Pragmatic Memory-System Support for Intermittent Computing using Emerging Non-Volatile Memory

Sivert T. Sliper, William Wang, Nikos Nikoleris, *Member, IEEE,* Alex S. Weddell, *Member, IEEE,* Anand Savanth, *Member, IEEE,* Pranay Prabhat, *Member, IEEE,* and Geoff V. Merrett, *Senior Member, IEEE*

***Abstract*—Intermittent computing (IC) is a key enabler for the vision of a trillion Internet of Things devices. By harvesting energy from the environment, and leveraging non-volatile memory (NVM) to retain computational progress across power cycles, IC enables untethered and battery-free devices to perform computation whenever ambient energy is available. The backbone of state retention is NVM, and recent advances in energy-efficient NVM have the potential to expand the application domain of IC significantly. Utilizing emerging NVM at the level of bit-cells, researchers have proposed non-volatile processors. However, these do not leverage hardware-software co-design, which can be used to overcome hardware limitations and to provide support for application-level constraints such as atomicity.**

In this paper, we propose MEMIC, a memory architecture tailored for IC devices with byte-addressable NVM. A core focus of MEMIC is to combine volatile- and non-volatile memory in such a way that the operations of IC are as efficient as possible, while also maximizing computational performance per joule. MEMIC uses volatile memory for energy efficiency, and non-volatile memory for data retention. To avoid double-buffered checkpoints and costly roll-backs when code needs to be re-executed, MEMIC is designed to track and minimize writes to non-volatile memory during failure-atomic sections. Our evaluation shows that MEMIC's instruction cache reduces workload completion time under intermittent operation by 41-70% and its data cache provides a further reduction of 13-39%.

***Index Terms*—intermittent computing, embedded systems, hardware-software co-design, low-power design.**

I. Introduction

INTERMITTENT Computing (IC) is a key enabling technology for widespread adoption of Internet of Things (IoT) devices. It enables them to make incremental computational progress by directly using energy harvested from the environment instead of relying on energy stored in a battery. Intermittent computing systems are thus free from batteries, which often increase cost and size as well as limit viable operating conditions.

Figure 1 illustrates intermittent operation, whereby brief on-periods of execution are interleaved with off-periods where the supply voltage recovers. The key ability of IC is to retain state so that execution can continue from where it left off after a power failure. This can be achieved in several ways. The most straightforward approach is to back up volatile state (registers, volatile memory etc.) in non-volatile memory (NVM) by saving a checkpoint [1]. When power returns after a failure, execution resumes from the last valid checkpoint.

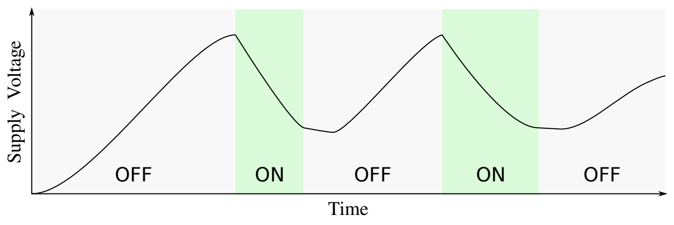


Fig. 1. Illustration of intermittent operation. The device operates during brief on-periods (typically in the order of milliseconds), until its stored energy is depleted. It then remains off until the supply voltage recovers.

Recent years have seen considerable research on software, programming models and compiler tools for IC, as well as the closely-related field of non-volatile processors (NVP). The former are largely focused on enabling IC on commercially-available microcontroller devices, and have also established criteria for correct execution and identified bugs that are unique to IC [2]. Researchers have, for example, shown that memory corruption can occur when re-executing sections of code that are not idempotent [3]. While it can often be avoided, re-execution is necessary when a failure-atomic section (FASE) needs to be restarted after a power failure. A classic example of a FASE is the transmission of a radio packet: if execution stops halfway through transmission, it will need to restart again from the beginning when power returns. Current software solutions for FASE support successfully avoid re-execution bugs, but incur excessive performance overhead, require rewritten software and/or lack portability across applications and hardware platforms as they must be tweaked for specific energy environments (power consumption, capacitor size etc.) [4]–[7]. This paper aims to alleviate these shortcomings by proposing pragmatic hardware support for the core operations of IC.

Research in NVP leverages recent developments in non-volatile bit-cells and registers to deliver devices that lose little to no state during a power failure. Generally, these aim to provide *implicit* non-volatility without software modification. Few works have, however, studied the combination of hardware and software support for IC. We argue that both areas

Manuscript received April 14, 2022.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising. All data supporting this study are openly available from the University of Southampton repository at https://doi.org/10.5258/SOTON/D2186. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under an iCASE award and Grant EP/P010164/1.

S. Sliper, A.S. Weddell and G. V. Merrett are with the School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ U.K. (e-mail: {S.Sliper,G.Merrett,A.Weddell}@ecs.soton.ac.uk.)

W. Wang, N. Nikoleris, A. Savanth and P. Prabhat are with Arm Research, Cambridge, CB1 9NJ, U.K. (e-mail: {William.Wang, Nikos.Nikoleris, Anand.Savanth, Pranay.Prabhat}@arm.com)

need innovation to support reliable and efficient IC. Utilizing energy-efficient and byte-addressable NVM and specialized circuits is necessary to efficiently perform the core operations of IC. On the other hand, software support is necessary to express application-level mechanisms like atomicity, and to overcome limited hardware resources. By combining appropriate hardware and software support, the result is a device that maintains the programmability of existing software-only methods while reducing software complexity and increasing performance per joule.

In this paper, we propose *MEMIC*, a memory system tailored for IC that improves efficiency and reliability while also simplifying IC software. The core aims of this paper can be summarized in three objectives. The first objective of *MEMIC* is to combine volatile and non-volatile memories to get the best of both worlds: volatile memory offers low latency and low per-access energy, whereas non-volatile memory offers persistence, lower leakage power, and potentially higher density. Secondly, MEMIC aims to provide efficient state retention and FASE support, with minimal software changes, by organizing and partitioning volatile and non-volatile memory in such a way as to enable efficient data versioning and checkpointing. Finally, the third objective is to maximize portability across software applications, and also across hardware platforms (circuit boards) by controlling checkpointing energy independently of the software's memory usage. MEMIC achieves these objectives by combining non-volatile memory, a volatile instruction cache, a customized volatile data cache, and a hardware undo-logger in an architecture that leverages synergies between these components.

The main contributions of this paper are:

- Simulation experiments, informed by production 28 nm MRAM [8] and 22 nm SRAM [9] memory compilers, showing the unique benefits instruction and data caching has in IC: in addition to lowering energy consumption during execution, they also effectively mitigate the overheads caused by frequently checkpointing and rebooting.
- With our modifications, the data cache is also used to bound checkpointing energy. This enables the device to operate under harsher conditions, and cuts the dependency between the software's memory footprint and checkpointing energy; thus ultimately enables a device that is portable between different circuit boards, software, and applications.
- Efficient FASE support through the use of a hardware undo-logger, activated only during FASEs, and placed behind a data cache so as to minimize undo-logger pressure (and thus its energy and performance overhead).
- *MEMIC*, a pragmatic memory system targeting modern MRAM-enabled low-power microcontrollers, that improves workload completion time by 13–39 %, using 13–39 % less energy, and operates under condition where state-of-the-art systems fail.

II. Motivation & Related Works

This section presents necessary background and related works on IC and emerging NVM, and motivates the need for a new memory system tailored for IC.

A. *Implementations of IC*

Related works on IC have proposed a large number of software-based methods that aim to provide fast and robust IC on commercially available microcontroller devices. These methods fit into three classes [10]:

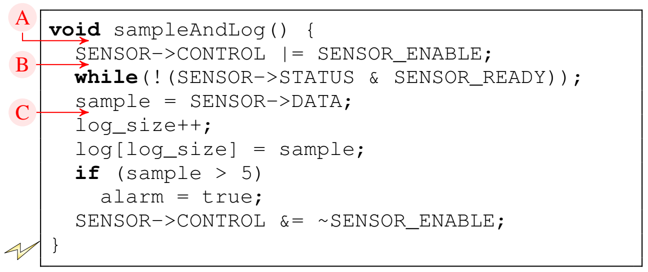
- Static IC saves checkpoints at pre-determined intervals, or at locations generated at compile time [11], [12]. After a power failure, execution can then recover state and continue from the most recent checkpoint.
- Reactive IC saves checkpoints when an imminent power failure is detected, for example when detecting that the supply voltage has dropped below a threshold [1]. After a power failure, execution recovers state and continues from exactly where it left off.
- Task-based IC changes the programming model and requires that the programmer divides the application into a set of tiny tasks that can be run atomically [4]. The task-based runtime does not save normal checkpoints, but instead ensures that the results from each task are persisted (saved to NVM), and keeps track of progress through the task-graph. After a power failure, the task-based runtime restarts execution from the most recently completed task.

There is also an alternative fourth method, footprint-based IC, suitable only for specific applications which have a static control flow [13]. Instead of checkpoints, a progress indicator (the footprint) and specific application data is saved. A runtime then uses the saved footprint to determine how to resume after a power failure. This method is, however, not general purpose, and so not considered for MEMIC.

Both static and task-based IC suffer from performance degradation due to frequent checkpointing or task-transitions, respectively [14]. Additionally, the task-based programming model is incompatible with existing libraries and code-bases and is not portable across hardware platforms, as it requires that programs be segmented into appropriately-sized tasks. Research on task-based methods have, however, also proposed several techniques to handle FASEs without risking re-execution bugs. These are discussed in following subsections. This work aims to combine the performance and software-compatibility of reactive IC with the FASE support of task-based IC.

B. *Inconsistencies caused by intermittent execution*

Figure 2 illustrates three possible faults when re-executing arbitrary sections of code. The function `sampleAndLog` reads a sensor value and appends it to an array of samples, `log`. If a value greater than five is sampled, the `alarm` should be set (assume that alarm is a global flag that gets checked and reset by the caller). A power failure occurs at the end of `sampleAndLog`, as denoted by ⚡. The points A, B and C mark potentially problematic checkpoint/task-boundary locations. Consider a naive approach that allocates all variables to NVM, does not checkpoint peripheral state, and allows re-execution (because of statically placed checkpoints, or failed reactive checkpoints). The following three scenarios



```
A  void sampleAndLog() {
     SENSOR->CONTROL |= SENSOR_ENABLE;
B    while(!(SENSOR->STATUS & SENSOR_READY));
     sample = SENSOR->DATA;
C    log_size++;
     log[log_size] = sample;
     if (sample > 5)
       alarm = true;
     SENSOR->CONTROL &= ~SENSOR_ENABLE;
⚡  }
```

Fig. 2. Pseudo code illustrating three possible faults when re-executing arbitrary sections of code.

describe three bugs that can occur when calling the function `sampleAndLog`, with a subsequent power failure at ⚡.

Scenario A illustrates a repeated-IO bug [2]. In the first on-period, a checkpoint was saved at A. Assuming that the sensor value was read to be > 5, `alarm` gets set to `true`. After the power failure, execution resumes from A, but assume that the sensor now reads ≤ 5; the alarm should not be set. The resulting state is corrupt, because the latest logged sensor value is < 5 and the alarm has been set; this is a state that could not have occurred were it not for the power failure.

In scenario B, a checkpoint was saved at B, and execution continued until the power failure at ⚡. After resuming from B, execution stalls indefinitely, because the sensor was never initialized and so the ready flag never gets set.

Finally, in scenario C, the checkpoint was saved just before incrementing `log_size`. Since this system allocates all variables to NVM, the state of `log_size` and `log` persists through the power outage. When power returns, `log_size` gets incremented yet again, and another log entry is saved. This bug lead to two log entries being saved on one call to `sampleAndLog`; in fact, the log would continue growing if power failed repeatedly at ⚡. Scenario C is an example of a write-after-read idempotency violation [3]. In fact, this same bug also occurs in scenario A.

Scenarios A and C can be protected against by allocating all variables to volatile memory and never re-executing code [15], by double buffering checkpoints, or by logging and rolling back changed state after a checkpoint/task-boundary [5]. The simplest way to avoid B is to avoid such checkpoint placements entirely, for example by executing (parts of) `sampleAndLog` as a FASE. Other works have focused specifically on the topic of saving and restoring peripheral state [6], [16].

C. *Supporting Failure-Atomic Sections (FASEs)*

Failure-atomic sections (FASEs) are sections of code that need to be restarted from the beginning if they are interrupted by a power failure. They are needed to express indivisible operations, such as radio transmissions, time-series sampling, and sampling of temporally correlated sensors. An example of a FASE, similar to the one mentioned in the introduction, is recording a contiguous window of time-series data from a sensor such as a microphone or accelerometer: the resulting data is only sensible if it was recorded without power interruptions. For such application-specific atomicity constraints, the FASE is annotated by the programmer, for example by using a wrapper function (as shown later in Listing 4). The programmer is responsible for ensuring that FASEs are small enough to be executed in a single power cycle; specifying a FASE that requires more energy than the device can muster leads to live-lock, because the FASE, by definition, cannot be subdivided. In particular cases, FASEs can be annotated automatically to prevent bugs such as A, B and C [2].

Task-based IC supports FASEs by default, since it is, in fact, based on dividing programs into a set of FASEs. The most commonly used method is data versioning, in the form of undo-logging or redo-logging, often accompanied by double-buffered checkpoints [4], [5]. Informed by annotation from the programmer (in the form of tagging variables and/or defining task-boundaries) and static analysis, custom compiler extensions instrument data versioning code for all variables deemed vulnerable to re-execution bugs. These specific variables can then be brought back to a consistent state (their old values) before re-executing a FASE. In the case of undo-logging, these variables are recorded in an undo-log before or during execution of a FASE. If the FASE completes successfully, the undo-log can be discarded. On the other hand, if the FASE has to be restarted, the undo-log must be applied first by writing logged data to roll back the state of NVM.

Traditional checkpoints are often used in addition to data versioning, as a means for more efficient state retention of variables that are not susceptible to re-execution bugs. To avoid corrupt checkpoints without making assumptions about the power supply, these can be double-buffered [4].

Reactive and static IC methods have often omitted FASE support [1], [11], [15], [17]. However, as long as they allocate all variables to volatile memory, they can support FASEs by saving a checkpoint immediately preceding the FASE, disabling checkpointing during the FASE, and finally enabling checkpointing again once the FASE has completed [6], [7]. This method can, however, lead to large overheads, especially if FASEs are frequent and checkpoints are expensive.

MEMIC employs reactive IC as the basis for efficient state retention during normal execution, and techniques from task-based IC for safe FASE support, and provides hardware support to accelerate both.

A complementary approach to FASE support is to reduce the chance of re-execution, by ensuring that a minimum of energy is stored before starting the FASE [7]. FASE support is still required, however, because some tasks may consume an unpredictable quantity of energy. The energy it takes to transmit a radio packet, for example, may depend on channel congestion, whether the intended receiver is listening, and other stochastic factors.

D. *Use of NVM for IC*

The first IC methods used flash memory as their NVM, but it was soon demonstrated that the byte-addressable and more energy-efficient ferroelectric RAM (FRAM) was a better candidate [1], [11]. Using a more suitable NVM meant that IC devices could operate using only the energy buffered in a few microfarads of capacitance already necessary for power conditioning purposes, as opposed to requiring supercapacitors to supply the checkpointing energy [1]. Today, magneto-resistive RAM (MRAM) and phase-change-memory (PCM) are emerging as replacements for flash in advanced process nodes for microcontrollers. MRAM is available from several foundries, in standalone memory products, and in some commercially-available microcontrollers [8]. Like FRAM, MRAM is byte-addressable, but it is also more energy-efficient and scalable to advanced process nodes. It is still, however, not as energy-efficient as SRAM. MRAM has one or two orders of magnitude higher write-energy, and several times more read-energy per access, than low-power SRAM. As such, it is clear that MRAM-enabled conventional microcontrollers still need SRAM. For microcontrollers made for IC, the hybrid SRAM and NVM architecture is a complex topic with many trade-offs.

E. Non-Volatile Processors (NVPs)

Many previous works realise reactive IC by building an NVP [18] - a processor with each state-containing volatile FF/RAM cell replaced with a non-volatile counterpart. Accesses during execution are served from the volatile cells, or volatile parts of the cells, and the non-volatile part is accessed only to checkpoint and restore state. This allows instant parallel save/restore of system state, also reduced overall save/restore energy. This approach has been demonstrated in [19] on a 130nm PZT FeRAM technology, leveraging byte-addressable and energy efficient FeRAM. However, FeRAM is not widely available on process nodes below 90nm due to scaling challenges [20]. The 130nm process results in a high energy consumption of 234 pJ/cycle at 16 MHz compared to modern 28nm/22nm microcontrollers which can achieve 4 pJ/cycle at 96 MHz [21].

Another 130nm FRAM-based NVP is demonstrated in [22]. This Arm Cortex-M0 implementation uses volatile retention FFs and ten 256-bit mini FRAM arrays to store FF state, finding a balance between a single central NVM array and completely distributed non-volatile FFs. This helps manufacturability and testability, with careful design, placement and synthesis to minimize the save/restore overheads compared to fully distributed NVPs.

An MRAM-based NVP is demonstrated in [23], achieving 145 pJ/cycle at 20 MHz on a 16-bit MSP430-class processor. The MRAM technology is based on a 3-terminal 2T1MTJ (2-Transistor 1-Magnetic-Tunnel-Junction) Spin Hall Effect device on a 90 nm CMOS platform. This is in contrast to the devices used in MEMIC evaluation, which are 2-terminal 1T1MTJ Spin-Torque-Transfer devices in 28/22nm foundry production [24], [25].

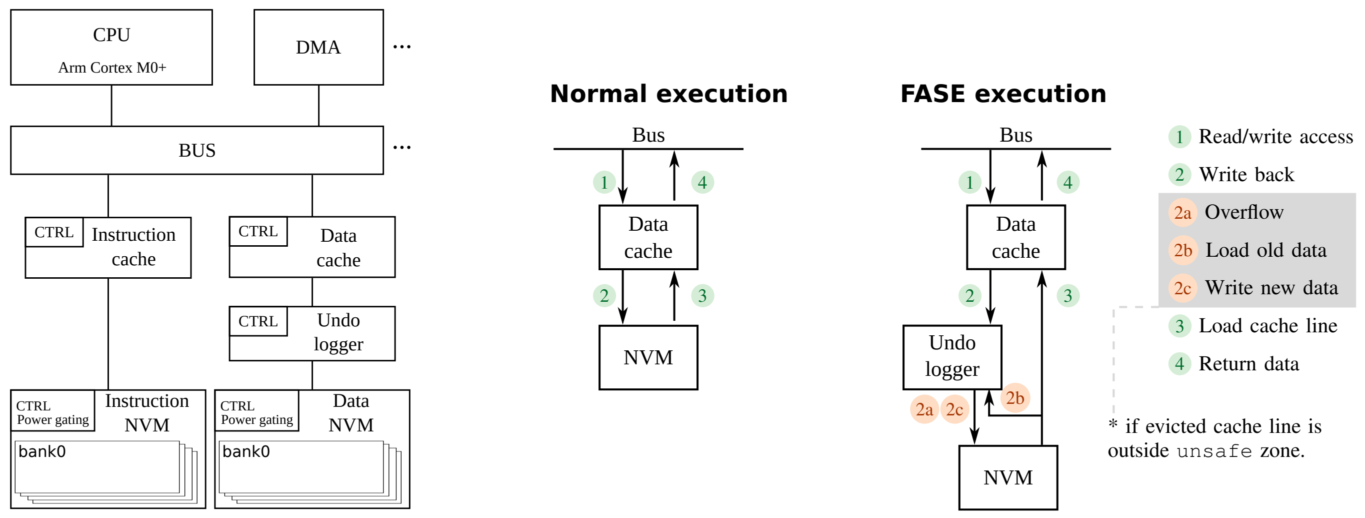
More recently, an ReRAM-based NVP with an NVSRAM macro is described in [19]. This achieves 33 pJ/cycle at 100 MHz on an 8051-class processor.

The deployment of an IC-capable SoC is a tradeoff between energy efficiency, performance, save/restore time and energy, programmability and development cost. Compared to programmable CPUs synthesized from standard logic libraries, all the reported NVPs achieve very fast and low-energy state save and restore. However, they present a number of deployment considerations. The use of non-volatile logic restricts them to older process nodes and impairs active energy efficiency. Synthesis, place and route of signals from a central NV controller may impede scalability. So far, only designs with a few thousand FFs have been demonstrated. Test and debug overheads have been addressed, but manufacturing yields will be lower for isolated, irregularly distributed NVM elements ([22]). NVPs currently require custom FF and RAM design and potentially non-standard fabrication, increasing their development cost compared to processors synthesized from standard libraries. Further, many SoCs may contain unmodifiable hard macro IP, and any state contained in these macros (for example analog trim) cannot be converted to non-volatile logic.

In contrast to the above reported NVPs, MEMIC proposes software-controlled checkpointing and FASE support, assisted by synthesizable logic on 28/22nm process technology using MRAM. No custom circuit design or fabrication is assumed, and production memory macros are evaluated as-is. While NVP research progresses towards better energy efficiency and deployment, there is still an important role for a system like MEMIC: a widely and easily deployable IC configuration which achieves energy efficiency, scalability and programmability while remaining maximally transparent to application programmers.

F. Hardware-supported IC

In contrast to NVPs, which use tight NVM integration at the level of flip-flops and memory cells, hardware-supported IC methods use functional subsystems or circuits to improve IC in an otherwise conventional SoC. *Clank* [26] proposes circuits that track memory accesses to detect, buffer and signal write-after-read sequences, so as to avoid idempotency violations. However, *Clank* does not protect against repeated IO bugs. *Freezer* proposes a simple hardware peripheral that controls accesses to SRAM and NVM, and tracks which blocks of SRAM have been written to in the current on-period, so as to avoid copying unmodified blocks [27]. As shown in our evaluation (Section IV-F), *Freezer* can be an effective way to reduce checkpointing energy. However, the worst-case checkpointing energy remains unchanged, so *Freezer* does not reduce the necessary amount of energy buffering. Additionally, *Freezer* considers only data memory, leaving out instruction memory. A hybrid MRAM-SRAM cache was proposed by Xie et al. [28] on a 480 MHz NVP simulated in *gem5*. It comprises a mix of SRAM blocks, which offer fast and energy-efficient access, and MRAM blocks, which offer non-volatility and higher density, and an access pattern predictor that intelligently allocates cache lines to either block type. This architecture may be efficient for a high-speed device with non-uniform memory access latency and energy (i.e. expensive access to main NVM). However, it is likely not applicable for ultra-low power IC devices that run at a few megahertz and thus would have similarly fast and energy efficient access to non-volatile cache blocks as to NVM. The proposed hybrid cache also does not include FASE support.



(a) Top-level architecture of *MEMIC*. (b) Data access during normal execution. (c) Data access during FASE execution.

Fig. 3. Overview of the *MEMIC* architecture, and data accesses during normal (undo logger disabled) and FASE execution (undo logger enabled). Annotatated steps are numbered in the order they are performed, although not all steps are performed for every memory access.

III. Memory-System Support for Reactive Intermittent Computing

This section describes the design of *MEMIC*. It begins by specifying design objectives, then presents the top-level architecture, followed by detailed description of the features that *MEMIC* implements to achieve the objectives.

A. Objectives

For the design of *MEMIC*, we set the following design objectives, derived from the background section and relevant literature [2], [3], [12], [29], [30]:

1) support FASEs, with minimal roll-back cost;
2) software-configurable limit of energy needed to suspend state, i.e. to back up volatile data in NVM;
3) minimal writes to NVM;
4) minimal suspend, restore, and roll-back energy;
5) minimal software complexity.

Firstly, *MEMIC* should support FASEs, as discussed in Section II; for the memory system, this implies support for rolling back state in case a FASE is aborted. Secondly, a limit on the energy it takes to suspend is required to guarantee that every suspend operation succeeds despite the finite energy buffer. This limit should be software-configurable such that the same *MEMIC* integrated circuit can be employed across a variety of applications, on a range of of printed circuit boards. Furthermore, this enables resiliency against adverse effects such as capacitor degradation over time and temperature by correspondingly adjusting the limit during deployment. In practice, this limit on the suspend energy can be implemented as a configurable limit on how many modified bytes are held in volatile (cache) memory. When this limit is reached, some modified state has to be saved in NVM before more modified state can be added. The final three objectives are optimization goals. Minimizing writes to NVM as well as suspend, restore, and roll-back overheads leads to better end-performance (i.e. application completion time). Minimizing software complexity eases the adoption of IC into a myriad of applications, ultimately accommodating widespread adoption of battery-less computing devices.

Prior works have demonstrated software implementations of the first two objectives. FASEs can be supported by data versioning or by inserting an extra checkpoint immediately before executing the FASE [6], as discussed in Section II-C. A limit on suspend-energy can be achieved by using software to track and limit modified state [15]. However, tracking modified state and supporting FASEs in software introduces additional software complexity by requiring special annotations, and can significantly degrade performance and energy efficiency.

Since the core operations of IC (restore, checkpoint, roll-back aborted FASE) consist of various forms of memory access tracking and control, they can be performed much more efficiently and transparently with hardware support than a pure software implementation.

B. Top-level architecture

To achieve the design objectives, *MEMIC* leverages existing memory subsystems like caches and an undo-log, specializes these for IC, and arranges them in a synergistic architecture that covers their weaknesses. Fig. 3a illustrates the top-level architecture, comprising a volatile instruction cache, a volatile data cache, a hardware undo logger, and non-volatile data and instruction memories. The overall goal is to combine volatile and non-volatile memory in a way that yields good energy-efficiency, by serving most accesses from volatile memory (cache) and carefully preserving state in NVM.

Caching of data and instructions is a well-known method which is used extensively to speed up memory accesses in high-performance computing. In the context of low-power computing, however, caching can also provide reduced energy consumption. The MSP430FR-series of low-power microcontrollers, for example, use a read-cache to reduce the energy consumption of their FRAM.

This paper shows that caching has further benefits in the context of IC. First, they mitigate the overheads of frequently rebooting, by loading data from NVM on demand instead of loading the entirety of data (and possibly instructions) to a separate SRAM during boot. Second, they inherently limit volatile state; a property *MEMIC* leverages and expands upon to enable software-control of the maximum suspend energy.

Employing data caching instead of a separate volatile memory can, however, complicate FASE support because software has less control over which variables are volatile and which are persistent, and must therefore assume that all variables are persistent. *MEMIC* solves this by using a hardware undo-logger. In a conventional hardware architecture, using an undo-logger would cause large overheads, as it logs every write access to NVM. However, in *MEMIC*, the undo-logger is placed behind the data cache, and thus only sees very infrequent write accesses (as shown later in Section IV-H).

MEMIC's caches both use pseudo-random replacement policy, which is a suitable for constrained devices due to its simplicity and robustness against cache-thrashing. The instruction cache is a write-through cache, as writes to instruction memory are assumed to be very infrequent (except during programming). The data cache, detailed in Section III-C, is a write-back cache to minimize NVM writes (and thus also pressure on the undo-logger during FASEs). Note that the caches and the undo logger are attached to the NVMs instead of being attached directly to the CPU. This arrangement ensures that memory accesses by peripherals (such as DMA) are treated the same as CPU accesses, and thus will not break idempotency or state retention. Figures 3b and 3c show memory operations during normal execution, and FASE execution, respectively; these are detailed in Section III-C and Section III-F. The `unsafe` zone allows certain memory accesses to bypass the undo logger, as detailed in Section III-F.

MEMIC requires that the platform has *some* energy stored in a capacitor (in the order of 1 µF), and that it has a supply voltage supervisor that signals a voltage warning when the supply voltage has dropped below a fixed threshold. The voltage warning can be achieved using a single voltage detector, and is usually already part of brown-out detection circuits on microcontrollers.

MEMIC provides software-configurable parameters to ensure portability across hardware platforms (i.e. different end-devices with varying capacitance and power consumption).

The key features implemented by *MEMIC* to achieve objectives 1-5 are as follows:

- a replacement policy that minimizes writes to NVM;
- **`MODMAX`:** a memory-mapped read/write register that sets a hard limit on the number of modified cache lines;
- **Undo-log:** Logging-support used for cache write-backs during FASE execution;
- **`unsafe` zone**: a region of memory that is excluded from undo logging.

These features are detailed in the following subsections.

C. Minimizing writes to NVM and limiting volatile state

Due to the relatively high write-energy and low endurance of NVM, design objectives 2 is to minimize writes to NVM. An effective way to achieve that is to employ a write-back data cache upstream of the NVM (shown in Fig. 3a). In contrast to write-through caches, which update both the internal version of a cache line and the downstream memory (NVM), write-back caches only update the internal version and thus reduce the number of NVM writes. Figure 3b shows the memory transactions to and from the cache. Most read or write accesses from the bus (chiefly from CPU or DMA), 1, are served by the cache (cache hit). Writes then only perform step 1, and reads perform 1 and 4 (return data). Read and write hits thus finish in a single clock cycle. However, if the cache does not contain the accessed cache line (cache miss), it has to load it first 3, consuming an extra clock cycle (assuming the NVM can be accessed in one clock cycle). If the cache has to evict a modified line to make room for the new line, a write back 2 occurs before the load 3; this cache miss with write back takes three clock cycles in total.

Cache evictions occur due to aliasing and limited capacity. The number of such evictions generally decreases with increased cache capacity and associativity[1]. Increasing the capacity or associativity, however, also increases power consumption and area. In addition to these well-known trade-offs, the configuration of the data cache also affects suspend energy: flushing a larger cache requires more energy.

An additional approach to reducing the number of write-backs in an associative cache is to bias the replacement policy, which selects the line in the set to be replaced when a cache miss occurs. By biasing the replacement policy such that it prefers evicting unmodified lines, the number of write-backs is reduced. We achieve this by modifying the data cache replacement policy such that, within a set, it always evicts an unmodified line if there is one; modified lines are only evicted if the set is filled with modified lines.

However, if the cache fills up with modified lines, the volatile state may grow larger than can safely be persisted on a power failure. Furthermore, some end-devices may have very stringent limitations on the energy buffer (capacitor). We therefore implement *MODMAX*, a hard limit on the number of modified lines in the data cache. It guarantees that the volatile state of the data cache never exceeds the limits of the energy buffer. When the total number of modified lines in the cache reaches *MODMAX*, the cache automatically evicts a single[2] pseudo-randomly chosen modified line so as to make it clean. The limit is software-configurable so that individual devices can be tuned for process variation, runtime variation and degradation over time, without requiring a variable suspend voltage threshold. In the simplest case, an appropriate value for *MODMAX* is found for a specific device, and set to a constant value during boot. In more advanced use cases, it can be adjusted throughout deployment. For example, as the storage capacitor degrades over the deployment lifetime of the

[1]Organizing a cache with set-associativity is common practice to reduce the number of conflicts (aliases) by enabling each cache line to have more than one possible storage location in the data array.

[2]Evicting multiple modified lines upon reaching *MODMAX* was also considered, but there is no benefit to spending N extra cycles to proactively write back N modified lines, versus spending one extra cycle to write back one line every time *MODMAX* is hit. In fact, evicting a single line at a time is preferable because it never writes back more lines than is necessary.

device, the limits can be reduced correspondingly so that the device remains functional.

Without *MODMAX*, the cache size would be limited by the minimum expected energy buffer. Or, put the other way, the chosen cache size would impose a minimum capacitor value on the end-device throughout its lifetime.

D. Suspend and restore during normal execution

When a power failure occurs during normal execution, a checkpoint is taken so that execution can resume from exactly where it left off when power returns. The suspend operation simply entails saving the processor registers to memory, then flushing the cache. This operation is powered by buffered energy alone, as we must assume the worst case where the power source has cut off completely.

When restoring after a power failure, the state is restored simply by loading processor registers. The cache logic automatically loads data as and when they are needed.

E. Suspend and restore during failure-atomic sections

Conversely, when a power failure occurs during a FASE, execution needs to restart from the beginning of the FASE, hence the current state must be discarded. To ensure that code outside of the FASE never gets re-executed, and to persist writes that occurred before the FASE, a checkpoint must be saved immediately before starting the FASE. When a FASE is aborted, the (volatile) state kept in the CPU registers and the cache is implicitly reset. Data which have been updated in NVM during the FASE, however, must be rolled back before execution can continue in the next on-period. *MEMIC* uses an undo logger to roll back such data, as described in the next subsection. Since the cache and CPU registers are not backed up when a FASE is aborted, the whole energy buffer can be used to apply the undo log, i.e. to roll-back persisted state to the beginning of the FASE.

F. Undo logging module

Because of possible memory corruption due to write-after-read and repeated-I/O violations (see Section II-B), it is imperative that any data that has been written to NVM during a FASE is rolled back before a FASE is restarted. Ideally, no data would be written to NVM during FASEs; roll-back would then simply consist of invalidating the cache and loading the checkpointed processor registers. However, due to limited cache capacity, write-backs may occur during FASEs too.

Taking inspiration from task-based IC [4], *MEMIC* employs undo-logging. However, in contrast to prior works, *MEMIC*'s undo-logger is only active while executing FASEs, is implemented in hardware, and is placed behind a write-back data cache. These are three factors which greatly reduce the energy overhead of the undo-logger, and also reduces its required logging capacity. When disabled, the undo logger simply forwards all writes to NVM without delay.

Figure 3c shows memory operations while the undo logger is enabled. Steps 1, 2, 3 and 4 are the same as during normal execution, except that write-backs to locations outside the `unsafe` zone are intercepted by the undo logger. When enabled, and a cache line write back 2 to an address outside the `unsafe` zone (see Section III-G) occurs, the undo logger first reads the old cache line from NVM and saves it, along with its address, in an internal *volatile* buffer 2b. Then, the write back is forwarded to NVM 2c, completing the write back. To prevent overflow, the undo logger automatically saves the oldest entry to NVM 2a when the size of the internal log exceeds a software-configurable threshold. The overflowed entries are saved at a location defined by a memory-mapped register. Software can then load and apply overflowed entries at the start of the next on-period. The overhead caused by the undo logger during cache write-backs is one cache line load (one clock cycle) when not overflowing, and an additional cache line write when overflowing (two clock cycles in total).

An alternative to undo logging would be to use a volatile write-back buffer that buffers cache write-backs during FASEs, and only commits the writes to NVM after the FASE has completed. However, the volatile state at the end of a FASE would then comprise both the modified cache lines and the lines held in the write-back buffer, hence increasing the necessary energy buffering. In contrast, when using an undo logger, the volatile state to be backed up is never larger than the maximum modified cache lines or the maximum undo log size, whichever is bigger. Both the undo log size (threshold) and the limit on modified cache lines are software-configurable.

G. The `unsafe` zone

We expect that the main use case for FASE is tasks such as taking sample windows of sensor data (e.g. audio or acceleration) or receiving radio packets, similar in nature to the *sample_window* function in Fig. 4. These are tasks that read a potentially large amount of data into structures that can readily be made insusceptible to write-after-read and repeated-I/O hazards; i.e. they can safely be overwritten if the FASE is re-executed, without first resetting the state. To reduce pressure on the undo log for applications that write excessive amounts of data (approaching the data cache size) during a FASE, *MEMIC*s undo logger implements an `unsafe` zone: *a region of memory which bypasses the undo logger*. The term *unsafe* is used to express that the protection normally provided by the undo logger does not apply; i.e. the programmer becomes responsible for avoiding re-execution bugs for variables they choose to allocate to the `unsafe` zone. A pair of memory mapped registers are used to define the base and bound of the `unsafe` zone. The `unsafe` zone is opt-in rather than opt-out, so that an unmodified program will execute correctly, albeit with sub-optimal performance. While specific variables that are allocated to an `unsafe` zone are excluded from the undo log, and thereby not protected against write-after-read and repeated-I/O hazards, all other variables are protected by default. In Fig. 4, the programmer knows that `data` can be safely ignored from logging, so allocates it to the `unsafe` zone to improve performance.

While we have assumed manual allocation of data arrays into `unsafe` in this section, methods to automatically detect write-after-read and repeated-IO hazards [2] could be applied to automatically allocate data to the `unsafe` region.

```
UNSAFE uint32_t [WINDOW_SIZE] data;

void sample_window(uint32_t *data) {
  for (int i = 0; i < WINDOW_SIZE; ++i;) {
    data[i] = readSensor();
    sleep(10, TIME_US);
  }
}

void main () {
  run_atomic(sample_window, data);
}
```

Fig. 4. A simple sensor-sampling FASE showing the usage of the `unsafe` zone and how FASE can be annotated.

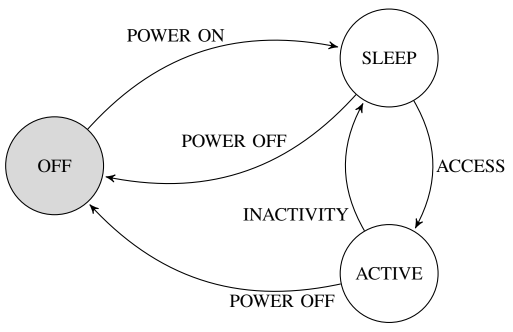


Fig. 5. State machine for power gating inactive memory banks

TABLE I
SIMULATION PARAMETERS.

Parameter	Value
CPU	Cortex M0+
CPU frequency	1.25 MHz
CPU active current	1.0 µA
Instruction memory size	128 kB (32 banks of 4 kB)
Data memory size	32 kB (8 banks of 4 kB)
Storage capacitor	10 µF
Supply power (P_{supply})	10 µW
Core voltage (V_{core})	1.8 V
On-voltage (V_{on})	2.6 V
Suspend Voltage Threshold (V_{warn})	2.1 V
SRAM read/write	0.33 pJ/b
SRAM leakage (active)	52.9 pA/b
SRAM leakage (retention)	11.2 pA/b
SRAM activate bank	94 pJ

IV. Evaluation

This section evaluates the performance of *MEMIC* against several relevant baselines. After describing the experimental setup, workloads and baselines, cache configuration is discussed. Then we separately explore the instruction cache and the data cache and compare them to their relevant baselines. We then investigate the permissible operating conditions for *MEMIC* and the baselines. Next, we perform a case study on a solar-powered IC device running a realistic sensing, computation, and logging workload. Finally, we briefly discuss NVM write-endurance.

For the purposes of reproducibility, all models and simulation scripts used to obtain the presented results, as well as the results themselves, are made available in the public dataset at https://doi.org/10.5258/SOTON/D2186 and simulation package at https://github.com/UoS-EEC/MEMIC.

A. Experimental setup

The evaluation is performed using *Fused* [10], an open-source *SystemC*-based simulator built for IC. Importantly for this work, *Fused* simulates energy and execution in a closed feedback loop, enabling the exploration of the effects of various design choices on the overall performance of a system that is intermittently powered. *Fused* has been proven to accurately model real hardware, including digital and analog components, running intermittently, and is therefore a suitable evaluation method for this work.

The simulation parameters are listed in Table I. A power source that outputs constant power, P_{supply}, is used as a simple model of real energy harvesters, which typically have decreased output current at higher output voltage (and vice versa). In each simulation time step, it adjusts its output current according to load voltage to achieve constant power output. Using a constant power source instead of a specific energy harvester model facilitates analysis (for example, energy can be calculated as the product of time and the power supply setting), and makes the results more readily transferable to a specific setup (energy harvester, capacitor, voltages etc.).

The device turns on when the voltage across the storage capacitor, v_{cap} reaches the on-voltage V_{on}. When v_{cap} discharges below the voltage warning threshold, V_{warn}, an interrupt is issued to trigger a suspend (or abort a FASE). After suspend completes, the device shuts off and charges back up to V_{on}. If v_{cap} drops below V_{core}, the device shuts off regardless of whether a suspend has completed; shut-off without completing suspend results in system failure, and should never occur in a properly configured device.

Dividing memories into banks allows the power gating of inactive banks to reduce leakage power. For example, if an application only uses a fraction of available memory for extended periods of execution, the inactive banks can be powered down. For NVMs like MRAM, that could mean powering down a bank completely (as it is non-volatile), or powering down certain (power-hungry) parts of the macro.

To ensure fair evaluation, all SRAM and MRAM memories are divided into banks with automatic retention modes. The caches are not power gated, as we consider them too small for this method to yield significant benefits.

This is consistent with modern practices for low-power devices. Importantly, it also ensures that our evaluation does not overestimate the leakage power of the baseline methods that employ separate SRAM memories. When comparing a small cache to a large SRAM, we cannot neglect the possibility that most of the large SRAM is in fact kept in a low-power retention mode when inactive. The power gating scheme employed for evaluation is shown in Fig. 5, and is loosely based on prior work [31], [32]. In essence, each bank is individually controlled, and will only become active when accessed, and will again enter a retention mode when an inactivity timer expires. These power mode transitions cost energy, so a trade-off exists between too frequently changing

TABLE II
WORKLOADS AND THEIR CODE AND DATA FOOTPRINTS.

Workload	Code Size (kB)	Data Size (kB)
bc	3.4	2.05
nn-gru-cmsis	7.12	8.56
matmul	4.46	4.37
fft-q31-cmsis	75.45	2.8
crc	5.17	4.03
aes	5.75	4.04
ar	3.98	2.29
qrencode	13.29	2.08
nn-gru-cmsis-fase	5.73	7.46

the power modes of the banks, and leaving them on for too long and thus consuming excessive leakage power. In this evaluation, we use an inactivity timer of 1000 clock cycles, which we found to give a reasonable balance.

The energy consumption of the SRAM and MRAM memories are modeled as read and write energy per accessed bit, leakage current per bit according to operating state (ACTIVE/SLEEP/OFF), and the transition energy between operating modes. The data and tag arrays of the caches are modeled to have the same access energy and leakage current per bit as the SRAM memory in ACTIVE-state. The parameters were obtained using Arm's production MRAM [8] and SRAM [9] compilers, targeting commercial 28 nm FDSOI and 22 nm bulk processes, respectively. While used in our experiments, the specific values of the MRAM parameters are confidential, and are therefore omitted from Table I (and altered in the public *MEMIC* simulation package). Note however, that since we release all source code associated with this paper, readers can readily repeat the presented simulations for the particular parameter values relevant to their target technology. Since this work focuses on the memory system, the CPU power consumption is modeled as a constant current.

B. Workloads

Table II lists the workloads used for evaluation, all of which are included in *MEMIC*'s simulation package. These workloads were selected based on being relevant to the domain and on having diverse code and data footprints and memory access patterns. Two of the workloads use the *Arm CMSIS5 library* without modification, showing that *MEMIC* is directly compatible with existing complex code-bases. All workloads except the last are computational workloads which perform computation on data stored in the program binary. The last workload, `nn-gru-cmsis-fase`, which exercises *MEMIC*'s FASE support by sampling sensor data, is described in Section IV-H.

C. Baseline

This subsection describes the baseline methods used for evaluation. In regards to instruction memory, *MEMIC* is compared to two baseline configurations, namely *ExecuteInPlace* and *LoadExecute*. *ExecuteInPlace* executes instructions directly from MRAM, which simplifies software and minimizes area. However, the read energy of MRAM is much higher than that of SRAM, so it may not be the most energy efficient solution. *LoadExecute*, on the other hand, executes instructions from a separate instruction-SRAM, using Direct Memory Access (DMA) transfers to load instructions from NVM to SRAM during boot. Thus *LoadExecute* decreases read energy significantly. However, *LoadExecute* also increases area and leakage power significantly, as the instruction SRAM must house as many bits as the instruction MRAM. A variation on *LoadExecute* could be to map the most active code to VM and the rest to NVM [33]. However, this requires the developer to statically analyze or profile the code before deployment to determine what to allocate to VM, and thus to make assumptions about how the device will operate in the future. In the context of IC, execution strongly depends on energy conditions, and so the ideal mapping found during profiling may not match conditions later on.

As the baseline for data memory, *AllocatedState* [1], [15] and *Freezer* [27] are used. Both baselines use a data-SRAM that is loaded from NVM during boot, and checkpointed when power fails. Note that the equivalent of an *ExecuteInPlace* for data is infeasible due to the high write energy of MRAM (in addition to the complexity of rolling back state after a failed FASE[3]). Similarly, methods that map certain portions of variable-sections (stack, `.data`, etc.) to NVM [33] would be ineffective with MRAM, as they increase NVM write-frequency. Increasing the write-frequency on MRAM is highly detrimental to performance as write accesses can require several orders of magnitude more energy than write accesses to SRAM. Note, however, that MRAM is still superior to FRAM, partly because it is compatible with more advanced process nodes, as discussed in the background section.

AllocatedState is a software-based IC method based on *Hibernus* [1], that checkpoints and restores all allocated volatile memory (i.e. `.data`, `.bss`, `.stack`) to and from NVM when power respectively fails or recovers. Our implementation of *AllocatedState* transfers data using DMA to improve efficiency. *Freezer* can be described as an optimization of *AllocatedState*, where only modified data is written to NVM when suspending state. Our implementation of *Freezer* [27] is a memory-mapped peripheral that tracks writes to data-SRAM in order to record which 32 B blocks have been written to. Each block's state is represented by a single *dirty bit* in the write tracker's register file. Software then uses the *dirty bits* to set up DMA transfers of modified blocks when suspending state. When there are multiple contiguous modified blocks, they get aggregated to one large DMA transfer to reduce setup-overhead. Note that all methods evaluated in this paper push the core CPU registers (`r0-r12`, `lr`, `pc`) to the stack before the stack/data is saved to NVM.

D. Cache configuration

Caches have various parameters that affect their power, performance and area (PPA), primarily the line width (LW),

[3]FASE support for a system using only non-volatile data memory would require double buffering, which in turn leads to greatly increased number of NVM accesses as data is copied between the buffers.

associativity (AS) and number of sets (NS). The capacity of the cache, is calculated as $LW \cdot AS \cdot NS$.

To find optimal cache parameters, we ran design space exploration using *Fused*. A total of 48 combinations of LW, AS and NS for instruction caches and 79 combinations for data caches of size 1–8 kB were simulated for all the computational workloads in Table II. Each workload was run for a number of iterations such that the total on-time exceeded five seconds, to ensure that each workload required several power cycles to complete. The number of power cycles to complete each workload ranged from 19 to 243 (best configuration on shortest workload to worst configuration on longest workload). For both the instruction and data cache, the cache configuration that yielded the lowest geometric mean completion time across all workloads was chosen for *MEMIC*.

The results are presented for the instruction and data caches in the next two respective subsections. For brevity, the presentation is focused on three distinctive workloads that have very different combinations of instruction and data footprints. The completion times of all workloads are shown later, in Table III (10 µF columns). Detailed results for all workloads are available in the public dataset.

E. Instruction Cache

In this subsection, we investigate the relative performance between two baseline methods and different instruction cache configurations (LW, AS and NS). For consistent results, all experiments in this subsection use the baseline *AllocatedState* method for data memory.

Figure 6 shows the energy consumption of instruction memory per executed instruction, and the workload completion time, for four different configurations along the horizontal axis. Each plot shows the result for an individual workload. The energy consumption of the SRAM instruction memory (baseline) and the instruction cache are both shown as SRAM (Read/Write/Leakage). The energy consumption of *ExecuteIn-Place* comprises NVM read energy and NVM leakage, as no cache or SRAM is used. The relatively high NVM read energy leads to much higher energy consumption, and thus longer charging time than other configurations. *LoadExecute* mitigates the NVM read energy, as each instruction is only read from NVM once (during boot). The drawback with *LoadExecute* is the area and leakage introduced by the SRAM memory, as well as the time and energy taken to load the instructions. For applications with a large instruction footprint, such as the `fft-q31-cmsis` workload, the overall time and energy consumption is dominated by loading instructions, as indicated by the significant increase in on-time and NVM read energy, as well as the increased SRAM leakage due to more SRAM banks being active. Even though only a small portion of the program is executed in each power cycle, *LoadExecute* loads the whole program.

The latter two configurations use an instruction cache in place of the instruction SRAM; the first of them is the best cache configuration for the specific workload, and the second is the best overall configuration (lowest geometric mean completion time across all workloads). In addition to reducing NVM reads compared to *ExecuteInPlace*, the instruction cache solves the instruction loading issue of *LoadExecute*, because the cache mechanism only reads instructions from NVM when they are needed. Additionally, the cached systems reduce area and leakage power compared to *LoadExecute*, because a small cache can cover a large instruction space.

Across all workloads, our results show that a 4 kB instruction cache, arranged as 16 B line width, 2-way set associativity and 128 sets, reduces the workload completion time by 60–77 % (71 % mean) and 41–70 % (49 % mean) compared to the baseline *ExecuteInPlace* and *LoadExecute* configurations, respectively. As these simulations were done using a constant-power supply, these time savings correspond to equal energy-savings. Furthermore, this 4 kB cache uses 1536 b of tag and 32 768 b of data SRAM bit cells, compared to the 1 048 576 b used by the 128 kB SRAM; a reduction of 97 %. Assuming the cache logic has little overhead over the access logic of the large SRAM, this could lead to a substantial area reduction.

F. Data Cache

In the previous subsection, a 4 kB instruction cache was found to be the best option. We now investigate the data memory to find the relative performance between *AllocatedState*, *Freezer*, and *MEMIC* configurations using a *data* cache. For fair comparison, and to isolate the effect of data memory, all simulations use the chosen 4 kB instruction cache.

Figure 7 shows the energy per instruction for *data* accesses and the total runtime, broken into active and charging, for four configurations along the horizontal axis. Each plot shows the result for an individual workload. The main inefficiencies of *AllocatedState* are caused by the data SRAM, which needs to be large enough to fit all volatile state, and the fact that all volatile state has to be loaded and backed up, regardless of whether it is modified (or even used) during the current on-period. For workloads with a small data footprint and good access locality, such as the `aes` workload, these inefficiencies may be acceptable. However, the NVM write energy (caused by suspend) grows with the data memory footprint, as seen when comparing `aes` and `nn-gru-cmsis` in Fig. 7. Suspend energy is explored further in Section IV-G. For workloads with poor access locality or a large active data set (`fft-q31-cmsis` and `nn-gru-cmsis`), leakage energy also grows because more SRAM banks stay active. *Freezer* significantly reduces the average NVM write energy by avoiding back-up of unmodified data for workloads with a significant data footprint where not all data is modified in every power cycle. The worst-case NVM write energy for *Freezer*, however, where all data has been modified, remains the same as for *AllocatedState* (in fact slightly worse due to tracking overhead). And *Freezer* does not improve on the other components of the total energy consumption, such as SRAM leakage. These inefficiencies and inconveniences are alleviated by using a data cache instead of data SRAM.

The data cache has higher energy consumption per access than SRAM, due to cache misses, the overhead of checking tags and, to some degree, loading more data than was requested (i.e. due to the cache line granularity). It can also

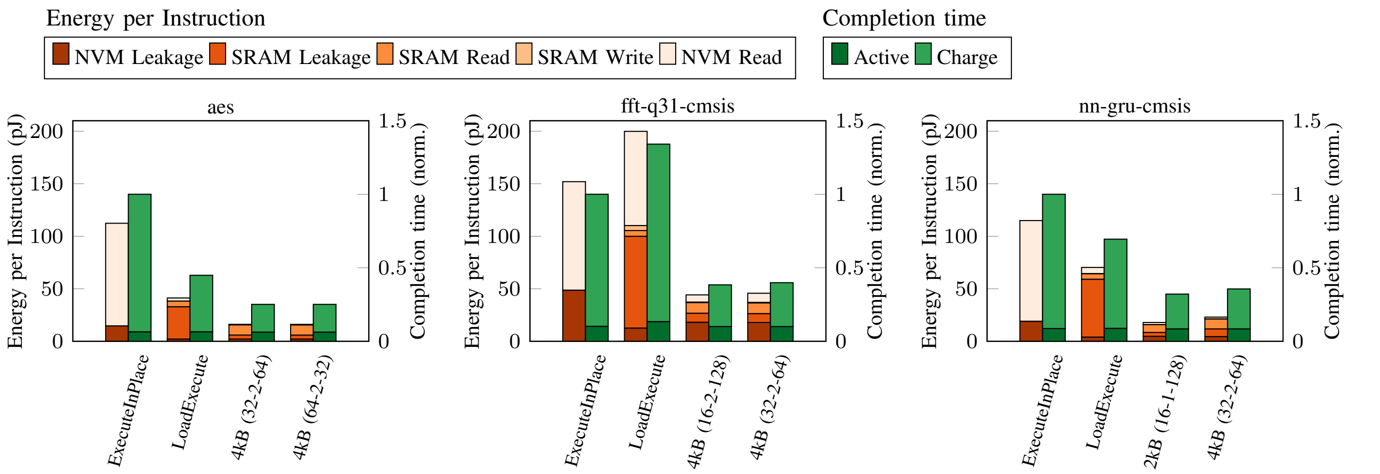


Fig. 6. Instruction access energies per executed instruction (left bars) and workload completion times (right bars, normalized to *ExecuteInPlace*) for different instruction memory architectures. The cache configurations are denoted as "size (Line Width-Associativity-Sets)".

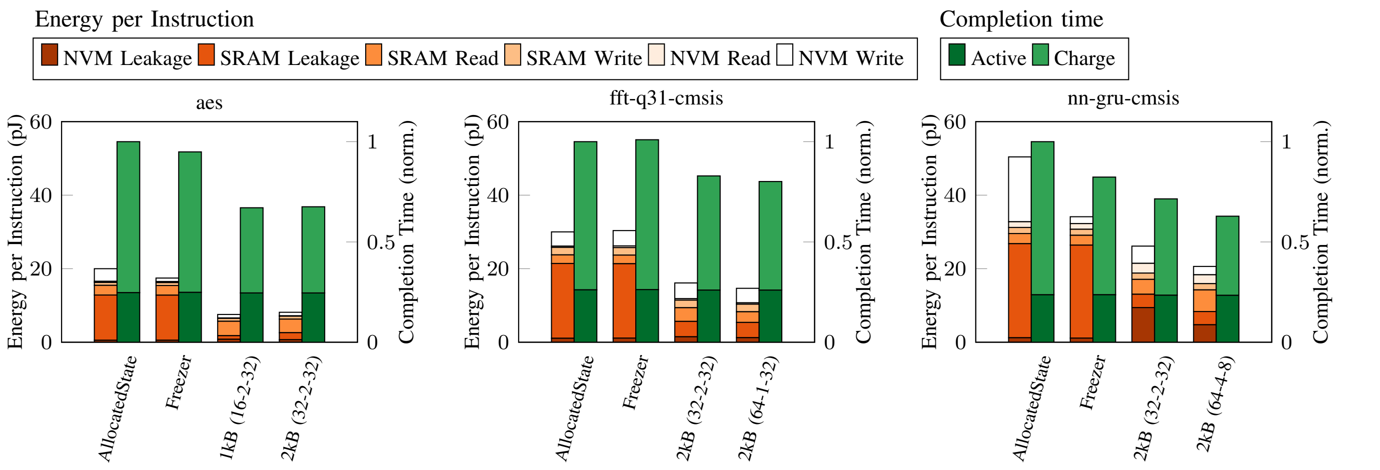


Fig. 7. Data access energies per executed instruction (left bars) and workload completion times (right bars, normalized to *AllocatedState*) for different data memory architectures. The cache configurations are denoted as "size (Line Width-Associativity-Sets)". Note that runtime is dependent on total power consumption, not just the component attributed to data access. Hence a large reduction in data access energy leads to a proportionally smaller reduction in total runtime. All configurations use the 4 kB (16-2-128) instruction cache.

lead to increased NVM leakage and access energy due to cache misses. The cache leakage energy, however, is much lower than that of the SRAM required by *AllocatedState* and *Freezer*, due to its smaller size.

Across all workloads, the 2 kB data cache, arranged as 32 B line width, 2-way set associativity and 32 sets, reduced the workload completion time by 17–39 % (26 % mean) and 13–39 % (23 % mean) compared to *AllocatedState* and *Freezer*, respectively. This 2 kB data cache uses 320 b of tag and 16 384 b of data SRAM bit cells, compared to the 262 144 b used by the 32 kB data SRAM; a reduction of 94 %.

G. Operating conditions

This section evaluates the energy required to suspend state and the workload completion times of *MEMIC*, *AllocatedState* and *Freezer* under different operating conditions. Certain systems may have less energy available for suspending state, either because of lower maximum supply voltage (hence also lower V_{warn}), or because the energy buffering capacitance is lower. We focus on the capacitor size, but the same analysis can also be solved for voltage. Figure 8 shows the suspend energy and minimum required capacitance for three workloads. Two extra configurations are shown in the figure; *MEMIC-MM32* and *MEMIC-MM16* show results for *MEMIC* when the number of modified cache lines is limited to 32 and 16, respectively.

The energy consumed for suspending state, $E_{suspend}$, was calculated from simulation results as follows:

$$E_{suspend} = E_{warn} - E_{done} + E_{supply} \quad (1)$$
$$= \tfrac{1}{2}CV_{warn}^2 - \tfrac{1}{2}Cv_{done}^2 + P_{supply}t_{suspend} \quad (2)$$

where E_{warn} is the stored energy when the voltage warning

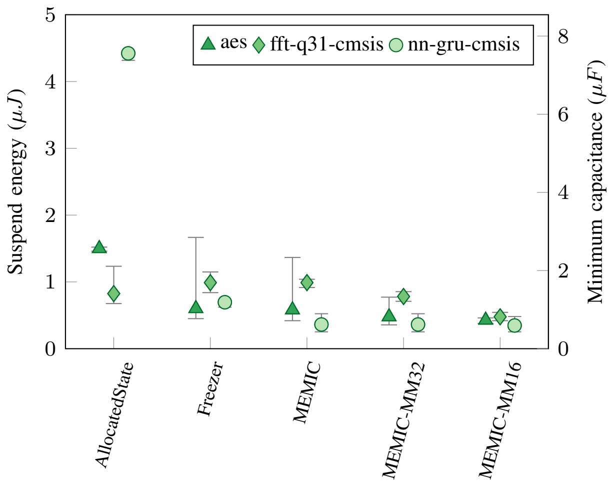


Fig. 8. Energy consumption of suspend (left vertical scale), and the corresponding minimum required capacitance (right vertical scale). The points show the mean value, and the bars show the minimum and maximum values. *MEMIC-MMxx* denotes *MEMIC* with the limit on modified lines (*MODMAX*) set to *xx*. *MEMIC* is the default configuration where *MODMAX* is set to the total number of lines in the cache.

is issued, E_{done} is the stored energy when suspend has completed, v_{done} is the capacitor voltage when suspend has completed, $t_{suspend}$ is the time it took to suspend, and C is the energy buffering capacitance.

Based on $E_{suspend}$, the minimum required capacitance, C_{min}, was calculated as follows:

$$E_{avail} = E_{warn} - E_{min} \tag{3}$$

$$= \tfrac{1}{2}CV_{warn}^2 - \tfrac{1}{2}CV_{core}^2 \tag{4}$$

$$C_{min} = \frac{2E_{suspend}}{V_{warn}^2 - V_{core}^2}, \tag{5}$$

where E_{avail} is the energy available for suspending and E_{min} is the stored energy when the stored voltage is equal to the minimum operating voltage for the system. Note that the required capacitance is linear with the required energy.

Table III lists the completion time of all workloads for three capacitor values, comparing *AllocatedState*, *Freezer* and *MEMIC*. All configurations use the 4 kB (16-2-128) instruction cache. *MEMIC*'s *MODMAX* parameter was adjusted according to capacitor size. We found empirically that *MODMAX*$= 400 \cdot C \cdot 10^6 / LW$ was a reasonable value (i.e. 400 B/µF) in this situation. By comparing Table III and Fig. 8, we can see why e.g. `nn-gru-cmsis` fails on *AllocatedState* when the capacitor size is 1.0–4.7 µF, but succeeds at 10 µF: ≈8 µF is the minimum required capacitance. In practice, these results mean that *AllocatedState* needs a larger capacitor and/or a higher suspend voltage threshold to run workloads with a large data footprint. This, in turn, shows that the application programmer has to re-evaluate the capacitor/suspend voltage threshold whenever the application changes; for a complicated end-device, this dependency between application and electrical properties can substantially complicate development. This also applies for *Freezer*, because, in the worst case where all data has been modified (e.g. a power failure after a long on-period), *Freezer* saves as much data as *AllocatedState*. *MEMIC*, on the other hand, never holds more modified state than is permitted by *MODMAX*, up to a maximum of the size of the data cache, regardless of the application. *MODMAX* enables the end-device to run with a smaller capacitor and/or a lower suspend voltage threshold; both of which can have several other benefits, such as improving the energy efficiency of the energy harvester and reducing conversion loss. As expected, reducing *MODMAX* can result in performance degradation, because the cache's ability to buffer writes is reduced. Compared to *MEMIC*, this performance degradation resulted in less than 2 % increased completion time for most workloads under the *MM32* and *MM16* configurations. For `fft-q31-cmsis`, however, the increase was 66 % and 212 % for *MM32* and *MM16*, respectively. The sharp degradation in `fft-q31-cmsis` is caused by frequent data writes with poor locality.

H. Case study: Solar-powered sensor node

To evaluate *MEMIC* under realistic conditions, we extended the base simulation model with a PV-cell, a boost regulator, and an accelerometer, as shown in Fig. 9, and implemented a realistic IoT workload. In each simulation time step, the single-diode PV-cell model [34] takes as input the luminance and the stored voltage v_{in}, to calculate its output current, which charges the 10 µF input capacitor C_{in}. v_{in} is then boosted up to 1.8 V by the boost regulator, with an efficiency of 80 %. The boost regulator is loosely modeled after the *Texas Instruments BQ25570* energy harvesting power management module. When v_{in} exceeds 1.4 V and the boost regulator's OOK (output OK) signal is high, the SVS connects the microcontroller and accelerometer to the 1 µF output capacitor C_{out}. Later, when v_{in} discharges below 0.3 V, SVS issues a voltage warning (WARN) which triggers a checkpoint (or FASE abort) in the microcontroller.

The sensing workload, `nn-gru-cmsis-logger`, records a window of 100 three-axis accelerometer values sampled at 1 kHz and uses the sampled data as input to a Gated Recurrent Unit neural network. The accelerometer is a *Fused* model of the *Bosch BMA280*. It communicates over SPI, and implements a 1 kB internal FIFO buffer. Its function and power consumption is modeled based on the device data sheet. The sample window is recorded within a FASE, so that the samples within a window are guaranteed to be continuous. While sampling, the accelerometer buffers data, and the CPU sleeps. When 100 samples (700 B[4]) have been buffered, the accelerometer requests an interrupt via GPIO, causing the CPU to wake up and read the data into memory; this concludes the FASE. The workload then proceeds to run the neural network intermittently over several power cycles.

Figure 10 shows the average completion time of the workload under different lighting conditions (across $20 - 120$ iterations), divided into the time spent charging, actively executing, and sensing while the CPU sleeps. Compared to the baselines, *MEMIC* is able to complete the workload under lower light conditions, and is the only workload to succeed at 800 lux. In the range of 1000–1600 lux, *MEMIC* completes the workload 6–27 % (21 % mean) and 10–31 % (22 % mean) faster than

[4]Each sample comprises 6 B of data and a 1 B header.

TABLE III
COMPLETION TIMES, IN SECONDS, OF ALL WORKLOADS FOR THREE CAPACITOR SIZES.

	AllocatedState			Freezer			MEMIC		
Workload	1.0 uF	4.7 uF	10 uF	1.0 uF	4.7 uF	10 uF	1.0 uF	4.7 uF	10 uF
fft-q31-cmsis	fail	0.21	0.18	fail	0.21	0.18	0.99	0.17	0.15
ar	5.11	3.41	3.31	4.75	3.39	3.23	3.51	2.64	2.50
aes	fail	21.26	18.38	fail	18.55	17.44	17.99	13.36	12.47
qrencode	23.79	17.12	16.34	16.02	17.14	16.36	15.46	12.73	12.19
nn-gru-cmsis	fail	fail	1.54	fail	1.40	1.27	1.49	1.14	1.10
matmul	fail	10.49	9.39	fail	10.49	9.39	7.46	5.91	5.66
bc	4.30	3.31	3.45	4.37	3.64	3.45	3.41	2.92	2.85
crc	fail	5.05	4.54	fail	4.53	4.34	6.16	3.96	3.50

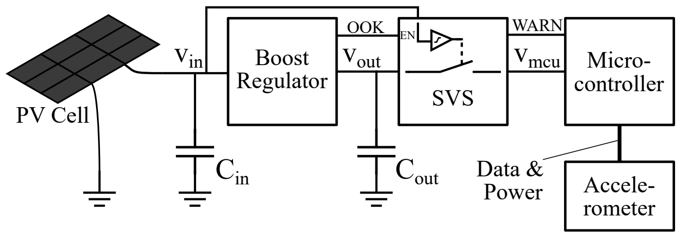


Fig. 9. Simulation model of a solar-powered IC device.

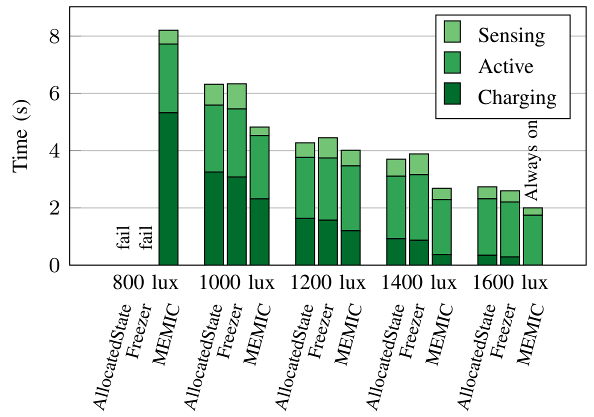


Fig. 10. Average completion time of the `nn-gru-cmsis-logger` workload on the solar-powered IC device (Fig. 9) under different lighting conditions. The two baseline configurations use the same instruction cache as *MEMIC*.

AllocatedState and *Freezer*, respectively. At 1600 lux and above, *MEMIC* stops power cycling, as the power input is higher than active power. The same applies for the baseline methods at 1800 lux and above, thus at high light levels all methods complete the workload in approximately the same amount of time (within 1 %).

By further analyzing the simulation data from the PV-cell case study, we can evaluate the overhead and efficacy of the undo-logger under realistic conditions. Across all simulations, up to 0.19 % of all write accesses *during FASE execution* were logged by the undo logger. When not using the `unsafe` zone, this grew to 0.65 %, albeit with insignificant performance overhead (completion time was within 1 %). Despite providing small improvements in this workload, the `unsafe` zone is an easy optimization to use, and can be important in particular workloads that write a large amount of data (approaching the data cache size) inside a FASE.

I. Endurance

Current MRAMs are limited in endurance, i.e. may fail after a specified number of writes to the same bit-cells. To assess whether *MEMIC* has an impact on NVM endurance, we instrumented *Fused*'s memory model to record the maximum total number of writes to a single location (byte) in data-MRAM. On average, *MEMIC* had 2 % more writes to the same location per completion of each workload, and thus does not significantly impact write endurance.

V. Conclusions and Future Work

Instruction caching substantially improves energy efficiency and performance under intermittent operation: partially by reducing access and leakage energy, but also by avoiding unnecessary loading of the entire program during boot, when energy is scarce and on-periods are short. By simulating 48 instruction cache configurations on eight workloads running intermittently, we found that a 4 kB instruction cache arranged as 16 B line width, 2-way set associativity and 128 sets, reduces workload completion time by 60–77 % (71 % mean) and 41–70 % (49 % mean) compared to the baseline *ExecuteInPlace* and *LoadExecute* configurations, respectively. By similarly simulating 79 data cache configurations, we found that a 2 kB data cache arranged as 32 B line width, 2-way set associativity and 32 sets, provided a further reduction of completion time by 17–39 % (26 % mean) and 13–39 % (23 % mean) compared to the relevant baselines *AllocatedState* and *Freezer*, respectively. As these simulations were carried out using a constant-power supply, these time savings correspond to equal energy savings. Both caches also present a substantial decrease in area, as they reduce the number of SRAM bit cells by over 90 % as compared to the SRAMs needed by the baseline methods.

Experiments running the eight workloads with three different capacitor sizes showed that the baseline methods failed to complete some workloads when the capacitor size was smaller than 10 µF, because their operation is conditional on the application's memory usage. In contrast, *MEMIC* successfully ran all workloads for the tested capacitor sizes by using *MODMAX* to adapt to operating conditions. Similarly, when modelling a solar powered IC device, running a realistic logging workload, *MEMIC* was able to complete the workload under lower light conditions, and with better performance (6–31 %) across light conditions ranging from 1000 lux to 1600 lux.

The *MEMIC* architecture and the simulation package released with this work can be used as a tool for further intermittent computing research. Important topics include wear-leveling for NVMs, accelerating other IC methods such as Chen et al. [35], memory power gating methods, scheduling of FASEs to minimize energy wastage caused by re-execution, and dynamically adapting *MODMAX* according to operating conditions. Given that the cache and undo-logger in *MEMIC* is placed directly in front of the NVM, *MEMIC* is also compatible with multiple CPU cores and could be combined with other works that enable IC for such systems [36].

References

[1] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Syst. Lett.*, vol. 7, no. 1, pp. 15–18, Mar. 2015.

[2] M. Surbatovich, L. Jia, and B. Lucia, "I/O Dependent Idempotence Bugs in Intermittent Systems," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 183:1–183:31, Oct. 2019.

[3] B. Ransford and B. Lucia, "Nonvolatile Memory is a Broken Time Machine," in *Proc. Workshop Memory Syst. Perform. Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:3.

[4] B. Lucia and B. Ransford, "A Simpler, Safer Programming and Execution Model for Intermittent Systems," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 575–585.

[5] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent Execution Without Checkpoints," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 96:1–96:30, Oct. 2017.

[6] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems," *IEEE Trans. Comput.*, vol. 68, no. 9, pp. 1390–1403, Sep. 2019.

[7] C.-K. Kang, C.-H. Lin, P.-C. Hsiu, and M.-S. Chen, "Homerun: Hw/sw co-design for program atomicity on self-powered intermittent systems," in *Proc. Int. Symp. Low Power Electronics Des.*, ser. ISLPED '18. New York, NY, USA: Association for Computing Machinery, 2018.

[8] E. M. Boujamaa *et al.*, "A 14.7Mb/mm ^{2} 28nm FDSOI STT-MRAM with Current Starved Read Path, 52Ω/Sigma Offset Voltage Sense Amplifier and Fully Trimmable CTAT Reference," in *2020 IEEE Symp. VLSI Circuits*. Honolulu, HI, USA: IEEE, Jun. 2020, pp. 1–2.

[9] G. Rangarajan, "Arm Delivers a Comprehensive Physical IP Platform for Optimized SoCs with TSMC 22nm ULP/ULL Process Technology," Oct. 2018.

[10] S. T. Sliper, O. Cetinkaya, A. S. Weddell, B. Al-Hashimi, and G. V. Merrett, "Energy-driven computing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2164, p. 20190158, Feb. 2020.

[11] B. Ransford, J. Sorber, and K. Fu, "Mementos: System Support for Long-running Computation on RFID-scale Devices," in *Proc. Sixteenth Int. Conf. Architectural Support Program. Lang. Operating Syst.*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 159–170.

[12] K. Maeng and B. Lucia, "Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing," in *13th {USENIX} Symp. Operating Syst. Des. Implementation ({OSDI} 18)*, 2018, pp. 129–144.

[13] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "Everything leaves footprints: Hardware accelerated intermittent deep inference," *IEEE Tran. Comput.-Aided Des. Integr. Circuits Syst. (TCAD)*, vol. 39, no. 11, pp. 3479–3491, 2020.

[14] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, ser. PLDI 2020. London, UK: ACM, Jun. 2020, pp. 1005–1021.

[15] S. T. Sliper, D. Balsamo, N. Nikoleris, W. Wang, A. S. Weddell, and G. V. Merrett, "Efficient State Retention Through Paged Memory Management for Reactive Transient Computing," in *Proc. 56th Annu. Des. Autom. Conf. 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 26:1–26:6.

[16] A. Rodriguez Arreola, D. Balsamo, G. Merrett, and A. Weddell, "RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems," *Sensors*, vol. 18, no. 2, p. 172, Jan. 2018.

[17] D. Balsamo *et al.*, "Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices," *IEEE Tran. Comput.-Aided Des. Integr. Circuits Syst. (TCAD)*, vol. 35, no. 12, pp. 1968–1980, 2016.

[18] Y. Liu *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proc. 52nd Annu. Des. Automat. Conf. on - DAC '15*. San Francisco, California: ACM Press, 2015, pp. 1–6.

[19] Z. Wang *et al.*, "A 65-nm ReRAM-Enabled Nonvolatile Processor With Time-Space Domain Adaption and Self-Write-Termination Achieving 4x Faster Clock Frequency and $>$ 6x Higher Restore Speed," *IEEE J. Solid-State Circuits*, vol. 52, no. 10, pp. 2769–2785, Oct. 2017.

[20] Y. Kato *et al.*, "Embedded feram challenges in the 65-nm technology node and beyond," in *Int. symp. app. ferroelectrics*. IEEE, 2006, pp. 81–84.

[21] Apollo4 blue, https://ambiq.com/apollo4-blue. Ambiq Micro.

[22] S. C. Bartling, S. Khanna, M. P. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams, "An 8mhz 75µa/mhz zero-leakage non-volatile logic-based cortex-m0 mcu soc exhibiting 100% digital state retention at vdd=0v with lt;400ns wakeup and sleep transitions," in *IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers*, 2013, pp. 432–433.

[23] N. Sakimura *et al.*, "10.5 a 90nm 20mhz fully nonvolatile microcontroller for standby-power-critical applications," in *IEEE Int. Solid-State Circuits Conf. Digest Tech. Papers (ISSCC)*, 2014, pp. 184–185.

[24] W. Gallagher *et al.*, "22nm stt-mram for reflow and automotive uses with high yield, reliability, and magnetic immunity and with performance and shielding options," in *IEEE Int. Electron Devices Meeting (IEDM)*, 2019, pp. 2.7.1–2.7.4.

[25] Samsung electronics starts commercial shipment of emram product based on 28nm fd-soi process. Samsung.

[26] M. Hicks, "Clank: Architectural support for intermittent computation," in *2017 ACM/IEEE 44th Ann. Int. Symp. Computer Architecture (ISCA)*, Jun. 2017, pp. 228–240.

[27] D. Pala, I. Miro-Panades, and O. Sentieys, "Freezer: A Specialized NVM Backup Controller for Intermittently-Powered Systems," *IEEE Tran. Comput.-Aided Des. Integr. Circuits Syst. (TCAD)*, pp. 1–1, 2020.

[28] M. Xie, C. Pan, Y. Zhang, J. Hu, Y. Liu, and C. J. Xue, "A novel stt-ram-based hybrid cache for intermittently powered processors in iot devices," *IEEE Micro*, vol. 39, no. 1, pp. 24–32, 2019.

[29] S. T. Sliper, W. Wang, N. Nikoleris, A. S. Weddell, and G. V. Merrett, "Fused: Closed-loop Performance and Energy Simulation of Embedded Systems," in *Proc. 2020 IEEE Int. Symp. Perform. Anal. Syst. Software (ISPASS)*. Boston, MA, USA: IEEE, Apr. 2020.

[30] M. Surbatovich, L. Jia, and B. Lucia, "Automatically enforcing fresh and consistent inputs in intermittent systems," in *Proc. 42nd ACM SIGPLAN Int. Conf. Prog. Lang. Des. Impl.*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 851–866.

[31] P. Prabhat *et al.*, "27.2 M0N0: A Performance-Regulated 0.8-to-38MHz DVFS ARM Cortex-M33 SIMD MCU with 10nW Sleep Power," in *2020 IEEE Int. Solid- State Circuits Conf. - (ISSCC)*, Feb. 2020, pp. 422–424.

[32] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," in *Proc. 29th Ann. Int. Symp. Computer Architecture*, 2002.

[33] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, apr 2017.

[34] A. Savanth, A. Weddell, J. Myers, D. Flynn, and B. Al-Hashimi, "Photovoltaic cells for micro-scale wireless sensor nodes: Measurement and modeling to assist system design," in *Proc. 3rd Int. Workshop Energy Harv. & Energy Neutral Sens. Syst.*, ser. ENSsys '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 15–20.

[35] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu, "Enabling failure-resilient intermittent systems without runtime checkpointing," *IEEE Tran. Comput.-Aided Des. Integr. Circuits Syst. (TCAD)*, vol. 39, no. 12, pp. 4399–4412, 2020.

[36] ——, "Heterogeneity-aware multicore synchronization for intermittent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021.