SCHEMATIC: Compile-time checkpoint placement and memory allocation for intermittent systems

Hugo Reymond, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Isabelle Puaut, Erven Rohou

SCHEMATIC: Compile-Time Checkpoint Placement and Memory Allocation for Intermittent Systems

Hugo Reymond*, Jean-Luc Béchennec†, Mikaël Briday†, Sébastien Faucou†, Isabelle Puaut* and Erven Rohou*

*Univ Rennes, Inria, CNRS, IRISA - name.surname@irisa.fr
†Nantes Université, École Centrale Nantes, CNRS, LS2N, F-44000 Nantes, France - name.surname@ls2n.fr

***Abstract*—**

Battery-free devices enable sensing in hard-to-access locations, opening up new opportunities in various fields such as healthcare, space, or civil engineering. Such devices harvest ambient energy and store it in a capacitor. Due to the unpredictable nature of the harvested energy, a power failure can occur at any time, resulting in a loss of all non-persistent information (*e.g.*, processor registers, data stored in volatile memory). Checkpointing volatile data in non-volatile memory allows the system to recover after a power failure, but raises two issues: (i) spatial and temporal placement of checkpoints; (ii) memory allocation of variables between volatile and non-volatile memory, with the overall objective of using energy as efficiently as possible.

While many techniques rely on the developer to address these issues, we present SCHEMATIC, a compiler technique that automates checkpoint placement and memory allocation to minimize the overall energy consumption. SCHEMATIC ensures that programs will eventually terminate (*forward progress* property). Moreover, checkpoint placement and memory allocation adapt to the size of the energy buffer and the capacity of volatile memory. SCHEMATIC takes advantage of volatile memory (VM) to reduce the energy consumed, by automatically placing the most used variables in VM.

We tested SCHEMATIC for different experimental settings (size of volatile memory and capacitor) and results show an average energy reduction of 51 % compared to related techniques.

***Index Terms*—Embedded systems, Intermittent computing, Memory management, Checkpointing**

I. INTRODUCTION

Battery-free devices allow sensing in hard-to-access locations, with low maintenance costs and environmental impact [1]–[4]. These devices harvest energy from their environment (sun, wind, vibrations) [5,6]. Instead of relying on a battery, they store the harvested energy in a small energy buffer (*super-capacitor*, *capacitor* for short in the rest of the paper). The harvested energy is usually neither sufficient nor stable enough to power the device continuously and perform long-running computations. Therefore, program execution spans over several computation phases, interleaved by power-off states. The underlying model of computation is known as *intermittent computing* [6].

Intermittent computing raises challenges for system designers since all volatile data (CPU registers, such as the program counter, and volatile memory, if any) is lost upon power-off. In such context, any progress made since the last power failure is lost, hindering the successful completion of the program. As of today, different approaches have been designed to ensure that programs will eventually terminate despite power failures (*forward progress* property [7]). Such techniques rely on specialized hardware or on specific software techniques. At a software level, many solutions have been developed to regularly save volatile data into non-volatile memory (*checkpointing*). Some solutions perform *static checkpointing* by computing at compile-time the locations in program code where backup operations will be performed [8,9]. Other solutions rely on measurements to estimate the energy available in the capacitor and trigger checkpointing operations just before the capacitor runs out of energy (*dynamic checkpointing* [10,11]).

Microcontrollers used in intermittent computing often have two kinds of memory: a fast, small and energy-efficient volatile memory (VM) and a larger non-volatile memory (NVM). The energy efficiency of VM would suggest storing all data in VM to save energy (even with efficient emerging NVM technologies such as FRAM, NVM accesses still consume up to $2.47\times$ more than VM accesses [12]). However, data allocated in VM are lost in case of power failures, and therefore have to be checkpointed in NVM, which introduces an overhead. Additionally, the size of VM is limited, and in general small (a few KB on most platforms). This is why hybrid VM/NVM architectures allow splitting the memory of a program between VM and NVM. Nonetheless, storing data in NVM may result in inconsistencies between VM and NVM when the application code is re-executed after a power failure (*memory anomalies* [13,14]).

Our contribution. Putting the placement of checkpoints and the memory allocation of variables under the exclusive responsibility of the developers is time-consuming, error-prone and may yield sub-optimal performance. We propose SCHEMATIC[1] to automate checkpoint placement and variable allocation in hybrid VM/NVM architectures. The objective of SCHEMATIC is to minimize the program's energy consumption while ensuring forward progress. At compile-time, SCHEMATIC places checkpoints and decides of the memory allocation of variables in polynomial time. At run-time, when a checkpoint location is reached, volatile data is saved in NVM and the platform waits until the capacitor is fully replenished.

[1] SCHEMATIC stands for Simultaneous Checkpoint Placement and Memory Allocation Tailored for Intermittent Computing

SCHEMATIC relies on an energy-aware approach: it assumes that the worst-case energy consumption (WCEC) of any activity in the system is known and leverages this knowledge to place checkpoints, such that all the code between two checkpoints can always be executed with no power failure. Therefore, SCHEMATIC ensures that any code will eventually terminate (forward progress). Execution never rolls back, thus no energy is wasted in re-executions.

In contrast to previous approaches that focus on either checkpoint placement or memory allocation, SCHEMATIC is, to the best of our knowledge, the first contribution to address both problems simultaneously. Additionally, SCHEMATIC takes into account the size of the VM, a consideration overlooked by related work. An extensive evaluation of SCHEMATIC compared to existing techniques shows that it allows significant energy reductions (51 % on average), and that it is able to run programs correctly until termination when other solutions cannot.

The remainder of this paper is organized as follows. Section II first gives a motivating example for SCHEMATIC and describes the research problem addressed. Section III is then devoted to an in-depth description of the compile-time and run-time operations carried out by SCHEMATIC. An experimental evaluation of SCHEMATIC is given in Section IV. SCHEMATIC is compared to related work in Section V. We discuss SCHEMATIC safety in Section VI. Finally, Section VII concludes the paper.

II. MOTIVATIONS AND PROBLEM STATEMENT

A. Motivating example

Let us motivate the rationale behind SCHEMATIC on a simple yet realistic example, depicted in Figure 1. The C code in the figure computes the sum of the elements of an array (variable *array)* and stores it in the scalar variable *sum*. Variable *sum* is subsequently used as a parameter in a call to function *f*. For the sake of demonstration, we assume that compiler optimizations do not promote variables to registers. Let us consider a static checkpointing scheme, that saves all volatile data (variables and registers) in NVM at predetermined code locations (through a call to function *save_state* in the figure).

```
1  int sum = 0;
2  save_state();
3  for(int i=offset; i<SIZE; i++)
4    sum += array[i];
5  /* ..... */
6  save_state();
7  class = f(sum);
8  save_state();
```

Fig. 1. Source code for a motivating example

1) **Motivation for energy-efficient memory allocation:** During the first phase of the program (lines 3–4), variable *sum* is frequently accessed. Thus, the energy gain of storing *sum* in VM counterbalances the cost of loading it from NVM at startup and saving it in NVM on a checkpoint. Allocating *sum* to VM is the best option in this case.

During the second phase of the program (line 7), assumed to take place much later, keeping *sum* in NVM is more appropriate, since loading it in VM would consume more energy than the energy gained for the unique access to *sum*. This motivates the need for a method that changes the memory allocation of variables during the program execution.

Since the size of the VM is limited, not all variables can be allocated in VM, even though it would be energy-efficient to do so. To avoid the cumbersome task of manual memory allocation of variables, compiler support for memory allocation, as provided by SCHEMATIC is obviously valuable.

2) **Motivation for joint memory allocation and checkpoint placement:** The decision-making process for checkpoint placement is inherently linked to memory allocation. For instance, it may happen in Figure 1 that there is not enough energy to execute the code between lines 2 and 6 if variable *sum* is allocated to NVM, whereas it becomes possible to do so when *sum* is stored in VM.

Conversely, the selection of a memory allocation depends on the location of the checkpoints. As an example, if a checkpoint is placed in the loop body (line 4) there may no longer be a benefit in allocating *sum* to VM as it would require checkpointing at each iteration. Furthermore, checkpoint locations are natural points to change the memory allocation of variables, since saving a checkpoint copies a variable in NVM and therefore hides variable migration overhead.

As one can see, the memory allocation and checkpoint placement problems are inter-dependent and should be solved jointly. The complexity of solving both problems simultaneously motivates compiler support.

SCHEMATIC provides such a compiler support for statically selecting the location of checkpoints in the code and jointly selecting the allocation of variables, that may change only at these points. The technique is fully automatic, minimizes the overall energy consumption (including checkpointing overhead) and copes with the limited capacity of the VM.

B. Problem statement

More formally, the research problem addressed by SCHEMATIC is the following.

Inputs and assumptions:

- SCHEMATIC assumes a hybrid architecture with NVM and VM, as displayed in Figure 2. The NVM is assumed to be large enough to store all application code and data. The VM is assumed to be smaller, with a size S_{VM}.
- The platform is equipped with a capacitor with known limited energy storage E_B.
- A safe yet precise worst-case energy consumption model is provided as an input to SCHEMATIC.

The energy available in the capacitor (E_B) depends on many factors such as manufacturing, aging or temperature [15,16]. The evaluation of E_B is discussed in Section VI and is considered as outside of the scope of this paper.

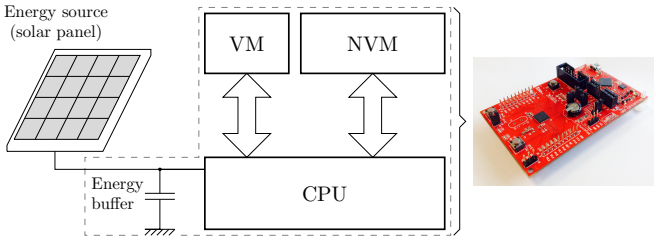

Fig. 2. Considered architecture

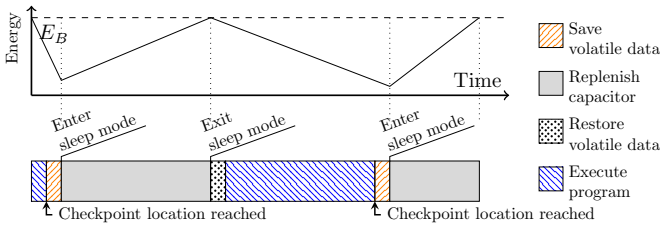

Fig. 3. Checkpointing strategy of SCHEMATIC

Problem statement: SCHEMATIC addresses the problem of checkpoint placement and variable allocation in intermittent computing systems, in order to minimize energy consumption.

Guarantees:

- *Forward progress.* Checkpoint placement in SCHEMATIC guarantees that any calculation eventually terminates: the execution will not remain trapped in endless re-executions due to repetitive power failures.
- *Absence of memory anomalies.* SCHEMATIC prevents code re-executions by making sure that the energy required to run the program between any two successive checkpoints is always below E_B, and that the capacitor is replenished at each checkpoint location. As a consequence, inconsistencies between VM and NVM (memory anomalies) are avoided by design.
- *Best-effort energy minimization.* Checkpoint placement and memory allocation are designed to minimize energy consumption on the most frequently executed paths.
- *Efficient usage of VM.* SCHEMATIC allocates variables in VM to reduce energy consumption, and transparently ensures that at any time, the volume of data allocated to VM is lower than or equal to S_{VM}.

III. SCHEMATIC: JOINT CHECKPOINT PLACEMENT AND MEMORY ALLOCATION

A. Overview of SCHEMATIC

SCHEMATIC provides a compile-time technique that both determines (i) the locations where volatile data is saved in NVM (checkpoint locations) and (ii) the allocation of variables (VM or NVM) between checkpoint locations.

At run-time, when a checkpoint location is reached, volatile data (processor registers, data allocated in VM) is saved in NVM (see Figure 3, label ▨). Then the system remains in standby until the capacitor is fully charged (□). During this standby period, the system is put into sleep mode and wakes up regularly to measure the voltage across the capacitor. Should a power failure occur during a standby period, the system goes back to sleep on restart. When measurements indicates that the capacitor is fully charged, the state of the program is restored (▩), and the execution is resumed (▧). The top curve in Figure 3 depicts the evolution of the state of charge of the capacitor during the different phases (program execution, energy replenishment).

SCHEMATIC determines variable allocations (VM or NVM) at compile-time. The allocation of a variable may change at run-time, but the points where the allocation may change (checkpoint locations) are defined at compile-time. Memory allocation is performed at the granularity of variables in the source code (scalars, structs, arrays considered as a whole).

SCHEMATIC analyzes and modifies a program using its Control Flow Graph (CFG) representation. The execution of a program may give rise to a number of execution paths at run-time, a path being defined as an ordered sequence of basic blocks, starting from the CFG's entry point and ending at one of its exit points.

SCHEMATIC positions the checkpoint locations and selects the allocation of variables with the objective of minimizing the energy required to run the most frequently executed paths (path frequency analysis is discussed in section III-A3). The flow of the algorithm is as follows:

1) Select a path to analyze.
2) Decide on checkpoints placement and memory allocation along this path, such that computations between checkpoints, given the selected allocation of variables, can be executed within the energy budget E_B.
3) Modify the program code to insert save/restore instructions at selected checkpoint locations and set the targets of memory accesses (VM or NVM) according to the selected memory allocation.
4) Repeat the process until all basic blocks of the CFG have been analyzed. The decisions made by SCHEMATIC along a path are final. When examining a yet unprocessed path, the memory allocations and checkpoint placements are inherited from the previous iterations.

The locations SCHEMATIC is considering for checkpoint placement are the CFG edges. During the program analysis, some of the potential checkpoints locations become *enabled*, meaning that instructions are inserted in the code to save and restore volatile data. The remaining checkpoint locations are *disabled*. They do not result in any overhead since checkpoint locations are selected at compile-time.

Sections III-A1 to III-A3 first describe SCHEMATIC checkpoint placement and memory allocation on a simple program. Section III-B then focuses on how SCHEMATIC handles function calls and loops. Finally, Section III-C derives the complexity of SCHEMATIC's analysis.

1) **Analysis on one path using the Reachable Checkpoint Graph (RCG):** In this section, we describe how SCHEMATIC analyzes a path on an example CFG depicted in Figure 4.a. The locations SCHEMATIC is considering for checkpoint placement

are edges AB, AC, BD and CD, resulting in four potential checkpoint locations (c_{AB}, c_{AC}, c_{BD} and c_{CD})[2]. Let us consider that the path (A, B, D) is analyzed first. Figure 4.b displays information along this path before analysis. The goal of the analysis is to set the memory allocation of *sum* on this path, as well as the status of the checkpoint locations c_{AB} and c_{BD}: *enabled* (a checkpoint will be inserted at this location) or *disabled*.

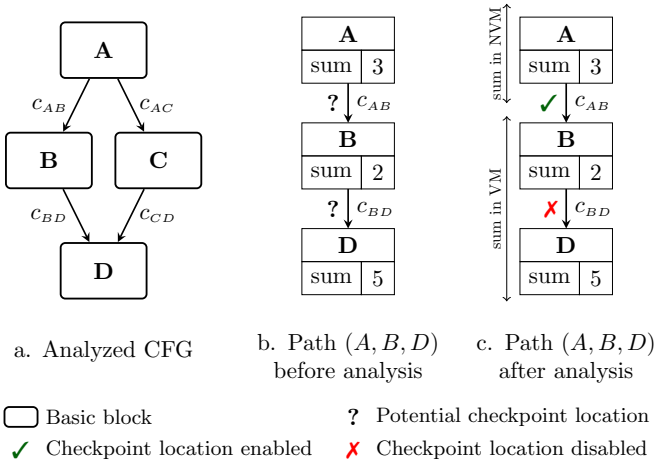

Fig. 4. Analysis of path (A, B, D) for checkpoint placement and memory allocation selection. Part a. depicts the analyzed CFG. Part b. depicts the status of path (A,B,D) before the analysis: potential checkpoint locations (c_{AB}, c_{BD}) are tagged with "?" because it is yet undecided if a checkpoint will be inserted at those locations; for each basic block the figure depicts the variables accessed (here only a variable named *sum*) and the number of accesses (read and write accesses aggregated for brevity). Part c. depicts the result of the analysis: a checkpoint is inserted on edge AB, variable *sum* is allocated in NVM before that checkpoint and in VM after.

The analysis uses the concept of *Reachable Checkpoint Graphs* (RCGs). A RCG is built from a given CFG path, and captures the energy consumed between potential checkpoint locations. A node of the RCG represents a potential checkpoint location c_x. An edge (c_1, c_2) in the RCG links two potential checkpoint locations c_1 and c_2. An edge (c_1, c_2) exists if it is possible to reach c_2 from c_1 in less than the energy budget E_B. Each edge is associated with an energy cost and a memory allocation. The cost corresponds to the energy needed to reach c_2 from c_1 considering the memory allocation of each variable (VM or NVM) that minimizes energy consumption. The energy cost includes:

- The energy to restore the volatile data at location c_1.
- The energy to execute the basic blocks along the analyzed path with the selected memory allocation. Selection of a memory allocation is detailed in the next paragraph.
- The energy to save volatile data into NVM at location c_2.

The absence of an edge from c_1 to c_2 means that there is no memory allocation that makes it possible to reach c_2 from c_1 with energy budget E_B.

[2]Without loss of generality, we assume that basic blocks need less than E_B energy to execute (those requiring more than E_B are split to fit in the energy budget).

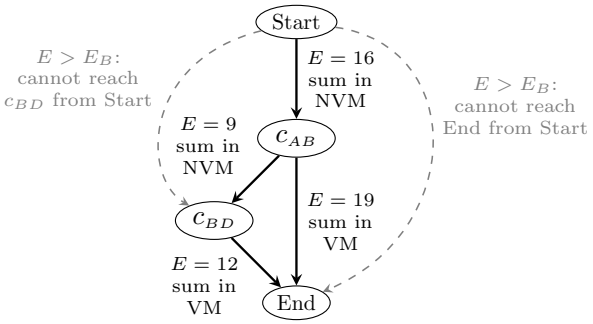

Fig. 5. Reachable Checkpoint Graph (RCG) for path (A, B, D) for an energy budget E_B of 20 *energy units*. Each edge in the RCG indicates the energy consumed (labeled E) to go from one potential checkpoint location to another one, for a given allocation of variables (here variable *sum*). Dashed lines indicate edges absent from the RCG due to E exceeding the budget E_B.

Two virtual nodes *start* and *end* represent the beginning and the end of the analyzed path. Each path from *start* to *end* corresponds to a valid placement of checkpoints for the analyzed path, with its corresponding memory allocations. The shortest path from *start* to *end* therefore represents the checkpoint placement and memory allocation that minimizes the energy consumed along the path. All checkpoint locations along the shortest path are enabled, while other checkpoint locations in the RCG are disabled. The selected memory allocations are the ones appearing on the edges of the shortest path.

Figure 5 represents the RCG for the CFG path (A, B, D). We consider a platform energy budget E_B of 20 *energy units*. Potential checkpoint locations for this path are c_{AB} and c_{BD}. The RCG contains two possible paths from *start* to *end*. We can either choose c_{AB} and c_{BD} as checkpoint locations – path (*start*, c_{AB}, c_{BD}, *end*) –, or only enable c_{AB} – path (*start*, c_{AB}, *end*). The shortest path in this case is (*start*, c_{AB}, *end*) thus we will only insert instructions to save/restore volatile data at checkpoint location c_{AB} (between the basic blocks A and B). The total cost of executing this path is 35 energy units. Figure 4.c depicts the result of the analysis.

2) **Selection of a memory allocation:** In order to build the RCG for a given path of the CFG, the energy required to go from a potential checkpoint location c_1 to another potential checkpoint location c_2 has to be determined, and this cost obviously depends on the allocation of variables referenced in the interval $[c_1, c_2]$ on this path.

The selection of the memory allocation of each variable is based on a cost function, that defines the gain obtained by placing a variable v in VM compared to NVM.

There is an energy gain when performing each access to VM instead of NVM: ΔE_W for a write access, ΔE_R for a read access. On the downside, there is an energy cost of $E_{\text{save/restore}}$ when allocating a variable v in VM, due to the overhead of restoring the variable at checkpoint c_1 and saving it in NVM at checkpoint c_2. The overall gain is therefore:

$$gain_v = \Delta E_W \times \mathrm{n}_W + \Delta E_R \times \mathrm{n}_R - E_{\text{save/restore}} \quad (1)$$

where n_W and n_R are the numbers of write and read accesses

to v in the interval. Repetitive accesses to a variable are required to recoup the overhead of checkpointing the variable.

Calculating this gain for all variables not only gives information on the benefit of placing a variable in VM, but also allows us to prioritize variables to be stored in VM in case of limited VM space. In order to maximize the overall gain for a limited memory size, SCHEMATIC sorts the variables by decreasing gain to size ratios. Thus, if multiple variables share the same gain, the smaller ones are allocated in priority, allowing to fit more variables in VM. Variables are allocated in VM contiguously, until either the list of variables with positive gains is exhausted or the VM is full.

Note that each variable v has a single address in NVM, attributed by the compiler toolchain. Additionally, when SCHEMATIC allocates a variable in VM in an interval $[c_1, c_2]$, it is given an address in VM, that does not change in the interval. Nevertheless, v may be attributed a different address in VM if loaded in VM for another interval.

Optimization of checkpointing by accounting for variable liveness. SCHEMATIC exploits information about the liveness of variables to reduce the checkpointing overhead. If a VM variable v is not used after a checkpoint $c2$, we do not need to save it. Similarly, if the first access to v after a checkpoint c_1 is a write access, we do not restore it, as its value will be overwritten.

The overhead $E_{\text{save/restore}}$ used in SCHEMATIC (see Equation 1) for a variable v depends on its size but also on its liveness, and is computed as follows:

$$E_{\text{save/restore}} = E_{\text{restore}} \times \text{live\_}c_1 + E_{\text{save}} \times \text{live\_}c_2 \qquad (2)$$

where E_{restore} represents the cost to restore v, E_{save} the cost to save v and live_X equals 1 if v is live at the checkpoint location X, and equals 0 otherwise.

3) **Path exploration and coverage issues:** To reduce consumed energy in priority for the most common paths, the different paths are iteratively analyzed, by decreasing frequency. The decisions made by SCHEMATIC along a path are final. Path prioritization is performed by extensive instrumentation of the code with varied input data, to gather execution traces, formed of sequences of executed basic blocks. Traces are sorted on a per-function basis. It may happen that despite extensive instrumentation, some portions of code never get executed. Paths are formed from these never-executed codes from the CFG, and are analyzed at the end of the algorithm to ensure complete code coverage.

When analyzing a new path p, only the segments of p that do not overlap with already analyzed paths are explored. The exploration of those segments inherits from the results for already analyzed paths, in the following way. Information is attached to the basic blocks from already analyzed paths: the memory allocation for the basic block, the energy left after its execution E_{left}, and the energy to leave to the basic block in order to be able to reach the next checkpoints E_{to_leave}.

When constructing the reachable checkpoint graph for a segment of path p containing already analyzed basic blocks, the criteria to determine if an edge (c_i, c_j) is present in the RCG slightly changes. Specifically, the following adjustments are made:

- When evaluating the edges originating from the *start* node, the criterion taken into account is E_{left}, rather than the energy budget E_B.
- Similarly, for the edges directed towards the *end* node, the criterion shifts from E_B to the difference between the energy budget and the energy to leave ($E_B - E_{to\_leave}$).

The energy left and energy to leave are recomputed and propagated after each new path analysis. Through the whole analysis, the energy left can only decrease while the energy to leave can only increase. By doing so, the analysis of new paths adapts to the constraints imposed by previously analyzed paths, guaranteeing a conservative checkpoint placement.

In our example, basic block A holds the information that the energy left after its execution is 4 (E_B = 20, A's execution cost is 16). The energy to leave to A in order to reach checkpoint c_{AB} is 16. This energy may increase if no checkpoint is placed at c_{AC}.

B. Handling of function calls and loops

1) **Handling of function calls:** The main challenge with functions is that they can be called from different places in the code. As checkpoint placement and memory allocation of a function is decided at compile-time and is context-independent, the decisions taken have to be the same for all calling contexts.

Functions are analyzed through a traversal of the function call graph, in reverse topological order, such that every function is always analyzed before its caller. SCHEMATIC currently handles non-recursive functions only[3]. A decision (checkpoint placement, memory allocation), taken for a given function in the call graph is imposed to the predecessor function(s) in the call graph. Analysis of functions is greedy: once a decision is taken for a given function, it is never reconsidered later on.

The analysis of leaf functions (functions with no successor in the call graph) proceeds as previously described in Section III-A. For a function f_{caller} that includes a call to a function f_{callee}, one must take into account the constraints established by the analysis of f_{callee}.

If f_{callee} does not include any checkpoint, then all of its basic blocks share the same memory allocation, as changes of memory allocation are performed on checkpoints. In this case, we can treat the function call to f_{callee} as a single basic block in the analysis of f_{caller}. On the other hand, if f_{callee} does have one or several checkpoints, then there may be different variable allocations when entering and exiting the function. Therefore, when analyzing f_{caller}, we must take into account the memory allocation and energy required to execute f_{callee} up to the first checkpoint(s) in f_{callee}, as well as the memory allocation and remaining energy when exiting f_{callee}. We further impose when analyzing a function that there is a single memory allocation when exiting the function, regardless of the number of exit points.

[3]We do not consider this as a significant restriction, since recursion is usually considered as a bad practice in embedded system development.

2) **Handling of loops:** We handle loops in a manner similar to functions. SCHEMATIC handles *natural loops* (strongly connected components of the CFG with a single entry point, called *loop header*). Without loss of generality, our description of loop handling in SCHEMATIC will assume a single *back-edge* (CFG edge from any basic block of the loop body back to the loop header) per loop. The source node of the back-edge is named *loop latch*.

Loops are analyzed through a bottom-up traversal of the loop nesting tree (*i.e.*, in the case of nested loops, inner loops are analyzed before outer loops).

A straightforward approach for handling loops would be to place a checkpoint on the loop back-edge and ensure one iteration of the loop can be executed within the energy budget. This guarantees the loop forward progress, but leads to unnecessary checkpointing overhead. Instead, we estimate the maximum number of loop iterations that can be executed before it is necessary to perform a checkpoint and then implement a conditional checkpointing scheme based on the number of iterations performed.

Loop analysis is performed in two steps, detailed below and shown in Algorithm 1:

Step 1. Analysis of one iteration (line 1). This first step processes one iteration of the loop using the algorithm of Section III-A. The algorithm is applied on the loop body with the back-edge removed. The result of the analysis is the memory allocation of the variables accessed in the loop body and the placement of checkpoints for one iteration of the loop. The algorithm will choose to place a checkpoint only if there is not enough energy to execute the entire loop body.

Step 2. Analysis of the entire loop (lines 2–10). The objective of this step is to decide if a checkpoint has to be placed on the loop back-edge, and how many iterations can be performed without saving volatile data. If the loop header and loop latch memory allocations are not the same, we need to place a checkpoint in order to change memory allocations between those two basic blocks (line 2).

If the loop header and the loop latch share the same memory allocation, it is not necessary to change memory allocation at each iteration. To reduce the overhead of checkpointing, a *conditional checkpointing* scheme is designed: save/restore operations occur once every num_{it} loop iterations,with num_{it} the number of loop iterations that can be executed within the energy budget of the platform E_B (lines 5–10).

When num_{it} is greater than the maximum number of iterations of the loop, no conditional checkpointing code is inserted. The maximum number of iterations of loops is provided using annotations.

Algorithm 1 Loop analysis in SCHEMATIC

▷ **Step 1. Analyze the loop body, without the back-edge**
1: ANALYZECFG(LoopBodyNoBackedge)
▷ **Step 2. Determine if a back-edge checkpoint is needed**
2: **if** $header.mem\_alloc \neq latch.mem\_alloc$ **then**
3: $backedge\_chkpt \leftarrow$ yes
4: **return**
5: $E_{loop} \leftarrow header.E_{left} - latch.E_{left}$
6: $\text{num}_{\text{it}} \leftarrow \left\lfloor \frac{E_B}{E_{loop}} \right\rfloor$
7: **if** $\text{num}_{\text{it}} > \text{max}_{\text{it}}$ **then**
8: $backedge\_chkpt \leftarrow$ no
9: **else**
10: $backedge\_chkpt \leftarrow$ every num_{it} iterations

C. Complexity considerations

In this section, we derive the complexity of the analysis of SCHEMATIC, based on the number of edges (E) and nodes (V) in a CFG.

To do so, we first focus on the complexity of the analysis of one path, that involves three main steps, namely building the RCG, identifying the shortest path in the RCG, and inserting checkpointing instructions into the code.

Building the RCG involves examining every possible combination of potential checkpoint location (c_i, c_j) where $i < j$. This process can be accomplished in polynomial time, specifically $O(\text{E}^2)$. The next step, finding the shortest path in the RCG, is accomplished using Dijkstra's shortest-path algorithm. The number of nodes in the RCG is bounded by $\text{V'} \leq \text{V} + 2$, as we add two virtuals nodes *start* and *end*. For the number of edges, we know that the RCG is a directed acyclic graph, thus there is at most $\text{E'} \leq \frac{\text{V'}^2 - \text{V'}}{2}$ edges in the RCG. Dijkstra algorithm has a time complexity of $O(\text{E'} + \text{V'} \times log(\text{V'}))$, thus finding the shortest path in the RCG has a complexity of $O(\text{V}^2 + Vlog(V)) = O(\text{V}^2)$. Finally, inserting checkpointing instructions into the code involves examining each edge of the analyzed path, and can be performed in linear time, specifically $O(\text{E})$. Overall, the time complexity of analyzing a path is $O(\text{V}^2 + \text{E}^2)$.

Since, in SCHEMATIC, a path is analyzed only if one of its basic block has not already been analyzed, there is at most one path analyzed per basic block. As a result, there will be a maximum of V analyses performed, leading to an overall polynomial complexity of $O(V \times (\text{V}^2 + \text{E}^2))$.

Empirically, the execution time of the analysis of SCHEMATIC is around 1 min (71 s on average) when applied on the benchmarks considered in Section IV.

IV. EXPERIMENTAL EVALUATION

Contrary to many intermittent computing systems, SCHEMATIC memory allocation enables the execution of applications even when their data exceeds VM size. This property is evaluated in Section IV-B. SCHEMATIC also provides automatic placement of checkpointing operations to guarantee forward progress. This facility is evaluated in Section IV-C. Section IV-D compares the energy consumption of SCHEMATIC with the one of related techniques. The memory allocation performed by SCHEMATIC is evaluated in Section IV-E. Finally, the impact of the capacitor size on consumed energy is studied in Section IV-F. The experimental setup is first presented in Section IV-A.

A. Experimental setup

a) **Target hardware and energy model:** Our experiments target the MSP430FR5969 [12], a low-power platform with a 64 KB ferromagnetic RAM NVM and a 2 KB SRAM VM. By default a 16 MHz operating frequency is assumed.

The worst-case energy consumption model used in experiments focuses on CPU energy consumption. We use the same energy model (taken from ALFRED [17]) for a fair comparison between all techniques. The energy spent per instruction is calculated from the instruction execution time and the type of memory access (VM or NVM) (see [17] for details). We do not currently consider the energy consumption of peripheral devices in our model, since the tested benchmarks are not using peripherals.

b) **Baselines:** We compared SCHEMATIC with four baselines: RATCHET, MEMENTOS, ROCKCLIMB and ALFRED. These techniques were selected because they all rely on static selection of checkpoints locations as SCHEMATIC does.

- RATCHET [9] is designed for systems only equipped with NVM. To deal with memory incoherence resulting from re-executions, RATCHET leverages compile-time instrumentation to place static checkpoints, in order to break write-after-read dependencies (such as incrementing a variable). Since RATCHET does not use VM, the CPU registers are the only volatile data to checkpoint. We use RATCHET as an All-NVM baseline.
- MEMENTOS [8] only uses VM as working memory and relies on compile-time selection of potential checkpointing locations. At runtime, MEMENTOS takes decisions about whether a checkpoint should be skipped or not, given the energy left. To estimate the energy available, it measures the voltage across the capacitor. MEMENTOS is used as an All-VM baseline, since it uses NVM only for checkpointing.
- ROCKCLIMB [18] is a compile-time checkpoint placement algorithm using only NVM as working memory. The first compiler pass of ROCKCLIMB systematically places checkpoints at loop headers and before function calls. Its second pass is responsible for inserting additional checkpoints, if needed, to ensure forward progress: it traverses the program CFG and adds checkpoints on the paths for which the energy consumption between successive checkpoints is higher than E_B. We re-implemented ROCKCLIMB and its *loop unrolling* optimization. That optimization unrolls loops to avoid saving checkpoints at each loop iteration (we nonetheless limit the unrolling factor to 10 to keep code size limited).
- ALFRED [17] is the only compile-time technique that, like SCHEMATIC, uses both VM and NVM as working memories. It reduces checkpointing overhead, by performing deferred restoration of variables (on their first read) and anticipated saving of variables (on their last write). ALFRED does not impose a checkpoint placement strategy and rather relies on existing work for static checkpoint selection. When reaching a checkpoint, only the CPU registers are saved in NVM, since all other volatile data has been saved previously. VM in ALFRED is used as much as possible, to reduce energy consumption.

For MEMENTOS and ALFRED, we placed checkpoints on loop latches, as described in the MEMENTOS publication [8].

c) **Tools and implementation:** Similarly to ALFRED, SCHEMATIC operates on the Intermediate Representation (IR) of the LLVM compiler infrastructure [19] (*version 8*), more precisely, on the textual representation of the IR generated by the *clang* compiler. To allow a fair comparison between techniques, all of them were re-implemented inside SCEPTIC [13,20], the infrastructure developed by the authors of ALFRED. SCEPTIC allows the analysis, modification and emulation of programs at the IR level. It enables precise monitoring of the energy consumption, allowing to understand how the available energy is utilized (computation, memory accesses, ...)

SCHEMATIC is implemented in multiple compilation passes. The first one (already implemented in SCEPTIC) gathers information about the targets of the load and store instructions. Then, the CFG of the program, augmented with the data gathered by the first pass, is fed into the central pass of SCHEMATIC, which selects checkpoints placement and memory allocation as described in Section III. The two final passes modify the program by setting the memory targeted by load/store operations according to the computed memory allocations and inserting save/restore operations.

In the current implementation of SCHEMATIC, variables accessed through pointers are systematically allocated in NVM. By doing so, we guarantee that their addresses do not change during the program execution. We believe this does not cause much performance loss since variables accessed through pointers are usually large arrays that SCHEMATIC would probably allocate in NVM.

The evaluation of the runtime behavior of SCHEMATIC, ALFRED, ROCKCLIMB, RATCHET and MEMENTOS relies on the SCEPTIC emulator, which executes programs at IR level, under intermittent power supply. To determine when power failures happen in a simple and reproducible way, the emulator relies on periodic power failures, through the notion of *time between power failures* (TBPF), which defines the time interval in cycles between power failures [9,18,21]. The SCEPTIC emulator monitors several program metrics at the IR level and provides bindings to map those metrics to machine-specific metrics, in particular the MSP430FR5969 energy consumption.

Execution traces of the basic blocks used by SCHEMATIC for checkpoint placement and memory allocation were generated using the SCEPTIC emulator by executing each benchmark 1000 times with randomly-generated inputs for each run.

d) **Benchmarks:** SCHEMATIC is evaluated on the MiBench2 benchmark suite [22]. We restrict our analysis to the benchmarks whose overall memory consumption is lower than 64 KB, the NVM size of the MSP430FR5969. We also rule out benchmarks *stringsearch* and *rsa* that does not execute correctly on the current version of SCEPTIC. The evaluated

benchmarks, that include the ones used in the evaluation of ALFRED in [17], are: *aes, basicmath, bitcount, crc, dijkstra, fft, randmath* and *rc4*.

B. Ability to support limited VM space

Volatile memory size can rapidly become a restricting factor on low-power embedded platforms, even with very small programs. To the best of our knowledge, no mixed VM/NVM memory allocation method takes this constraint into account. To highlight this issue, we have evaluated, for each studied technique (RATCHET, MEMENTOS, ROCKCLIMB, ALFRED and SCHEMATIC) if it could execute the benchmarks on a MSP430FR5969 board (64 KB NVM, 2 KB VM). Results are given in Table I.

RATCHET	MEMENTOS	ROCKCLIMB	ALFRED	SCHEMATIC
✓✓✓✓✓✓✓✓	✓✓✓✓✗✗✓✗	✓✓✓✓✓✓✓✓	✓✓✓✓✗✗✓✗	✓✓✓✓✓✓✓✓

✓: the benchmark can be executed with the limited VM size
✗: VM size is not large enough

TABLE I
ABILITY TO SUPPORT LIMITED VM SPACE. THE SERIES OF EIGHT SYMBOLS REPRESENTS THE BENCHMARKS: *aes*, *basicmath*, *bitcount*, *crc*, *dijkstra*, *fft*, *randmath*, *rc4*.

RATCHET and ROCKCLIMB use NVM as working memory so they do not require VM at all. They can therefore, by design, run all benchmarks. However, as NVM-only techniques, they do not benefit from the energy reductions that would be possible with VM.

MEMENTOS uses VM as working memory for all variables. Thus, it cannot run benchmarks with cumulated variable size larger than the VM size, which is the case for *dijkstra* (that needs 30 KB of VM), *fft* (16.7 KB) and *rc4* (6.5 KB).

ALFRED relocates memory accesses to use VM only when profitable regarding energy consumption. However, since it uses the same offset to access both data in VM and data in NVM, a large VM size (identical to NVM size) is needed, even if only a few bytes are actually accessed. This explains why ALFRED, like VM-only techniques, is unable to run benchmarks *dijkstra*, *fft* and *rc4*.

SCHEMATIC takes benefit of the energy reduction provided by VM while accounting for its limited capacity. It is therefore able to run all the benchmarks for any data size.

Overall, only RATCHET, ROCKCLIMB and SCHEMATIC are able to execute all benchmarks with the VM size of the MSP430FR5969 platform. However compared to RATCHET and ROCKCLIMB, SCHEMATIC takes benefit from the energy reduction resulting from the use of VM.

C. Ability to enforce forward progress

The ability of the studied techniques to enforce progress is evaluated by testing them on the same simple intermittency scenario, consisting of periodic power failures of period TBPF (*time interval between power failures*). The techniques were tested for different values of TBPF, that were chosen according to the duration of the benchmarks. We measured the execution time (in clock cycles, with all data in VM) of the benchmarks to select a range of TBPF that allows us to cover different situations, from no failure during the program execution to frequent failures (see Table II). The resulting TBPF are 1k, 10k, and 100k clock cycles. For each value of TBPF we set E_B to the average amount of energy that is consumed by the platform in the interval.

Benchmark	Execution time (in clock cycles)	Minimal number of power failures for TBPF= 1k	10k	100k
aes	1 079 363	1080	108	11
basicmath	169 599	170	17	2
bitcount	819 411	820	82	9
crc	41 133	42	5	0
dijkstra	1 381 746	1382	139	14
fft	377 578	378	38	4
randmath	15 062	16	2	0
rc4	437 335	438	44	5

TABLE II
EXECUTION TIME AND MINIMAL NUMBER OF POWER FAILURES FOR EACH *time between power failures* PER BENCHMARK (IN CLOCK CYCLES)

Table III shows if the benchmarks finish their execution or not, for all baselines and TBPF values.

Baseline	TBPF (cycles) 1k	10k	100k
RATCHET	✗✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓
MEMENTOS	✗✓✗✗✗✗✗✗	✗✓✓✓✗✗✗✗	✓✓✓✓✗✓✓✓
ROCKCLIMB	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓
ALFRED	✗✗✓✓✓✓✗✓	✓✓✓✓✓✓✗✓	✓✓✓✓✓✓✓✓
SCHEMATIC	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓

✓: the benchmark could terminate. ✗: the benchmark could not complete.

TABLE III
ABILITY TO ENFORCE FORWARD PROGRESS. THE SERIES OF EIGHT SYMBOLS REPRESENTS THE BENCHMARKS: *aes*, *basicmath*, *bitcount*, *crc*, *dijkstra*, *fft*, *randmath*, *rc4*.

ALFRED, MEMENTOS and RATCHET do not adapt to the platform characteristics thus they cannot ensure *forward progress* for small values of TBPF. ROCKCLIMB and SCHEMATIC, on the other hand, adapt their checkpoint placement to always ensure *forward progress*. We consider a TBPF of 10k as a good trade-off between extreme-intermittency and no-intermittency for the benchmarks considered. Therefore, we used this value for the experiments presented in Sections IV-D and IV-E.

D. Overall energy consumption

Reducing the energy consumption of intermittent programs is crucial, as it enables battery-less devices to accomplish more computations with the same amount of harvested energy. This also reduces the time spent replenishing the capacitor, thereby improving the system's throughput. This section evaluates the energy consumption of the eight considered benchmarks (intermittency management included) for periodic power failures.

Figure 6 displays the energy consumption of the considered benchmarks for a TBPF of 10k cycles for all analyzed techniques. Each bar in the figure splits the energy consumption in four categories:

- Computation (▨): energy for program execution, including the energy cost of memory accesses, and excluding the energy costs of re-executions after a power failure for the techniques that need re-executions.
- Save (▨): cumulated energy spent saving volatile data in NVM at checkpoint locations. This energy cost depends on the number of checkpoints saved and the size of checkpointed data.
- Restore (▨): cumulated energy spent restoring volatile data when re-starting the platform after a power outage.
- Re-execution (■): cumulated energy spent re-executing code for the techniques that roll-back to the previous checkpoint after a power failure.

Common observations can be made regardless of the benchmark under analysis.

First, SCHEMATIC allows in general a reduction of the overall energy consumption compared to the other solutions. On average, SCHEMATIC consumes 51 % less energy than the four baselines (the comparison was performed on the benchmarks that completed only).

Secondly, SCHEMATIC and ROCKCLIMB do not spend any energy in re-execution as they safely place checkpoints in the program and wait for full-replenishment of the capacitor.

Third, the energy consumption devoted to computations differs among techniques. This comes from their different memory allocations. As expected, MEMENTOS consumes the least energy in program execution since all data is allocated in VM, unlike RATCHET and ROCKCLIMB, which store all data in NVM. ALFRED and SCHEMATIC, on the other hand, opt for mixed memory allocation allowing them to consume less than All-NVM techniques.

Fourth, in comparison to the other techniques, SCHEMATIC saves checkpoints far less often and for a reduced subset of variables. Thus, the overall overheads for saving volatile data in SCHEMATIC are smaller than those of the other techniques.

Finally, SCHEMATIC and ROCKCLIMB tend to spend more energy in variable restoration compared to the other techniques. This comes from the fact that both techniques save and restore volatile data at all checkpoint locations, conservatively assuming that the platform goes into deep sleep and thus VM is lost. A more efficient handling of the sleep modes (*e.g.*, that would avoid saving data when there is enough energy to do so) is left for future work. It should be noted that, even in this case, there is still an incompressible cost for saving/restoring variables whose memory mapping is changed at the checkpoint.

In summary, checkpoint placement and memory allocation in SCHEMATIC results in significant reduction of energy consumption. As for the execution time, similar experiments were conducted and show analogous results with an overall execution time reduction of 54 % on average compared to related techniques.

E. Quality of memory allocation in SCHEMATIC

SCHEMATIC aims at reducing the energy consumed by a program by taking advantage of hybrid VM/NVM architectures. In order to measure how SCHEMATIC benefits from data allocation in VM, we compared the SCHEMATIC algorithm (joint checkpoint placement and memory allocation) to a modified version of SCHEMATIC called All-NVM, where no memory allocation in VM is performed (all data is stored in NVM). We focus on the energy spent for the computations, excluding energy used for intermittency management. We split the overall energy usage into three categories: computations without memory accesses, VM accesses and NVM accesses. Results are displayed in Figure 7.

For all benchmarks, memory allocation in VM allows to greatly reduce the energy usage: on average, SCHEMATIC requires 25 % less energy compared to All-NVM. The memory allocation of SCHEMATIC allows to efficiently exploit the VM capabilities, with 69 % of the memory accesses performed targeting VM on average, representing 33 % of the energy spent on computations.

F. Impact of capacitor size

The size of the energy buffer (E_B) plays an important role in the energy consumption of all techniques, as a small E_B results in more frequent power failures. The impact of the energy budget on the energy consumption is depicted in Figure 8 for the five techniques on benchmark *crc* (that was selected because it best illustrates the influence of E_B on energy consumption).

As one could expect, the overall cost of intermittency management (sum of ▨, ▨ and ■) decreases when E_B increases. However, the decrease is smaller for ALFRED and RATCHET than for SCHEMATIC, because their placement of checkpoints (and therefore energy for saving variables in NVM) does not depend on the platform characteristics, and is therefore constant. MEMENTOS performs conditional checkpointing given the energy available at runtime, thus it adapts to the variation energy budget. ROCKCLIMB adapts its checkpoint placement to the platform characteristics. However, as it places checkpoints on all loop headers and before all function calls, its checkpointing and restore overhead stays high. In contrast, for SCHEMATIC, both save and restore costs decrease when the value of E_B increases because fewer checkpoints are added to the program.

V. RELATED WORK

The early research in intermittent computing mostly focused on different methods to ensure that programs will eventually terminate (forward progress). To do so, snapshots of volatile data are regularly saved into non-volatile memory to keep track of the program progress. Some solutions, called *dynamic* checkpointing [10,11,23,24] rely on hardware components to monitor the available energy, and trigger a save operation right before a power failure. Other techniques focused on *static* checkpointing [8,9,18,25]–[27] consisting of inserting backup operations at regular intervals in the program binary. Dynamic checkpointing techniques feature important adaptability to the

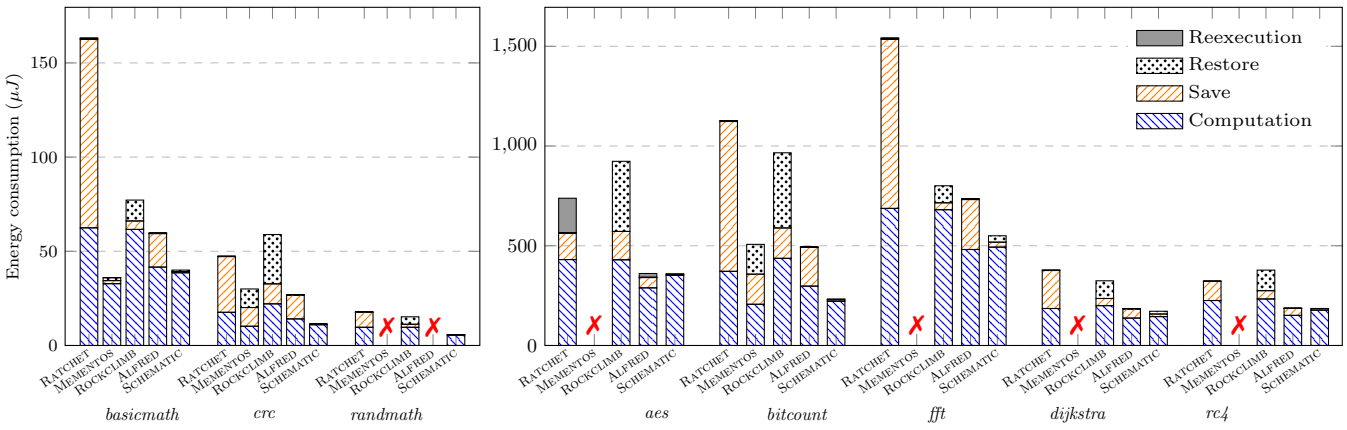

Fig. 6. Overall energy consumption of the eight benchmarks for the different techniques and a TBPF of 10k cycles. Benchmarks that could not be completed are marked with a red cross (✗).

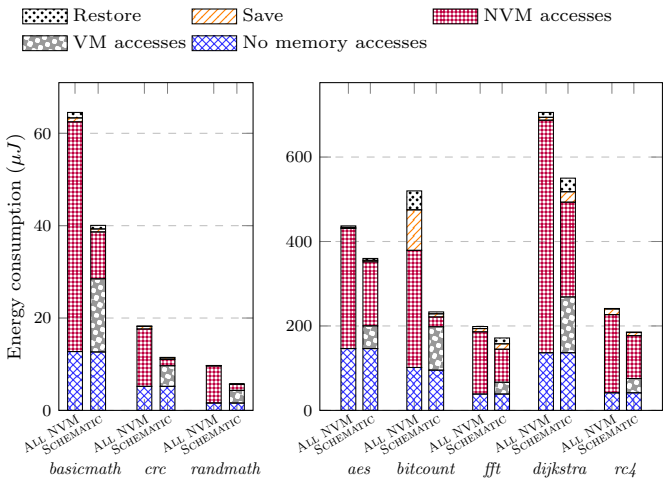

Fig. 7. Energy consumption of SCHEMATIC and *All NVM* on the benchmark suite (time between power failures of 10k clock cycles)

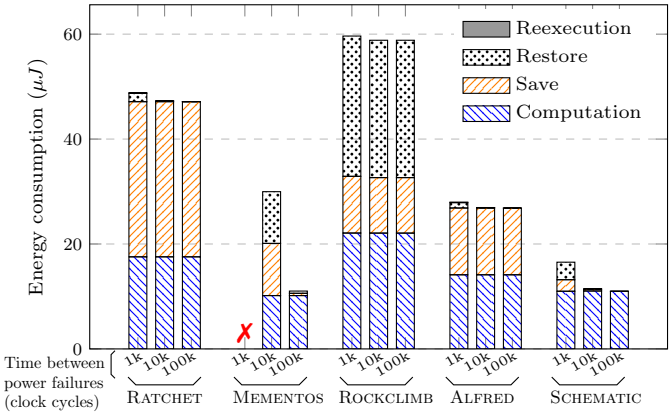

Fig. 8. Impact of capacitor size, for benchmark *crc*. For implementation simplicity on the ScEpTIC simulator, we do not directly vary E_B but vary the resulting TBPF, as a small E_B results in a small TBPF.

varying harvesting conditions, at the cost of an important run-time overhead. On the opposite, static checkpointing techniques come with low run-time overhead, as they allow precise control over the data to be checkpointed. Furthermore, static checkpointing techniques support atomic sections, which make them suitable for peripheral handling. SCHEMATIC falls in the category of static checkpointing techniques, and takes benefit of compiler insight about the code structure and its energy consumption.

Many techniques exist to save volatile data. Some techniques copy entire memory regions [8,11] while others are more precise. Being more precise nonetheless requires either to dynamically keep track of the memory usage [28]–[31], to rely on user annotations [32], or to perform compile-time analysis [9,17,33]. SCHEMATIC avoids saving entire memory regions using a simple compile-time analysis of variable liveness.

Early solutions often selected a single class memory as working memory: either VM [8,11,34] or NVM [9,10]. Using VM as working memory allows faster and more energy-efficient computations but comes at an important checkpointing overhead. On the other hand, NVMs do not need to be checkpointed but can be subject to memory anomalies: upon a restart after a power failure, a portion of the program may be re-executed and re-apply operations on NVM data [13,14], such as incrementing a variable twice. To mitigate this issue, many solutions have been developed [9,35].

Recently, Choi *et al.*, proposed a method called ROCKCLIMB [18] which prevents the consequences of code re-execution, by guaranteeing that no power failure can happen during program execution. To do so, ROCKCLIMB places checkpoints at compile-time such that from one checkpoint it is possible to reach the next checkpoint(s) with the energy of a full capacitor. Then, at run-time, ROCKCLIMB shuts down the platform when a checkpoint is reached, and resumes execution only when the capacitor is full. ROCKCLIMB and SCHEMATIC rely on the same technique to avoid memory anomalies. However, SCHEMATIC benefits from energy reduction due to

its dual VM/NVM memory allocation and its efficient loop and function handling.

So-called *task-based* solutions use both VM and NVM. They leverage compile-time variable liveness analysis [36,37] or user-provided annotations [32,33,38,39] to determine which variables are task-local and which ones are shared between tasks. The variables used only in a task are then allocated in VM, whereas the variables shared between tasks are allocated to NVM. While task-based techniques result in important energy savings, they do not not take into account the limited size of the VM. Moreover, some shared variables, because they are referenced frequently, would deserve to be allocated in VM. This observation made Jayakumar *et al*., propose an Energy-Aware memory mapping method for NVM-VM systems [40]. The proposed method evaluates, for each function, where to allocate the data and stack sections (VM or NVM), using energy measurements. Compared to SCHEMATIC, the allocation of entire sections is coarse-grain and therefore does not allow for variable-level allocations.

Maioli and Mottola recently developed ALFRED [17]. ALFRED works on a program already instrumented with a static checkpointing technique, and determines at compile-time where variables should be allocated (VM or NVM). It also introduces deferred and anticipated checkpointing by saving variables on their last access preceding a checkpoint and restoring them on their first access after a checkpoint. ALFRED uses the energy-efficient VM as much as possible, while enabling forward progress using NVM. However, memory allocation in ALFRED assumes that VM is large enough to store all variables. In contrast, SCHEMATIC accounts for the limited size of VM when deciding on variable allocation, and further performs memory allocation jointly with checkpoint placement.

VI. DISCUSSION

SCHEMATIC, as many intermittent computing systems [11, 18,41], relies on the energy buffer characteristics to ensure the system safety (forward progress and absence of memory anomalies, *cf*. II-B). However, the capacity of the energy buffer may change over time for a given capacitor due to aging or temperature variations [16]. Fortunately, models exist to reflect the impact of aging/temperature on capacitor size [42]. Such models can be used to: (i) select a capacitor size that will be valid for a certain period of time (resp. temperature range); (ii) program an over-the-air update [43] when the selected capacitor size is not valid anymore.

It is important to note that underestimating the energy buffer capacity or overestimating the WCEC results in conservative checkpoint placement but does not impact the device operability. Furthermore, any energy surplus resulting from the conservative approach will shorten the duration of the hibernation phase.

In the event of a power failure occurring between two checkpoints, our technique detects that it restarted from the same checkpoint twice and restores the system to its initial state. If such events occur frequently over time, one could recalculate checkpoint placement using a smaller capacitor size and perform an over-the-air update [43].

In this work we focused on the CPU energy consumption, but our technique could be extended to incorporate peripheral devices in the analysis. To achieve this, we would require WCEC estimation techniques that consider peripheral devices [44] and we would need the user to delimit atomic sections for code using peripheral devices, in which checkpoint placement would be forbidden.

Note on the usage of harvested energy: Since the rate at which energy builds up in the capacitor is by nature unpredictable, SCHEMATIC does not consider harvested energy but only the maximum *available energy* in the capacitor (E_B). Nevertheless, energy harvesting benefits SCHEMATIC, as it reduces the amount of time spent in sleep mode, waiting for the capacitor to replenish.

VII. CONCLUDING REMARKS

This paper presents SCHEMATIC, a compiler technique that automates checkpoint placement and memory allocation to minimize energy consumption in intermittent computing systems. SCHEMATIC ensures forward progress, and adapts checkpoint placement and memory allocation to architectural parameters (size of the energy buffer, volume of volatile memory). An experimental evaluation of SCHEMATIC shows that it is capable of executing benchmarks in situations where other solutions could not (short intervals between failures, small volatile memories). Experiments also shows that it allows significant energy reductions compared to existing systems.

As future work, one direction is to reduce the volume of data saved at each checkpoint by improving the liveness analysis of variables performed in SCHEMATIC. Pointer management could also be improved to allow the pointed variables to change their memory allocation using static point-to analysis. Finally, a direction would be to take advantage of the low-energy sleep modes of the MSP430, to avoid saving the entire volatile data (variables and registers) at every checkpoint.

ACKNOWLEDGEMENTS

This work has received a French government support granted to the Labex Cominlabs excellence laboratory and managed by the National Research Agency in the "Investing for the Future" program under reference ANR-10-LABX-07-01.

REFERENCES

[1] I. Zalbide, E. D'Entremont, A. Jiménez, H. Solar, A. Beriain, and R. Berenguer, "Battery-free wireless sensors for industrial applications based on UHF RFID technology," in *IEEE SENSORS*, 2014. DOI: 10.1109/ICSENS.2014.6985299

[2] L. Wang, Y. Yang, D. K. Noh, H. K. Le, J. Liu, T. F. Abdelzaher, and M. Ward, "AdaptSens: An Adaptive Data Collection and Storage Service for Solar-Powered Sensor Networks," in *30th IEEE Real-Time Systems Symposium*, 2009. DOI: 10.1109/RTSS.2009.8

[3] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *ACM Conference on Embedded Network Sensor Systems*, ser. SenSys, 2017. DOI: 10.1145/3131672.3131699

[4] B. T. Malik, V. Doychinov, A. M. Hayajneh, S. A. R. Zaidi, I. D. Robertson, and N. Somjit, "Wireless Power Transfer System for Battery-Less Sensor Nodes," *IEEE Access*, 2020. DOI: 10.1109/ACCESS.2020.2995783

[5] S. Priya and D. J. Inman, *Energy harvesting technologies*. Springer, 2009, vol. 21.

[6] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities," in *2nd Summit on Advances in Programming Languages (SNAPL)*, 2017. DOI: 10.4230/LIPIcs.SNAPL.2017.8

[7] M. Surbatovich, B. Lucia, and L. Jia, "Towards a formal foundation of intermittent computing," *Proc. ACM Program. Lang.*, 2020. DOI: 10.1145/3428231

[8] 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*, ser. ASPLOS XVI, 2011. DOI: 10.1145/1950365.1950386

[9] 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. DOI: 10.5555/3026877.3026880 pp. 17–32.

[10] 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 *27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, 2014. DOI: 10.1109/VLSID.2014.63

[11] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *Embedded Systems Letters, IEEE*, 2015. DOI: 10.1109/LES.2014.2371494

[12] *MSP430FR5969 datasheet*, Texas Instruments, 2012.

[13] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui, "Discovering the Hidden Anomalies of Intermittent Computing," in *International Conference on Embedded Wireless Systems and Networks*, ser. EWSN, 2021. DOI: 10.5555/3451271.3451272

[14] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Workshop on Memory Systems Performance and Correctness*, ser. MSPC, 2014. DOI: 10.1145/2618128.2618136

[15] A. Riaz, M. R. Sarker, M. H. M. Saad, and R. Mohamed, "Review on comparison of different energy storage technologies used in micro-energy harvesting, WSNs, low-cost microelectronic devices: Challenges and recommendations," *Sensors*, 2021. DOI: 10.3390/s21155041

[16] A. Gupta, O. P. Yadav, D. DeVoto, and J. Major, "A Review of Degradation Behavior and Modeling of Capacitors," in *ASME International Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Microsystems*, 2018. DOI: 10.1115/IPACK2018-8262

[17] A. Maioli and L. Mottola, "ALFRED: Virtual Memory for Intermittent Computing," in *ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys, 2021. DOI: 10.1145/3485730.3485949

[18] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022. DOI: 10.1109/RTAS54340.2022.00012

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004. DOI: 10.1109/CGO.2004.1281665

[20] A. Maioli, "Sceptic Repository." [Online]. Available: https://bitbucket.org/neslabpolimi/sceptic/

[21] B. Yarahmadi and E. Rohou, "So Far So Good: Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code," in *24th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2021. DOI: 10.1145/3493229.3493300

[22] Matthew Hicks, "MiBench2: MiBench benchmark suite ported for IoT devices." 2016.

[23] H. Williams, M. Moukarzel, and M. Hicks, "Failure Sentinels: Ubiquitous Just-in-time Intermittent Computation via Low-cost Hardware Support for Voltage Monitoring," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021. DOI: 10.1109/ISCA52012.2021.00058

[24] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems," in *54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2021. DOI: 10.1145/3466752.3480102

[25] J. Choi, Q. Liu, and C. Jung, "CoSpec: Compiler Directed Speculative Intermittent Computation," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. DOI: 10.1145/3352460.3358279

[26] M. Zhao, Q. Li, M. Xie, Y. Liu, J. Hu, and C. J. Xue, "Software assisted non-volatile register reduction for energy harvesting based cyber-physical system," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[27] M. Zhao, C. Fu, Z. Li, Q. Li, M. Xie, Y. Liu, J. Hu, Z. Jia, and C. J. Xue, "Stack-Size Sensitive On-Chip Memory Backup for Self-Powered Nonvolatile Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017. DOI: 10.1109/TCAD.2017.2666606

[28] N. A. Bhatti and L. Mottola, "Efficient State Retention for Transiently-powered Embedded Sensing," in *International Conference on Embedded Wireless Systems and Networks*, ser. EWSN, 2016. ISBN 978-0-9949886-0-7

[29] J. Choi, H. Joe, Y. Kim, and C. Jung, "ELASTIN: Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution," *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019. DOI: 10919/91451

[30] F. A. Aouda, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to NVRAM for transiently-powered systems," in *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014. DOI: 10.1109/ReCoSoC.2014.6861359

[31] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "MPU-based incremental checkpointing for transiently-powered systems," in *23rd Euromicro Conference on Digital System Design (DSD)*, 2020. DOI: 10.1109/DSD51259.2020.00025

[32] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, 2016. DOI: 10.1145/2983990.2983995

[33] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *OOPSLA*, 2017. DOI: 10.1145/3133920

[34] W. S. Lim, C.-H. Tu, C.-F. Wu, and Y.-H. Chang, "iCheck: Progressive Checkpointing for Intermittent Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. DOI: 10.1109/TCAD.2020.3046571

[35] M. Hicks, "Clank: Architectural support for intermittent computation," in *44th Annual International Symposium on Computer Architecture*, ser. ISCA, 2017. DOI: 10.1145/3079856.3080238

[36] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. ISBN 978-1-939133-08-3 pp. 129–144.

[37] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *SIGPLAN Not.*, 2015. DOI: 10.1145/2813885.2737978

[38] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *ACM/SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2019. DOI: 10.1145/3314221.3314583

[39] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Dynamic Task-based Intermittent Execution for Energy-harvesting Devices," *ACM Trans. Sen. Netw.*, 2020. DOI: 10.1145/3360285

[40] 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. DOI: 10.1145/2983628

[41] B. Yarahmadi and E. Rohou, "Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2020. DOI: 10.1007/978-3-030-60939-9_12

[42] F. Perisse, P. Venet, G. Rojat, and J. M. Rétif, "Simple model of an electrolytic capacitor taking into account the temperature and aging time," *Electrical Engineering*, 2006. DOI: 10.1007/s00202-004-0265-z

[43] D. Wu, L. Lu, M. J. Hussain, S. Li, M. Li, and F. Zhang, "$\mathcal{R}^3$: Reliable over-the-air reprogramming on computational RFIDs," *ACM Transactions on Embedded Computing Systems (TECS)*, 2017. DOI: 10.1145/3070720

[44] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems," *ECRTS*, 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.24