This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Intermittent Computing Emulation of Ultra-Low-Power Processors: Evaluation of Backup Strategies for RISC-V

Sebin Shaji Philip, Roberto Passerone, *Member, IEEE*, Kasim Sinan Yildirim, *Member, IEEE,* and Davide Brunelli, *Senior Member, IEEE*

***Abstract*—With the progress in energy harvesting circuits and the decrease in power requirements of processing, sensing, and communication hardware, we have the potential of freeing the Internet of Things devices from their batteries. However, removing batteries introduces frequent power failures due to the irregular power availability from the environment. This situation leads devices to compute intermittently under transient environmental power. Intermittent computing requires significant microarchitectural modifications on existing processor designs to ensure automatic computation progress despite the power failures. For example, built-in nonvolatile memory components should be integrated in processor architectures. Consequently, different microarchitectural automatic backup strategies need to be implemented. In this work, we introduce different processor state backup strategies based on an interrupt-based software approach, which do not need modifications to the microarchitecture of existing processors. Therefore, we present a systematic approach to emulate different processor architectures with varying backup strategies under transient power. To justify our claims, we make Ibex RISC-V core, a popular ultra-low-power processor architecture, suitable for intermittent computing. This is the first attempt to make a variety of existing and future ultra-low-power processor architectures easily exploitable for transiently-powered computing systems.**

***Index Terms*—Batteryless IoT Devices, RISC-V, Transient processing**

I. Introduction

The Internet of Things (IoT) forms a network of devices that can sense the environment through their sensors, perform computation and communicate wirelessly to interact with each other. IoT applications (e.g., smart homes and cities, autonomous vehicles, wearables) support various tasks in our daily lives to increase our comfort and efficiency. Most IoT devices span tiny and battery-powered computing platforms [1]. The non-functional properties such as power consumption and sustainability emerge as a crucial challenge standing in front of the future IoT applications [2], [3]. Thanks to the progress in energy harvesting circuits and the decrease in power requirements of processing, sensing, and communication hardware, we have the potential of freeing the IoT devices from their batteries [4]. However, removing batteries introduces frequent power failures due to the irregular power availability from the environment, that are not experienced when the amount of energy storage is not an issue [5].

S. S. Philip, R. Passerone and K. S. Yildirim are with the Department of Information Engineering and Computer Science, University of Trento, Italy. e-mail:sebin.shajiphilip@studenti.unitn.it, {roberto.passerone, kasimsinan.yildirim}@unitn.it

D. Brunelli is with the Department of Industrial Engineering, University of Trento, Italy. e-mail: davide.brunelli@unitn.it.

Upon a power failure, the batteryless device starts operating again only with the harvested energy into its tiny energy reservoir (e.g., a capacitor). Therefore, the execution of the software is interleaved with the energy harvesting intervals during which the batteryless device is off [6].

Today's batteryless platforms are composed of ultra-low-power microcontrollers, e.g., MSP430FR5969 [7], whose main architectural components, such as registers and main memory, are *volatile*. These microcontrollers comprise non-volatile secondary memory components, e.g., Ferroelectric RAM (FRAM) [8], to store information that will persist upon power failures. To mitigate the effects of unpredictable power failures and enable *progress of computation* while preserving *memory consistency*, several *software-aided* solutions have been proposed [9], [10]. Generally speaking, these solutions *back up* the volatile state of the processor into the nonvolatile memory so that the computation can be recovered from where it left upon reboot. Moreover, they ensure *memory consistency*, so that the backed-up state in nonvolatile memory will not be different from the volatile state or vice versa.

Batteryless platforms can also include nonvolatile processors (NVPs), which integrate built-in nonvolatile memory components in their microarchitecture. Thanks to these built-in nonvolatile logic components, NVPs automatically back up the processor state to internal memory elements when a power failure occurs, and restore the state when power becomes available [11]. This type of execution makes backup operations transparent to the programmer. Since backup and retention operations are fast compared to the software-aided solutions (e.g., on the order of a few microseconds [12], [13]), NVPs reduce leakage power by allowing the system to shut down when idle [14].

There is still a vast design space to explore new processors architectures and backup strategies targeting intermittent computing platforms. For example, the Ibex RISC-V processor core [15] is freely available and a good candidate for modern ultra-low-power IoT computing applications, as demonstrated in previous studies that compare and justify the suitability of "Zero-riscy" (former name of the Ibex core) for low power IoT applications [16]. In particular, one of the advantages of RISC-V is its layered and modular ISA architecture [17], which gives the flexibility to implement minimal instruction sets which are customized to the specific application. This is especially

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

beneficial in the context of heterogeneous architectures where cores of different complexity can be selectively activated to adapt for optimal power usage. At the same time, this processor has not been used by intermittent applications previously since it does not include an integrated nonvolatile memory, and therefore, it loses its state upon power failures. Researchers do not have the necessary emulation environment to introduce nonvolatile memory and processor backup strategies into this processor design. Furthermore, it is difficult to test, verify and benchmark different backup strategies under intermittent power supply. Prior art used either numerical simulations or ASIC implementations to evaluate different backup strategies and NVPs. However, numerical simulations are not sufficient to test the final system as a part of a real design with several other hardware components. On the other hand, ASIC implementations are expensive and are not available at the earlier stages of system design.

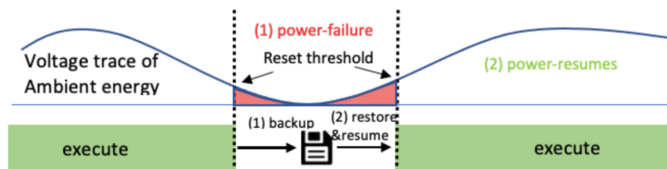

Fig. 1. The system behavior during power failure (context backup to NV memory) and power restore (context restore from NV memory and resume execution)

In this work, we introduce different processor state backup strategies based on an interrupt-based software approach, which do not need modifications on the microarchitecture of processors. Therefore, these strategies can be easily extended to a wide variety of different processors. To implement and evaluate our different backup policies, we used the non-volatile simulation framework developed especially for intermittent computing applications on off-the-shelf FPGAs [18]. We explore the effect of our backup strategies on the execution performance of intermittent programs on the Ibex RISC-V processor. More specifically, the main contributions of this article are as follows:

1) **Systematic Emulation.** We present a systematic approach consisting of a transient processing architecture to emulate processors with different backup strategies under transient power. This is achieved by interfacing a generic non-volatile memory to the processor bus, and by integrating an Intermittency Emulator module which provides power failure information on the basis of a defined energy profile.
2) **Development guideline to set optimal intermittent policy.** Based on the detailed analysis done on the results obtained from various backup policies, we formulate guidelines for selecting useful parameters, possibly changing and adapting over time (for a given trace), which could help answer the question *when to backup?*. These guidelines could be helpful in designing strategies and maximizing the efficiency for a wide range of applications in the future.
3) **Intermittent RISC-V and ultra-low-power Cores.** We introduce the fundamental building blocks to exploit and/or port ultra-low-power processor architectures as an intermittent computing platform. The combination of hardware (interrupt generator, intermittency emulator) and software (interrupt handlers, save and restore strategy algorithms) layers described in this paper demonstrates how to make the Ibex RISC-V core suitable for intermittent computing.

Figure 1 gives a high-level overview of the program execution flow during the backup/restore operations in the event of power-failure. The blue plot shows the available ambient energy as a function of time. Upon crossing a defined threshold, which depends on the particular policy, the processor saves its state in the non-volatile memory before losing power. Once sufficient energy is available again, the processor recovers the state, and resumes its normal operation.

The remainder of this article is organized as follows. Section II provides a brief overview of the current research and related work relevant to NVP, and backup/restore approaches. We discuss the components that enable the transient processing architecture and give brief implementation details in Section III. The results obtained for three different backup strategies are discussed, and their performance is compared in Section IV. Then, in Section V, we analyze the effect that the parameters that characterize the strategies have on performance according to different time profiles of the available energy and discuss possible approaches to be taken for each of the identified cases, including a mixed strategy when the parameters change over time. Finally, we present our conclusions and discuss future work in Section VI.

II. Related Work

The evolution of transient processing from CMOS processors to NVP, mainly improving speed and energy efficiency for backup/restore operations, is rooted in the advancements in memory technologies (from Flash, FRAM, MRAM, RRAM, TFET and NCFET) [19]. While in earlier designs the non-volatile (NV) element was a central off-chip memory, requiring sequential byte level transfer, emerging NVPs have embedded NV flip flops (NVFF) and gates that could retain their states at register level. Because of the large area overhead induced by NVFF [20], selective backup policies to central NV-memory is still employed, making use of advanced NV memory technologies [10], [21], [22], [23]. Moreover, backup optimization algorithms have been proposed to switch between write-back and write-through strategies to reduce the rollback induced by backup failures, achieving a remarkable reduction of inrush current [24].

The main design questions to be addressed while developing a central NV-memory/checkpoint-based failure resilient system is *what* to backup and *when* to backup, with regard to NV-system correctness [25]. Implementations of an *interrupt-driven* checkpointing technique using FRAM-enabled TI microcontrollers was proposed for transiently powered computers (TPCs) [23], [9]. Instead of tracking active registers in use, they back up the entire general purpose register file during checkpointing, even under Dynamic Frequency Scaling

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

(DFS) [26]. Compile time checkpoint triggering implementations, e.g. [27], [28], [29], [30], [31], [32] introduce minimum overhead for the programmer. For example, in [28], the triggering points are placed after each loop unroll and each function return compares the input voltage (V_{in}) against a *threshold* and performs a checkpoint if V_{in} is below the *threshold*. The compiler and runtime system implementation achieves both minimum overhead to the programmer and adaptive saving for backup strategy. Another strategy has been implemented in [27] that triggers checkpoints periodically using timers. In [29], the live-range analysis technique has been utilized by a specialized compiler that adaptively backs up only required volatile data which is performed in a dynamic checkpoint system. Inserting checkpoints during behavioral synthesis on ASIC implementations [33] has also been proposed. Moreover, checkpointing is usually implemented with high overhead in applications with loops, because a large amount of data needs backup during loop execution. In [34], authors reduce the amount of checkpointing data by analyzing data locality and shortening data lifetime in loops.

There have been previous studies conducted to evaluate the performance of forward progress achieved by employing different backup policies for several processor architectures [11]. The proposed policies, like backup every cycle (BEC), on-demand all backup (ODAB), and on-demand selective backup (ODSB), are for non-pipelined processors. An in-order execution with an N-stage pipeline architecture was studied using 'Shifted PC/Volatile FlipFlop (SPC/VFF)' as well as a 'Non-Volatile FlipFlop (NVFF)' [35], [11]. While a shifter buffer placed between each pipeline stage is responsible for probing the relevant program counter (PC) to save in the event of a power failure (SPC/VFF), NVFF automatically saves both the PC and the register file in each stage causing more time and energy overhead. Policies involving backing-up of more complex hardware units like the reorder buffer (ROB), instruction queue (IQ), map table, branch history buffer (BHT), branch target buffer (BTB), are proposed for an out-of-order (OoO) execution setup [35].

To cater for the basic use case of an energy harvester based low-power sensing application, we selected the integer-based RISC-V processor core (Ibex) with 2-stage in-order processing. It was found that the ODSB is the most energy-efficient for a comparatively stable energy source and the SPC/VFF performed well for pipelined in-order processors [11]. We have selected a similar approach by selectively saving the PC and minimal register file into the NV memory for an informed event of a power failure. These metadata are taken non-intrusively from the RISC-V core using an interrupt-based software approach, resulting in no change in the microarchitecture and the possibility to extend the same approach to a wide variety of processors.

To implement and evaluate our different backup policies and their performance, we have made use of the non-volatile simulation framework developed especially for intermittent computing applications on off-the-shelf FPGAs [18], [36] by integrating the RISC-V core. The framework consists of an Intermittency Emulator Module (IEMU) which gives informed predictions of a possible power failure, and a NV-memory module that helps emulate the behavior of a fast built-in NV-memory.

The choice of the architecture and policy design is highly influenced by the input power and its stability [11]. We have conducted a detailed study for a highly fluctuating real input trace [37], and formulated hypotheses for selecting useful parameters (for a given trace) which could help answer the question *when to backup?*. These observations could be helpful in designing strategies and maximizing the efficiency for a wide range of applications in the future.

III. Transient NV processing architecture

In this section, we present the transient processing architecture which includes the *Intermittent RISC-V core* setup, enabling the *Systematic Emulation* of different backup policies, and helping low-power sensing applications (running on energy harvesters) manage unstable power conditions and reliably *resume* their operations after a power shortage. The general system architecture of the transient processing, shown in Figure 2, can be divided into three main parts: (i) the transient processing core, (ii) the memory setup, and (iii) the intermittent setup. The processing core supports transient

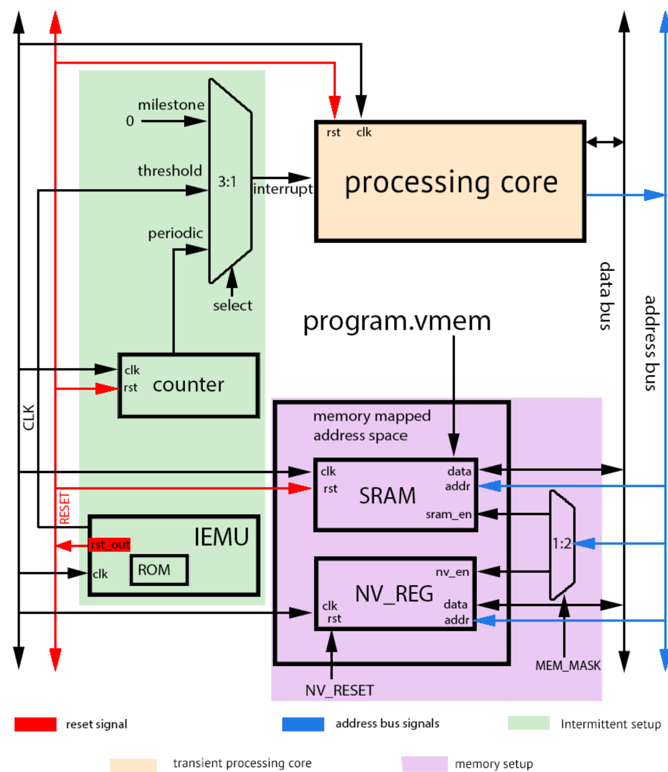

Fig. 2. The general architecture of transients RISC-V processing

operation by saving critical state (program counter, status registers, register file, etc.) to the non-volatile memory NV_REG. The processor that we have used is the Ibex (RV32IMC) RISC-V processor core [15]. The intermittent setup is used to emulate the power availability of real working hardware. This unit, which is designed as a separate module, is implemented together with the core and integrated directly through the interrupt line. Finally, the memory setup consists of the NV_REG

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

as well as the SRAM (static RAM for program and data), helping in program execution and backup/restore operations. The memories interface directly to the core through the address and data buses, and are activated by an appropriate address decoder. The entire system was integrated and implemented in mixed SystemVerilog/VHDL languages in Vivado and the results were simulated for verification. In the rest of this section, we provide the discussion of these parts in detail.

A. Intermittent setup

The intermittent behavior of the system is evaluated through three different strategies for a specified input voltage trace. The voltage trace is common to all strategies and is stored in a ROM (Read-Only memory) inside the IEMU. The IEMU periodically reads the values from the trace memory, and delivers status information to the rest of the circuit, to emulate the way the actual hardware behaves. There is a total of 1569 values in the ROM. Each value is retained for 2500 clock cycles, or 50 ms, bringing the total simulation time to 75 s. The IEMU works by comparing the trace voltage level to some thresholds. When the voltage falls below the *hard* threshold (or *reset* threshold), the IEMU activates a RESET signal, which is connected to all hardware components except the non-volatile memory NV_REG. The reset signal causes the components to lose their state, effectively emulating a power down cycle. The hard threshold of the IEMU is set to 2800 mV in all the cases, as transistor logic will not work below this voltage level. Other thresholds can be used with different strategies, as discussed below.

1) Threshold-based strategy: In the threshold-based strategy, a second threshold value is selected in the IEMU as a *soft* threshold, resting above the reset threshold. At any given time, the IEMU will give information about the status of the comparison between the current voltage level (from ROM) and the soft threshold. An interrupt is generated if the current voltage level falls below the soft threshold. By servicing the interrupt, the core can initiate a save operation to back up its internal state to non-volatile memory before the system loses power completely.

2) Periodic strategy: Unlike the threshold-based strategy, the periodic strategy does not rely on the input voltage trace to generate an interrupt to the core. Instead, an external dedicated counter is employed to generate the interrupt to the core periodically. The counter period can be tuned according to the voltage trace to yield maximum performance from this strategy.

3) Milestone-based strategy: The Milestone-based strategy does not rely on external interrupts to the core for a non-volatile save operation. Rather, the interrupt service routine is transformed into a regular function which can be called from within `main()` each time a number N_f of typical read-compute-transmit iterations of the code have been executed. The most frequent rate corresponds to saving the state at every code iteration (milestone = 1).

B. Memory setup

The processing core is connected to a memory mapped system as shown in Figure 2. The initial 64 kB of the address space is reserved for the SRAM which is the primary memory area into which program and data are stored when the processor is running. The peripherals, such as the sensors and the UART, are connected to the core through ports which are memory mapped, so that reading and writing from the application code is performed by a direct access to the address space assigned to their configuration and status registers.

The next 16 memory locations after the 64 kB are reserved for non-volatile (NV) memory, or NV_REG. For this, we have made use of a non-volatile simulation framework developed especially for intermittent computing applications on off-the-shelf FPGAs [18]. This FPGA-based framework, named NORM (Non-volatile memORy eMulator), is developed to emulate and verify the behavior of any intermittent computing system that exploits fast non-volatile memories.

The NV_REG is directly connected to the processor address and data bus, just like the SRAM and other peripherals. The address decoder (represented as “1:2” in Figure 2) determines when to activate the non-volatile memory. The address decoder switches between SRAM and NV_REG based on the address requested by the core (memory mask based address decoding). All the memory units connected to the core respect the Ibex Load-Store unit protocol specification [38] for memory transactions. The initial 12 words of the 16-bit wide NV memory locations we currently use are represented in Table I.

TABLE I
THE NV MEMORY LAYOUT

dirty bit	PC	general purpose regs.								Control and status registers	
(0/1)	mepc	a0	a1	a2	a3	a4	a5	a6	a7	mie	mstatus

The first word stores the *dirty* bit, which indicates whether the NV memory contains saved data or not. This is useful in determining whether a resume operation needs to be performed. The following word, called PC, contains the address of the instruction from where the code is to be resumed after a successful restore. The next consecutive eight locations store the state of the general purpose register (a0-a7), followed by the `CSR` register states required for interrupt services. The amount of non-volatile memory dedicated to storing general purpose register values can easily be expanded, according to the requirements of the application (i.e., how many registers are required to correctly resume the program execution). Here we limit the exposition to eight registers for simplicity, as the example periodic sensor reading application that we use for evaluation does not need to save more data. As discussed in the previous sections, saving to NV memory is performed by the interrupt service routine or by the main program, while the restore operation is carried out by the start-up code.

The NV_REG takes 120 μs to read/write one memory location. Instead, the SRAM has a 100 μs delay between data read/write request from the core until successful completion acknowledgment. We can emulate the delay of NV_REG to take a larger value depending upon the specification, in which case the results obtained will also vary accordingly.

C. Transient RISC-V (Ibex) core

Ibex is one of the popular open source 32-bit RISC-V CPU core (2-stage pipeline), suitable for low power embedded

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

control applications [15]. The parameters of the version of the Ibex core used in this paper are summarized in Table II. The core is operated at 50 kHz which is similar to a real-world low-power processing frequency of a sensor. The features exploited for saving processor states are the fast interrupt, the general purpose register file and certain CSRs (to enable interrupts and to read PC value when the core is interrupted).

TABLE II
MAIN IBEX RISC-V CORE FEATURES

Feature	Ibex specific feature
Pipeline	2-stage in-order pipeline (IF - Instruction fetch, ID/EX - Instruction Decode and Execute)
Enabled features	RV32IMC (M - Integer multiplication and division, I - Base Integer Instruction Set version 2.1, C - Compressed Instructions)
Disabled features	Instruction Cache, PMP (Physical memory protection), RV32B (Bit manipulation), Branch-target*, Branch-predict (static) and Write-back stage (third pipeline, experimental)
Memory	Single-port SRAM with 1 cycle read/write delay, 32 bit words. The Ibex core is directly connected to this memory through the address and data bus, where all the data and instructions are stored
No. of registers	32 general purpose registers (32-bit)
Register file	FPGA optimized register file (using RAM32M primitives)

Many features in the RISC-V specification are optional, and Ibex can be parameterized [15] to enable or disable some of them. *Branch-target: when enabled a separate simultaneous ALU instance is used for branch target calculation, to reduce stall from 2cc to 1cc (Branch taken).

The general program startup flow and the associated interrupt/NV save logic is illustrated in Figure 3. The main fragments of code corresponding to the various phases are also shown for completeness in Listing 1, 2 and 3. The core runs a sample C application that performs basic operations, such as reading sensor values from a memory mapped location, calculating its exponentially weighted moving average, and writing the output to a memory mapped UART for transmitting over the radio. This program, denoted as `program.vmem` in Figure 2, is loaded in memory at the processor startup. The sampling rate of the application is set to 2 Hz, which implies each value is read from the sensor, computation is performed and results transmitted every 0.5 seconds continuously, and includes a delay before each iteration.

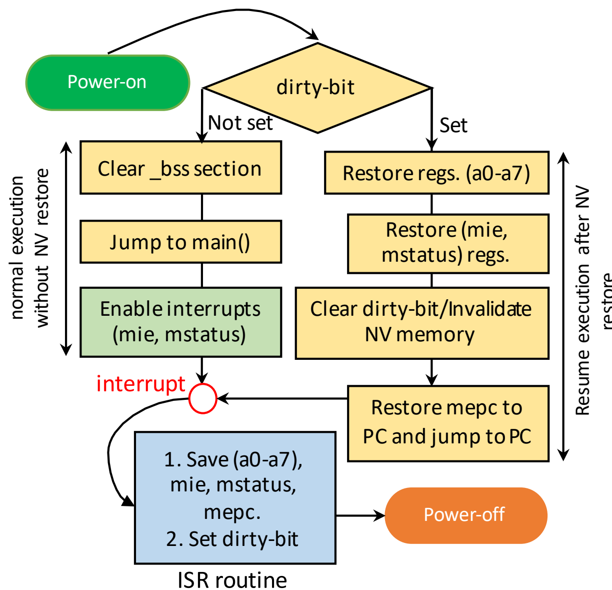

Fig. 3. The general program startup flow (with and without NV restore) and interrupt/NV save logic. **mie**: Machine Interrupt Enable; **mstatus**: Machine Status (containing global interrupt enable); **PC**: Program Counter; **(a0-a7)**: part of General purpose regs.; **mepc**: Machine Exception Program Counter (containing 'PC value before interrupt' or instruction address from which the program is to be resumed); **_bss**: Block starting symbol (containing unassigned statically allocated variables in program); **dirty-bit**: bit indicating whether to restore or not.

```
int main(int argc, char **argv) {
  /* mie fast interrupt enable */
  asm volatile("csrw 0x304, %0\n" : : "r"(0x00010000));
  /* enable global interrupt, mstatus register */
  asm volatile("csrw 0x300, %0\n" : : "r"(0x8));
  while (1) {
    /* logic to calculate Exponentially weighted moving
          average (ema) after reading from sensor and
          writing results back to radio (memory-mapped
          location) */
    current_sensor_reading = *sensor_mem_address;
    ema = (SMOOTHING_FACTOR * current_sensor_reading) +
          ((1 - SMOOTHING_FACTOR) * ema);
    ema = ema >> 2;
    *radio_transmit_mem_address = ema;
    /* sleep for 0.5s (code omitted for brevity) */
  }
}
```

Listing 1. The application program startup code and data acquisition loop in main.c. The SMOOTHING_FACTOR determines the decay rate of the exponentially weighted moving average.

After entering `main()` (shown in Listing 1, and as a light-green shaded box in Figure 3), the code writes the value '0x00010000' to the control register `mie` (0x304, Machine Interrupt Enable register) to enable the first fast interrupt signal (out of available 15 signals) and sets the global interrupt enable by writing 0x08 to `mstatus` (0x300, Machine Status register). This will enable the Ibex core to receive and service the fast interrupts given for the periodic and threshold based strategies. After compiling the program, we observe that 8 general purpose registers (a0-a7 (x10 - x17)) are used in the disassembled code for reading/computation and writing the results.

The application therefore also registers a dedicated interrupt service routine (shown in Listing 2 and as a light-blue shaded box in Figure 3), marked as `__attribute__((interrupt))`, whose main function is to write the dirty bit (the bit indicating the NV region has valid data), the current PC ('Program counter before interrupt' whose value is taken from the `mepc` register of the RISC-V processor in case of an interrupt, otherwise the function return address is stored in case of a milestone-based strategy), the content of a0-a7 and the state of the `mie` and `mstatus` control registers onto the NV region in that order, according to the memory layout outlined above.

Along with the application and the interrupt service routine, the program includes a startup code (shown in Listing 3, and as a set of light-orange shaded boxes in Figure 3) executed every time the core is powered up (either at the beginning, or when execution has to be resumed). The runtime startup code, `crt0.S`, is used to load the compiled C program into memory, since we are not using any OS or standard C-library

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems


```
/* The interrupt handler for NV save */
void fast_irq_handler_nv_write(void)
     __attribute__((interrupt));
void fast_irq_handler_nv_write(void) {
/* save registers (a0-a7) which are used in disassembled
     code (only a0 shown here) */
  __asm__ volatile("li t3, 0x10008");
  __asm__ volatile("sw a0, 0(t3)");

  /* read mstatus reg. and save the global interrupt
       enable bit */
  volatile uint32_t *nv_location_pointer = (uint32_t *)
       0x00010028;
  __asm__ volatile("csrr %0, 0x300;" : "=r"( temp ) );
  *(nv_location_pointer) = (temp & 0x80) >> 4;
  /* read and save mie reg. for fast interrupt enable */
  __asm__ volatile("csrr %0, 0x304;" : "=r"(
       *(nv_location_pointer+1) ) );
  /* read and save PC before interrut from mepc reg.*/
  nv_location_pointer = (uint32_t *) 0x00010004;
  __asm__ volatile("csrr %0, mepc;" :
       "=r"(*(nv_location_pointer) ));
  /* set dirty bit flag to indicate valid NV state */
  nv_location_pointer = (uint32_t *) 0x00010000;
  *nv_location_pointer = (uint32_t) 1;
}
```

Listing 2. Interrupt handler code in main.c

```
main_entry:
  /* jump to main program entry point (argc = argv = 0) */
  addi x10, x0, 0
  addi x11, x0, 0
  /* If dirty bit is set jump to dummy_main */
  li x1, 0x00010000
  lw x1, 0(x1)
  bnez x1, dummy_main
  /* Normal program startup without restore, so clear BSS
       */
  la x26, _bss_start
  la x27, _bss_end
  bge x26, x27, zero_loop_end
zero_loop:
  sw x0, 0(x26)
  addi x26, x26, 4
  ble x26, x27, zero_loop
zero_loop_end:
  jal x1, main
  jal x0, both
dummy_main:
  /* restore registers x10-x17 (a0-a7)*/
  li x1, 0x00010008
  lw x28, 0(x1)
  mv a0 ,x28
  /* restore mstatus register (for global interrupt) */
  lw x28, 32(x1)
  csrw 0x300, x28
  /* restore mie register (for fast interrupt) */
  lw x28, 36(x1)
  csrw 0x304, x28
  /* clear dirty bit/ Invalidate NV reg */
  sw x0, -8(x1)
  /* restore mepc and resume execution */
  lw x28, -4(x1)
  jalr x1, 0(x28)
```

Listing 3. The C program startup code in crt0.S (minimal version)

environment (bare-metal C program in this case). The script defines and initializes the interrupt vector tables, defines the reset handler which clears all the registers, clears the BSS section, initializes the stack section, and so on.

One of the main function of the script includes the logic for restoring the processor state from the NV memory and continuing the execution from the last save. This is done by reading the initial location of the NV memory address after a reset, to check if the dirty bit is set. The execution is continued from the previously saved PC value if the bit is set, after all the saved values from the NV memory are restored into a0-a7 and the control and status registers. After reading it successfully, the routine also resets the dirty bit to indicate that the NV region does not contain any valid data (the bit will be set again after saving a new state). If the dirty bit is not set, then the program proceeds to execute from the start of `main()`, after clearing the BSS section, ensuring normal program flow after reset.

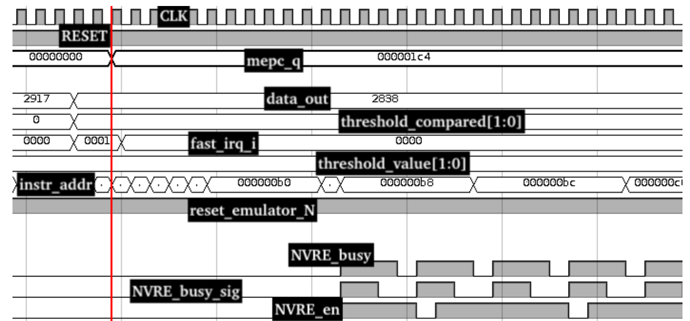

Fig. 4. NV save for threshold-based strategy (reset=2800mv and soft-threshold=2900mV). The signal names are as follows. **CLK**: Clock; **RESET**: Reset; **mepc_q**: mepc reg. value; **data_out**: power trace value from ROM; **threshold_compared[1:0]**: [hard-threshold-crossed, soft-threshold-crossed] the bit position *(*-threshold-crossed)* in this array is set if the corresponding threshold is crossed by the input trace; **fast_irq_i**: threshold based interrupt signal; **threshold_value[1:0]**: [hard-threshold, soft-threshold]; **instr_addr**: currently executing instruction address; **reset_emulator_N**: to simulate RESET/Power-down signal when input voltage trace crosses hard-threshold; **NVRE_busy**: NV reg. is busy; **NVRE_busy_sig**: same as **NVRE_busy** but drops one cycle earlier; **NVRE_en**: NV reg. enabled.

The output of the simulation for NV save is shown in Figure 4. It shows the threshold-based interrupt generated in `fast_irq_i` signal when the data from ROM (`data_out`) changes from 2917 to 2838. This causes `mepc_q` to capture the instruction address `0x000001c4` (PC value when the core got interrupted) followed by the NV-write represented by the `NVRE_en`, `NVRE_busy` signals. The core continues its normal execution after the NV write. Similarly in Figure 5, when the core boots up after a power failure (`reset_emulator_N`), the state stored in the NV memory (`NVRE_*`) is restored and the core continues its execution from `0x000001c4` in `instr_addr`.

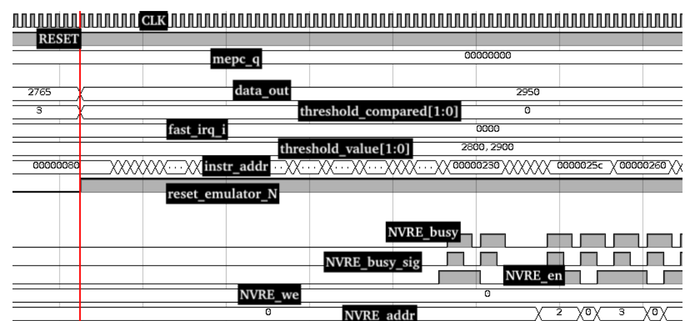

Fig. 5. NV restore for threshold-based strategy (reset=2800mv and soft-threshold=2900mV). See the caption of Figure 4 for an explanation of the signal names. For the extra signals: **NVRE_we**: NV reg. write enable; **NVRE_addr**: NV reg. address.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

IV. Evaluation

We have set up and collected data from all the three strategies and analyze and present the results in this section. Figure 6 shows the input voltage trace we used to obtain the results. This voltage trace is taken from [37], and it represents the harvested RF energy collected by walking around within about eight feet of an RFID reader. Out of all other available traces [39], we selected this trace in particular because it covers all the possible combinations of SAVE-RESTORE processor states of interest and therefore gives us a comprehensive result.

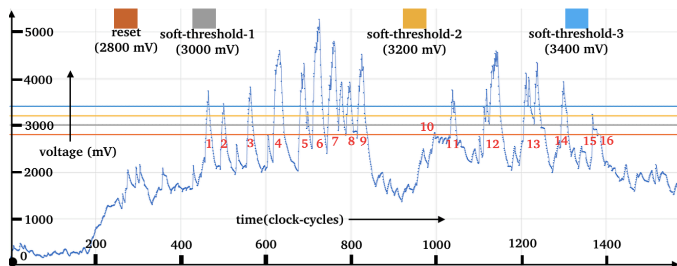

Fig. 6. The input voltage profile and threshold limits

We have chosen the hard-threshold limit (RESET voltage) as 2800 mV, below which a RESET signal is generated from the IEMU, and the Ibex core is reset. As we can see from the voltage trace, there are a total of 16 power-ON intervals {at 650, 550, 700, 1300, 1250, 1450, 2100, 1300, 1150, 100, 700, 2250, 2550, 900, 550, 50 ms respectively} during which the voltage crosses the hard threshold and the system powers up.

In the rest of this section, we analyze the behavior of the system upon each of the 16 power-on intervals. We color code the outcome of the save and restore operations according to the scheme outlined in Table III, going from dark grey for a double fail (both the save and the restore operation do not succeed) to white for a double success. This gives an intuitive and clear visualization of the strategy performance as a function of its operating parameters. Furthermore, we discuss the impact of the strategy in terms of CPU overhead.

TABLE III
Restore-Save status color codes

Color	Status
	Restore-Fail, Save-Fail (RF_SF)
	Restore-Fail, Save-Success (RF_SS)
	Restore-Success, Save-Fail (RS_SF)
	Restore-Success, Save-Success (RS_SS)

A. Threshold-based strategy

Three soft-threshold values above the hard-threshold (orange line), as shown in Figure 6, are selected to conduct the threshold-based NV strategy (3000, 3200, 3400 mv, respectively). Table IV gives the result of the threshold-based strategy in terms of correctness for various selected soft thresholds.

Since during the 1st interval there is no pre-existing saved value in the NV memory to be restored, we conventionally assume the outcome to be RF_SS. In all cases except for the 2900 mV threshold, the values computed are successfully saved and restored consecutively during the next boot cycles until interval 10. Intervals 10 and 16 (with power-on duration 100 and 50 ms, respectively) have a shorter power-ON interval compared to the others, and thus do not provide enough time to save the result to NV memory after receiving an interrupt. Also, in the case of 2900 (intervals 11 and 12) and 3000 (interval 11), the NV save fails because of the same reason. Observe in particular that even though the 3400 mV threshold is higher than 3200, it fails to save in interval 15, because the voltage trace does not reach the level 3400 before falling back below the reset level, thus not issuing the save operation (see Figure 6). This means that simply selecting a higher threshold does not necessarily guarantee better performance, but rather it highly depends on the actual voltage trace under consideration.

TABLE IV
Correctness of threshold based strategy

Threshold	Power-ON intervals															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2900																
3000																
3200																
3400																

From Table IV it is clear that the soft threshold 3200 mV yields the highest performance with 87.5% in terms of correctness (the percentage is calculated by taking all the Save-Success intervals out of the total 16 intervals), although 3400 mV (performance = 81.25%) is a higher threshold than 3200 (all Save-Success cases are considered to be Success). The 3000 mV threshold has similar results as 3400 (81.25%). Notice that lowering the soft threshold near the hard threshold (performance of 2900 = 56.25%) gives sub-optimal results compared to the others.

Table V records the actual CPU overhead (calculated from the number of clock cycles utilized) caused by the NV save/restore operations for different soft thresholds. The 3200 mV threshold has a slightly higher overhead (last column) compared to the other thresholds, since the increased number of RS_SS regions contribute additional cycles for NV operations (see Table IV). This clearly indicates the direct correlation between higher performance (in terms of correctness) and CPU overhead.

TABLE V
CPU overhead for threshold-based strategy

A (mV)	B (ms)	C (ms)	D (%)	E (ms)	F (%)
3000	64.4	17550	0.367	1212	5.3135
3200	73.1	17550	0.416	1212	6.0313
3400	69.08	17550	0.394	1212	5.6996

A - Soft threshold, B - Total SAVE /RESTORE duration, C - Total power-ON duration (above 2800 mV), D - Percentage, E - Total powered-ON duration except delay (nop), F - Percentage (excluding delay).

B. Periodic strategy

The performance of the periodic strategy depends on the relation between the period with which data is saved to NV memory and the time interval between the power down events, which we call the *power-ON* interval.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Figure 7 shows for each of the 16 power-ON intervals (horizontal axis) their corresponding duration in seconds (vertical axis). The average duration of the power-ON interval is equal to 1096.87 ms, while the shortest interval is the 16th, with a time duration equal to 50 ms. In terms of clock cycles, the average corresponds to 54844 clock cycles, while 50 ms are equal to 2500 clock cycles. To evaluate the strategy, we consider periods between the shortest and the average interval. The results are shown in Table VI. In this case, as expected, the lower the period, the higher the correctness rate. In particular, the average period achieves only 50% correctness, while choosing the shortest period is almost guaranteed to provide the correct data (100% correctness). Clearly, the problem with choosing the average period is that all the power-ON intervals whose duration is higher than the average (interval 1, 2, 3, 10, 11, 14, 15 and 16) fail to save the data in time to the NV memory since the processor does not receive an interrupt from the counter before reset. Similarly, for 20000 cc (400 ms), the intervals 10 and 16 have periods shorter than 400 ms and thus fail to save their state before reset. Therefore, it is interesting to note that combining information about the power-ON intervals and tuning the periodic interrupt generator accordingly, we can more or less achieve a predictable performance in terms of correctness.

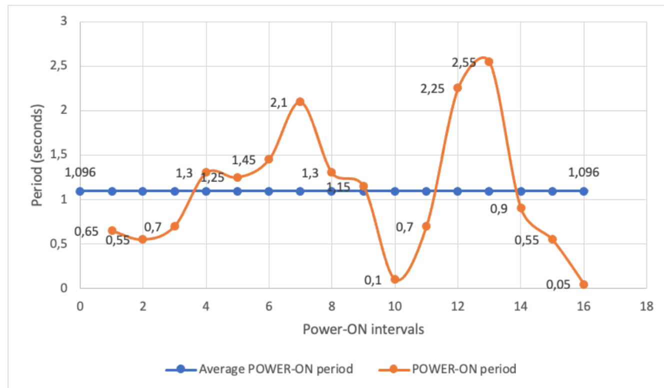

Fig. 7. Power-ON intervals of input voltage profile (x-axis = interval index, y-axis = power-ON interval in seconds)

TABLE VI
CORRECTNESS OF PERIODIC STRATEGY

Clock cycles	Power-ON intervals															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
54844																
35000																
20000																
3000																
2500																

In terms of overhead, we measured a drastic increase in the overall CPU usage from Table VII (column F) when the period of the counter is decreased. The average power-ON interval (54844 cc) gives us a negligibly low 3.4% overhead compared to the worst-case 67% of the shortest power-ON interval (2500 cc). The trade-off between performance (correctness) vs. overhead is clearly visible here.

TABLE VII
CPU OVERHEAD FOR PERIODIC STRATEGY

A (cc)	B (ms)	C (ms)	D (%)	E (ms)	F (%)
2500	812.2	17550	4.628	1212	67.013
3000	807.5	17550	4.60	1212	66.625
20000	115.22	17550	0.656	1212	9.51
54844	41.96	17550	0.239	1212	3.462

A - Period for counter generated interrupts (Clk cycles), B - Total SAVE /RESTORE duration, C - Total power-ON duration (above 2800 mV), D - Percentage, E - Total powered-ON duration except delay (nop), F - Percentage (excluding delay).

C. *Milestone-based strategy*

With a milestone-based strategy, data is saved only after it has been computed. Besides saving after every computing iteration, we also experiment with saving the data after 2 or 3 iterations. The results are shown in Table VIII.

TABLE VIII
CORRECTNESS OF MILESTONE BASED STRATEGY

No. of iterations before a milestone	Power-ON intervals															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2	2	1	2	2	2	2	4	2	2	0	2	4	5	2	1	1
3	2	2	2	3	2	3	4	3	2	1	2	5	6	1	2	1

The first column indicates the number of times the code has been executed before saving its state to the NV region. Also, note that the numbers in each interval cell for milestone 2 and 3 indicate the number of code iterations that the interval successfully executed. In the case of milestone 2, the dirty bit of the NV memory is set only in even iterations. It is cleared in all other cases. Similarly, for milestone 3, when the iteration reaches 3 or a multiple of 3 we set the dirty bit to indicate a valid state in NV memory (i.e., the data we saved after this into the NV memory will be the latest/correct state). Otherwise, the dirty bit is invalidated.

Since the program counter value saved to the NV region will always be the address of the milestone function call (after output computation and just before the 0.5 s delay loop), after a successful restore, the delay loop will always be executed first. This is the reason why in intervals 10 (milestone 1 and 2) and 16 (milestone 1), we do not have enough time to exit out of the 0.5 s delay, to calculate/transmit a new value and then save that into NV. This is also the reason that milestone 1 has a reduced performance of 87.5% , which would have been close to 100% otherwise. In the case of milestone 2 and 3, the save is failing just because the number of iterations is not a multiple of the milestone in that interval as shown. Milestone 2 has a performance of 68.75%, and milestone 3 has the worst performance of 25%.

Table IX shows the overhead for this strategy. The additional overhead of clearing the dirty bit after every iteration (if the iteration is not a multiple of the milestone) in case of higher milestones is also accounted for. Similar to the previous strategies, the total CPU utilization increases with lower milestones,

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE IX
CPU OVERHEAD FOR MILESTONE BASED STRATEGY

A	B (ms)	C (ms)	D (%)	E (ms)	F (%)
1	98.4	17550	0.561	1212	8.085
2	64.02	17550	0.364	1212	5.28
3	37.74	17550	0.215	1212	3.11

A - Milestones per calculation, B - Total SAVE /RESTORE duration, C - Total power-ON duration (above 2800 mV), D - Percentage, E - Total powered-ON duration except delay (nop), F - Percentage (excluding delay).

and the very frequent save operations contribute much towards the increase.

V. Discussion

Given the results outlined in the previous section, we proceed now to analyze the impact that the different parameters have on the performance of each individual strategy. There are three main performance indicators to consider when selecting an NV save strategy: (i) the desired level of correctness when restoring the data, (ii) the average power-ON interval duration of the voltage trace, and (iii) the CPU overhead introduced due to the extra NV save/restore operations which can be tolerated.

The best strategy, and the choice of parameters, clearly depends on the shape of the voltage trace over time, and on the processor performance. In the rest of the section, we consider different shapes which we represent in the form of a "terrain", as they remind of a hilly landscape. Figure 8 shows an example.

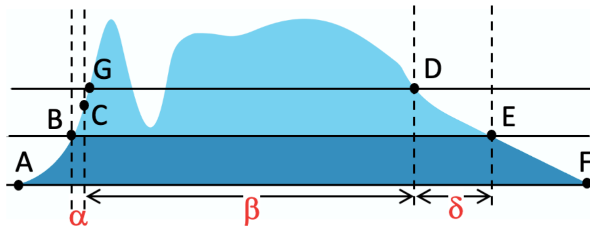

Fig. 8. Single terrain of input voltage trace. (x-axis = time, y-axis = voltage)

The dark blue area lies below the reset threshold, while the light blue area is above. The line GD represents the soft threshold value defined, which would be used as a reference voltage for carrying out NV operations. We identify several parameters that characterize both the trace and the processor execution:

- α - the time required from a power on event to the start of execution of the `main()` function;
- β – available time to execute the `main()` code before crossing the 'closing' threshold (From Figure 8, point D is the 'closing' threshold, because after D the trace continues to stay below the soft threshold and eventually goes below the reset threshold);
- δ - available time left between the soft threshold (NV write) and reset;
- $T_{STARTUP}$ = minimum time required to startup the `main()` code after reset (in our case this would be 201 cc and the corresponding power required would be the area under the curve BC);
- T_{SINGLE_EXEC} = minimum time required to execute a single output after reaching the `main()` code, this would be 72 cc;
- T_{NV_SAVE} = minimum time required for NV write after an interrupt is received, this would be 117 cc;

In our implementation, $\alpha = T_{STARTUP}$. Below, we consider each performance indicator.

A. Correctness

To achieve close to 100% correctness while restoring a saved state, we should always ensure that the last operation to be carried out should be an NV save before power down. We consider two cases:

1) case I: $\delta \geq T_{NV_SAVE}$: Here we assume that the shape of the voltage trace always provides enough time (δ) for the processor to perform a save operation when the trace falls below the first (soft) threshold and before it falls below the reset threshold. The best strategy depends on the time duration that the trace spends between the two thresholds.

- case I.a: $\exists\gamma, 0 \leq \gamma < T_{SINGLE_EXEC}$, such that $\delta = (T_{NV_SAVE} + \gamma)$. In this case, as shown in Figure 9, the processor has enough time to perform the NV save operation, but does not have enough time to complete another sensor read and computation cycle before power down.

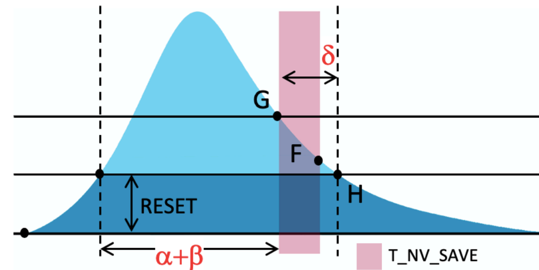

Fig. 9. (case I.a) where $\delta = T_{NV_SAVE} + \gamma$

Saving after crossing the threshold (point G) therefore saves the correct state. When this is the case, the threshold-based strategy is to be preferred compared to, for instance, the milestone-based strategy with milestone = 1. In fact, the threshold-based strategy saves the state only when necessary, and always when necessary, achieving full correctness with minimal overhead.

- case I.b: $\exists N, N \geq 1$, such that $\delta \gg (T_{NV_SAVE} + N \times T_{SINGLE_EXEC})$. In this case, shown in Figure 10, the voltage trace remains between the two thresholds for a long time before falling under the reset threshold. In a threshold-based strategy, the state is saved upon crossing the first threshold. The processor, then, has time to execute several sensor read and computation cycles, without another interrupt begin generated. The threshold-based strategy thus results in losing all the N values computed after the NV save. Employing a milestone-based strategy (setting milestone = 1 for best case results) will be more

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

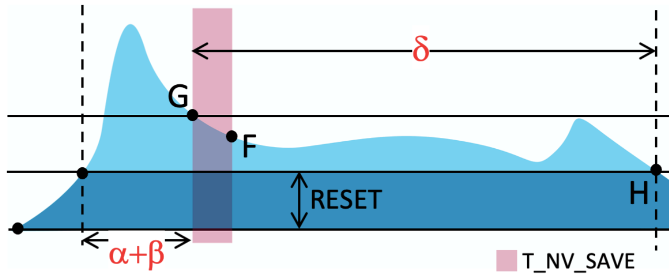

Fig. 10. (case I.b) where $\delta \gg (T_{NV_SAVE} + N \times T_{SINGLE_EXEC})$

effective to preserve all the values. Making use of the periodic strategy (with period = T_{SINGLE_EXEC}) will be equally effective as milestone based, but with added power overhead due to the interrupt generating counter peripheral.

2) case II: $\delta < T_{NV_SAVE}$*:* For this case, we assume that the time δ available between the crossing of the first threshold and the reset level is not sufficient for saving the state of the processor. In this case, clearly, the threshold-based strategy is not usable, as it will always result in an incorrect restore. We, therefore, consider the amount of time between the resume of program execution and the next power down $\beta + \delta$, assuming this is sufficient for at least one execution and one save.

- case II.a: $\beta + \delta \geq (T_{SINGLE_EXEC} + T_{NV_SAVE})$. The limit case, shown in Figure 11, is when there is time exactly for one execution of the application iteration, and one NV save operation. Then, selecting the milestone

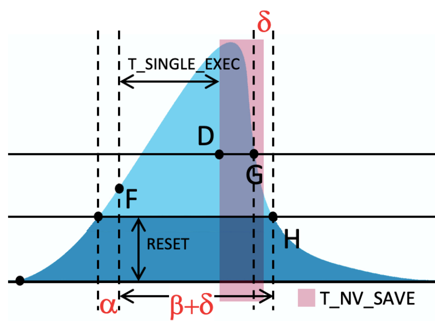

Fig. 11. (case II.a) where $\beta + \delta \geq (T_{SINGLE_EXEC} + T_{NV_SAVE})$

based (with milestone = 1, for best case) instead of the periodic strategy (as mentioned in case I.b) will always ensure that the immediate output generated is saved after each iteration of the code.
- case II.b: $\beta + \delta \gg (T_{SINGLE_EXEC} + T_{NV_SAVE})$. In this second case, there is ample time between power up and power down to execute several iterations, as shown in Figure 12.

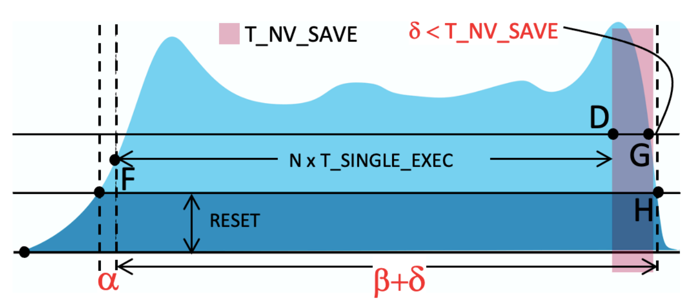

Fig. 12. (case II.b) where $\beta + \delta \gg (T_{SINGLE_EXEC} + T_{NV_SAVE})$

Because power down is infrequent compared to the iteration, opting for a periodic strategy, with a period equal to the power-ON interval, will be more efficient, both in terms of reduced CPU overhead and reduced power consumption for the NV save, instead of a milestone-based strategy. This can be seen in Table VI, with 2500 clock cycles.

B. Average power-ON interval

The length of the power-ON interval, as we have seen, affects the effectiveness of each strategy. We summarize the impact below.

1) Periodic strategy: Choosing a period which is equal to the average power-ON interval (54844 clock cycles from Table VI) yields a corresponding average performance. In our specific case study, this results in a 50% effectiveness (all Save-Success cases are considered to be Success) in terms of correctness. A shorter period increases performance, up to the minimum power-ON interval (2500 clock cycles from Table VI), yielding a 100% effectiveness in terms of correctness. At the same time, the overhead increases due to the execution of potentially unnecessary save operations. The power-ON interval is therefore the base metric to use to tune the period of the interrupt generating counter with respect to the requirements.

2) Threshold-based strategy: The duration of the power-ON interval is substantially uncorrelated with the specific variations of the voltage on the additional thresholds. As a consequence, it is difficult to predict the performance of a threshold-based strategy on the basis of this metric. Nevertheless, if the voltage profile is somewhat regular and/or periodic, then one could select more relevant thresholds based on the average power-ON interval, especially if the slope of the trace is taken into account to derive the timing parameters discussed before.

3) Milestone-based strategy: If N is the number of code iterations (within a power interval) that can be successfully executed after starting up the `main()` code (without considering the time required for NV saves), then the equation that relates a power-ON interval and N_f can be written as:

$$\text{power-ON interval} = \alpha + \Delta + N\left(T_{SINGLE_EXEC} + \frac{T_{NV_SAVE}}{N_f}\right)$$

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Here N and N_f are variables, which should be tuned to get the right hand side almost equal to the left hand side. The value of Δ will be the remaining time left between the final reset and last successful save operation.

Best-case results in terms of correctness can be easily achieved (guaranteed) by setting $N_f = 1$, causing however worst case CPU overhead. So, for optimum performance (in terms of energy usage) our aim should be to maximize N_f, satisfying the condition $\Delta < T_{SINGLE_EXEC}$.

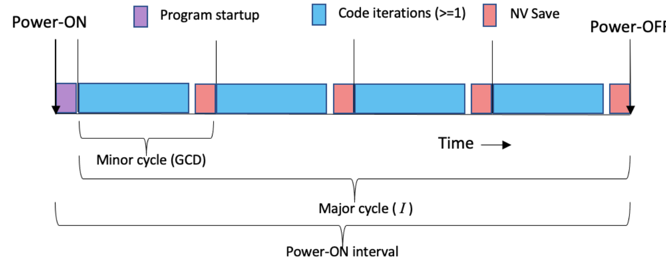

Fig. 13. The cyclic-like scheduling model for finding optimum N_f for milestone based strategy

As there are many power-ON intervals (with varying periods), we could find a common N_f that applies to all of them (given the voltage profile is somewhat regular and/or periodic). This common N_f will ensure the latest state is saved into NV for all the terrains before reset. The idea is that we find the GCD (Greatest Common Divisor) of all the intervals $I = (power_ON_interval) - \alpha$ and schedule our jobs (consisting of code iterations and NV saves) within each GCD (minor cycle) time-slot as show in Figure 13. This cyclic-scheduling, like NV saves, requires code iterations to fit inside (fine grained programs) each minor cycle, and there might be a possibility for the unused time between code iterations and NV save in each minor cycle. A much less complex implementation with almost the same results can be achieved by periodic strategy with the period set as a minor cycle (when an interrupt arrives, preemption is possible in this case).

C. CPU overhead

We have already discussed the impact of the different strategies in terms of CPU overhead. Nevertheless, when the threshold-based strategy is adopted, certain specific voltage trace shapes may have a large impact on performance, and that should be accounted for. One example is shown Figure 14, where the voltage level crosses the NV save threshold multiple times between power up and power down (i.e., within a β interval).

In this case, the threshold-based strategy introduces much higher CPU overhead. The worst case is achieved when the crossing frequency is approximately equal to T_{NV_SAVE}, as the NV save would happen back-to-back. Even if the trace satisfies case I.a, we could go for periodic/milestone based strategies in case of frequent crossings.

Similarly, under the same trace, a periodic strategy with period close to T_{NV_SAVE} would also result in maximum CPU overhead, as is also the case for a milestone-based strategy with $N_f = 1$.

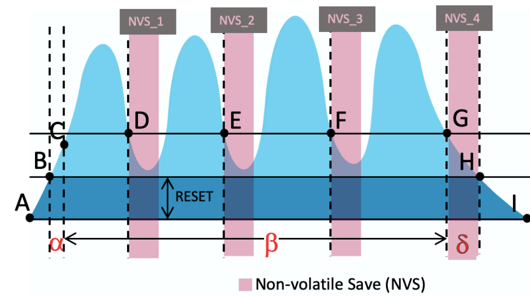

Fig. 14. Multiple threshold crossing within a single terrain. (x-axis = time, y-axis= voltage)

D. Development Guideline

When historic power traces are available, we present a guideline to set the optimal intermittent policy according to power-ON pulses as shown in Figure 15. A mixed strategy could be employed for various δ seen across different period of time in a day.

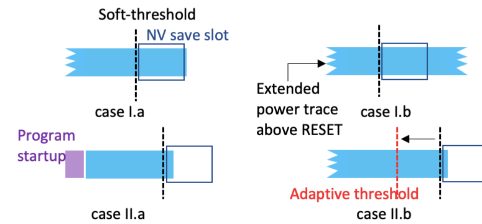

Fig. 15. Power-ON pulses modeled for each individual cases.

As we have seen in Section IV, the CPU overhead of the threshold-based strategy compares favorably to others for the same performance. So threshold-based saves are preferable when feasible. Case I.a is the ideal case, where threshold-based NV-save fits perfectly into the power-trace immediately after the soft threshold is crossed. The extra time left after NV-save in case I.b can be tackled either by adaptively reducing the soft-threshold (if possible), so that the NV-save is further pushed just before reset or adopt a cyclic schedule approach for the remaining time after save (i.e., schedule the group of power-ON pulses with case I.b characteristics with $minor - cycle = GCD(\delta - T_{NV_SAVE})$). Similarly, in case II, an adaptive increase in threshold (as shown for case II.b in Figure 15) for the specific time-of-day can be employed to get enough δ for NV save. If this is not feasible, the schedule based approach for either milestone/periodic is the best option (i.e., schedule the group of power-ON pulses with case II characteristics with $minor - cycle = GCD(\beta + \delta)$).

Here, we only considered the voltage trace characteristics and clock cycles used by the core to conduct our study. As an alternative, the actual power consumption by the core and other logic should be accounted for a more precise policy. A compiler-based run-time environment (or a plugin) could be integrated into LLVM/GCC, automatically inserting the save-restore code. This would ensure minimum programmer overhead and capture the registers for dynamically saving, based on live analysis.

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

VI. Conclusions

Thanks to the progress in energy harvesting circuits and the decrease in processing power requirements, many IoT devices will work batteryless in an intermittent manner, using the sole energy converted from the environment. Intermittent computing requires significant microarchitectural modifications on existing processor designs to ensure computation progress despite the power failures automatically. We presented a systematic approach to emulate different processor architectures under the Intermittent Computing domain. We wrapped the Ibex RISC-V core to be suitable for transient computing applications as a demonstration. Moreover, this paper presented different processor state backup strategies based on an interrupt-based software approach, which do not require modifications to the microarchitecture of existing processors. We compared the performance with irregular power and provided comprehensive development guidelines using an extensive set of measurements and tests with a typical IoT application used as a benchmark.

As of now, our backup strategies consider the voltage trace characteristics and the number of clock cycles to perform a backup operation. As an alternative, the actual power consumption of the processor and other logic elements can be considered to implement a more precise policy.

There are a number of standard benchmarks specific to the Ibex RV32IMC core [40], but these are oriented towards performance evaluation (like the CoreMark) and do not account for possible intermittent behavior. Therefore we have developed a specific application to validate our approach (shown as the data acquisition loop in Listing 1), which is representative of typical implementations. As future work, we will use real traces from different energy harvester setups (i.e., thermogenerators, piezoelectrics, micro-photovoltaic, etc.) and analyze how optimal policies change according to different scenarios and benchmarks.

References

[1] A. Baranov, D. Spirjakin, S. Akbari, A. Somov, and R. Passerone, “POCO: ‘perpetual’ operation of CO sensor node with hybrid power supply,” *Sensors & Actuators A: Physical*, vol. 238, pp. 112–121, 2016.

[2] I. Minakov, R. Passerone, A. Rizzardi, and S. Sicari, “A comparative study of recent wireless sensor network simulators,” *ACM Transactions on Sensor Networks*, vol. 12, no. 3, pp. 20:1–20:39, 2016.

[3] I. Minakov and R. Passerone, “PASES: An energy-aware design space exploration framework for wireless sensor networks,” *Journal of Systems Architecture*, vol. 59, no. 8, pp. 626–642, September 2013.

[4] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, “Camaroptera: A batteryless long-range remote visual sensing system,” in *the 7th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys’19)*, 2019, pp. 8–14.

[5] J. Zhan, G. V. Merrett, and A. S. Weddell, “Exploring the effect of energy storage sizing on intermittent computing system performance,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[6] G. Dalpiaz, A. Longo, M. Nardello, R. Passerone, and D. Brunelli, “A battery-free non-intrusive power meter for low-cost energy monitoring,” in *Proceedings of the 1st IEEE International Conference on Industrial Cyber-Physical Systems*, ser. ICPS 2018, Saint Petersburg, Russia, May 15-18, 2018.

[7] Texas Instruments. (2018) MSP430FR5969 launchpad development kit. Last accessed: 2018. [Online]. Available: http://www.ti.com/tool/MSP-EXP430FR5969

[8] Texas Instruments, Inc., “FRAM FAQs,” https://www.ti.com/lit/wp/slat151/slat151.pdf, 2014, last accessed: 2022.

[9] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, “Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[10] 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.

[11] K. Ma, X. Li, K. Swaminathan, Y. Zheng, S. Li, Y. Liu, Y. Xie, J. J. Sampson, and V. Narayanan, “Nonvolatile processor architectures: Efficient, reliable progress with unstable power,” *IEEE Micro*, vol. 36, no. 3, pp. 72–83, 2016.

[12] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, “A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops,” in *ESSCIRC (ESSCIRC), 2012 Proceedings of the*. IEEE, 2012, pp. 149–152.

[13] X. Li, S. George, K. Ma, W.-Y. Tsai, A. Aziz, J. Sampson, S. K. Gupta, M.-F. Chang, Y. Liu, S. Datta *et al.*, “Advancing nonvolatile computing with nonvolatile NCFET latches and flip-flops,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 11, pp. 2907–2919, 2017.

[14] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, “Microprocessor optimizations for the Internet of Things: a survey,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 7–20, 2018.

[15] lowRisc, “Ibex documentation,” https://ibex-core.readthedocs.io/en/latest/, 2021, last accessed: 2022.

[16] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, “Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications,” in *Proceedings of the 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[17] Calista Redmond, “RISC-V is ushering in a new era of silicon design and processor innovation,” https://www.hipeac.net/news/6921/risc-v-is-ushering-in-a-new-era-of-silicon-design-and-processor-innovation/, 2020, last accessed: 2022.

[18] S. Ruffini, L. Caronti, K. S. Yıldırım, and D. Brunelli, “Norm: An fpga-based non-volatile memory emulation framework for intermittent computing,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 4, 2022.

[19] F. Su, K. Ma, X. Li, T. Wu, Y. Liu, and V. Narayanan, “Nonvolatile processors: Why is it trending?” in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 966–971.

[20] Y. Wang, Y. Liu, S. Li, X. Sheng, D. Zhang, M.-F. Chiang, B. Sai, X. S. Hu, and H. Yang, “PaCC: A parallel compare and compress codec for area reduction in nonvolatile processors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 7, pp. 1491–1505, 2014.

[21] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, and J. Hu, “Checkpoint aware hybrid cache architecture for NV processor in energy harvesting powered systems,” in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016, pp. 1–10.

[22] B. Lucia and B. Ransford, “A simpler, safer programming and execution model for intermittent systems,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.

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

[24] Y. Liu, J. Yue, H. Li, Q. Zhao, M. Zhao, C. J. Xue, G. Sun, M.-F. Chang, and H. Yang, “Data backup optimization for nonvolatile SRAM in energy harvesting sensor nodes,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 10, pp. 1660–1673, 2017.

[25] K. Qiu, M. Zhao, Z. Jia, J. Hu, C. J. Xue, K. Ma, X. Li, Y. Liu, and V. Narayanan, “Design insights of non-volatile processors and accelerators in energy harvesting systems,” in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020, pp. 369–374.

[26] D. Balsamo, A. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, “Graceful performance modulation for power-

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2022.3169108, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

neutral transient computing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 738–749, 2016.

[27] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020, pp. 85–99.

[28] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, p. 159–170, March 2011.

[29] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 129–144.

[30] N. Bhatti and L. Mottola, "HarvOS: Efficient code instrumentation for transiently-powered embedded devices," in *Proc. IPSN*. Pittsburgh, PA, USA: ACM/IEEE, Apr. 18–21, 2017.

[31] M. Hicks, "Clank: Architectural support for intermittent computation," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 228–240, 2017.

[32] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. OSDI*. Savannah, GA, USA: ACM, Nov. 2–4, 2016, pp. 17–32.

[33] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs," in *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2013, pp. 216–224.

[34] F. Li, K. Qiu, M. Zhao, J. Hu, Y. Liu, Y. Guan, and C. J. Xue, "Checkpointing-aware loop tiling for energy harvesting powered nonvolatile processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 15–28, 2019.

[35] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 526–537.

[36] S. Ruffini, K. S. Yıldırım, and D. Brunelli, "Emulation of non-volatile digital logic for batteryless intermittent computing," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022.

[37] PERSISTLab. (2017) Input voltage trace. Last accessed: 2022. [Online]. Available: https://github.com/PERSISTLab/BatterylessSim/blob/master/traces/2.txt

[38] lowRisc, "Ibex load-store unit documentation," https://ibex-core.readthedocs.io/en/latest/03_reference/load_store_unit.html, 2021, last accessed: 2022.

[39] PERSISTLab. (2017) Available voltage traces. Last accessed: 2022. [Online]. Available: https://github.com/PERSISTLab/BatterylessSim/blob/master/traces/all-traces-grid.pdf

[40] Ibex. (2020) Ibex benchmarks. Last accessed: Feb. 2022. [Online]. Available: https://github.com/lowRISC/ibex/tree/master/examples/sw/benchmarks

0278-0070 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITA TRENTO. Downloaded on April 22,2022 at 10:31:31 UTC from IEEE Xplore. Restrictions apply.