

MPU-based incremental checkpointing for transiently-powered systems

Gautier Berthou, Kevin Marquet, Tanguy Risset, Guillaume Salagnac

▶ **To cite this version:**

Gautier Berthou, Kevin Marquet, Tanguy Risset, Guillaume Salagnac. MPU-based incremental checkpointing for transiently-powered systems. DSD 2020 23rd Euromicro Conference on Digital System Design, Aug 2020, Kranj, France. pp.89-96, 10.1109/DSD51259.2020.00025 . hal-03116944

HAL Id: hal-03116944
https://inria.hal.science/hal-03116944

Submitted on 20 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MPU-based incremental checkpointing for transiently-powered systems

Gautier Berthou, Kevin Marquet, Tanguy Risset and Guillaume Salagnac
Univ. Lyon, Inria, INSA Lyon, CITI
email: firstname.lastname@insa-lyon.fr

***Abstract*—Transiently-powered devices are a class of small devices powered by energy harvesting. Because such devices are subject to frequent power outages, many recent works propose to checkpoint data residing in volatile RAM into non-volatile RAM. In this article, we propose a new incremental checkpointing mechanism supported by a common hardware component, namely a Memory Protection Unit (MPU). This mechanism leverages the hardware interrupts of the MPU: volatile RAM is read-only on boot and is progressively unlocked as soon as protection violations occur. The MPU interrupt handler is designed to flag the corresponding volatile RAM blocks as *dirty*, *i.e.*, modified. When a power outage is foreseen to be imminent, the software simply has to copy the dirty blocks from volatile RAM into the non-volatile RAM to ensure application progress over power outages. We validate our approach analytically and in cycle-accurate simulation, and we show that the proposed solution can be easily implemented on real hardware.**

***Index Terms*—Low-power, NVRAM, Checkpointing, Transiently-Powered Systems**

I. Introduction

Intermittent computing is a new computation paradigm for IoT sensors and ultra-low power devices where power outage is the norm rather than the exception. Sensors are not powered by a traditional battery but by a capacitor charged by harvesting energy from the environment. Program execution can be interrupted at any time, leaving the device with no power at all for an unknown duration.

In intermittent computing, program progress is ensured by the presence of some form of non-volatile memory. Emerging non-volatile RAMs [1] are much more efficient to frequently *checkpoint* program state (*i.e.*, save a consistent program state in case of future power loss) than classical Flash-based storage. In some applications where the energy source can only provide low power to the platform (*e.g.*, radio harvesting), reboots can be extremely frequent and it is crucial to optimize the energy consumed by the *shutdown/reboot* processes.

The research community has focused its attention on *smart checkpointing*, *i.e.*, performing checkpoints only when they are really necessary. Some other works propose to decrease the amount of RAM copied to NVRAM, either by considering the role of each region of the RAM [2] or by considering incremental checkpointing [3]. We propose a novel technique of incremental checkpointing which uses a hardware component that is present on most low-power micro-controllers today: the memory protection unit (MPU). An MPU can be configured to avoid costly checksums formerly used to check that a memory region has changed. This work proposes to investigate MPU write-access violation interrupts in order to flag volatile RAM regions as modified, to be saved into non-volatile RAM before the platform runs out of power.

After presenting our mechanism in Section III, we propose, in Section IV, an analytical model of the energy consumption of this mechanism and a comparison to a classical checkpointing mechanism. This analysis shows that even for short amounts of RAM (*e.g.*, 20 kB), MPU-based incremental checkpointing is much less energy consuming. We implemented our MPU-based incremental checkpointing mechanism and we ran it on a cycle-accurate simulator of the FRAM-based MSP430FR5739 micro-controller. The cycle-accurate simulations, presented in Section V, confirm the interest of this new technique for transiently-powered devices checkpointing.

II. Related Work: Transiently-Powered Systems and Checkpointing

A. Transiently-Powered Devices

Following the recent development of IoT [4], [5], it appears that low-power consumption is a crucial issue for IoT devices. Recent advances in low-power radio transmission [6], [7] allow high form factor reduction for sensors [8], but using battery remains an important problem because of battery charge process and battery size.

Harvesting technologies have evolved to extend the energy source to other power sources than light [9], [10], [11], [12]. These new energy harvesting technologies enable a possible scenario where sensors would survive for decades without maintenance, simply by *harvesting* energy from the environment. This gave rise to the notion of *Transiently-Powered Systems* (TPS) [13] and their computing paradigm: intermittent computing. Transiently-powered systems are tiny devices powered by energy harvesting, supporting frequent unexpected power losses and ensuring progress by saving program state to a non-volatile storage element [14], [15], [16], [17].

The first studies on intermittent computing TPSs used Flash memory as non-volatile storage [13], but most recent works rely on new non-volatile RAM (NVRAM) technologies [15], [18], [16], [19], [20], [17], [21], [22]. The use of NVRAM offers many possibilities: it can be used to save and restore program state much more efficiently than Flash, but can also be used as regular RAM by the program. Various memory hierarchies can be designed. Storing data in NVRAM makes each access slower and/or more energy-expensive and brings

new consistency problems [23], but reduces the time for checkpointing. Many intermittent computing systems use *hybrid-memory* models (volatile and non-volatile RAM).

For several years, the only commercial CPUs using NVRAM and RAM were those of the MSP430FR family from Texas Instruments which use FRAM technology. Hence many of the aforementioned intermittent computing experiments leveraged these platforms. For instance, the MSP430FR5739 CPU contains 16 kB of NVRAM (FRAM) and 1 kB of SRAM.

B. Checkpointing Mechanisms

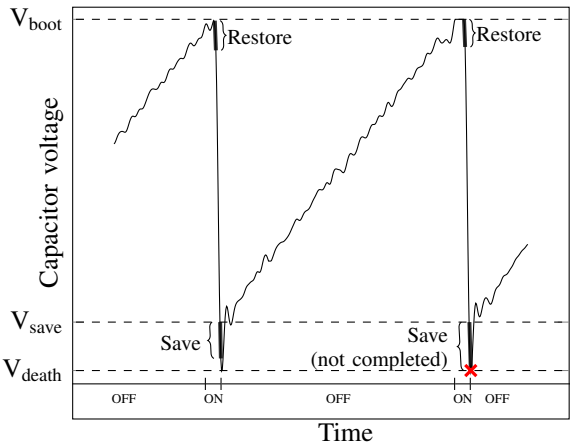


Fig. 1: Ideal checkpointing On/Off cycles of a platform supplied by energy harvesting. Checkpoint operations are performed when voltage drops below V_{save}. Checkpoint can succeed or fail, application and devices are always restored in a consistent state but no progress is made if checkpoint fails.

Checkpointing consists in saving the program state and restoring it, usually after a crash occurrence. We are studying checkpointing in the context of intermittent computing for low-power embedded devices. This context is quite different from the usual context of checkpointing, *i.e.*, distributed systems and high performance computing. In TPS, checkpointing must solve three problems: program state persistence (proposed by all works), peripheral state persistence [20], [21], [22], [24] and persistent-time keeping [25]. In this paper we focus on the first problem: saving the program state into NVRAM and restoring it, with a minimal energy cost. Fig. 1 illustrates an ideal situation where checkpointing is done solely before power outage[1]. In general, it might be difficult to predict exactly when checkpointing should be performed. The copy from RAM to NVRAM is, on most devices, performed by a DMA peripheral in efficient burst copies of memory blocks.

Mementos [14], running on Intel WISP [26] proposes to periodically interrupt the application and measures the remaining energy level. If the energy level is below a certain threshold, Mementos saves the CPU registers and copies the contents of RAM to flash memory, making checkpointing transparent to the application program. Hibernus [15] introduced a hardware device to trigger the checkpoint operation only when needed (in order to get close to the ideal situation depicted in Fig. 1).

Some works proposed static insertion of checkpoints, [18], or dynamic insertion [17], [27], [16]. The IBIS tools [28] ensure memory consistency when computing directly in non-volatile RAM.

Several works have proposed solutions for improving checkpointing efficiency, *i.e.*, minimizing energy spent for checkpointing. Aït-Aoudia *et al.* [3] propose *incremental checkpointing* scheme to reduce the number of NVRAM writes as much as possible. Bhatti and Mottola [29] improved this proposition by distinguishing RAM regions (*stack*, *heap*, *etc.*) and other [30], [31] proposed further improvements. These works define incremental checkpointing. In order to selectively save memory regions from volatile to non-volatile memory, the entire volatile RAM must be mirrored in the checkpointing image at least once (or more in case of double-buffering for instance), which is always the case since a checkpoint image must enable the system to repopulate its RAM, amongst others, upon restoration process.

Few works leverage hardware for checkpointing. Bartling *et al.* [2] propose to design a non-volatile micro-controller which automatically saves all CPU and peripheral registers to FRAM, upon detecting a power loss.

In this article, we propose a new checkpointing mechanism that relies on a very common hardware component present in every low-power device for isolation purposes: the Memory Protection Unit (MPU).

III. A New MPU-based Checkpointing Mechanism

Our proposal is to use an MPU to optimize the quantity of data saved into NVRAM, *i.e.*, to get as close as possible to the ideal case where only data that have been modified during the life-cycle are saved into NVRAM. We first recall the principle of MPUs which are provided in many embedded devices, then we explain our proposal of incremental checkpointing mechanism and we review the parameters that might change from one MPU to another.

A. Memory Protection Units

Ultra low-power embedded systems do not have hardware support for memory virtualization through Memory Management Units (MMU). An MMU is energy-expensive, mainly because of the presence of the Translation Lookaside Buffers, made of associative memory. However, many embedded systems include hardware support for memory isolation thanks to MPUs which enable to *manage access rights* to some memory region.

The characteristics of an MPU substantially vary from one platform to another but their goal is the same: guarantee memory integrity and fire interrupts upon access violation. Table I gives the characteristics of MPUs present on the MSP430FR5739 and ARM Cortex-M platforms. The criterion "Tunable for Checkpointing" indicates whether our technique will be easy to implement on that MPU, as precised hereafter.

[1] We refer to the period within which the platform is continuously powered-on and computes normally as *life-cycle*.

TABLE I: Characteristics of the MPUs from two low-power architectures.

	MSP430FR5739	ARM Cortex M3&M4
Working range	NVRAM range	Entire memory
Region count	3	8 with 8 subregions per region
Region size	Customizable	Customizable One region at a time
Tunable	No	Yes

```
void application_main(void)
{
  compute_1();
  *((int*) 0x668) = 42;
  compute_2();
  *((int*) 0x01068) = 24;
  compute_3();
}
```

Fig. 2: C code serving the explanation.

B. Incremental Checkpointing Mechanism

The proposal of this paper is to use MPU as hardware support for incremental checkpointing. But we do not regularly compute checksums on memory regions as in previous works [3]. The idea is to keep track of whether a region has been modified since last checkpoint or not. Let us note D the set of *dirty* regions, *i.e.*, regions that have been modified since the last checkpoint.

When the platform starts executing, D is empty ($D = \emptyset$) and all regions of volatile RAM are write-protected by the MPU. When a write occurs in the region R_i of RAM, the MPU triggers an interrupt. The interrupt handler marks the region dirty: $D := D \cup \{R_i\}$ and unprotects the region. The interrupt handler then returns to the faulting instruction, *i.e.*, the instruction that caused the write violation access. Now that the region is unprotected, the write access can complete without further interrupt and the execution resumes as usual. In practice, D may be implemented as a bitfield.

The code sequence in Fig. 2 illustrates this mechanism with a RAM of 8 kB and an MPU capable of protecting 8 different regions R_1 to R_8:

1) Initialization and execution of the system. $D = \emptyset$.
2) The program performs a write in RAM at address 0x668 (address in region R_2). This triggers an MPU interrupt.
3) The interrupt handler is executed. $D = \{R_2\}$ and the MPU is configured to stop protecting R_2.
4) The failed write to address 0x668 is now executed correctly. This is possible only if the MPU is "Tunable " for checkpointing. Indeed, when an interrupt is raised by the MPU by instruction i, the execution flow returns – after interrupt handler execution – to the instruction *following* instruction i. Here, we need to execute i again. This can be parameterized on an ARM MPU as indicated in Table I but not on an MSP430FR MPU.
5) The program continues its normal execution after the write.
6) Then the program writes in RAM at address 0x1068 (in region R_5). Again, this triggers an MPU interrupt.
7) The interrupt handler is executed. $D = \{R_2, R_5\}$ and the MPU is configured to stop protecting R_5.
8) The write at address 0x1068 is actually executed.
9) The program continues its execution and at some point, the energy subsystem triggers an interrupt leading the OS to checkpoint volatile memory: only regions 2 and 5 are copied to NVRAM.

This technique requires the MPU to be able to protect the RAM. The energy gain, compared to saving the entire RAM, will depend on characteristics of the MPU and of the executed applications. In the following, we give details on all these characteristics, and we explore their impact in the next section.

IV. Analytical Analysis of Checkpointing Performance

In this section, we focus on the energy required to checkpoint memory from volatile RAM to non-volatile RAM. The energy consumed by this process depends on several parameters. The model depicted in this section, as well as the results that come out of it, may be used as a base for design space exploration, in terms of hardware specifications and/or software specifications.

A. Modeling MPU-based Incremental Checkpoint

The performance of our proposal heavily depends on some parameters, listed in Table II. The amount of RAM used S_{words} is important because the bigger the memory is, the more vital it becomes to save energy by selectively checkpointing fractions of memory to NVRAM. The performance of the DMA f_{DMA} directly impacts the time needed to perform a checkpoint.

We call α the average dirtiness ratio, *i.e.*, the average proportion of regions that have been modified since last checkpoint when a new checkpoint arrives. This parameter α is important: a small value gives our proposal better results. In many transiently-powered systems, little energy is available between consecutive checkpoints, allowing the execution of a few thousands or millions of instructions each time, hence α is indeed expected to be low. The amount of regions, the application behavior and the frequency of power loss (or equivalently the duration of life cycle), influence also the performance of our proposal.

The amount of regions N_{reg} that the MPU can handle is also crucial: if this number is too high, the platform will spend a lot of time handling interrupts. But on the other hand, with a small number of regions, the checkpointing is less incremental and closer to a full copy of the RAM contents.

Our proposal relies on a standard micro-controller equipped with an MPU. Although the MPU is a hardware component, it is driven by software, which means that there is some time overhead due to running instructions. This overhead is specific to incremental checkpointing. There are two sources

TABLE II: Model parameters and their default values (used in Section V).

Symbol	Description	Unit	Typical value
S_{words}	Amount of RAM used by the application	Word	2^{13}
f_{DMA}	DMA bandwidth	Word/second	8×10^6
P_{plat}	Power drawn by the whole platform, without CPU, DMA nor MPU	Watt	1.65×10^{-2}
P_{DMA}	Power drawn by the (active) DMA	Watt	ϵ
P_{MPU}	Power drawn by the (active) MPU	Watt	ϵ
P_{CPU}	Power drawn by the CPU (in active mode)	Watt	3.96×10^{-3}
α	Average ratio of dirty regions during one life-cycle	-	0.1
N_{reg}	Number of regions handled by the MPU	-	16
$t_{overhead}$	Time to check if a region must be copied	Second	3×10^{-6}
t_{int}	Execution time of the MPU interrupt handler	Second	5×10^{-6}
$E_{critical}$	Average amount of energy wasted due to re-executing code if the MPU interrupt occurred during a critical section	Joule	ϵ

of software-related time overhead. First, this mechanism is interrupt-based which introduces an overhead, named here t_{int}, that is the time to handle the MPU interrupt. The interrupt handler must simply flag the concerned memory region as dirty, and unlock that region to allow further modifications from the software until the memory is saved in the checkpointing process. Second, during the checkpointing process, the software must check every region dirtiness flag to determine whether they must be copied or not, this overhead is called $t_{overhead}$. Both overheads are expected to be small, but not negligible, within the order of a few microseconds for each.

This model also needs insight about some electronics aspects of the platform. We distinguish four parts: the micro-controller itself that consumes P_{CPU}; the DMA that consumes P_{DMA} apart from the micro-controller; the MPU that consumes P_{MPU}; and the rest of the platform, including peripherals, that consumes P_{plat}. The power consumption of the MPU, P_{MPU}, is only accounted in the incremental checkpointing since the full copy does not need the MPU and thus it can be turned off. The different power consumptions are platform-dependent and furthermore, P_{plat} also depends on the application since the amount and the nature of enabled peripherals depend on the state of the application at a given point in time. When yielding our results for the analytical part, we chose to arbitrarily set P_{plat} to 16.5 mW which corresponds to a consumption of 5 mA under a 3.3 V supply; P_{DMA} and P_{MPU} to be negligible; and P_{CPU} to be 3.96 mW, which corresponds to 1.2 mA, the consumption of the active mode of the MSP430FR5739 – in which words are 2 bytes long – as we measured it.

From these parameters, we describe below the equations that analytically compute the energy spent using our incremental checkpointing solution (E_{inc}) and the energy spent doing a copy of all RAM (E_{copy}).

As checkpointing is done using a DMA, *i.e.*, without using the CPU, the energy required to checkpoint the whole RAM (of size S_{words}) is simply computed with equation (1):

$$E_{copy} = \frac{S_{words}}{f_{DMA}} \times (P_{DMA} + P_{plat}) \quad (1)$$

The power consumption of the MPU is not accounted in Equation (1) since E_{copy} represents the non-incremental checkpointing.

To compute the energy used by our mechanism, we now introduce the necessary terms. To save a single region using DMA (CPU off), the required energy is expressed as:

$$E_{region} = \frac{R_{words}}{f_{DMA}} \times (P_{DMA} + P_{MPU} + P_{plat})$$

with $R_{words} = \lceil \frac{S_{words}}{N_{reg}} \rceil$ the number of words in a region.

The energy consumed by MPU interrupt handling corresponds to the energy spent by the MPU interrupt handler plus the energy needed to re-run a portion of code, if for instance the interrupt occurred during an atomic section (*e.g.*, a "syscall", see [20], [21], [22] for details on peripheral handling in TPS). This energy is abstracted by $E_{critical}$ which is averaged over all the MPU interrupts. It is important to model this energy because in some cases, frequent interrupts might prevent the application from efficiently making progress. In the analysis presented in Section V, we have not taken this parameter into account ($E_{critical} = \epsilon$ in Table II), but in the cycle-accurate simulation in Section V-A, it is simulated. Hence the energy of dirtiness detection is expressed as:

$$E_{detect} = t_{int} \times (P_{CPU} + P_{MPU} + P_{plat}) + E_{critical}$$

The energy to checkpoint only dirty regions corresponds to:

$$E_{dirty} = N_{dirty} \times E_{region} + t_{overhead} \times (P_{CPU} + P_{MPU} + P_{plat})$$

with $N_{dirty} = \lceil \alpha \times N_{reg} \rceil$ the average amount of dirty regions per checkpoint.

Finally, the energy dedicated to incremental checkpointing during an entire life-cycle energy is given by:

$$E_{inc} = E_{dirty} + N_{dirty} \times E_{detect} \quad (2)$$

B. Comparison with Classical Complete RAM Copy

In this section, we evaluate the benefits of our proposal using the analytical estimation provided in previous section. The parameters that are not indicated in the following figures have default values mentioned in Table II. These values have been obtained by informal measurement on the MSP430FR5739.

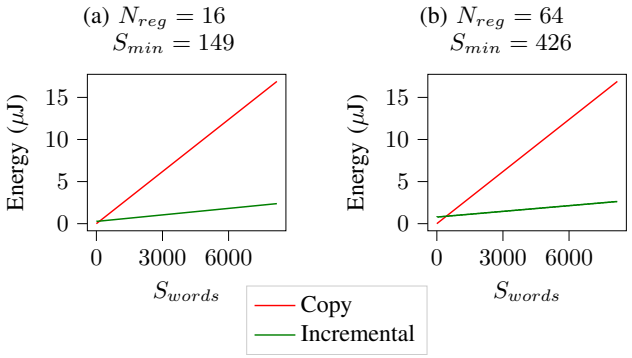


Fig. 3: Energy consumption of checkpointing mechanism with respect to size of RAM for different values of N_{reg}.

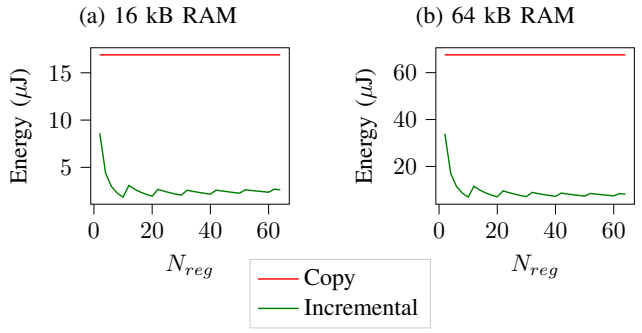


Fig. 4: Energy consumption of checkpointing mechanism with respect to the number of MPU regions for different RAM sizes.

The precision of these values is not very important, what is important is the aspect of the evolution of energy consumption when some parameter (*e.g.*, the size of the RAM) changes.

a) Impact of RAM size: Fig. 3 shows the energy consumption with respect to the RAM size for different values for N_{reg} (results are similar for $\alpha = 0.3$). Our intuition is confirmed: when α is small, our mechanism is better than a copy of the entire used RAM. This is normal as our mechanism avoids the copy of all not-dirty regions. Another intuition is confirmed, for the same reason: the larger the amount of RAM used is, the better our mechanism is.

The third observation we make from these curves is that our mechanism is not always more efficient with an increase of the number of regions. We detail this phenomenon in the next section. The energy curves have many similarities when making parameters vary. Hence, we can define S_{min} such that $\forall S_{words} > S_{min}, E_{inc}(S_{words}) < E_{copy}(S_{words})$. In other terms, S_{min} is the minimal amount of RAM words to make the incremental checkpointing worth using in comparison to the classical full copy. Note that S_{min} actually depends on the other parameters as listed in Table II. The values of S_{min} are fairly low, which implies that the MPU-based approach consumes less energy than the traditional copy for the needs of realistic applications. However, it is necessary to mitigate these results by taking into account the influence of other parameters; this is done in paragraph *c)* below.

b) Impact of the number of regions: If N_{reg} is too high, the platform will spend a lot of time handling MPU interrupts. But if N_{reg} is too small, the checkpointing is less incremental and resembles more the classical full copy with detrimental overhead. This is illustrated by Fig. 4: for the considered amounts of RAM, the optimal number of regions is under 10.

c) Impact of other parameters: It is not possible, in figures presented above, to see the influence of two important parameters. The results presented above were based on the typical values mentioned in Table II. But Fig. 5 illustrates the influence of other parameters.

A high α decreases the efficiency of incremental checkpointing (see Fig. 5a). Indeed, when α grows towards 1, the system has to copy an amount of data that becomes closer to the amount of data required by the classical full copy. In that case, the overhead due to MPU interrupt handling makes it a challenge, for the incremental checkpointing, to keep being beneficial. However, even with stable energy harvesters that are able to run the platform longer and thus to achieve a greater α such as solar-based harvesters, the incremental checkpointing requires only a few thousands of words to show better performance.

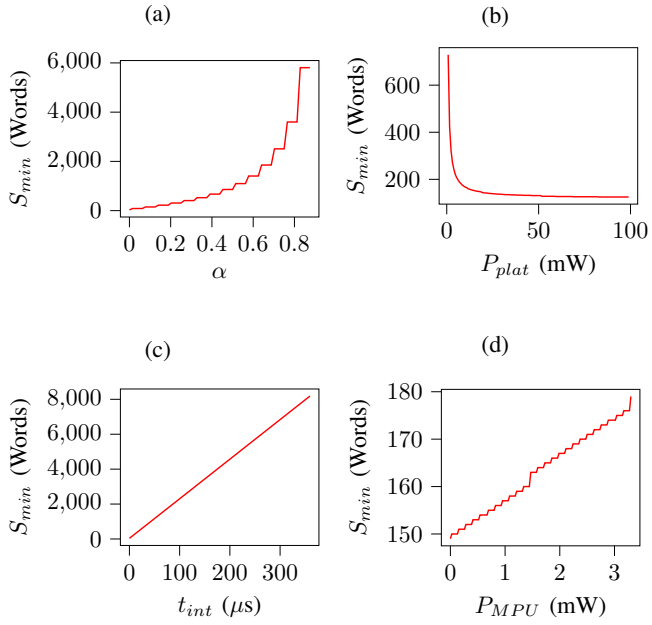


Fig. 5: Impact of (a) α, (b) P_{plat}, (c) t_{int} and (d) P_{MPU} on the minimal amount of RAM words to make the incremental checkpointing worthwhile (S_{min}).

The lesser P_{plat} is, the less efficient the incremental checkpointing is (see Fig. 5b).

The execution time of the MPU interrupt handler is crucial (see Fig. 5c). A longer execution time means a higher energy budget allocated for checkpointing, since the micro-controller

and the peripherals are consuming energy in the meantime.

The electronic properties of the MPU can differ from a micro-controller to another and its configuration may have an impact on its consumption. This motivates the need to study the influence of P_{MPU} on the performance of the incremental checkpointing. Fig. 5d shows that, if the MPU consumes more, S_{min} increases accordingly, meaning that there must be a greater amount of memory to be checkpointed in order to keep the incremental checkpointing beneficial. However, S_{min} values are still low, even when P_{MPU} is high. Thus, even a complex, power-consuming MPU would not hinder the benefits of the incremental checkpointing.

Some extreme values were shown on purpose in Fig. 5, in order to show the low yet realistic requirements that make this proposal efficient. For instance, t_{int} is not expected to be greater than a few microseconds, however Fig. 5c shows that greater values would require a greater S_{min} but still realistic from the application's perspective. Another example is when α is very high in Fig. 5a. In practice, the application is not expected to modify most of the memory contents before a power outage occurs, yet greater values of α would require a realistic value of S_{min}. In any case, S_{min} is always low, making the incremental checkpointing often worth using over a classical full copy.

V. Cycle-Accurate Simulation of MPU-based Incremental Checkpointing

In this section, we validate our analytical results thanks to a simulation platform. One limitation of the analytical approach is that α is always the same whereas it can change from one life-cycle to another. In cycle-accurate simulation, as in reality, α changes every life-cycle.

A. Simulation Platform

We implemented a simulator [32] of an MSP430FR5739 board on top of ArchC [33] but the MPU is modified to handle up to 16 regions instead of 3, as well as to cover volatile RAM address range instead of solely the non-volatile RAM address range. The memory capacity of the platform was also virtually modified in order to propose 20 kB volatile RAM and 40 kB non-volatile RAM.

ArchC is a language for CPU architecture and Instruction Set Architecture description. It aims at generating cycle-accurate SystemC code for simulation purposes. We wrote an ArchC model of the MSP430X instruction set, in order to run the binary images compiled for MSP430FR5739 without any modification.

While the genuine software part is run in a cycle-accurate fashion, the driver part is run symbolically. Instead of actually running every single instruction of a driver call, the whole routine is bypassed and only its functional effects are simulated: time advances, some amount of energy is taken from the energy budget and the peripherals states are updated. The time duration and energy consumption are directly taken from measurements on a real platform.

The simulator is able to simulate continuous supply as well as energy harvester with a power manager that stores energy into a capacitor and powers the device under test through a voltage regulator. In the latter case, when the capacitor voltage drops below a certain threshold, the simulator generates an interrupt and calls the software-defined interrupt handler as it would be done on the real platform. Then, when the capacitor voltage drops further to a lower threshold, the simulator virtually switches the device off, refills the capacitor and restarts the device by running the reset entry of the kernel interrupt vector.

The MPU of the MSP430FR5739 is not suited for dirtiness detection for several reasons: only the non-volatile RAM is managed by the MPU and the interrupt system is not capable of returning to the faulting instruction (which is needed for our method, as explained in the scenario, p. 3). However, we chose to benefit from the simulation environment to implement a very simple, realistic MPU, with up to 16 regions, able to cover the whole memory (volatile and non-volatile as well), and able to re-execute a faulting instruction. Such an MPU does not correspond to the actual one that is proposed by the MSP430FR5739 micro-controller, but it resembles the ones that can be found on ARM micro-controllers.

B. Benchmark Applications

The simulation is tested against home-made benchmarks because no TPS benchmark using peripherals exists yet. Our benchmark consists in the following applications:

- **Quicksort** initializes and sorts an array of pseudo-random data. The array size is statically defined.
- **RSA** initializes and performs an RSA encryption onto an array of data. The array size is statically defined.
- **Complete-WSN** uses accelerometer, temperature sensor and radio. It senses acceleration and temperature several times and sends the data over radio as small packets. The radio is sleeping while sensing data and the accelerometer is always on. The packet size is statically defined.

All applications are declined into several instances, one instance per combination of S_{words} and N_{reg} values if the MPU is enabled, one instance per value of S_{words} for the full copy version without MPU. Each instance has its own memory access signature, that impacts α. However, α is also impacted by the available energy within a single life-cycle, *i.e.*, the weaker the life-cycle, the lower is expected to be α. In all the results presented here, the available life-cycle energy is 120 µJ.

C. Checkpointing Mechanism and System Layer

The aforementioned applications run on top of Sytare [20]. Sytare grants application *and* peripheral state persistence across power losses. Within the context of this work, we only changed the mechanism that makes the volatile RAM persist in NVRAM, by integrating MPU information and selectively copying from volatile RAM to NVRAM based on that information. The results shown for the full copy correspond to the original version of Sytare.

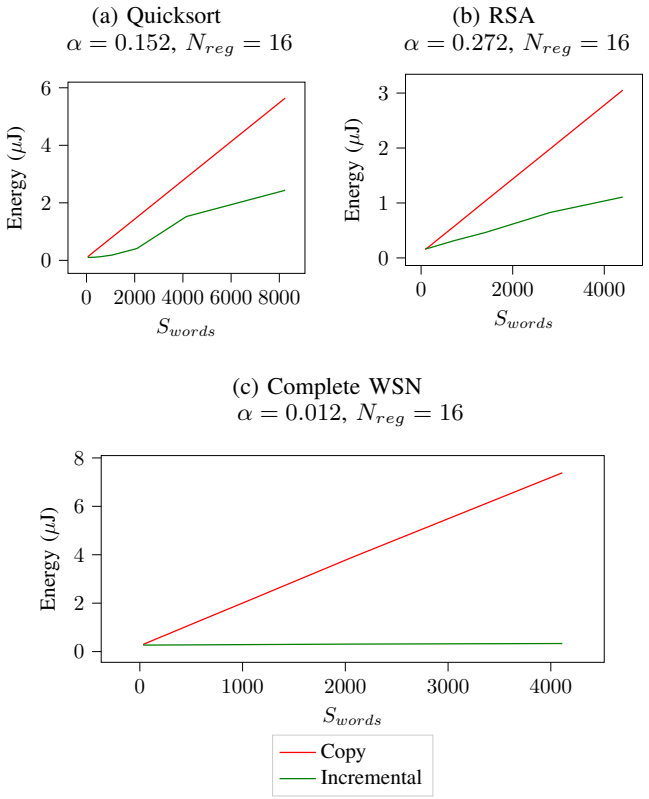


Fig. 6: Energy consumption of checkpointing with respect to size of RAM for different applications, when the energy budget is 120 µJ.

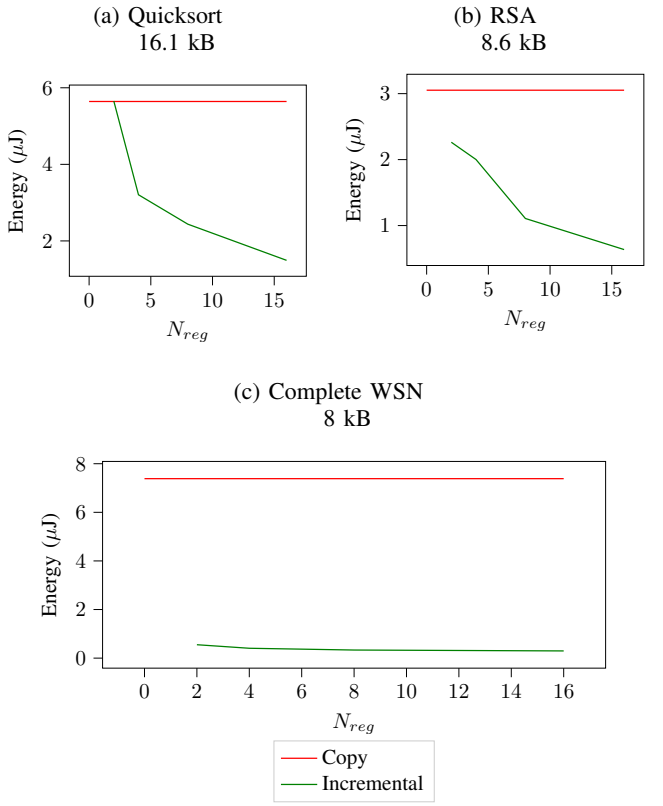


Fig. 7: Energy consumption of checkpointing with respect to the amount of MPU regions for different applications, when the energy budget is 120 µJ.

D. Validation of Analytical Results

We now validate our analytical model against our simulation platform. The α values are computed as the average α of the first 32 life-cycles for all executions involved in a given graph.

a) Impact of RAM size: Fig. 6 shows how the needs in RAM impact the energy required for checkpointing, for the applications defined above. The results are similar to the anticipated values given by the model.

The major discrepancy is the fact that, in practice, α is not a constant. Indeed, depending on where the application resumes and on the energy budget of the next life-cycle, the application does not require the same regions, nor the same amount of regions, to be checkpointed for the next life-cycle. The results show that, when the application uses more than a hundred RAM words (S_{min} is around 100), it is always better to use the incremental checkpointing rather than full RAM copy.

b) Impact of the number of regions: Fig. 7 shows how the amount of MPU regions impacts the energy required for checkpointing, for the applications defined above. Analytical results showed that incremental checkpointing is always better than a full RAM copy. To this extent, simulation results are alike. The visible steps of Fig. 4 do not appear in Fig. 7 as we made the amount of regions vary at fine grain in Fig. 4, whereas we made it vary along powers of two in Fig. 7.

E. Discussion

Our proposal is inspired from incremental checkpointing mechanism [3], but it uses hardware support rather than computations to detect used regions. Even with hardware support, we only benefit from incremental checkpointing if the size of RAM is big enough. We showed in this paper that the conditions under which incremental checkpointing is better than an unselective copy are realistic today, and they will be even more realistic in the future as the amount of embedded RAM is likely to increase.

We have performed analytical exploration (Section IV) and more precise validation by cycle-accurate simulation. Our simulations validate our analytical analysis, even though there are some small discrepancies with the results, due to the simplifications made in the analytical model.

One slight difference between the analytical model and our simulation should be precised here. In our analytical model, we only consider the amount of RAM used, whereas in simulation, the total amount of RAM and the amount of RAM used are two distinct quantities. However, we chose not to complicate the analytical model.

Also, it may be noticed that the duration of a life-cycle, which depends on the energy available in the capacitor of the TPS, is not important to appreciate the benefits of incremental checkpointing. It only has an impact on the value of α: with

shorter life-cycles, α will be lower as fewer write instructions will be executed. With our proposal, we rather target small systems, that cannot be powered by solar panels, nor embed a big capacitor, which is the case for many applications. It would be interesting to study extensively the value of α with regard to the power source, the size of the capacitor and the application needs, but it is outside the scope of this paper.

VI. Conclusion

We have explained how incremental checkpointing can be made extremely efficient by using an MPU, which is a very common hardware component in recent ultra-low power devices. We have analytically explored the conditions to achieve this efficiency and we have validated, in simulation, the analytical results. Under reasonable assumptions, our mechanism is more efficient than merely copying the RAM used by an application, even when using a DMA. Our perspectives are to compare our approach to a hardware component that would be specifically designed to optimize checkpointing, to tackle the software overheads.

Acknowledgment

This work is partially supported by Inria (IPL ZEP).

References

[1] T. Shull, J. Huang, and J. Torrellas, "Defining a high-level programming model for emerging NVRAM technologies," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, 2018.

[2] S. Bartling, S. Khanna, M. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams, "An 8MHz 75uA/MHz zero-leakage non-volatile logic-based Cortex-M0 MCU SoC exhibiting 100-percent digital state retention at VDD=0V with ¡400ns wakeup and sleep transitions," in *Intl. Solid-State Circuits Conf. (ISSCC)*, 2013.

[3] F. Aït-Aoudia, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to NVRAM for transiently-powered systems," in *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2014.

[4] C. Morais, D. Sadok, and J. Kelner, "An IoT sensor and scenario survey for data researchers," *Journal of the Brazilian Computer Society*, 2019.

[5] J. R. Shah and B. Mishra, "Iot enabled environmental monitoring system for smart cities," *2016 International Conference on Internet of Things and Applications (IOTA)*, 2016.

[6] P. Mercier and A. Chandrakasan, *Ultra-Low-Power Short-Range Radios*. Springer, Cham, 2015.

[7] A. Kozłowski and J. Sosnowski, "Energy efficiency trade-off between duty-cycling and wake-up radio techniques in IoT networks," *Wireless Personal Communications*, 2019.

[8] V. Iyer, R. Nandakumar, A. Wang, S. B. Fuller, and S. Gollakota, "Living iot: A flying wireless platform on live insects," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[9] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola, "Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences," *ACM Trans. Sen. Netw.*, 2016.

[10] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-based battery-free programmable sensing platform," *IEEE Transactions on Instrumentation and Measurement*, 2008.

[11] Y. Lee, S. Bang, I. Lee, Y. Kim, G. Kim, M. H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw, "A Modular 1 mm^3 Die-Stacked Sensing Platform With Low Power I2C Inter-Die Communication and Multi-Modal Energy Harvesting," *IEEE Journal of Solid-State Circuits*, 2013.

[12] A. Colin, E. Ruppel, and B. Lucia, "A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[13] 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*, 2014.

[14] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on RFID-scale devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[15] 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*, 2015.

[16] J. van der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016.

[17] K. Maeng, A. Colin, and B. Lucia, "Alpaca: intermittent execution without checkpoints," *PACMPL*, 2017.

[18] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015.

[19] M. Hicks, "Clank: Architectural support for intermittent computation," *SIGARCH Comput. Archit. News*, 2017.

[20] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A lightweight kernel for nvram-based transiently-powered systems," *IEEE Transactions on Computers*, 2019.

[21] A. R. Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "RESTOP: retaining external peripheral state in intermittently-powered sensor systems," *Sensors*, 2018.

[22] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui, "Intermittent asynchronous peripheral operations," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019.

[23] B. Ransford and B. Lucia, "Nonvolatile Memory is a Broken Time Machine," in *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014.

[24] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, 2019.

[25] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017.

[26] M. Buettner, R. Prasad, A. Sample, D. Yeager, B. Greenstein, J. R. Smith, and D. Wetherall, "RFID sensor networks with the Intel WISP," in *Conference on Embedded Network Sensor Systems (SenSys)*, 2008.

[27] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, 2018.

[28] M. Surbatovich, L. Jia, and B. Lucia, "I/O dependent idempotence bugs in intermittent systems," *PACMPL*, 2019.

[29] N. Bhatti and L. Mottola, "Efficient State Retention for Transiently-powered Embedded Sensing," in *International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2016.

[30] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices," *ACM Trans. Embed. Comput. Syst.*, 2017.

[31] S. Ahmed, M. H. Alizai, J. H. Siddiqui, N. A. Bhatti, and L. Mottola, "Towards smaller checkpoints for better intermittent computing: Poster abstract," in *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2018.

[32] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "Accurate power consumption evaluation for peripherals in ultra low-power embedded systems," in *2020 Global Internet of Things Summit (GIoTS)*, 2020.

[33] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "Archc: a systemc-based architecture description language," in *16th Symposium on Computer Architecture and High Performance Computing*, 2004.