

# DiCA: A Hardware-Software Co-Design for Differential Check-Pointing in Intermittently Powered Devices

Antonio Joia Neto, Adam Caulfield, Chistabelle Alvares, Ivan De Oliveira Nunes

Rochester Institute of Technology

aj4775@rit.edu, ac7717@rit.edu, cda5542@rit.edu, ivanoliv@mail.rit.edu

**Abstract**—Intermittently powered devices rely on opportunistic energy-harvesting to function, leading to recurrent power interruptions. Therefore, check-pointing techniques are crucial for reliable device operation. Current strategies involve storing snapshots of the device’s state at specific intervals or upon events. Time-based check-pointing takes check-points at regular intervals, providing a basic level of fault tolerance. However, frequent check-point generation can lead to excessive/unnecessary energy consumption. Event-based check-pointing, on the other hand, captures the device’s state only upon specific trigger events or conditions. While the latter reduces energy usage, accurately detecting trigger events and determining optimal triggers can be challenging. Finally, differential check-pointing selectively stores state changes made since the last check-point, reducing storage and energy requirements for the check-point generation. However, current differential check-pointing strategies rely on software instrumentation, introducing challenges related to the precise tracking of modifications in volatile memory as well as added energy consumption (due to instrumentation overhead).

This paper introduces *DiCA*, a proposal for a hardware/software co-design to create differential check-points in intermittent devices. *DiCA* leverages an affordable hardware module that simplifies the check-pointing process, reducing the check-point generation time and energy consumption. This hardware module continuously monitors volatile memory, efficiently tracking modifications and determining optimal check-point times. To minimize energy waste, the module dynamically estimates the energy required to create and store the check-point based on tracked memory modifications, triggering the check-pointing routine optimally via a non-maskable interrupt. Experimental results show the cost-effectiveness and energy efficiency of *DiCA*, enabling extended application activity cycles in intermittently powered embedded devices.

**Index Terms**—Intermittent Computing, Energy Harvesting, Check-pointing

## I. INTRODUCTION

In contrast to traditional devices that depend on batteries or external power sources, energy harvesting devices capitalize on ambient energy from the surrounding environment to fuel their operations. By leveraging opportunistic energy sources such as solar power [19] and kinetic energy [8], [9], intermittent computing enables energy harvesting devices to operate under unpredictable power interruptions. By eliminating the need for batteries, these devices become more sustainable and environmentally friendly, as they reduce electronic waste [30]. Moreover, battery-less devices offer increased convenience and autonomy since they do not require frequent battery replacements or recharging. This allows for significantly reduced

device size and weight, enabling sleeker and more compact designs [18]. By removing the need for large batteries, battery-less devices open up new possibilities for miniaturization and integration into various applications [9].

On the other hand, power disruptions on intermittent devices present challenges to reliable operation, including data loss, difficulties in state preservation, task resumption, and system instability [22], [28]. As a result, the execution of applications in intermittent devices follows cyclic patterns, where task execution occurs during periods of power availability, followed by power depletion. These cycles require strategies to maintain state across power depletion cycles, ensuring correct operation.

A *naive* solution to this problem is the exclusive use of Non-Volatile Memory (NVM), such as FRAM, to store all data. While a completely FRAM-based solution ensures reliability, it increases energy consumption due to increased access latency when compared to volatile memory, such as SRAM. Conversely, an exclusively SRAM-based implementation offers high energy efficiency while lacking reliability across power depletion cycles [7], [17]. An alternative approach is to integrate check-pointing into the execution cycles of intermittent devices. Check-pointing involves regularly saving the volatile system state to NVM, creating snapshots that capture the current execution context. Therefore, after power depletion, the system can restore the latest check-point and resume execution properly.

One approach to implement the check-pointing is to modify the original software with additional logic to determine when a minimum energy level has been reached and save the program context to NVM [11], [29], [31]. Although functional, these techniques also extend the program’s runtime and thus incur additional energy costs, reducing the original application’s activity cycle. Alternative techniques propose one-time check-pointing [6], [7] that is triggered when the supplied voltage falls below a certain threshold. However, the latter does not track the modifications made to the volatile memory (VM), requiring the entire VM to be copied to NVM at each check-point. This process also incurs a significant energy cost.

Differential check-pointing is an approach aimed at minimizing the amount of data copied from the VM to the NVM. The fundamental concept is to track modified memory addresses and copy only the modified memory blocks to NVM for each check-point. This approach has led to a notable reduction in both run-time and energy costs associated with

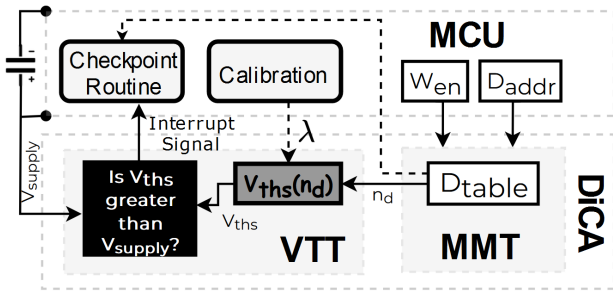


Fig. 1. Illustration of  $\mathcal{D}iCA$ 's high level architecture

check-pointing and has been systematically employed in prior work [2], [4], [10]. However, prior differential check-pointing schemes are implemented by instrumenting the application source code, still introducing energy and run-time overhead.

### A. Contributions

Given the popularity of intermittent computing applications, we argue that future devices could be manufactured with minimal hardware support to facilitate check-pointing and reduce associated energy and run-time costs. With that premise in mind, we propose  $\mathcal{D}iCA$ : a Differential Check-point Assistant based on a hardware/software co-design.  $\mathcal{D}iCA$  eliminates software instrumentation and application code modifications and dynamically determines optimal differential check-pointing times based on the amount of memory to be saved to NVM. More specifically,  $\mathcal{D}iCA$  comprises:

- A lightweight (and energy efficient) hardware module – called Memory Modification Tracker – used to efficiently mark modified segments in VM. This module enables tracking of differential memory modifications without requiring any instrumentation or code modification, simplifying and optimizing the check-point generation.
- A new interrupt source that optimizes check-point timings and reduces energy consumption. This approach dynamically estimates the minimal supply voltage required to perform the check-point based on the number of segments modified in VM. When the dynamically defined threshold is reached,  $\mathcal{D}iCA$  hardware generates a non-maskable interrupt to initiate the check-pointing procedure.
- A software routine that interacts with the  $\mathcal{D}iCA$  hardware to copy appropriate memory segments from VM to NVM based on the optimal parameters configured by  $\mathcal{D}iCA$  hardware module.

The intangibility of applying hardware modifications to real devices has been a significant obstacle for check-pointing techniques, leading to a reliance on software instrumentation. Our approach is rooted in the premise that minimal hardware modifications are both feasible and realistic considering the recent popularity of and demand for energy harvesting devices. We believe that, given their distinct purposes, these devices can benefit from simple, practical, and inexpensive hardware modifications to enhance their performance without requiring massive architectural overhauls.

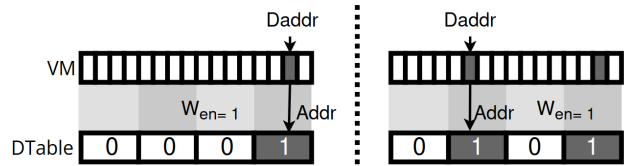


Fig. 2. Illustration of DTable memory tracking

### B. Scope

Battery-less and intermittent computing devices are typically implemented with micro-controller units (MCUs) that run software at bare-metal, have low-cost, and are energy efficient. In this work, we focus on ultra low-energy MCUs (e.g., Atmel AVR ATmega [5], TI MSP430 [16]) which feature single-core 8- or 16-bit CPUs running at low clock frequencies (usually 1 to 16 Mhz). They use between 4 and 16 KBytes of SRAM as VM while the rest of the address-space is available for NVM. We implement  $\mathcal{D}iCA$  prototype atop an open-source version of TI MSP430 from openCores [15].

### C. Organization

This paper is structured as follows. Section II presents the high-level ideas in  $\mathcal{D}iCA$  design. Section III delves into the details of  $\mathcal{D}iCA$  architecture and specifies  $\mathcal{D}iCA$  hardware and software components. Section IV discusses the implementation of  $\mathcal{D}iCA$  prototype, the experimental set-up, and presents  $\mathcal{D}iCA$  empirical evaluation. Section V discusses related work and Section VI concludes the paper.

## II. $\mathcal{D}iCA$ HIGH-LEVEL OVERVIEW

$\mathcal{D}iCA$  is a hardware/software co-design. It includes an inexpensive hardware module that tracks modified VM segments. Compared to methods that perform this tracking in software, it reduces energy consumption, enabling more instructions belonging to the original application to be executed per power cycle.  $\mathcal{D}iCA$  hardware also implements a new interrupt source that triggers the check-point generation, i.e., the process of copying modified VM segments to NVM, based on the estimated required time and available energy. Figure 1 illustrates  $\mathcal{D}iCA$  architecture.

### A. $\mathcal{D}iCA$ Hardware

$\mathcal{D}iCA$  hardware has two sub-modules: Memory Modification Tracker (MMT) and Voltage Threshold Tracker (VTT).

MMT divides the volatile memory into blocks. It detects whether these blocks have been written to by monitoring two internal CPU signals: the write enable bit (denoted  $W_{en}$ ), which indicates whether the MCU is writing to the memory, and the  $D_{addr}$  signal, that defines the memory address being written to when  $W_{en} = 1$ . Whenever  $W_{en}$  is active,  $\mathcal{D}iCA$  takes the value of  $D_{addr}$  and uses it as an index to set a bit in a register vector called (Dirty-bits Table) DTable, indicating that the block to which the address  $D_{addr}$  belongs to has been modified. Figure 2 depicts  $\mathcal{D}iCA$  updating DTable as VM blocks are modified. To track the differential changes,  $\mathcal{D}iCA$

also has a controller that allows software to reset DTable after loading the prior check-point.

(VTT) generates an interrupt signal based on how many memory segments in VM have been modified (as detected by MMT). It dynamically calculates a threshold voltage supply value, denoted  $V_{ths}$ . The value of  $V_{ths}$  is determined by counting the number of modified VM memory blocks, i.e., the number of active bits in DTable, to allow sufficient time for the check-pointing routine. When the supplied voltage falls below  $V_{ths}$ , indicating an impending power depletion, a non-maskable interrupt is triggered.

### B. DiCA Software

DiCA software component is implemented as an interrupt service routine (ISR) associated with the VTT-generated interrupt. Based on DTable, the ISR captures a snapshot of modified segments in VM and CPU registers, and copies them to NVM. After the successful check-point generation, the system is powered off until the energy harvesting component recharges the power supply. When the supplied voltage reaches a full charge threshold (denoted  $V_{full}$ ), the MCU restarts and DiCA software restores the most recent check-point, allowing the system to resume operation from the suspended state.

## III. DiCA IN DETAILS

This section details DiCA design. For quick reference, Table I summarizes the notation used in the rest of the paper.

TABLE I  
NOTATION SUMMARY

Notation	Description
DTable	Bit-vector indicating memory blocks that have been modified since the last check-point
DTable <sub>size</sub>	Size of DTable (in number of bits)
DTable [d]	d-th bit in DTable
$n_d$	Number of active bits in DTable
<i>reset</i>	A 1-bit CPU signal that sets all the DTable bits to zero
$W_{en}$	A 1-bit signal that indicates if the MCU is writing to memory
$D_{addr}$	CPU signal containing memory address being modified by the MCU when $W_{en} = 1$
<i>BSS</i>	Block size shift
<i>SP</i>	Stack Pointer
<i>SP<sub>Lim</sub></i>	Stack Pointer Limit: lowest memory address for data allocation in the CPU stack architecture
NVM and VM	Non Volatile Memory and Volatile Memory
VM <sub>size</sub>	Volatile Memory size
VM <sub>min</sub> , VM <sub>max</sub>	Maximum and Minimum memory addresses of VM
<i>SF<sub>mask</sub></i>	Stack Frame mask
VM <sub>size</sub> <sup>block</sup>	Size of a memory block
$V_{ths}$	Supply voltage threshold
$V_{ths}(n_d)$	Function mapping $n_d$ to $V_{ths}$

### A. Memory Modification Tracker (MMT)

MMT is a hardware sub-module designed, in its simplest form, to monitor the memory locations written by the CPU. Central to this component is the DTable, a peripheral that enables efficient differential check-point generation, obviating the need for instrumentation or application-specific code modifications.

DTable is a bit-vector where each bit maps to a memory block, sequentially, in VM. Memory blocks are of pre-defined size denoted as  $VM_{size}^{block}$ . DTable's size is determined by the total size of the VM (VMsize) divided by  $VM_{size}^{block}$ .  $VM_{size}^{block}$  defines the granularity of memory tracking and is a design parameter chosen at MCU manufacturing time. To simplify the hardware implementation, we restrict  $VM_{size}^{block}$  to powers of 2.

MMT monitors the CPU signals  $W_{en}$  and  $D_{addr}$  which are used by the CPU to write to memory. As part of the underlying CPU behavior,  $W_{en}$  is set to 1 whenever a write access to memory occurs, whereas  $D_{addr}$  contains the address of the memory location being written when  $W_{en} = 1$ . Therefore, whenever  $D_{addr}$  is within VM and  $W_{en} = 1$ , MMT determines a DTable index (*Addr*) by shifting the relative address of VM ( $D_{addr} - VM_{min}$ ) by BSS bits, where  $BSS = \log_2 VM_{size}^{block}$ . Then MMT sets the bit corresponding to *Addr* in DTable to 1. Figure 2 illustrates DTable functionality. The specification of MMT basic behavior is presented in Definition 1 (see Section III-B for MMT extended version that ignores de-allocated stack frames during check-point generation).

#### Definition 1: Memory Modification Tracker Model

$$i \in [1, DTable_{size}]$$

$$Addr := (D_{addr} - VM_{min}) \gg BSS$$

$$DTable[i] := \begin{cases} 0 & \text{if } reset \\ 1 & \text{if } (i = Addr) \wedge \\ & (D_{addr} \in VM) \wedge W_{en} \\ DTable[i] & \text{Otherwise} \end{cases}$$

**Rationale.** MMT detects differential memory changes between the last check-point and the current memory state. This feature reduces the number of memory blocks that must be copied to NVM in the next check-point, as unmodified data remains consistent in NVM and need not be copied. To support this functionality, the MMT module incorporates a control bit (*reset*) that clears DTable on each power cycle (i.e., at MCU boot). Figure 3 illustrates the differential check-pointing process across subsequent check-points based on DTable.

### B. Extending MMT to Ignore De-Allocated Stack Frames

MMT basic design only sets DTable bits to track memory modifications. However, it does not clear DTable if a memory block is no longer in use by the program. Since functions are called (and returned from) multiple times in most programs, MMT basic design would check-point function stack frames that are no longer in use. Considering nested function calls, this check-pointing approach would unnecessarily include a large number of memory blocks related to stack frames that are no longer in use. To address this issue, MMT is extended

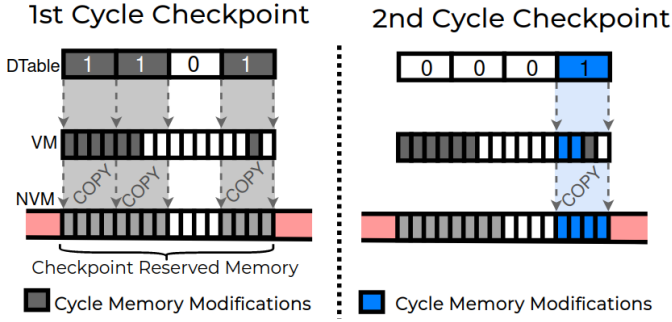


Fig. 3. Visualization of differential check-pointing using DTable

to clear DTable bits associated with stack frames of functions upon their completion.

The stack frame cleaning is depicted in Figure 4. It uses a bit mask, called  $SF_{mask}$ , that masks DTable entries. This mask indicates whether the corresponding D-table bit is no longer in use.  $SF_{mask}$  values are determined based on two values. The first,  $SP_{LIM}$ , defines the lowest memory address<sup>1</sup> that can be used for stack allocation in the MCU. The second,  $SP$ , is the current stack pointer, i.e., a CPU signal that contains the address of the top of the currently allocated stack. Therefore, at any given time, the memory region between  $SP$  and  $SP_{LIM}$  is unallocated.

To produce  $SF_{mask}$ , the relative memory positions of  $SP$  and  $SP_{LIM}$  with respect to the VM are computed by subtracting  $VM_{min}$  from both input parameters.  $VM_{min}$  indicates the lowest memory address of VM. Next, the indices that correspond to these relative memory addresses in DTable are obtained by shifting the resulting values by  $BSS$ . This generates the indices  $ID_{SP}$  and  $ID_{SP_{LIM}}$ . The mask is then generated by setting all indices between  $ID_{SP}$  and  $ID_{SP_{LIM}}$  to 0 and leaving all others as 1. MMT behavior, extended with the stack frame cleaner, is specified in Definition 2.

### C. Voltage Threshold Tracker

To reduce power consumption, DiCA establishes the ideal moment to start the check-pointing routine. To this end, it implements a non-maskable interrupt source that is triggered when the system's supplied voltage ( $V_{supply}$ ) falls below a dynamically defined threshold ( $V_{ths}$ ).  $V_{ths}$  serves as a proxy for the amount of energy required for the check-pointing routine to fully execute before  $V_{supply}$  drops below to a level that is insufficient to sustain MCU operation.  $V_{ths}$  is calibrated based on the number ( $n_d$ ) of active bits of in DTable (see Section III-D for details). When  $V_{ths}$  is reached, the interrupt is triggered.

$n_d$  default value is 0 after loading a check-point. It is incremented by one whenever a DTable value changes from 0 to 1 (as detected with the operation  $[\neg DTable[Addr] \wedge W_{en}]$ ). Conversely, when a bit in the DTable is cleared (due to stack

<sup>1</sup>The use of the "lowest memory address" as the limit assumes that the stack grows downwards in the underlying MCU architecture.

**Definition 2: Memory Modification Tracker with Stack Frame Cleaner:**

- Stack Frame mask:

$$i \in [1, DTable_{size}]$$

$$ID_{SP} := ((SP - VM_{min}) \gg BSS)$$

$$ID_{SP_{LIM}} := ((SP_{LIM} - VM_{min}) \gg BSS)$$

$$SF_{mask}[i] := (i \leq ID_{SP}) \vee (i \geq ID_{SP_{LIM}})$$

- DTable with Stack Frame Cleaner:

$$DTable[i] := \begin{cases} 0 & \text{if } SF_{mask}[i] \vee reset \\ 1 & \text{if } (i = Addr) \wedge \\ & (D_{addr} \in VM) \wedge W_{en} \\ DTable[i] & \text{Otherwise} \end{cases}$$

frame cleaning)  $n_d$  is decremented. To determine the number of bits cleared due to stack frame cleaning, DiCA checks the previous stack pointer index, denoted  $ID_{SP}^{t-1}$  and subtracts it from the current index  $ID_{SP}^t$ . If the result is greater than zero, it is subtracted from  $n_d$ . The value of  $n_d$  is then used to compute the  $V_{ths}$ , as detailed in Section III-D. The interrupt signal generation is specified in Definition 3.

**Definition 3: Voltage Threshold Tracker Model:**

- Stack Frame reduction counter:

$$ID_d := ID_{SP}^t - ID_{SP}^{t-1}$$

- Dtable bit counter:

$$n_d := \begin{cases} 0 & \text{if } reset \\ n_d + 1 & \text{elif } (\neg DTable[Addr] \wedge W_{en}) \\ n_d - ID_d & \text{elif } (ID_d > 0) \\ n_d & \text{Otherwise} \end{cases}$$

- Interrupt Signal:

$$IT_{sig} \leftarrow V_{supply} < V_{ths}(n_d)$$

### D. Voltage Threshold ( $V_{ths}$ ) Calibration

$V_{ths}$  is determined by two factors:  $n_d$  and a constant  $\lambda$ . A device calibration phase is conducted at system deployment time to determine  $\lambda$ . The calibration process involves fine-tuning  $\lambda$  to the specific MCU in order to obtain an optimal  $V_{ths}(n_d)$  function for that particular device. Similar to prior work [6], we assume that the voltage supply decay between 3.6V (fully charged supply) to 2.0V (minimal operational threshold) is linear. Therefore,  $\lambda$  is determined by measuring the total amount of blocks that can be copied in one full device power cycle ( $N$ ) and dividing 1.6V by  $N$ .

**Implementation of  $V_{ths}(n_d)$ :** In order to avoid hardware multiplications, the product of  $\lambda$  and  $n_d$  is computed using

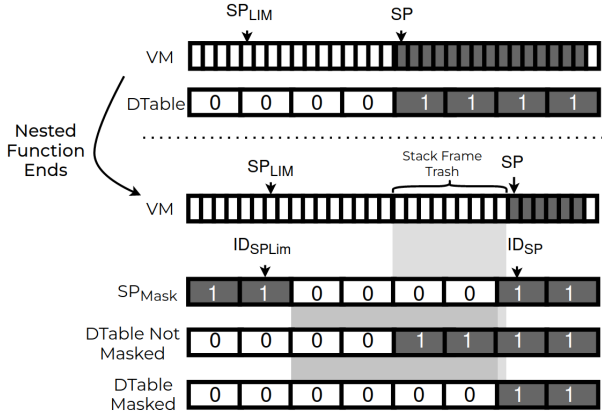


Fig. 4. Visualization of the DTable stack frame trash cleaning

addition operations. Initially, the threshold voltage  $V_{ths}$  is set to  $V_{min}$ . As  $n_d$  increases or decreases, an additional register  $n'_d$  tracks the values of  $n_d$  with a unitary increment or decrement. This additional register is necessary because  $n_d$  can decrease by more than one value at a time, due to the stack frame cleaner. This is specified in Definition 4.

*Definition 4:*  $V_{ths}$  Calibration Model:

$$\begin{aligned}
 V_{ths}(0) &= V_{min} \\
 n'_d &:= \begin{cases} n'_d + 1 & \text{if } n_d > n'_d \\ n'_d - 1 & \text{if } n_d < n'_d \\ n'_d & \text{if } n_d = n'_d \end{cases} \\
 V_{ths} &:= \begin{cases} V_{ths} + \lambda & \text{if } n_d > n'_d \\ V_{ths} - \lambda & \text{if } n_d < n'_d \\ V_{ths} & \text{if } n_d = n'_d \end{cases}
 \end{aligned}$$

### E. Check-Point Generation

When the check-pointing ISR is triggered ( $IT_{sig} = 1$ ), DiCA software executes to copy VM memory blocks that are marked in DTable to a dedicated region in NVM. Enough space in NVM should be reserved for this purpose.

Before the check-pointing process, a flag stored in the NVM is set to *True*, indicating the active state of the check-pointing process. Once check-pointing is completed, the flag is unset. If the check-pointing process is not successfully completed, the value of  $\lambda$  can be adjusted to a more conservative value for the subsequent power cycle.

In order to generate the VM snapshot using the DTable, the DiCA software iterates through each bit of DTable. If a bit is 0, the iteration proceeds to the next bit. If the bit is 1, the memory block associated with that bit (of size  $VM_{size}^{block}$ ) is copied to its corresponding position in NVM. This process is shown in Algorithm 1.

### Algorithm 1: Memory check-pointing using DTable

---

**Data:** DTable, DTable<sub>size</sub>; NVM = pointer to the first position of nonvolatile memory related to the check-point, VM = pointer to the first position of the ram memory,  $b = VM_{size}^{block}$

```

for  $i \leftarrow 0$  to DTablesize - 1 do
  if DTable[ $i$ ] is 1 then
    memcpy(NVM[ $b * i$ ], VM[ $b * i$ ],  $b$ );
  end
end

```

---

### F. Execution Resumption

When resuming execution in a new power cycle,  $V_{ths}$  can be re-calibrated by adjusting  $\lambda$ . The system must also check the integrity of the current check-point. If no check-point is found or if it is corrupted (or incomplete), the application starts anew. If a valid check-point exists, it must be reloaded by copying the check-point data from NVM to the VM, and restoring the CPU registers. Additionally, after copying the check-point data to NVM, DTable and  $n_d$  are cleared. This step sets DiCA up for check-pointing the next power cycle. Finally, the Program Counter (PC) is loaded with the address that was meant to be executed by the application in the previous power cycle immediately before DiCA interrupt was triggered.

## IV. PROTOTYPE & EXPERIMENTS

We synthesize DiCA using a Xilinx Artix-7 FPGA [33], on a Basys-3 [13] prototyping board. The Xilinx Vivado tool-set [32] was used for synthesizing the on top of the openMSP430 [15] MCU core. DiCA was written in the Verilog hardware description language. Each module implements the logic outlined in Section III. DiCA was implemented in 368 lines of Verilog code for the MMT and VTT hardware modules (including their integration with the underlying openMSP430 core) and 201 lines of C code for DiCA software ISR and check-point recover.

In addition to the FPGA-based prototype, we perform complementary experiments using a low-energy device for realistic energy results. We use the MSP430FR2476 MCU for these experiments, featuring 8kB of SRAM (VM) and 64KB of FRAM (NVM) and running at a CPU clock frequency of 1MHz.

### A. Profiling $VM_{size}^{block}$

An important manufacturing time decision is to determine  $VM_{size}^{block}$  in DiCA. This decision has a direct impact on the number of bits in DTable, thereby affecting the hardware size. However, it also plays a vital role in enhancing the granularity of memory-tracking, which has the potential to reduce check-pointing time and energy consumption. To determine the optimal value of  $VM_{size}^{block}$ , we profile the check-pointing runtime against various values. The results for  $VM_{size}^{block}$  varying from 8 to 512 Bytes are presented in the Figure 5.

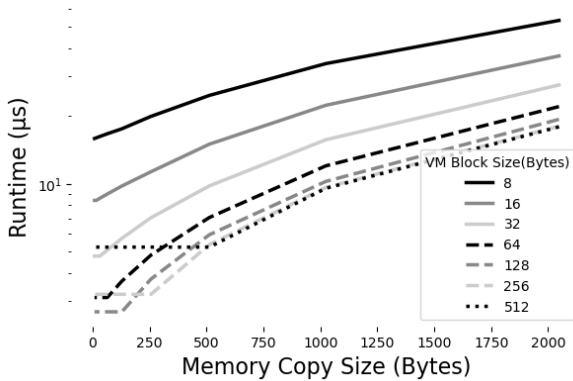


Fig. 5. Profiling of copy run-time vs.  $VM_{size}^{block}$

The results show that smaller  $VM_{size}^{block}$  incur longer run-times for copying the same amount of memory, even for small memory amounts. This can be attributed to the inherent overhead associated with the copy operation, encompassing memory address calculations, data transfers, and synchronization. Moreover, smaller  $VM_{size}^{block}$  result in larger DTable sizes, necessitating a more extensive search to identify modified bits, which further contributes to the augmented check-pointing run-time. Conversely, as the  $VM_{size}^{block}$  increases, it becomes apparent that larger values are constrained by a minimum overhead, which increases proportionally due to the granularity of memory-tracking.

Based on these experiments, we determined that the optimal  $VM_{size}^{block}$  for the MSP430FR2476 is 128 Bytes, as it provides the most favorable balance between run-time and granularity. When selecting such parameters, it is crucial to take into account the characteristics of the device’s memory and its clock frequency. These factors can vary across different devices and significantly influence the profile. Consequently, it is important to highlight that our prototype’s profile may not be optimal to other devices or MCU models. That being said, we use 128-Byte blocks as the reference value for the remainder of the experiments in *DiCA* evaluation.

### B. Hardware Footprint Overhead

We assess the hardware cost in terms of additional Look-up Tables (LUTs) and flip-flops/registers (FFs). The increase in LUTs reflects the additional chip cost/size attributed to combinatorial logic. The increase in FFs indicates the additional state required by sequential logic. Figure 6 shows the hardware cost of unmodified openMSP430 and the additional cost of *DiCA* hardware when configured to monitor memory blocks from 16 to 512 Bytes.

The cost of *DiCA* hardware is maximized when  $VM_{size}^{block}$  is at the lowest value of 16-Bytes. With this configuration, *DiCA* hardware incurs the maximum overhead with additional 730 LUTs and 561 FFs. However, this additional cost decreases as  $VM_{size}^{block}$  increases. For instance, configuring *DiCA* hardware to monitor 512-Byte blocks only requires 49 LUTs and 58 FFs. For a 128-Byte configuration, which we determined has an ideal check-pointing runtime (see Section IV-A), additional 114 LUTs and 106 FFs are required. Configuring *DiCA*

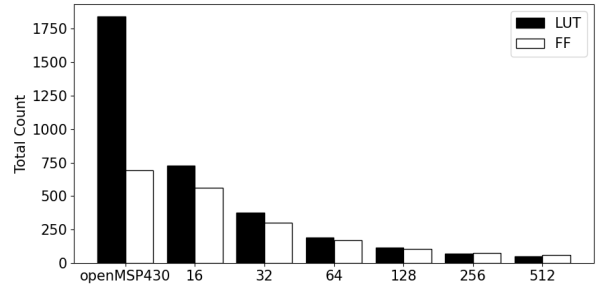


Fig. 6. Hardware cost of unmodified openMSP430 compared to additional cost of *DiCA* with  $VM_{size}^{block}$  of 16 to 512 Bytes

hardware to monitor 128-Byte blocks causes an increase of  $\approx 8.7\%$  relative to the unmodified openMSP430 core.

### C. Hardware Energy Overhead

While the added hardware modules eliminate the need for software-based memory tracking (and associated energy consumption), they also drain energy at runtime. We use Vivado synthesis tool to estimate *DiCA*’s power consumption on the Basys3 FPGA board. In this analysis, we consider *DiCA* configured for 128-Byte memory blocks. The MCU, including openMSP430 default set of peripherals and *DiCA*, consumes 89 mW of static power whereas *DiCA* hardware alone is reported<sup>2</sup> to draw less than 1 mW. Therefore, *DiCA* is responsible for less than 1.1%, of the device’s static power consumption.

The dynamic power drawn depends on the frequency of memory writes performed by the software. We consider an application that writes to all memory blocks in a loop to evaluate the worst case. In this scenario, the unmodified openMSP430 (along with peripherals) draws 234 mW of dynamic power. When equipped with *DiCA*, the dynamic power increases to 241 mW, representing a  $\approx 2.9\%$  increase.

We note that these estimates are based on the FPGA deployment and may vary once the MCU design is manufactured as an integrated circuit.

### D. Power Cycle Efficiency

To evaluate the efficacy of *DiCA*, we compare its performance with prior related work with respect to the amount of power cycles required to complete five distinct computations. Our benchmark considers five well-known algorithms: an AES encryption block cypher, a matrix multiplication, a SHA256 cryptographic hash function, a bit counting function, and a Depth-First Search (DFS) algorithm. To gauge how well our approach performs in comparison to other methods that trigger the check-point based on Voltage thresholds, we implemented an optimized version of Hibernus++ [6] (see related work in Section V). Our implementation of Hibernus++ triggers the check-point within the time required to copy VM (8 kB) before

<sup>2</sup>Vivado does not report energy consumption units under 1mW, treating such small values as negligible. Thus, 1mW is the upper bound for *DiCA* consumption because it is not possible to obtain precise measures below this value. In reality, however, *DiCA* may consume significantly less than 1 mW.

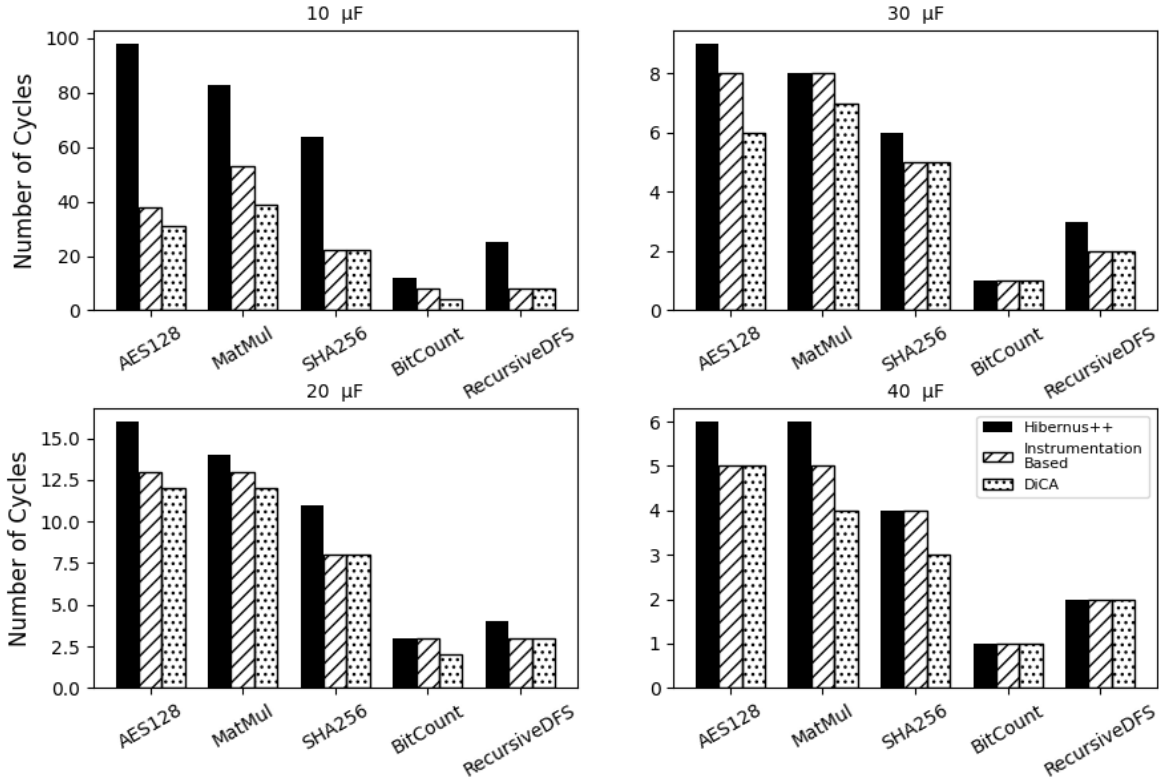


Fig. 7. Number of power cycles required to compute five application test-cases: *DiCA* vs. two related check-pointing methods

energy depletes. Furthermore, to compare our method with existing differential check-pointing approaches, we integrated a software instrumentation version of a memory modification monitor into Hibernus++, drawing upon prior work [3] as a reference.

In our experiment, we set up four distinct experimental configurations, each considering an MCU equipped with a capacitor of different capacitance, used to store harvested energy. By varying the capacitance values, our objective is to examine the influence of varying power supply decays on the performance of *DiCA*. The results are presented in Figure 7.

Across the four experimental capacitance configurations, we observed that *DiCA* consistently outperforms Hibernus++. It also outperforms the instrumentation-based differential check-pointing in most cases, requiring fewer execution cycles to run the benchmarked algorithms. The impact of this outcome becomes more pronounced in setups characterized by lower capacitance, where the rate of power supply decays is elevated. This occurs because, with more power cycles, the check-pointing routine tends to occupy a more substantial portion of the MCU execution time. As a result, the difference prior methods and *DiCA* is more pronounced.

It’s important to note that *DiCA* and other techniques from the literature are not mutually exclusive. *DiCA* is compatible and interoperable with a diverse array of existing strategies,

allowing for a potential combination of approaches to further enhance the efficiency and effectiveness of intermittent applications.

## V. RELATED WORK

### A. Traditional Check-Pointing

Differential (a.k.a. incremental) check-pointing [1], [21], [23], [26], [27] is widely studied for traditional computing systems. These systems are developed to operate on high-end devices and thus prioritize performance. Since they do not consider devices that operate on intermittent power supplies or have limited computational/storage resources, these techniques rely on tasks that low-end MCUs are not capable of performing, such as maintaining complex data structures [1], [20], or relatively expensive hardware support [14], [25], [26] to compute the differentials and store the check-point. Check-points in these systems also have different purposes, e.g., fault tolerance, load-balancing, and data concurrency across parallel servers.

### B. Check-Pointing in Intermittently Powered Devices

Energy-efficient check-pointing schemes are required for intermittently powered devices that harvest their own supply of energy. Compile-time techniques [11], [29] instrument the application source code with additional function calls that check the device’s current power. They are placed at specific

points in the program’s control flow, such as each backward edge of a loop or function returns. To properly create a check-point, the program context must be written in NVM, including all registers and data currently in use. Therefore frequent check-pointing can result in additional energy losses.

To reduce the overhead, runtime techniques [6], [7] determine the proper time for a check-point to be made while the device operates. The time to create a check-point is determined by employing a hardware-based interrupt to check the voltage supply periodically, and a check-pointing routine is executed once the voltage reaches a minimum threshold, reducing the number of check-points. In contrast with DiCA, prior runtime techniques save all registers and the entire VM at each check-point.

An alternative approach to further reduce the size of check-points is to save just the changes in a similar manner to differential check-points in traditional computing. However, computing the differentials in an energy-efficient and low-cost manner for intermittent computing devices is challenging. Some techniques use software to iterate over all addresses of VM [10] and compare them to the prior check-point. Others compare the hashes of memory blocks [4]. DICE [3] computes the differentials by maintaining an internal *modification record*, and the application software is instrumented with function calls that records differentials. DINO [22] extends C’s programming model by providing compiler-aided analysis to place *task-boundaries*, where check-points and data versioning takes place. As an alternative, new programming abstractions that operate on data in NVM at all times have been proposed. For instance, Chain [12] proposes a new program abstraction that guarantees that the state of self-contained tasks is preserved in NVM. Unlike DiCA, these techniques either depend on additional software, software modification via instrumentation, or new programming abstractions.

## VI. CONCLUSION

We proposed DiCA, a lightweight hardware/software co-design to support efficient differential check-pointing for intermittently powered devices. DiCA eliminates the need for application code modifications or instrumentation, simplifying and optimizing check-point generation. In addition, it implements a non-maskable interrupt to dynamically estimate optimal check-pointing times, thereby increasing the active period of applications during a power cycle. DiCA interrupt triggers a software routine that complements the hardware, efficiently copying modified memory segments from volatile to non-volatile memory. We implemented and evaluated DiCA with an FPGA deployment. DiCA open-source prototype is available at [24].

## ACKNOWLEDGEMENTS

We thank the ICCAD anonymous reviewers for their constructive comments and feedback. This work was supported by the National Science Foundation (Award #2245531) as well as a Meta Research Award (2022 Towards Trustworthy Products in AR, VR, and Smart Devices RFP).

## REFERENCES

- [1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, “Adaptive incremental checkpointing for massively parallel systems,” in *Proceedings of the 18th annual international conference on Supercomputing*, 2004, pp. 277–286.
- [2] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, “Efficient intermittent computing with differential checkpointing,” in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 70–81.
- [3] —, “Efficient intermittent computing with differential checkpointing,” in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 70–81.
- [4] F. A. Aouda, K. Marquet, and G. Salagnac, “Incremental checkpointing of program state to nvram for transiently-powered systems,” in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2014, pp. 1–4.
- [5] Atmel, “8-bit atmel microcontroller with 16/32/64/128k bytes in-system programmable flash,” Jan 2015.
- [6] 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.
- [7] 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 Systems Letters*, vol. 7, no. 1, pp. 15–18, 2014.
- [8] S. P. Beeby, M. J. Tudor, and N. White, “Energy harvesting vibration sources for microsystems applications,” *Measurement science and technology*, vol. 17, no. 12, p. R175, 2006.
- [9] S. Beeby and D. Zhu, “Vibration energy harvesting: fabrication, miniaturisation and applications,” *Smart Sensors, Actuators, and MEMS VII: and Cyber Physical Systems*, vol. 9517, p. 951703, 2015.
- [10] N. Bhatti, L. Mottola *et al.*, “Efficient state retention for transiently-powered embedded sensing,” in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, 2016, pp. 137–148.
- [11] N. A. Bhatti and L. Mottola, “Harvos: Efficient code instrumentation for transiently-powered embedded sensing,” in *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2017, pp. 209–219.
- [12] 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.
- [13] Digilent, “Basys 3 artix-7 FPGA trainer board: Recommended for introductory users,” 2018. [Online]. Available: <https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/>
- [14] B. Egger, Y. Cho, C. Jo, E. Park, and J. Lee, “Efficient checkpointing of live virtual machines,” *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 3041–3054, 2016.
- [15] O. Girard, “openMSP430,” 2009.
- [16] T. Instruments. Msp430 ultra-low-power sensing & measurement mcus. <http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html>.
- [17] H. Jayakumar, A. Raha, and V. Raghunathan, “Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 264–269.
- [18] T. J. Kazmierski and S. Beeby, “Energy harvesting systems,” *Principles, Modeling and Applications; Springer Science+ Business Media LLC: New York, NY, USA*, 2011.
- [19] A. Khaligh and O. C. Onar, *Energy harvesting: solar, wind, and ocean energy conversion systems*. CRC press, 2017.
- [20] W. Kim, C. Park, D. Kim, H. Park, Y.-r. Choi, A. Sussman, and B. Nam, “{ListDB}: Union of {Write-Ahead} logs and persistent {SkipLists} for incremental checkpointing on persistent memory,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 161–177.



- [21] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.
- [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] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 284–295.
- [24] A. J. Neto, A. Caulfield, and I. De Oliveira Nunes, "Dica open source prototype," <https://github.com/RIT-CHAOS-SEC/DiCA>.
- [25] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 75–86, 2011.
- [26] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.
- [27] B. Randell, P. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *ACM Computing Surveys (CSUR)*, vol. 10, no. 2, pp. 123–165, 1978.
- [28] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014, pp. 1–3.
- [29] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 159–170.
- [30] Y. K. Tan, *Sustainable energy harvesting technologies: Past, present and future*. BoD–Books on Demand, 2011.
- [31] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention." in *OSDI*, 2016, pp. 17–32.
- [32] Xilinx, "Vivado design suite user guide," 2017.
- [33] Xilinx Inc., "Artix-7 FPGA family," 2018. [Online]. Available: <https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html>