Incremental checkpointing of program state to NVRAM for transiently-powered systems

Fayçal Ait Aoudia, Kevin Marquet, Guillaume Salagnac

Incremental checkpointing of program state to NVRAM for transiently-powered systems

Fayçal Ait Aoudia, Kevin Marquet, Guillaume Salagnac
Université de Lyon, INRIA
INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, France
faycal.ait-aoudia@insa-lyon.fr kevin.marquet@insa-lyon.fr guillaume.salagnac@insa-lyon.fr

***Abstract*—As technology improves, it becomes possible to design autonomous, energy-harvesting networked embedded systems, a key building block for the Internet of Things. However, running from harvested energy means frequent and unpredictable power failures. Programming such *Transiently Powered Computers* will remain an arduous task for the software developer, unless some OS support abstracts energy management away from application design. Various approaches were proposed to address this problem. We focus on *checkpointing*, i.e. saving and restoring program state to and from non-volatile memory. In this paper, we propose an incremental checkpointing scheme which aims at minimizing the amount of data written to non-volatile memory, while keeping the execution overhead as low as possible.**

I. Introduction

Many challenges remain to be solved before the Internet of Things (IoT) becomes a reality. Some of these challenges are already faced by today's resource-constrained embedded systems, like smart cards or wireless sensor networks. These platforms have very little computing power, with a slow CPU and a few kilobytes of memory. They must run on limited energy budgets, and at the same time meet long-term lifetime requirements.

But for these systems to become pervasive in our environment, additional difficulties are to be addressed. One is that of energy management. "Smart dust" [1] nodes will be embedded in hard-to-reach places (*e.g.* for smart building applications, in the ceilings) and will have to run autonomously for many years. Even assuming very low power operation, fitting each node with a large-enough battery is not practical. On the contrary, energy harvesting techniques are very promising because they more naturally fit the low power / long lifetime constraints of IoT applications. Gathering energy from the environment has two drawbacks, though: first, the output power level is very low, so harvested energy must be buffered before use, *e.g.* in a battery or a super-capacitor. Second, the environment is by nature unpredictable, so the system must be able to somehow react to variations in energy availability. These systems are called [2] *transiently powered computers* (TPCs). Some systems (*e.g* smart cards) have similar consumption constraints and applications running on them are therefore very limited (*e.g* non-contact payment). More recently, several approaches have been proposed by the research community to lift these constraints, enabling execution of general programs despite frequent power failures. One such proposal is *Mementos* [2] which saves the program's state in Flash memory before power loss and restores it when energy is available.

Contrary to Mementos, we advocate for the use of *non-volatile RAM* (NVRAM). Several technologies of NVRAM exist and some of them are ideal for use in TPCs because 1) they do not consume much energy when accessed 2) they retain data when not powered 3) they are much faster than Flash memory. For instance, phase-change memory (PCM) has a read/write latencies around 12/100 ns, compared to 15/1000 ns for Flash NOR memory [3]. In this work, we consider hybrid platforms that embed both NVRAM and regular, volatile, SRAM. Keeping main memory in volatile RAM is needed for two reasons. First, existing NVRAM technologies are still time consuming when write-accessed : compare the 12/100 ns access times of PCM to the 2/2 ns of SRAM [3]. Second, a software bug or a hardware failure can lead to a system crash. If all memory is persistent, then the system is likely to reboot in an error state.

The problem we address in this paper is therefore to efficiently save program state to NVRAM before power loss, and then restore it when energy is available. The solution we describe basically consists in :

1) Saving a copy of the program state (a *checkpoint*) before power failure. Our solution is incremental: only the parts of the program state that have been modified since the last checkpoint are copied in NVRAM ;
2) Doing so *at the right time*, *i.e.* just before loss of power, in order to avoid useless and energy consuming checkpointing operations.

II. Related work

Different approaches were suggested by the research community to allow executing of non-trivial programs on transiently powered computers. In *Dewdrop* [4], Buettner *et al.* observe that the usual policy of booting as soon as possible is actually suboptimal, because the user program may not have enough energy to do something useful before a power failure happens. Instead, they propose to artificially wait (in software) for the energy buffer to be charged "enough" before starting the user program. Dewdrop dynamically adjusts this threshold level to optimize the application quality of service. But its main drawback is that the user program must still be able to complete its execution within the energy budget, that is at worst, within one full charge of the energy buffer.

Zhang *et al.* propose to overcome this limitation with *QuarkOS* [5]. Their idea is to divide the execution of the program into tiny fragments and to introduce software sleeps between them to allow for the energy buffer to replenish. QuarkOS is thus based on the hypothesis that every software task is indeed divisible. Purely computational tasks will

obviously break down nicely, but other common tasks like sensing and networking involve hardware peripherals which may not offer the fine-grained level of control required by QuarkOS. For example, the authors use a camera sensor that permits to break down the sampling of a single pixel into several steps.

With Mementos [2], Ransford *et al.* propose to save a *checkpoint* of the program state before a power failure occurs. Later on, when the platform reboots, the program state is restored using the most recent checkpoint. The idea is to spread the execution of the user program over multiple charge/discharge cycles of the energy buffer, so-called *lifecycles*. At compile time, Mementos inserts *trigger points* at each loop latch and function return. Each trigger point calls a Mementos function that measures the available energy level and performs a checkpoint if it is below a fixed threshold.

This approach is really promising in that it allows general program execution to transparently survive power failures. However, it has several drawbacks. First, measuring the energy level is time and energy consuming because it involves an ADC read. Because Mementos makes such measurements very often, it incurs a severe overhead on the application. Second, the developer is required to set many parameters arbitrarily. In this paper, we present several mechanisms to address these limitations.

III. Proposal

Our proposal involves two separate modules: the *checkpointing module* and the *energy monitoring module*. The former is responsible for saving and restoring checkpoints of the user program state to and from NVRAM. The energy monitoring module is responsible for triggering a checkpoint before power failure.

A. The checkpointing module

The job of the checkpointing module is to save the program state to NVRAM (and obviously to restore this state at the beginning of a new lifecycle). For reasons exposed in the introduction, we consider a platform with both NVRAM and regular, volatile, RAM. All read-only contents (executable code, read-only data) is stored in NVRAM, as well as (infrequently written) OS data. But for performance reasons we keep application variables as well as the main execution stack in regular RAM. The resulting memory layout is illustrated in figure 1. Note that the OS stack space is only used during checkpoint save and restore operations, so it can remain volatile without problems. To save the program state, the checkpointing module has to save the CPU state (registers), the program data and the program stack to NVRAM.

Copying all program state "as-is" like Mementos does would be inefficient in NVRAM, because most of the emerging memory technologies present latencies and dynamic energy significantly higher for write operations than for read operations [3]. For this reason, we use an incremental checkpointing strategy in order to minimize the quantity of data written. The idea is to divide the program state into same-sized blocks, and to only save the blocks which have changed since the last checkpoint.

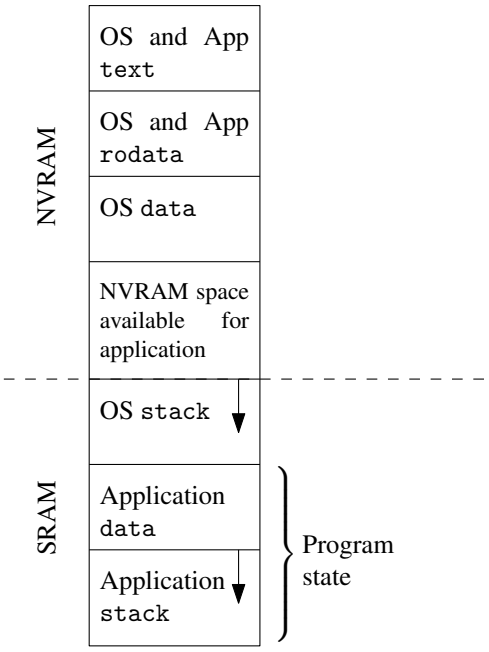

Fig. 1. Memory layout of the system

Because power failures may happen at any instant, we have to keep at least one valid checkpoint image at all times, and save program state to a second image. We denote the former by *active image* and the latter by *scratch image*. Thus even if the system crashes while checkpointing, the active image is not compromised and will be restored at the next lifecycle. But after a successful checkpoint, only the most recent image is useful. The roles of the two images are switched, and the new scratch image now contains an obsolete state.

A checkpoint image is implemented as an array of pointers, as illustrated in figure 2. Both the block pool and the two checkpoint images are located in non-volatile memory. Let's assume that the system just performed a successful checkpoint: the active image contains now our last checkpoint and the scratch image contains our next-to last checkpoint. The first and fourth blocks of the program state must have been unchanged between these two instants, because both images agree on their contents. But other blocks pointed by the scratch image have been obsoleted and will need to be garbage collected.

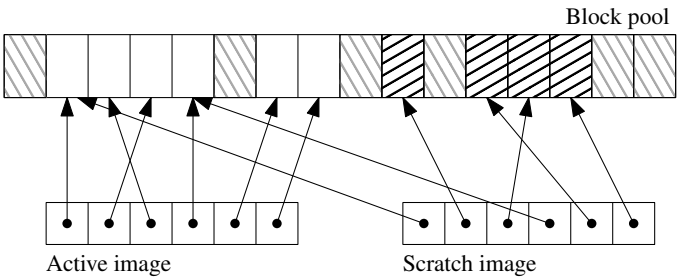

Fig. 2. Persistent data structure. In the block pool, white rectangles are valid blocks (i.e. they are part of the active image). Gray hatched rectangles represent free blocks. Black hatched rectangles represent *obsolete* blocks. These will be marked free by the GC during the next collection cycle.

We can now sum up the different steps of our checkpointing operation:

1) Push the CPU registers in the program stack,
2) Perform a garbage collection to reclaim obsolete blocks (cf below),
3) Save the program state according to algorithm 1,
4) Switch the roles of the two checkpoint structures.

At the beginning of the next lifecycle, the restore operation will simply consist in copying the content of the active image into the program space in SRAM, and then restoring the registers by poping them from the stack. The application program will resume without noticing the power failure.

Algorithm 1 Algorithm use to save the relevant program's memory space incrementally

Input: P[1..n] ▷ Program state, as n blocks
Input: A[1..n] ▷ Active image
Output: S[1..n] ▷ Scratch image
  for $i = 1$ to n **do**
    if HASH$(P[i]) \neq$ HASH$(A[i])$ **then**
      $new\_block \leftarrow$ ALLOC_BLOCK()
      COPY$(new_block, P[i])$
      $S[i] \leftarrow new\_block$
    else
      $S[i] \leftarrow A[i]$
    end if
  end for

Garbage collection: To manage the pool of checkpoint blocks, we implement a simple garbage collector (GC). This GC must be resilient to power failure, which means that if a crash happens during a GC operation, system integrity must not be compromised. For that purpose, knowing that the GC manage same-sized blocks, we use a bitmap to identify free blocks.

Another desirable property is that the duration of the whole checkpointing operation should be kept as constant as possible. This makes the task of the energy monitoring module easier, because it can wait more before triggering a checkpoint. For that reason, we don't wait until the block pool is exhausted before recycling space. Instead, we do incremental garbage collection at each checkpoint, before program state is actually saved. A collection cycle consists in freeing all blocks not referenced by the active image.

B. The energy monitoring module

The goal of the energy monitoring module is to determine when a checkpoint should be made. This task is critical for the performance of our solution. Indeed, if the checkpointing process is triggered too late, we may not have enough energy left to complete the checkpoint, and we will lose the progress of the whole lifecycle. On the other hand, if we trigger the checkpointing process too early, it will be finished before the actual power failure occurs and the remaining energy will be wasted. The task of checkpointing at the right time is complicated by the facts that the amount of harvested energy is unpredictable. Moreover, the consumption of the executed program is generally not constant. Also, our system should be able to adapt to different program profiles (sensing, computation, networking...) as well as different harvested energy profiles.

Like Mementos [2], we trigger a checkpoint when the energy buffer voltage becomes less or equal than a certain threshold, denoted V_c, fixed at compile time. To monitor the energy buffer, Mementos inserts *trigger points* in the control flow of the application. Each trigger point pauses the application, reads an ADC to measure the capacitor voltage, and either resumes execution or starts a checkpoint operation. Because of the frequency, and the significant cost of these ADC reads, Mementos imposes a large overhead on execution time [2], and the application only gets a small fraction of all CPU cycles. To improve on this idea, we propose to use a hardware timer instead of explicit calls, and to dynamically adapt the timer period in order to minimize the actual number of ADC reads. The optimal scenario is when the timer is programmed once at boot time and expires when the voltage is just below to V_c. But accurately predicting the duration of the lifecycle is not realistic. Instead, we use an adaptive policy.

We implement two strategies for programming the timer. The first one uses an exponentially adapted polling interval similar to what *Dewdrop* uses to charge the energy buffer. At boot time, the timer is set with a compile-time constant. When it expires, if the threshold V_c is not reached yet, the timer is reprogrammed with a new period higher, equal or smaller than the previous period, depending on the distance remaining from the measured voltage to V_c.

The other method is based on the observation that, while the system is active, it draws much more power than can be harvested, hence the discharge of the energy buffer is roughly linear. The idea is to program the timer at the time using a constant fixed at the compile time. Then, when the timer expires, we use a linear extrapolation based on the consumption since boot to compute the new timer value so that, when it will expire, the energy buffer voltage will be close to V_c.

IV. Experimentations and Results

We evaluate the benefits of our approach on a modified version of the Mementos simulator. Initially based on the MSPsim platform emulator, Ransford added[1] support for energy harvesting and consumption. We add support for simulating NVRAM but we use the same energy harvesting traces. Our benchmarks, also taken from Mementos in order to get comparable results, are the following:

`1e5` loops 65500 times on a `nop` instruction,
`crc` performs a 16 bits CRC,
`rsa` performs a 256 bits RSA hashing.

Figure 3 illustrates the different execution phases of an application instrumented with our OS over a few lifecycles. On the very first boot, the OS initializes its persistent data structures (magenta). On a later reboot, the OS restores the active checkpoint to SRAM and sets a timer (yellow), then jumps to application code (solid black line).

[1] http://github.com/ransford/mspsim

When the timer expires, the energy monitoring module measures the energy left and decides what to do:

- if it is above V_c (blue) then we reprogram the timer and resume the application,
- if it is below V_c then we start a checkpointing operation. Either it succeeds (green) if enough energy is available, or it fails (red) because of power loss.

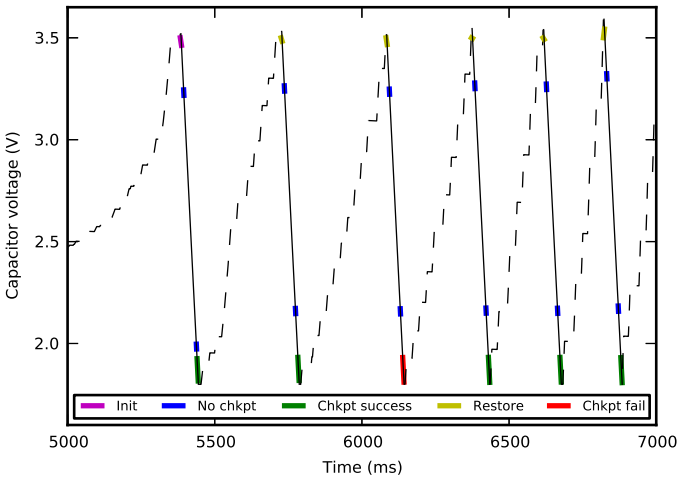

Fig. 3. Execution of an instrumented program. Black lines show the capacitor voltage: dashed when the platform is dead, and solid when the platform is active. Color highlights that OS code is executing (see text above for a description of various OS operations). The checkpoint threshold V_c is fixed to $1.9V$.

Our simulation results for the two energy monitoring strategies presented in the previous section are presented in the tables of figure 4. As we can see, the two methods present roughly the same CPU overhead and execute the tasks in approximately the same number of lifecycles. The linear extrapolation method requires less calls to the OS, but it is more complex and requires more CPU cycles at each call. The `rsa` program suffers from a significant overhead (about 50%) because it works on a larger data structure, making the checkpointing quite longer than for other benchmarks.

However, the benefits of our incremental checkpointing approach are well illustrated by this test case. Because the large data structure does not change very quickly, each checkpoint can reuse most of the previously saved blocks, and writes only 20-25% of new blocks. As writing in NVRAM is expected to be much more energy consuming than reading, we argue that this is a significant advantage over Mementos. However, we do not have a hardware prototype and quantitative comparison is currently future work.

V. Conclusion and Further Work

In this paper, we present an incremental checkpointing scheme, enabling execution of general programs on a transiently powered system comprising both volatile memory and NVRAM. Early simulation results show that our mechanism has acceptable CPU overhead compared to the execution of the same program with infinite energy.

We have different research perspectives on this work. Rather than considering a fixed threshold V_c, we intend to make our OS to *learn* it over time. Indeed, this threshold depends on

Exponential adaptation

Trace	crc			1e5			rsa		
	lifecycles	overhead	chkpt ratio	lifecycles	overhead	chkpt ratio	lifecycles	overhead	chkpt ratio
#1	4	12.0	83.3	3	8.7	70.0	7	53.9	23.3
#2	4	12.1	83.3	3	8.6	70.0	7	53.7	24.0
#3	5	11.9	83.3	3	8.6	70.0	7	53.8	24.0
#4	4	12.1	83.3	3	8.6	70.0	7	54.2	24.0
#5	3	11.3	83.3	3	8.9	70.0	8	53.8	24.0
#6	6	11.3	80.0	3	8.8	70.0	7	53.6	23.3
#7	3	11.4	83.3	3	9.3	70.0	7	54.6	24.0
#8	3	11.3	83.3	3	8.6	70.0	7	53.6	23.2
#9	3	11.3	83.3	3	8.7	70.0	7	54.0	24.0
#10	4	9.6	83.3	3	10.0	70.0	11	53.3	23.9

Linear Interpolation

Trace	crc			1e5			rsa		
#1	4	13.0	83.3	3	9.5	70.0	7	52.8	22.7
#2	4	13.0	83.3	3	9.4	70.0	8	54.9	24.4
#3	4	13.0	83.3	3	9.5	70.0	8	53.5	22.9
#4	4	13.0	83.3	3	9.5	70.0	8	52.4	22.7
#5	4	13.0	83.3	3	9.5	70.0	8	53.4	23.6
#6	4	13.0	83.3	3	9.5	70.0	8	52.6	22.0
#7	4	12.7	83.3	3	9.6	70.0	9	53.0	23.3
#8	4	12.7	83.3	3	9.5	70.0	8	53.3	23.0
#9	4	12.6	83.3	3	9.4	70.0	9	53.4	23.7
#10	4	12.6	83.3	3	9.0	70.0	7	53.1	23.8

Fig. 4. Simulation results of the three benchmarks on various energy-harvesting traces, for our two timer adaptation strategies. For each program, we show in how many lifecycles the program completes, the execution overhead (percentage of CPU cycles spent in the OS), and the average ratio of saved blocks during checkpointing operations.

the amount of data to save and can be very different from one application to another. With this in mind, we want to study other types of programs including communication and sensing activities.

Then, one important step will be to validate our energetic models. First, we need to measure real NVRAM latencies and energy costs, and include them in the simulator. This will allow to validate quantitatively the benefits of our checkpointing mechanism. Second, we intend to make experiments with different types of harvesters (*e.g.* solar panels) to make our approach more generic.

References

[1] M. Buettner, B. Greenstein, D. Wetherall, and J. R. Smith, "Revisiting smart dust with rfid sensor networks," in *Workshop on Hot Topics in Networks (HotNets-VII)*, 2008.

[2] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on rfid-scale devices," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*, 2011, pp. 159–170.

[3] L. V. Cargnini, L. Torres, R. M. Brum, S. Senni, and G. Sassatelli, "Embedded memory hierarchy exploration based on magnetic RAM," in *Faible Tension Faible Consommation (FTFC), 2013 IEEE*, 2013.

[4] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An energy-aware runtime for computational rfid," in *8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, 2011.

[5] P. Zhang, D. Ganesan, and B. Lu, "Quarkos: Pushing the operating limits of micro-powered sensors," in *14th Workshop on Hot Topics in Operating Systems, HosOS'13*, 2013.