA-based design, which includes CPU, RAM, NVM, and *MeTra*. It also comprises an on-chip synchronous communication bus, serving as the primary communication backbone. *MeTra* monitors CPU-RAM interactions by tracing address and write-enable signals during write operations. The CPU communicates with *MeTra* via memory-mapped registers over the bus, allowing configuration and trace data retrieval.

Fig. 2 depicts *MeTra*'s internal architecture, which comprises three primary components: a trace memory, two bit-shift modules, and a dedicated counter. *MeTra* interfaces with the RAM through the RAM address bus (*ram_addr*), RAM write-enable signal (*ram_wen*), and RAM clock (*ram_clk*) inputs to monitor RAM changes. These changes are logged in the trace memory that the CPU can access via data output signal (*data_out*). Additionally, the counter is used to adjust V_{th}, reflecting the number of RAM changes that occur between power outages. The trace memory is a bit-addressable, two-dimensional array, where each bit is identified using a pair of coordinates—row (row) and column (col). These coordinates are derived from the RAM address bus (*ram_addr*). To facilitate this conversion, *MeTra* uses two bit-shift modules, which generate the row and column coordinates through the row select (*row_sel*) and column select (*col_sel*) signals.

To manage memory tracing, we introduce a granularity parameter in *MeTra*, which defines how many bytes of RAM correspond to a single bit in the trace memory. This mapping, determined by the granularity parameter, is referred to as the *RAM block size*.

A coarse-grained tracing granularity (a larger RAM block size per bit) reduces hardware and energy costs, making it suitable for applications with a high spatial locality in RAM usage. In contrast, a fine-grained tracing granularity (a smaller RAM block size per bit) is beneficial for applications with low spatial locality in RAM utilization. Two granularity settings were used: 4 bytes and 8 bytes. A granularity of 4 bytes means that each 4-byte RAM block is mapped to 1 bit in the trace memory; similarly, a granularity of 8 bytes maps every 8-byte RAM block to 1 bit. For example, an 8 KB RAM with a block size of 4 bytes (granularity = 4) would require 256 bytes of trace memory, whereas using a block size of 8 bytes (granularity = 8) would reduce the trace memory requirement to 128 bytes. Since our CPU has a 32-bit architecture, the minimum feasible granularity is 4 bytes, corresponding to the CPU's native word size. Thus, 4 bytes of RAM are traced per bit in the trace memory. When a write operation occurs in RAM, *MeTra* identifies the corresponding cell in the trace memory and changes the bit from 0 to 1, indicating that the data in that particular RAM block has been modified.

MeTra utilizes a counter to trace the number of RAM blocks that must be saved to NVM during a snapshot. This count is essential to dynamically adjust V_{th} required to trigger a snapshot, ensuring all necessary RAM blocks are reliably saved. After determining the *row_sel* and *col_sel* coordinates, the current value of the corresponding bit-cell in the trace memory is checked and updated to 1. If that value was previously 0, it indicates that the RAM block has not been modified before. Consequently, the counter increments only under two conditions: (a) when a write operation is enabled (*ram_wen* is 1) and (b) the bit was previously 0. This counter is then used to fine-tune V_{th}, ensuring the reliable saving of modified RAM blocks to NVM during snapshots. An interrupt is triggered in the event of a counter overflow, prompting the CPU to adjust V_{th} (*overflow int.*).

MeTra operates in parallel with the CPU and memory system, introducing no time overhead during normal program execution, as the trace memory is not accessed until an interrupt is triggered when the system voltage, V_{sys}, drops below V_{th}, initiating state retention. While *MeTra* consumes energy when active, this overhead is minimized by shutting it down once the crosspoint is reached. Time overhead is only incurred during the save phase, proportional to the number of modified RAM blocks, and during the restore phase.

The code in Listing 1 illustrates how consecutive memory locations are written one after another starting at address 0x60000010, and can be seen together with Fig. 3 which shows Vivado's Integrated Logic Analyzer (ILA), illustrating *MeTra*'s internal operation. *MeTra* monitors these accesses and decodes the target address into internal row and column indices, setting the corresponding bit in the trace memory, and incrementing a counter on first-time accesses. Fig. 3a shows the case where the values are written to previously unwritten RAM locations. When *ram_wen* goes high, a write to RAM occurs. Therefore, *MeTra* translates the RAM address into *row_sel* and *col_sel* signals, which identify the corresponding bit in the trace memory. If the bit was previously unset, it is then marked, and the counter is incremented to reflect a new region of RAM that must be saved at snapshot time. At the start, *row_sel*, *col_sel*, the trace memory, and the counter are set to zero. As the write operations starts (Listing 1), the pointer index is first written to RAM, which is traced and mapped to row 43 and column 16 in the trace memory.

Since this address is being written for the first time, the trace bit is set and the counter increments from 0 to 1. The next RAM write occurs at location of memory 0x60000010,

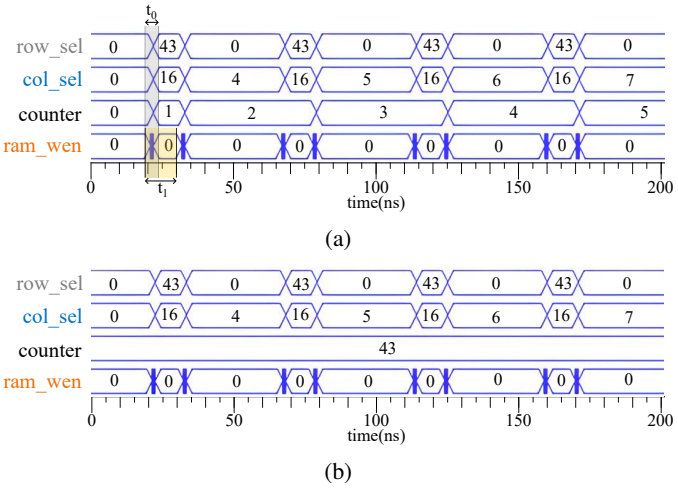


Fig. 3: Vivado Internal Logic Analyzer (ILA) snapshot: (a) new RAM writes, (b) writes to previously used locations.

which maps to row 0 and column 4. As this address was also previously unwritten, the counter increments again. This pattern continues, with the counter updating whenever a new RAM location is accessed. However, each subsequent update to the pointer index writes to the exact location (row 43, column 16). Since the corresponding bit is already set in the trace memory, the counter does not increment, reflecting *MeTra*'s design to avoid redundancy. The system clock period is 10 ns. A RAM write is signalled by asserting *ram_wen* for 1 clock cycle. Note that the assertion time of *ram_wen* and the settling time of *row_sel* and *col_sel*, t_0, are significantly shorter than the RAM's write cycle t_1 (as can be seen in Fig. 3, $t_0 < t_1$). This ensures that the corresponding bit in the trace memory is reliably set before the next *ram_wen* assertion occurs, guaranteeing accurate tracing of every write operation.

In the second instance (Fig. 3b), data is written to the same RAM addresses as before. This is evident from the unchanged *row_sel* and *col_sel* values. Since all these locations have already been marked in the trace memory, the counter remains unchanged. Before taking a snapshot, the CPU reads the trace memory via the on-chip communication bus, ensuring that only the modified RAM blocks, previously mapped into the trace memory, are saved to non-volatile memory (NVM) at the time of the snapshot.

In this work, we assume RAM operates reliably, as is typical in embedded and FPGA-based systems. *MeTra* monitors RAM write transactions on the system bus, ensuring all logical changes are captured without needing access to physical RAM internals. This enables selective retention of only modified regions. Detection or handling of physically faulty memory regions is beyond the scope of this work and can be addressed using standard error correction or diagnostics if required.

Fig. 4 illustrates the execution flow between *MeTra* and the CPU during each EH cycle. When power becomes available, the CPU checks whether a snapshot is available. If no snapshot exists, such as during the system's first run, the entire RAM is saved to NVM, and *MeTra* is then activated to trace RAM changes made during the application run. Conversely,

```
pRAMmem = (u32 *)0x60000010; //Assigning address to
 the pointer, Counter increments
*pRAMmem++ = 0x01234567; //Data written to a new RAM
 location. Counter increments for data and no increment
 when the pointer index updates
*pRAMmem++ = 0x87654321;
*pRAMmem++ = 0xabcdef12;
*pRAMmem++ = 0x12abcdef;
[...]

 pRAMmem   = (u32 *)0x60000010;
*pRAMmem++ = 0xabcdef12;
*pRAMmem++ = 0x87654321;
*pRAMmem++ = 0x01234567;
*pRAMmem++ = 0x87654321;
[...]
```

LISTING 1: Sequential RAM writes triggering *MeTra*'s access tracing logic.

if a snapshot is available, the CPU restores the RAM and its internal registers from NVM, allowing the application to resume from where execution was interrupted. During restore, data is copied from NVM back into RAM, followed by reloading the CPU's internal registers content by popping this from the stack, which was also preserved in NVM. *MeTra* is then activated to monitor subsequent RAM changes.

Once *MeTra* is activated, it starts tracing RAM writes. Each time a RAM write occurs, *MeTra* converts the RAM address (ram_addr) to row and column coordinates and checks the corresponding bit cell in the trace memory. If the bit cell is '0', it is set to '1', and the counter increments to trace the number of RAM locations that need to be copied to NVM. If the bit cell is already '1', the counter remains unchanged, as the location was previously recorded.

The value of V_{th} is stored in RAM, and *MeTra* uses the

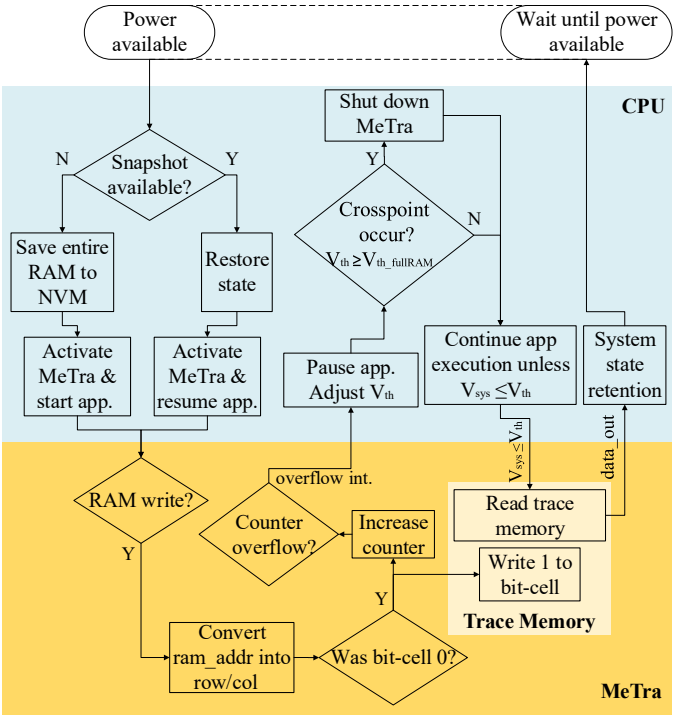


Fig. 4: Execution flow between *MeTra* and CPU.

counter overflow interrupt (*overflow int.*) to notify the CPU when V_{th} needs adjustment.

We empirically determined the energy required to save the entire RAM to NVM and used this to compute the corresponding voltage threshold, $V_{th\_fullRAM}$, needed to safely complete the copy, taking into account the system's storage capacitance, C_{store}, and minimum operating voltage, $V_{sys,min}$. When using selective saving with *MeTra*, voltage thresholds can similarly be computed using Eq. 5, as described below, where each increment corresponds to the energy required to save one RAM block to NVM. *MeTra* traces the number of modified memory blocks using a counter. Each time this counter overflows (e.g., reaching 0x1F for a 5-bit counter), an interrupt is triggered, prompting the CPU to update the estimated voltage threshold V_{th}. The crosspoint condition is evaluated during each counter overflow interrupt. If V_{th} exceeds the threshold required for a full RAM save ($V_{th\_fullRAM}$), a crosspoint is detected. This indicates that selective saving with *MeTra* is no longer energy-efficient, and a full save should be performed instead. Upon reaching this crosspoint, the CPU sends a shutdown signal to *MeTra*, disabling it until the next energy harvesting cycle. If the crosspoint has not yet been reached, the CPU reads the trace memory to determine which modified RAM blocks to save in NVM, initiating the selective state retention process.

The maximum value of this counter, K_{max}, is determined by the desired number of adjustment steps for V_{th}. Here, V_{th}, at each overflow interrupt, i, is updated using the equation:

$$i \cdot K_{max} \cdot E_{\text{block}} = C_{\text{store}} \cdot \frac{V_{th}[i]^2 - {V_{\text{sys,min}}}^2}{2}, \qquad (5)$$

where E_{block} represents the energy required to save a single RAM block to NVM, C_{store} the energy storage, and $V_{sys,min}$ represents the minimum voltage required to maintain regular system operation. Eq. 5 can be implemented as a lookup table for real-time updates to V_{th}.

The system's granularity and energy requirements are influenced by two key factors: (a) the minimum V_{th} adjustment step, which can be coarse (e.g., 0.5V), resulting in fewer interrupts but less precise adjustments, or fine (e.g., 0.05V) offering more customized adjustments at the cost of more frequent interrupts, and (b) the energy required to save a single RAM block, E_{block} can be determined based on the write time of NVM technology, read time of RAM, CPU instructions required for this copy operation and the size of the RAM block. When determining the size of the RAM block, which affects the trace memory size and counter, the CPU architecture must be considered, whether it is an 8-bit, 16-bit, or 32-bit, along with the system application's RAM usage patterns, such as random versus consecutive RAM writes. For instance, a 32-bit CPU typically writes data to RAM in four-byte blocks, making the minimum trace granularity one bit per 4-byte RAM block.

In the current design, the trace memory remains active throughout the CPU's operation to continuously trace memory changes, except when the system reaches the crosspoint, where the energy cost of selective saving exceeds that of saving the full RAM to NVM. At this point, *MeTra* is disabled to avoid unnecessary overhead.

C. System Architecture and FPGA-Based Implementation

We use Arm Cortex-M1 which is a general purpose, 32-bit RISC soft microprocessor that is optimized for FPGA integration. Arm Cortex-M1[1] supports all Thumb and Thumb-2 instruction set (Armv6-M). This approach facilitates efficient code execution on the processor, making it well-suited for the constrained memory in embedded systems. We used the Arty S7 (S7-50 variant) board for FPGA implementation. The Arty S7 is a development board featuring the AMD Spartan 7 FPGA. The S7-50 variant includes a Spartan 7 FPGA[2] with 32,600 LUTs, 65,200 flip-flops, and 2,700 Kbs of block memory, along with various user interfaces such as switches, buttons, and LEDs and is compatible with AMD's Vivado Design Suite.

Our design includes two memory systems. The first system, used by the CPU as RAM, is created using the Block Memory Generator IP and is connected to the CPU through the AXI BRAM Controller IP. The RAM size is 8 Kbytes, aligning with the typical RAM capacity of MCUs used in small embedded systems and sensor applications[3] [4].

The second memory system in our design serves as NVM and is also 8 Kbytes in size. Similar to the RAM, it is constructed using the Block Memory Generator and AXI BRAM Controller IPs. This FRAM-based NVM requires 120 ns for both read and write operations, reflecting the per-bit access time of 120 ns as reported in [28].

MeTra is integrated as an IP into the system via the AXI bus. As outlined in Sec. IV-B, *MeTra*'s trace memory is addressable and can be accessed by the CPU to identify which areas of RAM have changed so that they can be saved from RAM to NVM during snapshot time. Additionally, the three inputs from the RAM's block memory generator IP (*ram_addr*, *ram_wen*, *ram_clk*) are connected to the inputs of *MeTra*.

MeTra connects to the system bus like any other peripheral and does not require explicit synchronization during normal operation, as it passively monitors RAM writes without interfering. During the state-saving phase, the CPU accesses the trace memory through the same bus, relying on standard arbitration and synchronization mechanisms provided by the system interconnect (e.g., AXI on FPGA or AMBA on ASIC). We used the Vivado Design Suite (Version 2018.2), a comprehensive development environment for designing, synthesizing, and implementing Xilinx FPGA designs. Additionally, the Vivado Software Development Kit (SDK) was utilized to generate the Board Support Package (BSP), which is a layer of software containing hardware-specific firmware, device drivers, and other routines that allow the embedded operating system to function on our design. Keil (version 6) was used for the development of benchmarks. Finally, Vivado was used

[1]Arm Cortex-M1 Technical Reference Manual. Available at https://documentation-service.arm.com/static/5e8e18c2fd977155116a3d48

[2]Arty S7: Spartan-7 FPGA Development Board. Available at https://digilent.com/shop/arty-s7-spartan-7-fpga-development-board/

[3]LPC122x: 32-bit Arm Cortex-M0 microcontroller Product Data Sheet. Available at https://www.nxp.com/docs/en/data-sheet/LPC122X.pdf

[4]STM32F030x4 STM32F030x6 STM32F030x8 STM32F030xC Product Data Sheet. Available at https://www.st.com/resource/en/datasheet/stm32f030c8.pdf

again to integrate the hardware and software designs, which were then deployed on the FPGA.

V. Evaluation and Experimental Results

This section presents a performance analysis and highlights the benefits of using *MeTra* to selectively save RAM to NVM, thereby extending the application's active time on the CPU.

A. Overview of Benchmarks

We assess the results using three computational benchmarks: matrix multiplication, array sorting, and AES. The first benchmark performs matrix multiplication on two square matrices, each size 20 x 20 elements, and stores the result in the output matrix. The function employs three nested loops to calculate the product by summing the products of corresponding elements. Matrix multiplication plays a crucial role in numerous fields and applications, including computer graphics, machine learning, and control systems. The second benchmark involves sorting 300 elements in an array using the quicksort algorithm with stack-based iterations. The algorithm partitions the array around a pivot element, prioritizing the larger sub-array by pushing it onto the stack first. This approach optimizes stack depth and balances the workload. Array sorting is widely used in various applications, including databases, graph algorithms, and real-time systems, where efficient, resource-constrained, on-the-fly sorting is essential. The third benchmark executes AES-128 encryption, encompassing key operations like S-Box lookups and Galois field calculations to encrypt a 16-byte block (state) with a 16-byte key. This benchmark incorporates all the essential steps of AES encryption, including substitution, permutation, mixing, and key expansion.

B. Effect of Selective State Saving on V_{th}

We analyze the impact of selective RAM saving on V_{th}. Leveraging *MeTra*'s tracing, V_{th} is dynamically adjusted based on RAM changes, minimizing time and energy overhead while ensuring reliable state retention and recovery.

The top part of Fig. 5 shows the first case where the entire RAM needs to be saved while the system voltage, V_{sys}, drops from the maximum voltage of 3.6 V ($V_{sys,max}$) to 2.0 V ($V_{sys,min}$), at which point the system shuts down. With a C_{store} of 50 µF, the maximum energy stored, E_{store} at

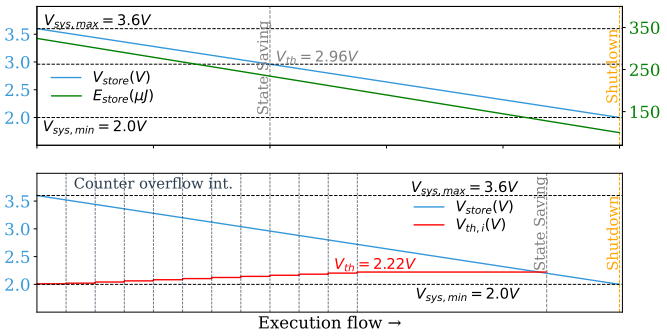


Fig. 5: Behaviour of *MeTra* when running matrix multiplication, with RAM block size of 4 bytes.

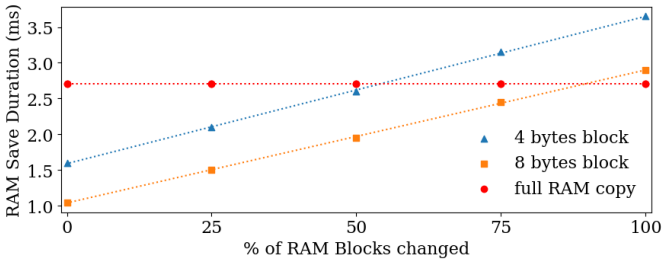


Fig. 6: Main memory (RAM) utilization and corresponding save time to NVM for granularity values of 4 and 8 bytes.

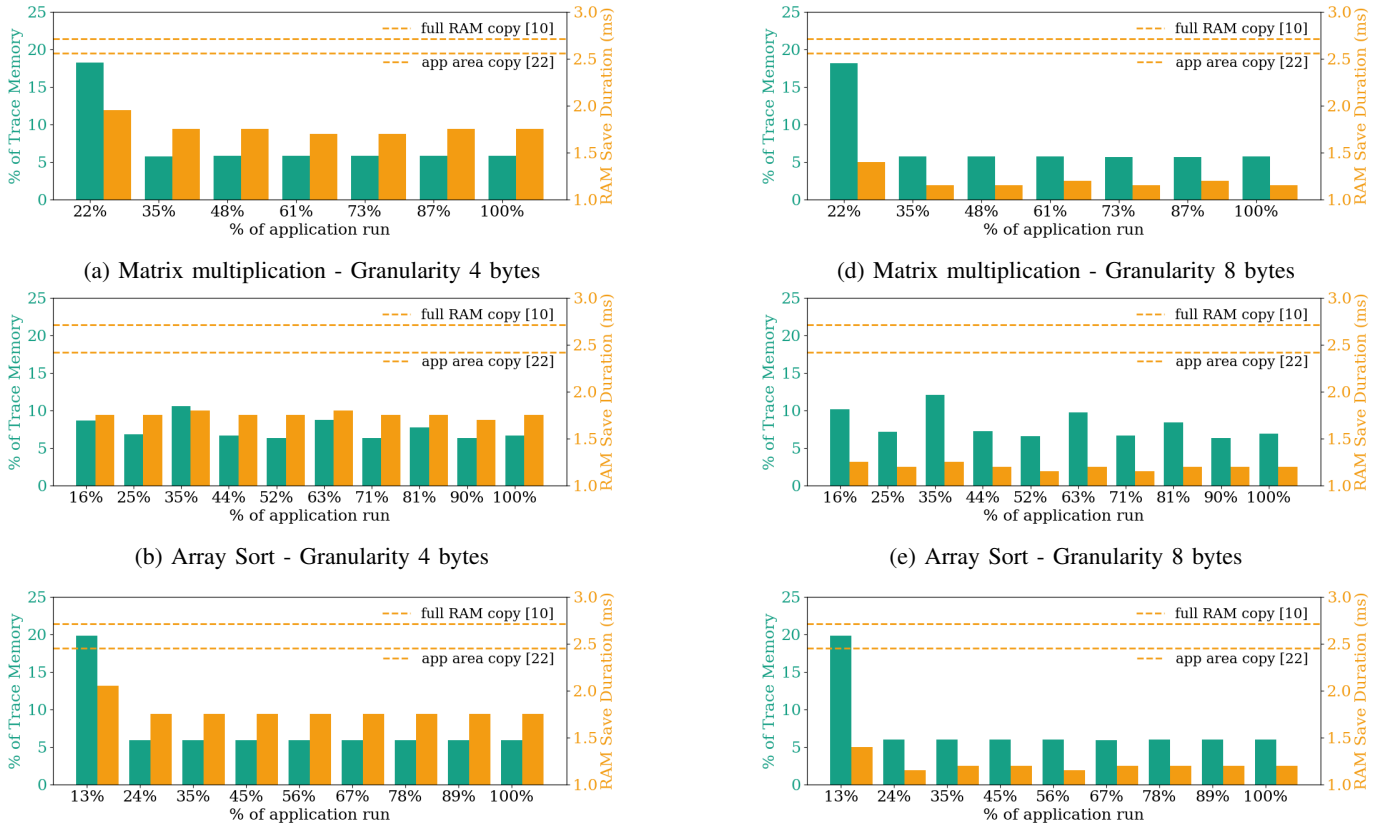
$V_{sys,max}$ is 324 µJ. Saving the complete RAM to NVM took 2.71 ms, which accounts for reading from the RAM, CPU instructions, bus transfer, and saving to NVM. The energy required for this operation is 218.54 µJ. As V_{sys} approaches 2.96 V, it will have enough energy to copy data from the RAM to the NVM.

The lower part of the plot shows the matrix multiplication example, where 18.21% of the RAM was utilized. In this case, the system updates its V_{th} value each time a counter overflow interrupt is raised by *MeTra*. In our implementation, the maximum counter value is 32, which means 32 bit-cells are marked in the *MeTra* trace memory. With a granularity value of 4 bytes, this translates to 128 bytes. So, each time 128 bytes are modified, V_{th} is updated accordingly. In this example, 18.21% of RAM utilization corresponds to 1492 bytes being utilized, causing the counter to overflow 11 times, which sets the V_{th} value to 2.22 V. This allows the extra energy that would have been required to copy the full RAM to be instead utilized for computation and other useful CPU operations.

C. Impact of Trace Memory Granularity on Save Time

We evaluated *MeTra*'s performance by analyzing the time required to save varying portions of RAM to NVM under different granularity settings. The RAM save duration includes the time required to push CPU registers to RAM, read trace memory in *MeTra*, and copy the modified regions of RAM to NVM. We varied RAM modifications at 25%, 50%, 75%, and 100% and measured the time required to save it to NVM using *MeTra* with 4- and 8-byte granularity settings, comparing it to a full RAM save without *MeTra*.

Fig. 6 presents a comparison of the time required to save RAM to NVM for varying percentages of RAM changes. The x-axis indicates the percentage of RAM blocks changed, ranging from 0% to 100%. The y-axis represents the time needed to complete saving data from RAM to NVM in milliseconds. The horizontal dashed red line marks the time required for a full RAM save operation, serving as a benchmark for evaluating RAM-saving efficiency at different granularities. The blue line (4-byte granularity) and the green line (8-byte granularity) represent the time required to save data as the percentage of modified RAM blocks increases from 0% to 25%, 50%, 75%, and 100%. The intersection points between these lines and the horizontal dashed line indicate the thresholds at which selective RAM saving using *MeTra* exceeds the time required to save the entire RAM. This additional delay is due to the overhead of the CPU acquiring the trace memory and



(a) Matrix multiplication - Granularity 4 bytes

(b) Array Sort - Granularity 4 bytes

(c) AES - Granularity 4 bytes

(d) Matrix multiplication - Granularity 8 bytes

(e) Array Sort - Granularity 8 bytes

(f) AES - Granularity 8 bytes

Fig. 7: Comparative analysis of trace memory usage and RAM-to-NVM save duration for matrix multiplication, array sorting, and AES at granularities of 4 and 8 bytes.

identifying RAM changes before initiating the save process. For instance, when 0% of RAM is modified, the delay introduced by *MeTra* is 1.6 ms for 4-byte granularity and 1.05 ms for 8-byte granularity. Depending on the granularity and the extent of RAM changes, a crossover point exists where it becomes more efficient to disable *MeTra* and save the entire RAM instead. The figure also shows that a coarser granularity (granularity of 8 bytes) enables faster saving of selected RAM compared to a finer granularity (granularity of 4 bytes). With coarser granularity, the time required to read trace memory and make decisions decreases, as smaller trace memory in *MeTra* need to be processed. However, this reduction in overhead comes at the cost of increased memory copy time, as larger blocks of data are copied in each operation. The optimal size of RAM blocks for copying depends on the system architecture and on whether RAM changes occur in a dispersed manner or exhibit spatial locality. In a 32-bit architecture, data is transferred in 32-bit chunks (i.e., 4 bytes). This represents the smallest possible block size for data transfer, providing the finest granularity. However, such fine granularity results in the highest overhead for trace memory reads and decision-making processes, representing the worst-case scenario for these operations. As a result, a trade-off must be made to determine the optimal block size (tracing granularity) that balances the decision-making overhead associated with trace memory and the efficiency of RAM copy operations. In this study, we explore only 4-byte and 8-byte blocks. Further exploration of the optimal RAM block size is left to future work.

D. Impact of Computational Progress and Trace Granularity on Save Times

This subsection explores the correlation between computational progress and the proportion of RAM changes traced by *MeTra*, focusing on how selective saving impacts memory copy time and can extend the CPU's active operating time. The analysis evaluates three benchmarks—matrix multiplication, array sorting, and AES—under two granularity settings: 4 and 8 bytes, highlighting the efficiency gains achieved through selective state retention. Fig. 7 presents a comparative analysis of the percentage of RAM changes and the corresponding save times at different stages of application runtime.

To simulate realistic power interruptions, program execution was interrupted at random points for state saving. At each interruption, the internal state is saved, including the CPU registers and modified RAM regions flagged by *MeTra*'s trace memory. Once power is restored, the program resumes from the exact point of interruption, repeating this process multiple times to cover the complete execution of the benchmark.

For the matrix multiplication benchmark (Fig. 7a), execution was interrupted at 22%, 35%, 48%, 61%, 73%, and 87%

of a full run. At the first interrupt, 18.21% of the trace memory was marked for selective copying from RAM to NVM. The save state process, which included storing the CPU core registers to NVM and copying 18.21% of modified RAM to NVM, took 1.95 ms with a granularity of 4 bytes. In contrast, saving the entire RAM would have required 2.71 ms. For subsequent saves, only about 5.8% of the RAM changed consistently, resulting in RAM save duration of approximately 1.7 ms. This indicates that the matrix multiplication initially caused a relatively higher amount of RAM changes. However, as execution progressed, changes to RAM were consistently minimal, requiring only about 5.8% of the RAM to be copied during subsequent save state operations.

For the array sorting benchmark (Fig. 7b), execution was interrupted at 16%, followed by 25%, 35%, 44%, 52%, 63%, 71%, 81%, and 90% of the run. Notably, the percentage of trace memory flagged for RAM changes ranged from 6.30% to 10.60%. Unlike the first benchmark, there was no initial spike in RAM changes at the first interruption (16%). Instead, 10.60% of the traced memory was marked at 35%, reflecting RAM changes accumulated during 25% to 35% of the application's runtime. Because the amount of RAM changes remained consistent during application execution, the RAM save duration remained steady, ranging from 1.7-1.8 ms.

The AES benchmark exhibited a trend similar to that of matrix multiplication, with a higher percentage of RAM changes during the initial stage of the application. Specifically, 19.82% of trace memory was marked at 13% of the application's runtime. Subsequently, only 5.86% of trace memory was marked, indicating that after the initial phase, relatively few RAM regions were modified. These results demonstrate that during the execution of these applications, only a small portion of the RAM is actively utilized or changed between EH cycles. Consequently, saving the entire RAM to NVM for each power interrupt is largely redundant.

We also measured the time required to save only the RAM segments utilised by the application, specifically the .data, .bss, and .stack segments, when power interruption occurs, following the approach proposed in [22]. It is worth noting that our benchmarks do not use dynamic memory allocation, which is often avoided in compact embedded systems as discussed in Sec. II. The measured save times for the RAM regions used by the applications were 2.56 ms for matrix multiplication, 2.42 ms for array sorting, and 2.45 ms for the AES benchmark. While these times are lower than saving the full RAM by avoiding saving unused RAM regions, they are still significantly higher compared to *MeTra*'s selective RAM saving.

The time savings achieved with granularity of 8 bytes are even more significant, as illustrated in Fig. 7d, 7e, and 7f. For instance, in the matrix multiplication benchmark, granularity of 8 bytes reduced the RAM save duration to 1.4 ms while saving 18.21% of RAM, compared to 2.71 ms required to save the entire RAM. Similarly, saving 5.71% of RAM took just 1.15 ms with granularity of 8 bytes, compared to 1.7 ms with granularity of 4 bytes. This pattern is consistent across all the benchmarks. For the matrix multiplication benchmark, where 5.71% to 18.21% of the RAM was modified, the corresponding save time improved by 28% to 37.27% with a granularity of 4 bytes when compared to the full RAM save duration. When the granularity was set to 8 bytes, the save time improvements increased further, ranging from 48.34% to 57.56%. In the array sorting, the percentage of RAM change ranged from 6.30% to 10.60%, and the corresponding save time improved from 33.58% to 37.27% with a granularity of 4 bytes when compared to the full RAM save duration. In contrast, a granularity of 8 bytes yielded improvements between 53.87% and 57.56%. Similarly, for the AES, where the RAM changes ranged from 5.86% to 19.82%, the corresponding save time improved by 24.35% to 35.42% with a granularity of 4 bytes when compared to the full RAM save duration. With a granularity of 8 bytes, save time improvements ranged from 48.34% to 57.56%. To summarize, performing selective RAM saving through *MeTra*, compared to saving the full RAM or only the application area, results in substantial improvements in RAM save duration.

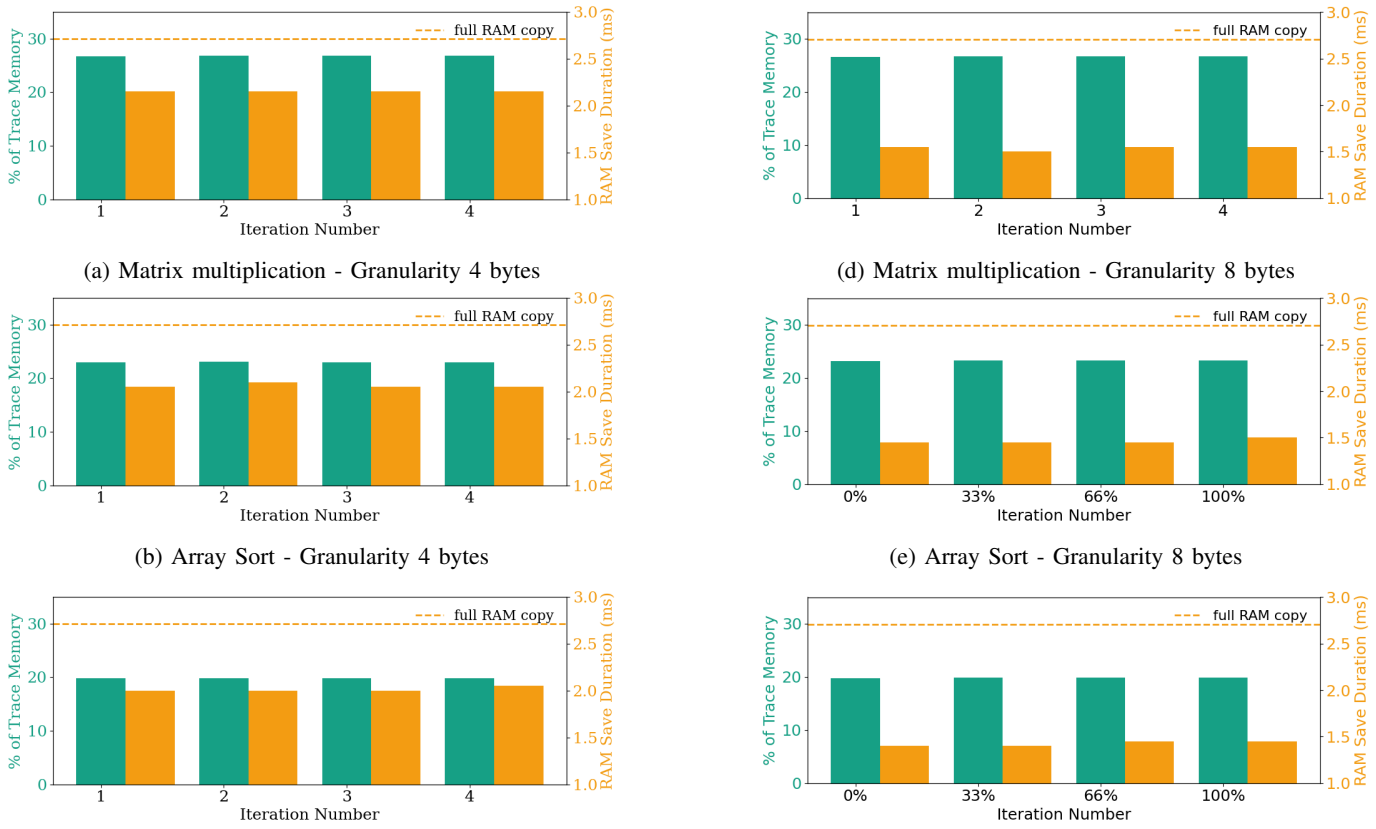
E. Impact of Benchmark Data and Trace Granularity on Memory Use and Save Time

This subsection analyzes the impact of processing new benchmark input data across multiple iterations of the same application on memory usage and saving time. Benchmarks were executed multiple times with varying data to emulate real-world use cases where similar computations are performed repeatedly, such as processing incoming sensor data or performing matrix multiplications on different images. Thus, we expanded the comparative analysis, shown in Fig. 8, which illustrates the percentage of RAM changed and the time required for saving across multiple iterations of the same benchmarks, with each iteration representing a complete application execution.

Although each iteration represents a self-contained task, analyzing the total memory usage during a complete task shows that only selective parts of the memory are actively utilized, even during full task execution. This reinforces our claim that selective memory retention is effective, reducing unnecessary memory-saving operations. Additionally, this analysis evaluates the impact of new input data, confirming that re-running the same task with different inputs does not significantly change the amount of RAM utilized.

As in previous analyses, we consider two granularity settings: 4- and 8-bytes. During each iteration of the benchmark, RAM changes were traced, and at the end of each iteration, the modified RAM was saved to NVM as well as the content of core registers. The y-axis indicates the percentage of traced RAM to be saved to NVM and the time required, while the x-axis represents the application iteration number.

For the matrix multiplication with a granularity of 4 bytes, saving the internal state consistently took 2.15 ms across all iterations, with trace memory usage ranging from 26.76% to 26.81%. In the array sorting benchmark, the state saving time showed minor variations, ranging from 2.05 ms to 2.1 ms, with trace memory usage between 22.95% and 23.10%. The AES demonstrated stable state saving times of approximately 2 ms, with trace memory usage ranging from 19.78% to 19.82%.



(a) Matrix multiplication - Granularity 4 bytes

(b) Array Sort - Granularity 4 bytes

(c) AES - Granularity 4 bytes

(d) Matrix multiplication - Granularity 8 bytes

(e) Array Sort - Granularity 8 bytes

(f) AES - Granularity 8 bytes

Fig. 8: Comparative analysis of trace memory utilization and RAM-to-NVM save duration across multiple iterations of matrix multiplication, array sorting, and AES at granularities of 4 and 8 bytes.

As observed earlier, using a granularity of 8 bytes delivers significant performance improvements by reducing memory tracing overhead. In this configuration, any modifications to eight-byte memory blocks were traced in the trace memory. For the matrix multiplication, saving times ranged from 1.5 ms to 1.55 ms, with trace memory usage between 26.56% and 26.66%. This marks a notable enhancement compared to the 4-byte granularity configuration and the full RAM saved to NVM, where state saving times were 2.15 ms and 2.71 ms, respectively. Similarly, for the array sorting, the state saving with granularity of 8 bytes ranged from 1.45 ms to 1.5 ms, with trace memory usage between 23.14% and 23.24%. For the AES, the state saving times ranged from 1.4 ms to 1.45 ms, with trace memory usage between 19.73% and 19.82%. These results demonstrate a clear improvement over the granularity of 4 bytes.

In summary, using granularity of 8 bytes in the matrix multiplication, array sorting, and AES benchmarks significantly reduced the time required to save the internal state across all benchmarks compared to granularity of 4 bytes. The coarse-grained granularity minimizes the overhead associated with RAM tracing and can be an effective strategy for enhancing system state performance, mainly when RAM changes exhibit spatial locality.

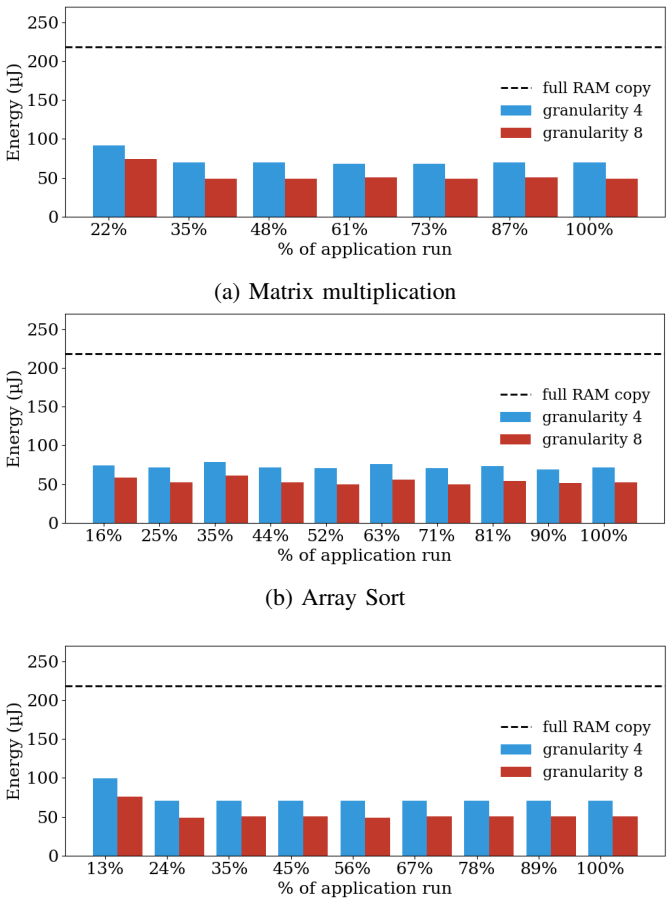
F. Energy Comparison: Selective vs. Full RAM Saves

This subsection analyzes the energy consumption associated with saving modified RAM at different granularities (4 bytes and 8 bytes) during benchmark execution. The comparison highlights the efficiency of selective RAM saving versus a full RAM save. Fig. 9 illustrates the energy required to save modified RAM at various stages of the benchmark's execution, contrasting it with the energy usage for a full RAM save.

For matrix multiplication (Fig.9a), at first interrupt, when 18.21% of the trace memory was marked, 92 µJ were consumed for saving state with a granularity of 4 bytes. In contrast, saving the entire RAM would have required 218.54 µJ. For subsequent interrupts, only about 5.8% of the RAM changed consistently, resulting in energy consumption of approximately 69.66 µJ. For the same amount of RAM changed but traced with granularity of 8 bytes, the energy consumption ranged from 48.59 µJ to 73.81 µJ.

For the array sorting benchmark (Fig.9b), the percentage of traced RAM ranged from 6.30% to 10.60% at various stages of the application run, consuming between 69.12 µJ and 78.30 µJ with granularity of 4 bytes. In comparison, energy consumption with granularity of 8 bytes ranged from 49.73 µJ to 60.56 µJ.

The AES benchmark (Fig.9c) caused RAM changes between 5.86% and 19.82%, resulting in corresponding energy



(a) Matrix multiplication

(b) Array Sort

(c) AES

Fig. 9: Energy consumption to save selected RAM to NVM during state saving.

consumption of 70.34 µJ to 99.27 µJ with granularity of 4 bytes, while energy consumption ranged from 48.97 µJ to 75.97 µJ with granularity of 8 bytes.

Considering that a full RAM save requires 218.54 µJ, which involves saving largely redundant RAM data, selective saving through *MeTra* with granularity of 4 bytes achieved energy efficiency improvements between 57.90% and 68.89% for matrix multiplication, 64.17% to 68.37% for array sorting, and 54.57% to 67.81% for AES.

The energy efficiency improvements for the same amount of RAM changes traced with granularity of 8 bytes ranged from 66.22% to 77.76% for matrix multiplication, 72.29% to 77.24% for array sorting, and 65.24% to 77.59% for AES.

While our prototype is implemented on an FPGA platform to demonstrate practical feasibility, the *MeTra* architecture is also applicable to ASIC designs. Adopting *MeTra* in ASIC would introduce additional area and power overheads, primarily due to the inclusion of trace memory and associated control logic. These overheads scale with RAM size, but can be optimized through careful granularity selection and hardware co-design. We anticipate a moderate increase in power consumption to operate *MeTra*; however, this is offset by the system-wide energy savings achieved through efficient state retention. A comprehensive ASIC-level analysis of area and power overheads is left for future work. To give a practical perspective, Table II presents the FPGA resource utilization for different granularities (4-byte and 8-byte), with the percentage of total resources used shown in brackets below. This offers a relative estimate of the hardware overhead introduced by *MeTra* for each type of resource.

TABLE II: Resource utilization by the system, with and without the memory tracing system *MeTra*.

Resource	Total	Without *MeTra*	*MeTra* Gran 8	*MeTra* Gran 4
LUT	32600	8614 (26.42%)	10559 (32.39%)	11923 (36.57%)
LUTRAM	9600	321 (3.34%)	358 (3.73%)	358 (3.73%)
FF	65200	9588 (14.71%)	11288 (17.31%)	12381 (18.99%)
BRAM	75	16.50 (22.00%)	16.50 (22.00%)	16.50 (22.00%)
DSP	120	3 (2.50%)	3 (2.50%)	3 (2.50%)

VI. Conclusion and Future Work

This paper presents *MeTra*, a custom hardware module for efficient system state retention by tracing changes in RAM between power outages. *MeTra* dynamically adjusts V_{th} at runtime to optimize energy usage and maximize active time. Prototyped on an FPGA with an Arm Cortex-M1, it reduces state-saving time by 48.34%–57.56% and improves energy efficiency by 65.24%–77.76% in FRAM-based systems through selective saving of only modified memory regions.

Our future work includes exploring dynamic power gating of trace memory, where memory tracing circuitry is selectively disabled during low memory activity phases to reduce static power consumption without affecting correctness, and investigate how *MeTra* can support multi-core systems, requiring coordinated trace handling and state retention mechanisms across cores.

Acknowledgment

This work is supported by the grants UK EPSRC NIA EP/W022877/1 and UK EPSRC IAA EP/X525601/1. Experimental data can be found at DOI:10.5258/SOTON/D3726 (https://doi.org/10.5258/SOTON/D3726). The authors would like to thank Sergey Mileiko and Mohd Firdaus Hirman Ritom for their support.

References

[1] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 939–954.

[2] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM conference on embedded network sensor systems*, 2017, pp. 1–6.

[3] S. Mileiko, O. Cetinkaya, D. Mackie, R. Shafik, and D. Balsamo, "A tegbased non-intrusive ultrasonic system for autonomous water flow rate measurement," *IEEE Transactions on Sustainable Computing*, vol. 8, no. 3, pp. 363–374, 2023.

[4] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, 2019, pp. 8–14.

[5] J. R. Smith, *Wirelessly powered sensor networks and computational RFID*. Springer Science & Business Media, 2013.

[6] D. Sartori and D. Brunelli, "A smart sensor for precision agriculture powered by microbial fuel cells," in *2016 IEEE sensors applications symposium (SAS)*. IEEE, 2016, pp. 1–6.

[7] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui *et al.*, "Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 368–381.

[8] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet-of-things," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–13.

[9] D. Balsamo, A. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Graceful performance modulation for power-neutral transient computing systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 5, pp. 738–749, 2016.

[10] 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.

[11] A. Colin and 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.

[12] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 85–99.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] ARM Limited, *Cortex-M1 Technical Reference Manual*, ARM Limited, 2008, aRM DDI0413D, Revision: r1p0, Accessed: September 17, 2024. [Online]. Available: https://documentation-service.arm.com/static/5e8e18c2fd977155116a3d48

[20] P. Singla and S. R. Sarangi, "A survey and experimental analysis of checkpointing techniques for energy harvesting devices," *Journal of Systems Architecture*, vol. 126, p. 102464, 2022.

[21] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Dynamic task-based intermittent execution for energy-harvesting devices," *ACM Transactions on Sensor Networks (TOSN)*, vol. 16, no. 1, pp. 1–24, 2020.

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

[23] 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.

[24] 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.

[25] 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.

[26] C. Comar, C. Dross, F. Gilcher, and Y. Moy, "Dynamic memory management in critical embedded software," 2022. [Online]. Available: https://www.adacore.com/uploads/techPapers/DynamicMemoryManagement.pdf

[27] MathWorks. (2021) MISRA C:2012 Dir 4.12 – Dynamic memory allocation shall not be used. *MATLAB Documentation*. MathWorks. Accessed: 2025-07-22. [Online]. Available: https://www.mathworks.com/help/bugfinder/ref/misrac2012dir4.12.html

[28] T. Daulby, A. Savanth, A. S. Weddell, and G. V. Merrett, "Comparing nvm technologies through the lens of intermittent computation," in *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, 2020, pp. 77–78.

Osama Bin Tariq earned his M.Sc. and Ph.D. degrees in Electronic Engineering from Politecnico di Torino, Italy. He is currently a Research Associate with the MicroSystems Research Group at Newcastle University, U.K. His research interests include sensing systems, machine learning applications, and energy-efficient, intermittently powered embedded systems.

Theodoros D. Verykios received a BEng and a Ph.D. in Electronic Engineering from the University of Southampton (UK) in 2015 and 2019, respectively. He previously worked as a Senior Hardware Engineer at Qualcomm and a Senior CPU Research Engineer at Huawei R&D. He is currently a Visiting Research Fellow at the University of Southampton and a Staff Systems Performance Engineer at Qualcomm. His research interests include energy efficient embedded systems and intermittently powered computing systems.

Geoff V. Merrett (GSM'06-M'09-SM'19) received the B.Eng. (Hons) and Ph.D. degrees from the University of Southampton, U.K., in 2004 and 2008, respectively. He currently holds a Personal Chair in Electronic and Software Systems at the School of Electronics and Computer Science, University of Southampton, U.K. His research interests are in energy management of mobile/embedded systems and self-powered devices, and he has published over 200 journal and conference articles on these topics. Prof. Merrett Program Co-Chair of the IEEE COINS 2024 conference, and General Chair of EWME 2016 and ENSsys 2013-15. He is a Member of the IET, and a Fellow of the HEA.

Domenico Balsamo received the PhD degree in computer engineering from the University of Bologna, Italy, in 2015. He is a senior lecturer in electronic systems focusing on embedded systems hardware and software co-design, co-leading the MicroSystems Research Group, Newcastle University, U.K. He has more than ten years of experience in energy-efficient embedded systems and low-power pervasive computing design as part of different EPSRC, EU FP7 and EU HORIZON 2020 projects, which led to more than 40 publications in top-tier journals and conference proceedings.