EXTREMIS: Static Frequency Switching for Battery-less Devices

Veronica Rovelli
Politecnico di Milano
Italy
veronica.rovelli@mail.polimi.it

Andrea Maioli
Politecnico di Milano
Italy
andrea1.maioli@polimi.it

Luca Mottola
Politecnico di Milano
Italy
luca.mottola@polimi.it

ABSTRACT

We present EXTREMIS, a compile-time pipeline that improves energy consumption of battery-less devices by ensuring that memory operations occur at the most efficient device frequency setting. Different memory operations incur different energy consumption depending on a device's current operating frequency. Volatile memory operations, for example, are generally most efficient at the highest frequency, whereas non-volatile memory operations may require wait cycles that make lower frequency setting more energy savvy. EXTREMIS reorders the instructions without violating data dependencies and inserts instructions to change the operating frequency depending on program flow and memory access patterns, reconciling their energy overhead with the gains they possibly enable. This is achieved by solving a series of optimization problems at compile-time. Our evaluation shows that, compared to a static frequency setting, EXTREMIS reduces a program's energy consumption by up to 11%, without incurring in any extra cost.

CCS CONCEPTS

• **Computer systems organization → Embedded software**.

KEYWORDS

Frequency scaling, non-volatile memory, intermittent computing.

ACM Reference Format:
Veronica Rovelli, Andrea Maioli, and Luca Mottola. 2024. EXTREMIS: Static Frequency Switching for Battery-less Devices. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Battery-less embedded sensing devices unlock new deployment scenarios [3, 14, 21, 37] while reducing environmental impact and maintenance costs [7]. However, ambient energy sources are irregular and provide limited energy [12] and devices frequently experience unpredictable energy failures. Computations become *intermittent* [7]: executions unfold between periods of active computation and periods to recharge energy buffers.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Energy failures cause devices to lose the content of volatile memories. To ensure forward progress, devices must periodically save their computational state onto Non-Volatile Memory (NVM) and restore it after energy failures [10, 11, 13, 27, 28, 34, 41]. Mixed-volatile platforms [22] facilitate state-retention operations, as they feature directly addressable NVMs.

Problem. Ensuring battery-less devices operate in the most energy-efficient setting is crucial. One key operating parameter is *operating frequency* [5]. Existing approaches dynamically scale this parameter depending on available energy [4, 8, 9, 17]. As we shown in Sec. 2, they consider the highest operating frequency as optimal, as this generally results in lower energy consumption per clock cycle.

Existing works, however, overlook the difference between MCU execution and accesses to peripherals and NVM. MCUs generally run faster than the latter. Accesses to peripherals or NVM while the MCU runs faster require *wait cycles*, wasting time and energy. Consequently, the highest operating frequency is not always the most efficient setting, as a lower operating frequency that incurs no wait cycle may improve energy consumption for specific operations.

We focus on NVM accesses. Battery-less devices can execute three types of memory operations: volatile memory accesses, non-volatile memory accesses, and operations on registers. These operations have different energy consumption due to different access speeds. Consider the MSP430FR2355 [23] MCU, which features a register-configurable operating frequency up to $24MHz$ and FeRAM as built-in NVM. Accesses to SRAM complete within the same cycle they execute. The $24MHz$ setting is the most efficient operating frequency for this, consuming $201pJ$ per access at $3V$. Instead, FeRAM has a maximum access speed of $8MHz$ and accesses at higher frequencies require *wait cycles*, as the FeRAM controller lacks a data cache to mitigate access latency [24]. At $24MHz$, one access to FeRAM costs $604pJ$ in contrast to $320pJ$ at $8MHz$.

Statically selecting a single operating frequency for the entire program thus results in sub-optimal performance, as some operations necessarily execute at a non-optimal setting. In contrast, dynamically adapting the frequency setting requires information on the operation being executed, which existing techniques do not account for, as they mainly focus on available energy. Further, switching frequency at runtime introduces an energy overhead, as the device needs to execute additional instructions necessary to alter the value of frequency-regulating registers. For example, the MSP430FR2355 requires three operations to do so.

Contribution. We design EXTREMIS (**E**fficient e**X**plorations and **T**transformations for f**RE**quency opti**M**izations in **I**ntermittent **S**ystems): a compile-time pipeline that analyzes the program's instructions, evaluate all possible instructions reorderings, and group together the instructions with the same optimal operating

frequency without altering data dependencies. The key idea is maximizing the number of instructions executed at their optimal frequency while minimizing the number of frequency-switching operations. Sec. 3 describes how we achieve this by means of a series of integer linear programming (ILP) problems that account for available operating frequencies and their energy consumption, NVM access latency, and switching costs.

We implement a prototype of EXTREMIS and we evaluate its performance in energy consumption and workload completion time using ScEpTIC [30, 32], a state-of-the-art emulation environment for intermittent systems. We compare EXTREMIS against two static frequency configurations; one minimizes the energy consumption of NVM accesses and the other does the same for volatile memory accesses. As we articulate in Sec. 4, EXTREMIS demonstrates to energy gains by up to 11%, while reducing the time to complete the given workload and without introducing extra costs.

2 RELATED WORK

Research in intermittent computing concentrates on efficiently ensuring forward progress, either using checkpointing techniques [10, 11, 13, 25, 29, 34] or task-based programming abstractions [16, 27, 28, 41]. Techniques to slice or reduce the size of program state [6, 29, 40] or efficiently map memory accesses to different memory types also exist [31]. Our work complements these efforts, as it reduces energy consumption *during* program execution.

Works improving the energy consumption of running code also exist. These are not necessarily specific to intermittent computing but may be applied in this domain nonetheless. They vary device duty cycles and MCU operating modes [33, 38, 39], use maximum power point tracking to maximize harvested energy [9, 36], or dynamically tune devices operating voltage and frequency to ensure optimal performance [8, 17]. Other techniques [26, 33, 38] ensure devices achieve energy-neutrality by tuning sensors sampling rates [33] and data transmit rates [38].

Closest to our efforts are dynamic frequency and voltage scaling techniques for intermittent computing [4]. Because of the rapid sweeps of the MCU voltage operating range that occur at every active cycle, better energy efficiency may be achieved if the MCU operates at the most efficient frequency setting based on the current capacitor voltage [5]. Custom hardware is necessary to achieve this functionality, coupled with a dedicated software driver, to compensate the lack of built-in hardware support [4].

The unique trait of our work is to capture how the most efficient frequency setting changes depending on the kind of memory operation, independent of current capacitor charge. This orthogonal dimension is not considered in other works and makes our efforts largely complementary compared with existing literature.

3 EXTREMIS

As we argue in Sec. 1, the optimal program operating frequency changes throughout the computation, depending on memory access patterns. We call f_{base} the operating frequency that, when statically set for the entire program execution, results in best energy performance, whereas we call f_{perf} the operating frequency that can possibly produce a performance gain for specific instructions, yet is globally sub-optimal. EXTREMIS sets f_{base} as the default frequency and temporarily switches to f_{perf} whenever convenient.

3.1 Problem

We express the energy required to execute a sequence of instructions S at frequency f as

$$E(S,f) = \sum_{x \in \{\text{vm,nvm,reg}\}} n_x \cdot e_x(f) \cdot (1 + wc(x,f)) \quad (1)$$

where x represents the operation type, that is, *vm* and *nvm* for instructions accessing volatile and non-volatile memory, respectively, and *reg* for register operations; n_x is the number of instructions of type x in sequence S; $e_x(f)$ is the energy for executing an operation of type x at frequency f; and $wc(x,f)$ is the number of additional wait cycles required to execute an operation of type x at frequency f. Usually, $wc(x,f) > 0$ only if $x = nvm$ and f is greater than the maximum frequency supported by NVM, as battery-less devices usually lack a data cache for non-volatile memory [22, 24].

Given a sequence of instructions S executing at f_{base}, we execute it at f_{perf} only if the performance gain exceeds the energy costs of frequency switching. This requires setting a series of registers and can be computed as $e_{switch}(f) = n_{switch} \cdot e_{reg}(f)$, where n_{switch} is the number of operations required to switch the frequency and f is the frequency when the switch occurs. This cost is paid twice, one to set the frequency to f_{perf} and one to set it back to f_{base}, as this is the default frequency the programs run at. Therefore, we switch the operating frequency whenever

$$E(S,f_{base}) - E(S,f_{perf}) > e_{switch}(f_{perf}) + e_{switch}(f_{base}) \quad (2)$$

Evidence shows that programs include very short sequences of operations of the same type [40], and sequences containing operations of different types may result in sub-optimal performance when executing either at f_{base} or f_{perf}, as a frequency is optimal for only one operation type. Despite the simplicity of Eq. 2, isolating sequences of instructions that lead to a performance increase through frequency switching is non-trivial.

We reorder instructions and place frequency switching operations to maximize the number of instructions executing at their optimal setting. Next, Sec. 3.2 describes the EXTREMIS compile-time pipeline; Sec. 3.3 and Sec. 3.4 illustrate step⟨2⟩ and step⟨3⟩.

3.2 Compile-time Pipeline

Fig. 1 shows the compile-time pipeline of EXTREMIS, which is independent of forward progress mechanisms and target architecture. EXTREMIS only requires support for dynamic frequency switching through dedicated instructions, such as altering registers [22]. It takes two inputs: a platform model and a program. The platform model specifies the supported operating frequencies, the instructions controlling the operating frequency setting, the maximum speed of each memory type, and an energy model of the different

Profile Tag	Meaning
E_LOOP(*n*)	The loop usually executes at least *n* times.
E_BRANCH(*r*)	The branch execution ratio is *r*.
E_SAVE(*name*, *fixed*)	The function *name* saves the program state; *fixed* specifies if basic blocks containing the routine can be reordered.
E_RESTORE(*name*)	The function *name* restores the program state.

Table 1: EXTREMIS profile tags.

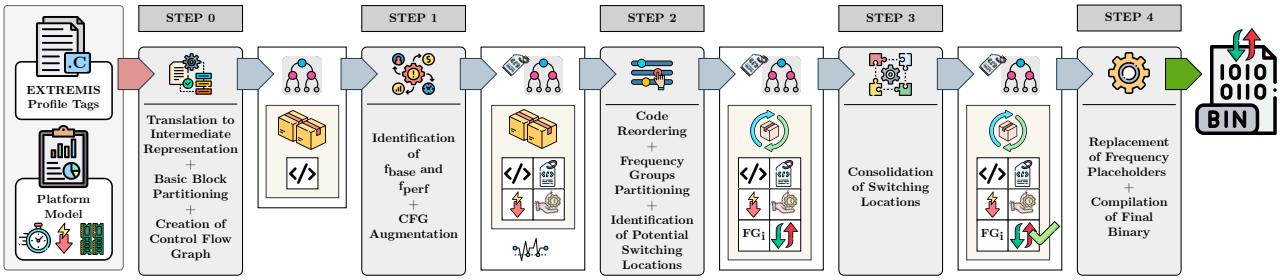


Figure 1: EXTREMIS compile-time pipeline.

device operations for every supported operating frequency. This information is normally found in platform data sheets.

We ask developers to instrument the source code with EXTREMIS profile tags, shown in Tab. 1. These are macros to specify program-specific information, consisting of the minimum number of expected iterations of a loop (*E_LOOP*) and the execution ratio of each branch of conditional statements (*E_BRANCH*). This information may be obtained through code profiling or static analysis [5].

At step⟨0⟩, EXTREMIS transforms the program source code into an intermediate representation close to machine code, yet with no architecture-specific elements, such as register names. During this step, EXTREMIS partitions the program into basic blocks, consisting of sequences of operations with a single entry and exit point and builds the program control flow graph (CFG).

At step⟨1⟩, we determine f_{reg_vm} as the most efficient frequency for register operations and volatile memory accesses, and f_{nvm} as the maximum operating frequency for NVM accesses with no wait cycles. EXTREMIS analyzes the energy cost of each basic block at f_{reg_vm} or f_{nvm}, and uses platform model information and Eq. 1 to determine the default program operating frequency f_{base} between f_{reg_vm} or f_{nvm}. The other frequency is set to f_{perf} and is used to improve performance. If f_{base} is higher (lower) than f_{perf}, the next steps evaluate whether decreasing (increasing) the operating frequency. Based on this, EXTREMIS augments the program CFG with metadata including the cost of each instruction at f_{base} or f_{perf} and profiling information retrieved from profile tags.

Step⟨2⟩ identifies *potential* locations for frequency switch operations. For every basic blocks, we maximize instructions executing at their most efficient operating frequency while minimizing frequency switching costs. Fig. 2 shows an example, where f_{perf} is the optimal frequency of blue instructions, whereas f_{base} is the optimal frequency of the red ones. For simplicity we only show memory accesses. EXTREMIS reorders instructions to group together those with the same optimal operating frequency without affecting data dependencies. In Fig. 2, EXTREMIS moves instruction 2 before instruction 4 and instruction 7 after instruction 4, retaining the data dependencies between instructions $1-8$ and $5-6$.

Next, EXTREMIS partitions the reordered instructions into *frequency groups* $(FG_1, ..., FG_n)$, consisting of instructions that execute at the same operating frequency. In Fig. 2, EXTREMIS partitions the instructions into three frequency groups. Any two successive frequency groups FG_i and FG_{i+1} execute at two different frequencies, or their instructions would be in the same frequency group. Between FG_i and FG_{i+1}, EXTREMIS inserts a frequency switch *placeholder* p_i, which indicates a location where to switch the frequency. Frequency p_i is the optimal frequency of the subsequent frequency group FG_{i+1}. In Fig. 2, EXTREMIS inserts two placeholders, p_1 and p_2 to switch the frequency to f_{perf} and f_{base}, respectively.

To ensure each basic block starts and ends at the default operating frequency f_{base}, EXTREMIS ensures there is an odd number of frequency groups. When the optimal frequency of the first frequency group FG_1 is f_{perf}, EXTREMIS inserts a placeholder before FG_1 to switch to f_{perf}. After the last frequency group FG_n when f_{base} is not its optimal operating frequency, EXTREMIS similarly inserts a placeholder to switch back to f_{base}. This ensures each basic block always starts at f_{base}. The resulting partitioning ensures that each frequency group FG_i execute at f_{base} (f_{perf}) if i is odd (even).

EXTREMIS achieves the result in Fig. 2 by solving an ILP problem that evaluates all possible reorderings of basic block's instructions that do not invalidate data dependencies, inserts placeholders at program's locations where a frequency switch increases performance using Eq. 2, and computes the resulting execution cost using Eq. 1. For each basic block, EXTREMIS keeps the solution that minimizes the execution energy, consisting of the most efficient frequency configuration. Further details are available in Sec. 3.3.

Step⟨3⟩ aims at fine-tuning the output of step⟨2⟩, removing redundant placeholders. To do so, we analyze the program at a higher granularity than step⟨2⟩ and target all pairs of frequency groups (FG_x, FG_y) such that FG_x is the last frequency group of a basic block BB_i and FG_y is the first frequency group of the subsequent basic block BB_{i+1}. Both FG_x and FG_y necessarily execute at f_{base}. The reasoning here is opposite compared to step⟨2⟩: we remove the frequency switch operations after FG_y and before FG_x whenever they produce a cost higher than the additional energy of executing FG_x and FG_y at f_{perf} instead. This entails checking that

$$2 * c_{switch}(f_{base}) > C(FG_x, f_{perf}) - C(FG_x, f_{base}) + \\ + C(FG_y, f_{perf}) - C(FG_y, f_{base}) \quad (3)$$

Fig. 3 shows an example of a program after step⟨2⟩. The start of basic block BB_2 has a better performance at f_{perf} and therefore its frequency group FG_1 is empty. We consider FG_3 of BB_1 as FG_x and FG_1 of BB_2 as FG_y. As a result, in step⟨3⟩ we remove the switching placeholders p_2 of BB_1 and p_1 of BB_2, as their cost is higher than executing the instructions contained in FG_3 of BB_1 and FG_1 of BB_2 at f_{perf}. Further details are in Sec. 3.4.

At the end of step⟨3⟩, the placeholders left in the code represent the *final* locations of frequency switching. At step⟨4⟩, EXTREMIS replaces the placeholders with the actual frequency switching instructions and compiles the program.

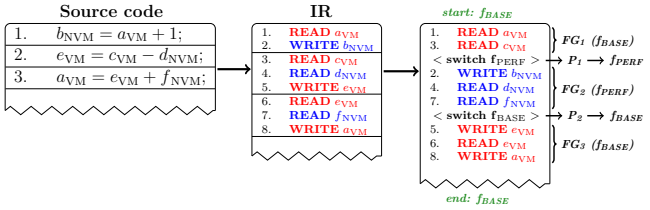


Figure 2: Code reordering example at step⟨2⟩.

3.3 Identifying Frequency Groups

We formulate the ILP problem that identifies the location of frequency switch operations for every basic block.

Problem formulation. Each frequency group FG_i executes at f_{base} (f_{perf}) if i is odd (even) and no two consecutive frequency groups execute at the same frequency. Without loss of generality, the ILP problem is formulated to create the even-indexed frequency groups FG_x, accounting for one frequency switch before their execution to set the frequency to f_{perf} and another one after their execution to set the frequency back to f_{base}. The odd-indexed frequency groups consequently execute at f_{base}. We consider the following quantities:

- n: number of instructions in the basic block.
- k: number of data dependencies in the basic block.
- dep: data dependency tuples; a tuple (x, y) indicates that instruction y requires data computed by x.
- $\texttt{ev}_\texttt{i}$: energy variation of instruction i when executed at f_{perf} or f_{base} computed as $e(i, f_{perf})$ - $e(i, f_{base})$.
- sc: cost of one switch operation to set f_{perf} and one to set it back to f_{base} computed as $e_{switch}(f_{base}) + e_{switch}(f_{perf})$.

Consider how a negative (positive) value of $\texttt{ev}_\texttt{i}$ indicates an increase (degrade) in performance when switching frequency from f_{base} to f_{perf} for that specific instruction, excluding the cost for switching frequency that represents the overhead necessary to this end. We define the following decision variables:

(1) $\texttt{fg}_\texttt{i}$: index of the frequency group where the i-th instruction belongs to, which represents the output of the ILP that EXTREMIS uses to partition a basic block into frequency groups and place frequency switch operations accordingly.

(2) n_even_fg: the number of even frequency groups, that is

$$n\_even\_fg = \sum_i 1, \forall i = 1, ..., n \mid fg_i \; mod \; 2 == 0 \tag{4}$$

(3) total_ev: performance variation for the basic block, that is

$$total\_ev = \sum_i ev_i + n\_even\_fg \cdot sc, \forall i = 1, ..., n \mid fg_i \; mod \; 2 == 0 \tag{5}$$

Being $\texttt{cv}_\texttt{i}$ negative when cost decreases, our goal is to identify the lowest possible total_cv, provided data dependencies are respected. Therefore, the ILP problem objective function is:

$$\begin{aligned} &minimize \;\; total\_cv \\ subject\; to \;\; &fg_x \leq fg_y, \forall (x, y) \in dep \\ &fg_i \geq 1 \; \forall i = 1, ..., n \end{aligned} \tag{6}$$

Special cases. Additional care is required with function calls. They may access global variables or data in the callee stack frame, potentially introducing data dependencies with the instructions in the basic block of the function call. We account for this when computing the *dep* array by analyzing functions' body and propagating the resulting data dependency information.

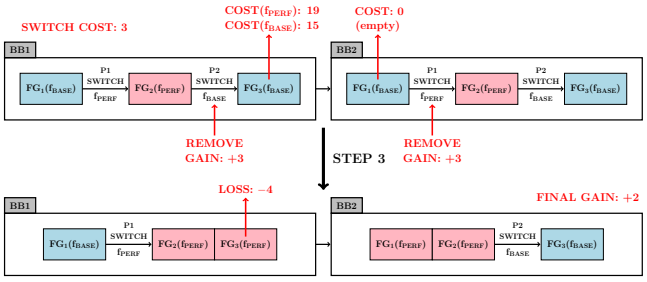


Figure 3: Consolidation example at step⟨3⟩.

Function calls also alter the execution flow. After a call, the next instruction to execute is in a different basic block, whose start frequency is f_{base}. To ensure the frequency is set to f_{base} for function execution, we include the cost of switching to f_{base} before a function call executing at f_{perf} and for the cost of switching back to f_{perf} after function returns. This ensures the frequency switch happens only if it produces an increase of performance, despite the frequency requirements of function execution.

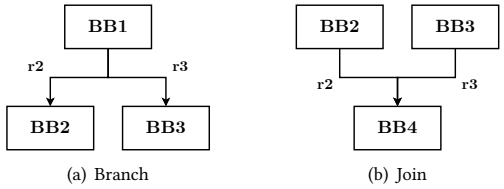
A similar problem arises with checkpointing, where saving the program state occurs through a function call. We do not alter the location of the checkpoint routine and, if specified with the corresponding profile tag, also do not reorder basic blocks containing these operations, so not to invalidate the placement strategies [13, 34, 40]. For just-in-time approaches [10, 11], EXTREMIS collects the current frequency, switches to f_{base} if necessary, executes the checkpoint, and restores the frequency to f_{perf}.

Complexity. Given a basic block of n instructions, in principle, the ILP problem considers $n!$ possible instruction reorderings and 2^{n+1} different frequency switch configurations for each possible reordering. Therefore, the complexity of the ILP problem is $O(n! \cdot 2^n)$. In practice, data dependency greatly limit the number of *valid* instruction reorderings. To further reduce complexity, we introduce two additional constraints. These constraints are *only* instrumental to avoid checking solutions that are necessarily sub-optimal; therefore, they do not impact the optimality of the solution.

The first constraint limits the number of frequency switching operations that may occur in a single basic block, therefore limiting the number of frequency groups considered. The previous ILP formulation, indeed, considers frequency switching to potentially happen after any instruction, leading to n different frequency groups per instruction. Due to switching overhead, the maximum number of switching operations is limited by the energy cost reduction. Therefore, the maximum number of switching operations corresponds to the maximum possible energy cost reduction divided by the cost of switching, that is

$$n\_max\_switch = \lceil \frac{|\sum_i ev_i|}{sc} \rceil \cdot 2, \forall i \mid ev_\texttt{i} < 0 \tag{7}$$

The 2 factor is necessary as *sc* includes the cost of two switching operations. The maximum number of frequency groups is $max\_fg = n\_max\_switch + 1$, as frequency switch happens after every frequency group except the last one.



(a) Branch (b) Join

Figure 4: Examples of splits in control flow.

We also add a constraint that limits the maximum value of fg_i to $min(\text{max\_fg}, n)$. This entails limiting the maximum number of frequency groups to n to address cases where ev_i is higher than sc, which would result in $max\_fg > n$, that is, an essentially meaningless solution. We finally introduce an extra parameter, $\texttt{independent}_\texttt{i}$, which indicates if instruction i is dependent on another instruction, that is, $\texttt{independent}_\texttt{i}$ is 1 if $\nexists j \; s.t. \; (j,i) \in dep$, or 0 otherwise. Independent instructions are instructions with no data dependencies can be placed anywhere in a basic block. To reduce the number of possible instruction reorderings, we conventially place all independent instructions i with $ev_i \geq 0$ in the first frequency group, and limit all independent instructions i with $ev_i < 0$ to the first two frequency groups, as they may improve performance when run at f_{perf}. The analytical formulation is available [35].

3.4 Consolidating Frequency Groups

Programming structures like branches and loops split the control flow. This means different basic blocks may execute after or before others, as Fig. 4 shows for BB_2 and BB_3. We refer to these basic blocks as *parallel* basic blocks.

This affects the analysis to evaluate the removal of frequency switching placeholders according to Eq. 3. In fact, removing the last frequency switch to f_{base} before a split in the control flow affects the starting frequency of all parallel basic blocks, but this may may increase performance at f_{perf} only in a few of them. This is the case of BB_2 and BB_3 in Fig. 4(a). Further, removing the last frequency switch to f_{base} before parallel basic blocks re-join creates a situation where the frequency setting of following basic block depends on what branch the execution is coming from, as for BB_4 in Fig. 4(a).

To avoid reducing program performance, EXTREMIS evaluates control flow structures with parallel basic blocks together, considering the execution ratio of parallel basic blocks provided through profile tags. Consider a split as in Fig. 4(a). EXTREMIS removes the last frequency switching placeholders from the basic block before the parallel ones and the first frequency switching placeholders from all parallel basic blocks if:

$$\begin{aligned} (1+n_p) * e_{switch}(f_{base}) > \\ E(FG_{entry}, f_{perf}) - E(FG_{entry}, f_{base}) + \\ + \sum_i x \cdot (E(FG_{1_i}, f_{perf}) - E(FG_{1_i}, f_{base})), i \in P \end{aligned} \tag{8}$$

where n_p is the number of parallel basic blocks; P is the set of parallel basic blocks; x_i is the execution ratio of the parallel basic block i; FG_{entry} is the last frequency group of the basic block before the parallel basic blocks; and FG_{1_i} is the first frequency group of the parallel basic block i. The situation of Fig. 4(b) where parallel basic blocks joins into a single basic block is dual to Eq. 8.

We apply a similar reasoning to loops, where we consider the minimum number of loop iterations. Further details are available [35].

4 EVALUATION

In the following, Sec. 4.1 discusses our experimental setup and Sec. 4.2 illustrates the experiment results.

4.1 Setup

Reproducing energy harvesting conditions is difficult [15, 19]. Therefore, we opt for system emulation and we evaluate our technique using ScEpTIC [30, 32], an open-source emulation environment for intermittent devices. ScEpTIC emulates the execution of LLVM IR [1], an intermediate representation of the program's source code, and is widely used in existing literature [31].

Platform and tools. We consider the MSP430FR2355 [23] as target platform, a mixed-volatile MCU from the popular MSP430 family [22], equipped with a $100\mu F$ capacitor. We use Hibernus [11] as forward progress technique. We set n_{switch} to 3, as the MSP430 mixed-volatile platform allows developers to regulate the operating frequency using 3 registers [23].

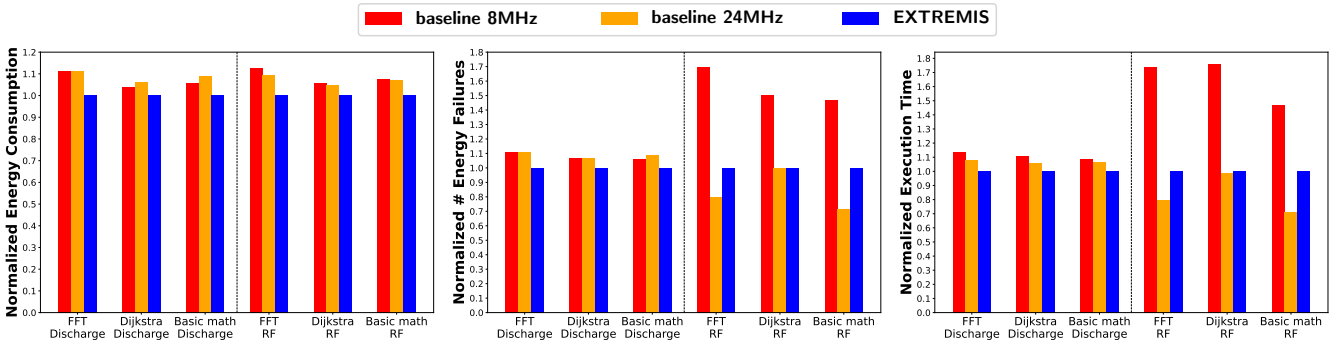
We consider two energy patterns, `RF` and `Discharge`. The former represents energy sources that supply short bursts of energy during device active cycles. We reproduce this behavior using the RF energy source used in Mementos [34]. The latter is a synthetic energy source that represents energy sources with a scarce energy throughput, such as kinetic energy from vibrations [3], which does not supply energy to devices during their active periods [31]. We reproduce this behavior by modeling a synthetic energy source that supply $3.6V$ when the emulated device is powered off.

We model the ILP problem of Sec. 3.3 using MiniZinc [2]. Our prototype analyzes the intermediate representation exposed by ScEpTIC, interfaces with MiniZinc to retrieve the solution of the ILP problem, and places frequency switching operations as described in Sec. 3. In all our experiments, MiniZinc returns a solution in less than 5 minutes, which we set as an upper bound.

Baselines and metrics. We compare EXTREMIS against static frequency configurations, which is how existing system support for intermittent computing operates [11, 27, 34, 40]. We consider static configurations at $24MHz$ and $8MHz$, which are the maximum MCU operating frequency and the maximum frequency supported by the NVM of the MSP430FR2355 [23], respectively. These are the most efficient configurations developers can choose. EXTREMIS considers $8MHz$ as f_{nvm} and $24MHz$ as f_{vm}.

We mainly focus on the energy consumed to complete a given workload. To gain deeper insights, we also collect the number of energy failures occurred before workload completion and the workload completion time. The former helps us understand if a reduction in energy consumption produces a tangible effect, consisting in a reduction of the active cycles required to complete a given workload. The latter provides a numerical indication to understand the impact of this reduction in terms of completion time.

Benchmarks. We use benchmarks from MiBench2 [20], a popular benchmarking suite for intermittent techniques [28, 29, 31]. We consider three benchmarks: `basic math` that runs mathematical operations not directly supported in hardware; `FFT` that computes the discrete Fourier transformation of signals, and `Dijkstra`, which



(a) Energy consumption (b) Number of energy failures (c) Workload completion time

Figure 5: Experiment results with Discharge and RF energy patterns.

runs the Dijkstra algorithm to identify the shortest path between nodes in a graph. We execute 500 iterations of each benchmark.

We produce the LLVM IR of each benchmarks using Clang v8.0.1 [1] with the default optimization level. We consider the memory layout of popular compilers for embedded systems [18, 28], which promote local variables to global variables. Following state-of-the-art solutions [27, 31, 32, 40], we allocate global variables onto NVM and the stack segment onto volatile memory. Instructions are stored onto NVM. The MSP430FR2355 [23] FeRAM controller has a 64-bit instruction cache but no data cache [24]. Therefore, considering the limited cache size and benchmarks' structure, we set ScEpTIC to simulate an instruction cache hit rate of 85% [24].

4.2 Results

Fig. 5 shows the experimental results normalized to EXTREMIS. Most trends depend on the energy pattern.

`Discharge`. The left side of Fig. 5(a) reports the energy consumption of EXTREMIS and selected baselines with the `Discharge` energy pattern. The ability of EXTREMIS to execute memory instructions at their most efficient frequency yields a lower energy consumption. On average, this is 7% lower for EXTREMIS compared to the most efficient baseline of $8MHz$, with a peak of 11% when running `FFT`.

The `Discharge` energy pattern does not supply energy during active cycles. Compared to the baselines, the energy saved allows EXTREMIS to execute more instructions, resulting in fewer energy failures and a lower workload completion time, as the left side of Fig. 5(b) and Fig. 5(c) show, respectively. EXTREMIS experiences 11% fewer energy failures and complete benchmarks up to 13% faster than the static $8MHz$ frequency setting.

`RF`. Experiments with the `RF` energy pattern show a similar trend in energy consumption, as the right side of Fig. 5(a) shows. EXTREMIS demonstrates up to a 11% lower energy consumption than the most efficient baseline, which here the $24MHz$ static frequency setting.

Compared to experiments with `Discharge`, the trends change in the number of energy failures and workload completion time, as the right side of Fig. 5(b) and Fig. 5(c) show, respectively. The performance difference between the baselines is higher compared to the `Discharge` scenario, with $8MHz$ experiencing up to $2x$ more energy failures and a $2x$ higher completion time than the $24MHz$ static setting. This is caused by the additional energy that energy sources with the `RF` energy pattern supply during active cycles.

Wait cycles cause the $24MHz$ static setting to consume 88% more energy than the $8MHz$ one to access NVM, while taking the same amount of time [23]. The $24MHz$ static setting uses the additional energy available during active cycles to cover the cost of NVM operations, while executing volatile memory accesses and register operations more efficiently than the $8MHz$ static setting. EXTREMIS executes these accesses at $8MHz$, saving energy but not time, and requiring additional operations and time to switch to $24MHz$. Compared to $24MHz$, EXTREMIS executes fewer instructions during active cycles when frequency groups contain a heterogeneous mix of volatile and non-volatile memory operations.

This is the case of `FFT` and `basic math`, where EXTREMIS experiences 20% more energy failures than the $24MHz$ static setting, resulting in a 24% longer workload completion time. In `Dijkstra`, EXTREMIS can create more homogeneous frequency groups, reaching a performance comparable to the $24MHz$ static setting. Compared to the $8MHz$ static setting, on average, EXTREMIS experiences 55% fewer energy failures and a 65% shorter workload completion time.

We conclude that EXTREMIS performance is linked to the amount of ambient energy provided during active cycles. The less there is, the more crucial is not to pay for the energy overhead due to wait cycles, if any. Energy sources are irregular and the scenario may switch from something akin to `RF` to something similar to `Discharge` [3]. EXTREMIS inherently adapt to such changes.

5 CONCLUSION

We presented EXTREMIS, a compile-time pipeline that instruments battery-less devices' code to ensure memory operations occur at the most energy-efficient frequency setting. To do so, we reconcile the energy overhead due to executing frequency changes with the gains they possibly enable. EXTREMIS works at compile-time based on the solution to ILP problems that determine where and how to insert instructions to change frequency. Compared to the most energy-efficient static frequency setting, EXTREMIS reduces energy consumption up to 11%, while reducing workload completion time.

Acknowledgments. This work is partially supported by the Swedish Science Foundation (SSF) and by the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3 - Call for tender No. 1561 of 11.10.2022 of Ministero dell'Università e della Ricerca (MUR); funded by the European Union - NextGenerationEU.

REFERENCES

[1] The LLVM Compiler Infrastructure. https://llvm.org/ (last access: September 9th, 2024).

[2] MiniZinc high-level constraint modelling language and solver. https://www.minizinc.org/ (last access: September 9th, 2024).

[3] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys '20)*.

[4] S. Ahmed, Q. Ain, J. H. Siddiqui, L. Mottola, and M. H. Alizai. 2020. Intermittent Computing with Dynamic Voltage and Frequency Scaling. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks (EWSN '20)*.

[5] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[6] S. Ahmed, M. H. Bhatti, N. A. Alizai, J. H. Siddiqui, and L. Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*.

[7] Saad Ahmed, Bashima Islam, Kasim Sinan Yildirim, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola, Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* (2024).

[8] D. Balsamo, A. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Graceful Performance Modulation for Power-Neutral Transient Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[9] D. Balsamo, B. J. Fletcher, A. S. Weddell, G. Karatziolas, B. M. Al-Hashimi, and G. V. Merrett. 2019. Momentum: Power-Neutral Performance Scaling with Intrinsic MPPT for Energy Harvesting Computing Systems. *ACM Transactions on Embedded Computing Systems* (2019).

[10] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[11] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[12] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[13] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[14] Q. Chen, Y. Liu, G. Liu, Q. Yang, X. Shi, H. Gao, L. Su, and Q. Li. 2017. Harvest Energy from the Water: A Self-Sustained Wireless Water Quality Sensing System. *ACM Transactions on Embedded Computing Systems* (2017).

[15] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. *SIGOPS Operating Systems Review* (2016).

[16] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[17] Benjamin J. Fletcher, Domenico Balsamo, and Geoff V. Merrett. 2017. Power Neutral Performance Scaling for Energy Harvesting MP-SoCs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*.

[18] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. The nesC language: A holistic approach to networked embedded systems. *Acm Sigplan Notices* (2003).

[19] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*.

[20] M. Hicks. MiBench2 - MiBench porting to IoT devices. https://github.com/impedimentToProgress/MiBench2 (last access: September 9th, 2024).

[21] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[22] Texas Instruments. MSP430 family of MCUs. https://www.ti.com/msp430 (last access: September 9th, 2024).

[23] Texas Instruments. MSP430-FR2355 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr2355.pdf (last access: September 9th, 2024).

[24] Texas Instruments. MSP430 FRAM controller datasheet. https://www.ti.com/lit/an/slaa498b/slaa498b.pdf (last access: September 9th, 2024).

[25] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[26] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. 2007. Power Management in Energy Harvesting Sensor Networks. *ACM Transactions on Embedded Computing Systems* (2007).

[27] B. Lucia and B. Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[28] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[29] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[30] A. Maioli. ScEpTIC documentation and source code. http://sceptic.neslab.it/ (last access: September 9th, 2024).

[31] A. Maioli and L. Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys '21)*.

[32] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021)*.

[33] S. Peng and C. P. Low. 2012. Throughput optimal energy neutral management for energy harvesting wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*.

[34] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[35] Veronica Rovelli. Energy Eciency in Intermittent Computing Systems through Static Frequency Scaling Techniques. https://hdl.handle.net/10589/223509 2024.

[36] M. M. Sandhu, K. Geissdoerfer, S. Khalifa, R. Jurdak, M. Portmann, and B. Kusy. 2020. Towards Optimal Kinetic Energy Harvesting for the Batteryless IoT. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*.

[37] N. Saoda and B. Campbell. 2019. No Batteries Needed: Providing Physical Context with Energy-Harvesting Beacons. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys)*.

[38] V. Sharma, U. Mukherji, V. Joseph, and S. Gupta. 2010. Optimal energy management policies for energy harvesting sensor nodes. *IEEE Transactions on Wireless Communications* (2010).

[39] L. Spadaro, M. Magno, and L. Benini. 2016. Poster Abstract: KinetiSee - A Perpetual Wearable Camera Acquisition System with a Kinetic Harvester. In *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[40] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[41] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18)*.