A Formal Model for Optimal Autonomous Task Hibernation in Constrained Embedded Systems

C. Brandolese, W. Fornaciari, L. Rucco
Politecnico di Milano – DEIB
Piazza L. Da Vinci, 32 – 20133 Milano, Italy
{brandole,fornacia,rucco}@elet.polimi.it

***Abstract*—This paper proposes and studies an autonomous hibernation technique and optimal hibernation policies aimed at minimizing the power consumption, while allowing stateful processing in constrained embedded systems with long-lasting lifetime requirements. To this purpose the paper models the energy contributions for hibernating the system—by saving the memory status on an external non-volatile memory and completely powering off the system—rather than maintaining the system in a sleep mode with memory retention—with problems of static leakage power—between two consecutive bursts of processing. Thanks to a simplified yet formal notion of system state, the paper rigorously determines the optimal conditions for deciding whether to hibernate or not the system during idle periods. Hibernation policies have been implemented as a module of the operating system and results demonstrate energy savings up to 50% compared to trivial hibernation approaches. Moreover, the hibernation policy proved to be robust and stable with respect to changes of the application parameters.**

I. Introduction

In many applicative scenarios that require embedded devices with long-lasting lifetime, typical of monitoring or self-learning/adaptive systems, the need for low or ultra-low power consumption is an intrinsic requirement. Tasks performed by such systems are often periodic or event-driven with a certain occurrence probability. This implies that the typical operation profile is an alternation between cycles of computation/sensing activities interleaved with relatively long sleeping or inactive periods. Considering this behavior, static leakage power, which critically affects new generation microcontrollers as the integration pushes towards deep nanometers scale, often represents a non negligible contribution to the overall energy consumption during sleeping periods. This often leads to favor ultra-low power and extremely resource-constrained microcontrollers to more flexible and powerful new generation architectures. The key idea is to investigate a trade-off between energy efficiency, on one hand, and flexibility, maintainability and extensibility, on the other. Ultra-low power devices are, in fact, often too constrained in terms of memory and computational resources, require tiny operating systems, non-standard programming languages, lack effective dynamic reprogramming mechanisms and do not support process-based OS architectures. Resorting to high-end or mid-range microcontrollers partly reduces most of these limitations, at the cost of a significant power consumption, even in idle and sleeping periods. The most effective way to avoid this effect is to switch the microcontroller off during idle periods and to resume the system immediately before active cycles, at the cost of losing the memory status of the running processes and tasks. To avoid this effect, the system should be hibernated, by saving the memory image on external non-volatile memories and recreate the status as the system resumes. This entails that the selection of non-volatile external memories must account for both electrical characteristics (read/write energies) and functional limitations (number of writing cycles), because of the potentially high number of swap-out/swap-in operations needed throughout the lifetime of the system. New-breed magnetoresisteve (MRAM), ferroelectric (FeRAM) or phase-changing RAM offer a good solution to both problems thanks to their unlimited number of write cycles and their limited energy consumption. Based on such considerations, we proposed [1] a powerful and energy-efficient hardware platform and operating system for Wireless Sensor Networks (WSN), capable of overtaking some of the highlighted limitations.

Due to the timing and energy overheads required to swap-out the system status, shut-down the operating system and resume it, the optimal choice whether to hibernate the system or put it into sleeping mode—with memory retention—is a very harsh problem, though simpler suboptimal policies exist. The decision, in fact, depends on a broad range of factors, such as the number of active processes, their periods and their sizes in RAM[1], the specific energy levels of the microcontroller and the electrical characteristics of the external memory. The *hibernation policy* itself, moreover, should be autonomous, effective, dynamic, robust, computationally efficient and reliable.

This paper presents the main theoretical achievements concerning the definition of the hibernation policy and the experimental results obtained by stressing the model properties as well as by assessing the performance of its actual implementation. Though the proposed approach is general enough to be adopted in a broad class of embedded systems, we considered for the proof of concept a WSN node since it represents a typical class of resource-constrained devices with long lifetime requirements. The paper is organized as follows: Section II presents an overview on the related work; Section IV describes the hibernation architecture; Section V details the formal model and the hibernation policy, whose properties and effectiveness are discussed and proved in Section VI.

[1]Note that the `.text` section of processes is stored in the main Flash of the micrconotroller, since it does not changes during the execution. The `.data` and `.bss` sections, as well as the stack and the heap of each process are instead stored in RAM and need to be saved on an external non-volatile memory during an hibernation period without main memory retention.

`/13 $26.00 © 2013 IEEE
DOI 10.1109/DSD.2013.50

II. Related Work

A widely explored power management technique in constrained embedded systems—and Wireless Sensor Networks in particular—is duty cycling. According to Anastasi et al. [2], duty-cycling can be classified in Topology Control and Power Management techniques, the latter in turn divided in two main branches: low duty-cycle MAC protocols and sleep/wakeup protocols. We will focus on the last class of approaches, as they try to achieve objectives which can be considered akin to those of the present paper. This class of models can be divided in three schemes, namely on-demand (e.g. [3], [4]), asynchronous (e.g. [5]) and scheduled rendezvous (e.g. [6], [7], [8]). Our approach shows some similarities with the last scheme as far as the operating profile is concerned.

Beyond classical duty cycling problems, which in general consider alternation between wake and sleep cycles with main memory retention[2], there is another critical factor that must be taken into consideration. As microcontroller technology steps down to deep nanometer scale, the increasing value of leakage current in non-active states poses important challenges, especially in presence of long lifetime requirements.

The problem of managing the power consumption of devices through dedicated software layers has already collected more than a decade of history, since the first appearance of works expressly dedicated to the topic [9], [10], [11]. By the time, the research on power management has taken two main directions: one broad vein related to dynamic power management and another, less explored, concerning static leakage power. In the first class of problems fall most of the works on dynamic voltage and frequency scaling, as well as clock-gating techniques. These approaches, however, target dynamic power reduction, which is a topic complementary but orthogonal to that of the present paper.Static leakage power, unfortunately, has lately become an hard problem to the same extent. While significant research effort has been initially directed to dynamically power-off idling components, such as unused cache blocks [12], [13], [14], the specific interest on microcontrollers seems to have been quite upstaged. Considering the main memory and in particular the SRAM, it should be noted that the average leakage power can be up to ten times higher than other parts of logic inside a microcontroller, simply considering the transistors count. To mitigate this problem, memory standby-mode scheduling approaches have been proposed [15][16], accounting also for the scratchpad mapping of most frequently used code. More often, however, in embedded scenarios that need long operational lifetime the problem of static leakage power in sleeping states with memory retention has been simply escaped by resorting to less-consuming, ultra-low power and loose nanometer-scale microcontrollers, often extremely resource constrained. This lead to an harsh trade-off between energy efficiency and purely functional parameters like programming flexibility, usability and interoperability. To the best of our knowledge, though some contribution exist concerning hibernation for general purpose computing machines and servers, no literature exists that tackles this specific issue for high-end microcontroller based embedded systems.

[2]Otherwise accepting the risk of loosing the memory status of the application that only accounts for very simple applicative scenarios.

III. Operating model

In this section the operating model of the targeted system is described to clarify the role of the hibernation mechanism. We can logically distinguish two phases: *data processing* and *data sensing*.

In particular, data processing consists of analyzing a given set of sampled data (and possibly transmit a result), once a given number of samples have been collected or when a certain threshold of the monitored parameter is reached. Operations of data sensing, on the other hand, entail sampling a certain sensor with an opportune frequency and push the measurement into a queue that will be analyzed during the next processing phase. Reading a sensor and adding the retrieved measurement to a queue are simple operations, such that maintaining the system in full active mode during this phase is indeed a waste of energy. So a possible solution is to hibernate the system once no more processes are performing data processing.

During the hibernation period, some data sensing operations may be required with a fine-grained timing: in this case, the system may be resumed in a low power mode and with a lower clock frequency, just to perform sensor readings and put the measurement in the associated queue. This operation is very fast (typically less than one millisecond) and does not require the boot of the entire operating system, neither to resume the process in charge of analyzing the entire set of data. Sensing operations are performed by a special module, called *smart sensing*, which, together with the hibernation manager, composes the power management infrastructure of the system. Though the smart sensing module is not object of this work (see [1] for details), we briefly introduce its two-phase model:

- *Pre-boot phase.* A smart sensing pre-boot function (embedded in the RTC interrupt service routine) determines whether the systems should be awakened in full active mode or in smart sensing. In this latter case, it reads the required sensor and pushes the measurement into the queue associated to the requiring process, then set the RTC to the next resume time and shuts down the system again.
- *Run-time phase.* A smart sensing daemon, implemented as kernel thread, runs in background while the system is in full active mode and determines if a measurement should be retrieved from the sensor on behalf of some processes.

Figure 1 shows the typical operating model of the proposed system. During the full active mode (T_{on}), the microcontroller is clocked at its maximum frequency, the main memory is powered-on and contains the images of the running process As said, during the full active periods, the active processes perform analyses, calculation and, possibly, transmission on the local data collected from sensing operations. Once all the processes have finished their calculation, the hibernation policy

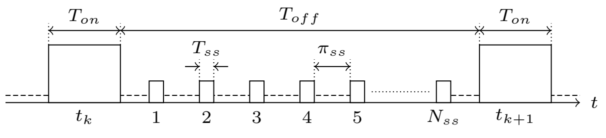

Fig. 1. Typical system operation

determine whether the system can undergo hibernation and whether it is convenient or not. To hibernate the system the process images are saved on the external non-volatile memory[3] the RTC is set to the next resume time and the system is switched off completely. The entire hibernation mechanism, as well as the policy, is indeed very articulated and is detailed in the next sections. During the hibernation period (T_{off}) the system is powered off and resumed in a low power mode by the smart sensing pre-boot function for the time strictly necessary for sampling and storing the measurement (T_{ss}). Since different processes may require samplings at different frequencies, potentially slightly unaligned, a system of deferrable timers has been implemented to group the sensing periods of different processes at a common frequency, with a given tolerance. In such a way, it is possible to resume the system in smart sensing mode and perform sensing on behalf of different processes together. This improves the energy efficiency of the system, since the number of times that system is resumed for smart sensing is significantly reduced.

Assuming that smart sensing aggregation has already been performed, the power consumption of sensing operations can simply be modeled by an average power P_{ss}. Referring to Figure 1, the energy for sensing in the time interval between the two processing phases is thus given by $P_{ss} \cdot T_{off}$.

As anticipated, hibernation requires two different decision: when it is *possible* to hibernate and whether it is *convenient* or not. The next session presents the scheme to answer the former question, whereas the model and the policy in Section V support the latter decision.

IV. Hibernation scheme

In this work we will refer to *processes* as generic instances of a running program, regardless of their actual implementation, that can be either a complete process, a task, a thread, and so on. To avoid considering trivial operation of the system, we will concentrate on periodic processes or asynchronous ones, which wait for events that occur with a certain statistical distribution. In case of non periodic processes, in fact, once all of them have completed their execution, the system can be switched off, since it accomplished to its goal. If, on the other hand, even a single process remains perpetually running, the system should remain active and can never be hibernated. Before going into the details of the the hibernation scheme, described by the state diagram of Figure 2, it is worth noting that to achieve maximum effectiveness and flexibility, the swap-in/swap-out mechanisms should allow operating on

[3]To allow an unlimited number of writings an MRAM has been used in the prototype.

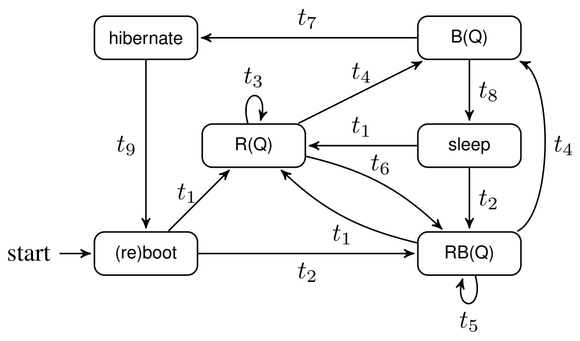

Fig. 2. Hibernation State Diagram

the *status* of single processes, process groups or the entire application.

In the FSM diagram the following conventions have been adopted. State R(Q) represents the condition in which all processes Q are running or ready; state RB(Q) indicates that some processes are running or ready and some are blocked, e.g. because in the idle period or because waiting for a (periodic) event; state B(Q) indicates that all the processes are blocked; state sleep is reached once all processes are blocked and the system is put into low-power mode, but with the main memory still powered; state hibernate indicates the condition in which the microcontroller is completely powered-off. Finally, the state (re)boot indicates either the first boot or the system reboot after hibernation. We assume that the entire hibernation process is managed through two demons: an hibernation daemon and a resume daemon. The *Hibernation daemon* keeps track of the status of all the processes and once all are in a blocked state, takes the decision whether to either hibernate or entering sleep mode, according to the hibernation policy. The hibernation daemon, moreover, saves the process descriptors in a fixed area of the non-volatile memory, and finally, when hibernating, swaps-out the memory images of the processes on the external memory. The *Resume daemon*, activated by an external real-time clock, swaps-in from the external non-volatile memory to the main internal memory the images of the processes that must be run and recreate their execution contexts in the operating systems. Note that the status of the operating system does not necessarily need to be saved, as it can be reconstructed from descriptors and images. These two daemons behaves according to the following formal state transition rules:

$t_1 : q_r \wedge |B| = 1$
$t_2 : q_r \wedge |B| > 1$
$t_3 : q_r$
$t_4 : q_b \wedge |R| = 1$
$t_5 : (q_b \wedge |R| > 1) \vee (q_r \wedge |B| > 1)$
$t_6 : q_b \wedge |R| > 1$
$t_7 : Policy \rightarrow Hibernate$
$t_8 : Policy \rightarrow Sleep$
$t_9 : RTC \rightarrow Wakeup$

where the event q_r indicates that a process q becomes ready or running, while q_b indicates that it assumes a blocked status. The set R and B represent the lists of running (or ready) and blocked processes, respectively, used by the daemons to keep track of the execution status. The notations $|R|$ and $|B|$ represent the cardinality of these lists immediately before an event q_r of q_b occurs. Note that an event q_r increments

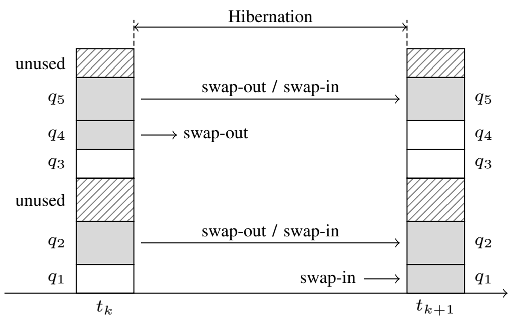

Fig. 3. Memory status in two subsequent processing phases: Hibernation

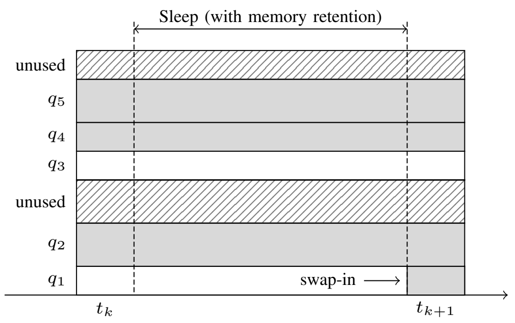

Fig. 4. Memory status in two subsequent processing phases: sleep

$|R|$ by one and decrements $|B|$ by one, while the event q_b has the opposite effect. As it can be noted, the described mechanism abstracts from a specific implementation and proposes a general guideline for implementing hibernation for embedded systems suffering from non-negligible static leakage power. According to this theoretical scheme, we actually implemented an hibernation mechanism in our system, based on the hardware/software architecture presented in [1].

V. Model

The model described in the following has the goal to define the optimal decision whether to hibernate the system or not whenever all processes are blocked. Such a decision depends on the difference between the energy consumed for hibernation and the energy necessary to maintain the microcontroller in a sleep state –with memory retention– during the interval between two subsequent processing phases. This, in turns, depends on which processes are active at the present time t_k and which will need to be executed at the next processing phase. Let $Q = [q_1 \; q_2 \; \ldots \; q_p]$ be the ordered set of all processes composing the application and $W = [w_1 \; w_2 \; \ldots \; w_p]$ be the size of their images in RAM, expressed in bytes. Furthermore, let t_k denote the current time and t_{k+1} be the time of the next processing phase. The status of the main memory at t_k and t_{k+1} is depicted in Figures 3 and 4. These figures show that the application consists of 5 processes $q_1, \ldots, q_5$ and that at time t_k processes q_2, q_4 and q_5 are active (i.e. loaded in main memory) while q_1 and q_3 are not. The figures also show that a portion of the memory is permanently unused.

When hibernating (Figure 3) all the processes active at t_k must be swapped-out and all those active at t_{k+1} need to be swapped-in. This is necessary to guarantee that the processes' status is maintained over time.

When the system is not hibernated between t_k and t_{k+1} (Figure 3) it is only necessary to swap-in new processes, that is q_1 in the example. Note that even though process q_4 does not need to be in memory at time t_{k+1}, it is not necessary to swap it out immediately after t_k, but rather its swap-out can be deferred until a decision to hibernate will actually be taken at some subsequent time.

With respect to the timing behavior of processes, three situations are possible: periodic, aperiodic and asynchronous.

- *Periodic processes.* The timing behavior of the processes is completely described by the vector $\Pi = [\pi_1 \; \pi_2 \; \ldots \; \pi_p]$ of their periods. This is the optimal case, since the timing of all processes is completely known in advance.
- *Aperiodic deterministic processes.* Each process has a deterministic behavior but the times of its next wake-ups are determined by the process itself, i.e. the process reschedules itself just before suspending. At a given moment, thus, only a few next wake-up times[4] are known for each process, limiting the observability of the system to a finite time period.
- *Asynchronous processes.* The wake-up time of a process is determined by some external, unpredictable event. In the best case, a distribution of their periods may be know. Such processes can only be treated resorting to a stochastic model, not treated in the following[5].

In the following we will concentrate on periodic and aperiodic processes, that is those processes whose timing behavior can be determined in advance. Asynchronous processes, on the other hand, are not considered here[6].

Assuming that processes are either periodic or deterministic aperiodic, at a given time t_k it is possible to determine both the next wake-up time and which processes will be active at that time. We define the state of the system at time t_k as a boolean vector:

$$S_k = [s_{k,1} \; s_{k,1} \; \ldots \; s_{k,p}] \tag{1}$$

where $s_{k,j} = 0$ if process q_j is currently swapped-out onto the external non-volatile memory, while $s_{k,j} = 1$ if the process is residing in RAM. The evolution of the system over time can thus be modeled as a sequence of pairs

$$(t_0, S_0), \; (t_1, S_1), \; \ldots, \; (t_k, S_k), \; \ldots \tag{2}$$

[4]Exactly only one, in the worst case.

[5]It has been proved by the authors that the formal model for stochastic processes cannot be solved in closed form as it becomes combinatorial in the number of such processes. The main reason is that no closed-form expression can be found to express the next wake-up time t_{k+1} and, consequently, the next state. The complete proof is omitted here for the sake of conciseness.

[6]If the number of executions of asynchronous process is small compared to that of periodic and aperiodic ones, their effect can be shown to produce small "perturbations" with respect to the ideal behavior. Furthermore such perturbation will affect the decision to be taken only for the time interval $[t_k, t_{k+1}]$ in which they fall.

TABLE I
ENERGY CONTRIBUTION FOR PROCESS TRANSITIONS

$S_{k,j} \rightarrow S_{k+1,j}$	E_H	E_S
$0 \rightarrow 0$	0	0
$0 \rightarrow 1$	$E_i W_j$	$E_i W_j$
$1 \rightarrow 0$	$E_o W_j$	0
$1 \rightarrow 1$	$(E_o + E_i) W_j$	0

indicating the current absolute time and the state of all processes. Considering the example of Figures 3 and 4, the two states are represented as:

$$(t_k, [01011]),\ (t_{k+1}, [11001]) \tag{3}$$

For a periodic system the evolution is know for the entire future, while for a system with periodic and aperiodic processes it is know only until the latest known wake-up time of the aperiodic processes.

A. Local formulation

The first formulation of the model is based on the assumption that to decide whether to hibernate the system or not between t_k and t_{k+1} it is sufficient observing only the states S_k and S_{k+1}. The system should be hibernated between t_k and t_{k+1} if and only if the energy E_H required for hibernation and wake-up is less than the energy E_S for maintaining the system in sleep mode in this time interval, that is when $E_H < E_S$. The first step consists of expressing E_H and E_S as a function of the present and next states S_k and S_{k+1}, of the process sizes W, of the time interval $(t_k; t_{k+1})$, the energies E_i and E_o to swap a single byte in and out, respectively, the average smart sensing power P_{ss}, and the power consumption of the system in sleep mode P_S. Furthermore, the energy E_{OS} to shut-down and to boot the operating system must also be accounted for, in the case of hibernation.

Table I summarizes the energy contributions associated to the possible state transitions for a single process q_j. Furthermore, we must also note that the energy $(t_{k+1} - t_k)P_{ss}$ for smart sensing is the same when the system is hibernated and when it is in sleep mode. In both cases, in fact, the system must perform the same operations. Affecting both E_H and E_S, this contribution can be omitted. Finally, we note that the energy for maintaining the system in sleep mode is $(t_{k+1} - t_k)P_S$. Combining these contribution with process states, the energy required to hibernate and wake-up the system is:

$$E_H = (S_k \cdot W_j)E_o + (S_{k+1} \cdot W_j)E_i + E_{OS} \tag{4}$$

where the scalar products $S_k \cdot W_j$ and $S_{k+1} \cdot W_j$ evaluate to the total size of the process images at times t_k and t_{k+1}, respectively. To simplify the notation, we define the *norm* of a state S_k as:

$$||S_k|| = S_k \cdot W \tag{5}$$

where the sizes' vector W is left implicit. Using this definition, Equation (4) becomes:

$$E_H = ||S_k||E_o + ||S_{k+1}||E_i + E_{OS} \tag{6}$$

Defining as $\tau_k = t_{k+1} - t_k$ the time interval between the present and the next state, the energy for maintaining the system in sleep mode is:

$$E_S = (\, ||S_{k+1}|| - ||S_k \wedge S_{k+1}||\,)E_i + \tau_k P_S \tag{7}$$

where the difference of the norm expresses the total size of the processes that are not present in S_k but only in S_{k+1}: these are, in fact, the processes that need to be swapped-in.

Hibernation is convenient if $E_H < E_S$, that is when $\Delta E = E_H - E_S < 0$. Combining Equations (6) and (7) yields:

$$\Delta E = ||S_k||E_o + ||S_k \wedge S_{k+1}||E_i + E_{OS} - \tau_k P_S \tag{8}$$

and solving $\Delta E < 0$ for τ_k, yields the condition:

$$\mathcal{C}_l : \quad \tau_k > \frac{||S_k||E_o + ||S_k \wedge S_{k+1}||E_i + E_{OS}}{P_S} \tag{9}$$

This last inequality determines the minimum time interval after which it is convenient to put the system into hibernation. We refer to this as *local condition* and indicate it with $\mathcal{C}_l$.

It must be noted, though, that the time interval τ_k must also be longer than the time to write and read the non-volatile memory for swap-out/swap-in operations:

$$T_{SWAP} = ||S_k||T_B + ||S_{k+1}||T_B \tag{10}$$

where T_B is the time for reading or writing a single byte on the non-volatile memory[7], plus the time T_{SO} to shut-down and reboot the operating system. This condition can be thus expressed as:

$$\tau_k > T_{SWAP} + T_{SO} \tag{11}$$

To avoid considering this last constraint, Equation (9), which expresses the minimum time that makes hibernation convenient, must imply Equation (11), i.e.:

$$\frac{||S_k||E_o + ||S_k \wedge S_{k+1}||E_i + E_{OS}}{P_S} > T_{SWAP} + T_{SO} \tag{12}$$

Equation (12) can be split in four contribution and rewritten as $T_1 + T_2 - T_3 + T_4 > 0$ where:

$$\begin{aligned} T_1 &= ||S_k|| \left(\frac{E_o}{P_S} - T_B \right) = \frac{||S_k||}{P_S}(P_W + P_A - P_S)T_B \\ T_2 &= \frac{||S_k \wedge S_{k+1}||E_i}{P_S} \\ T_3 &= ||S_{k+1}||T_B \\ T_4 &= \frac{E_{OS}}{P_S} - T_{SO} = \frac{(P_A - P_S)T_{SO}}{P_S} \end{aligned} \tag{13}$$

and P_W represents the power of write operations consumed by the memory and P_A is the microcontroller power in active state. Observing the expression of the different times in Equation (13) it is possible to determine a worst-case boundary condition on the maximum amount of memory swappable. Since for all microcontrollers is always true that $P_A > P_S$, the term T_1 is always positive and as a worst-case we set $T_1 = 0$. The term T_2 can never be negative and can become zero when states S_k and S_{k+1} have an empty intersection.

[7]For magneto-resistive memories read/write operations require the same time. A more general formulation can be easily derived for asymmetric times.

Again, to consider the worst-case, we set $T_2 = 0$. Under these assumptions the initial inequality becomes $T_3 < T_4$, that is:

$$||S_{k+1}|| < \frac{(P_A - P_S)T_{SO}}{P_S T_B} \tag{14}$$

This inequality poses an upper bound on the swappable memory, exceeding which Equation (11) must be considered explicitly in the model. In Section VI we will show that this constraint is generally satisfied for micrcontrollers and operating systems adopted in the embedded system domain.

B. Global formulation

The model derived so far limits the observation to the next state S_{k+1}. It may happen, though, that the state S_{k+1} and S_{k+2} are close enough in time to be convenient not to hibernate, between t_{k+1} and t_{k+2}. In this case, the new state at t_{k+1} is given by $S_{k+1} \vee S_{k+2}$. In fact, if the system is sleeping between t_{k+1} and t_{k+2}, then the processes to be swapped-in at t_{k+2} can also be swapped-in at t_{k+1}, without changing the behavior of the system. Assuming that for the next n states $S_{k+1}, \ldots, S_{k+n}$ the system will not hibernate, the processes that must be swapped-in at each of the n future states may, in theory, be swapped-in altogether at t_{k+1}.This is only true, as said, in theory. In practice, without affecting the form of the model, processes will be swapped in only when actually needed. To express this general condition for hibernation based on the observation of the n states $S_{k+1}, \ldots, S_{k+n}$ we define:

$$S^{(n)}_{k+1} \quad = \quad S_{k+1} \vee S_{k+2} \vee \ldots \vee S_{k+n} \tag{15}$$

and note that for $n = 1$ we obtain the expression $S^{(1)}_{k+1} = S_{k+1}$ appearing in Equation (9). By extending the concept of next state from S_{k+1} to the transitive closure $S^{(n)}_{k+1}$ of the OR operation over the next n states, Equations (6) and (7) become:

$$E_H = ||S_k||E_o + ||S^{(n)}_{k+1}||E_i + (t_{k+n} - t_{k+1})P_S + E_{OS} \tag{16}$$

$$E_S = (\,||S^{(n)}_{k+1}|| - ||S_k \wedge S^{(n)}_{k+1}||\,)E_i + (t_{k+n} - t_k)P_S \tag{17}$$

Also the global formulation entails that hibernation is convenient if and only if $E_H < E_S$. Noting that $(t_{k+n} - t_k) = (t_{k+n} - t_{k+1}) + (t_{k+1} - t_k)$ the equation for ΔE becomes:

$$\Delta E = ||S_k||E_o + ||S_k \wedge S^{(n)}_{k+1}||E_i + E_{OS} - \tau_k P_S \tag{18}$$

and thus the global condition $\mathcal{C}_n$ for hibernation is:

$$\mathcal{C}_n : \quad \tau_k > f(n) = \frac{||S_k||E_o + ||S_k \wedge S^{(n)}_{k+1}||E_i + E_{OS}}{P_S} \tag{19}$$

which guarantees that hibernating is convenient in the time interval between t_k and t_{k+1}.

C. Model analysis

The global condition $\tau_k > f(n)$ states that the correct choice between hibernation and sleep, in the time period between t_k and t_{k+1}, requires observing the next n states. The actual value of n is, however, unknown. Studying the function $f(n)$ we observe that it admits an absolute maximum and a theoretical lower bound. The transitive closure $S^{(n)}_k$ appearing in $f(n)$ is monotonic, since the boolean OR operator can only add new ones to the vector. This implies that:

$$\max_n ||S_k \wedge S^{(n)}_{k+1}|| = ||S_k|| \tag{20}$$

and this occurs as soon as $S^{(n)}_{k+1} \supseteq S_k$. This is guaranteed to happen for periodic processes, while it is only an upper bound in the case of aperiodic processes. Let denote with t_{k+n_M} the time at which $f(n)$ reaches its maximum $f(n_M) = f_M$. Since, thus, f_M is an absolute maximum, if $\tau_k > f_M$, that is when:

$$\mathcal{C}_M : \quad \tau_k > \frac{||S_k||(E_o + E_i) + E_{OS}}{P_S} \tag{21}$$

then the decision to hibernate is guaranteed to be optimal. We refer to this inequality as *global maximum condition*, or $\mathcal{C}_M$.

As anticipated, the function $f(n)$ also admits a lower bound f_L when $S_k \wedge S^{(n)}_{k+1} = \emptyset$. Thus, when $\tau_k < f_L$, that is when:

$$\mathcal{C}_L : \quad \tau_k < \frac{||S_k||E_o + E_{OS}}{P_S} \tag{22}$$

the decision not to hibernate is guaranteed to be optimal. We will refer to this inequality as *global minimum condition*, or $\mathcal{C}_L$. It is fundamental to note that the two conditions $\mathcal{C}_M$ and $\mathcal{C}_L$ are only *sufficient* but not necessary.

In the general case, that is with a specific look-ahead of n states, the value of $f(n)$ will necessarily fall between f_M and f_L. As anticipated, the specific value of n to be used for a particular combination of processes, energies, sizes and for each specific state cannot be determined in general, as it requires an exhaustive analysis of the system evolution, which, moreover, is only possible for periodic processes. We thus introduce the parametric threshold f_α defined as:

$$f_\alpha = f_L + \alpha(f_M - f_L) \tag{23}$$

with $0 \leq \alpha \leq 1$. This leads to the new, general, condition:

$$\mathcal{C}_\alpha : \quad \tau_k > f_\alpha \tag{24}$$

where α is to be determined experimentally in order to minimize the energy consumed by the system.

VI. Results

To evaluate the proposed model, we performed several experiments aimed both at assessing its efficiency, robustness and reliability on one hand, and the energy saving obtained in adopting the hibernation policy on the other. The intrinsic properties of the model have been verified through an integrated simulation flow developed to replicate the evolution of a real system and to stress its operating parameters in order to get information about the hibernation policy performance. A highly optimized simulation flow has been implemented in C++ to meet the severe computational requirements of the stress tests and stability analysis on a wider spectrum of tests.

The impact on energy efficiency have been mainly measured by testing the policy on a real embedded platform, that we recently proposed in [1] and developed to support hibernation. This platform—originally conceived for wireless sensor networks—is based on the STM32F2 microcontroller,

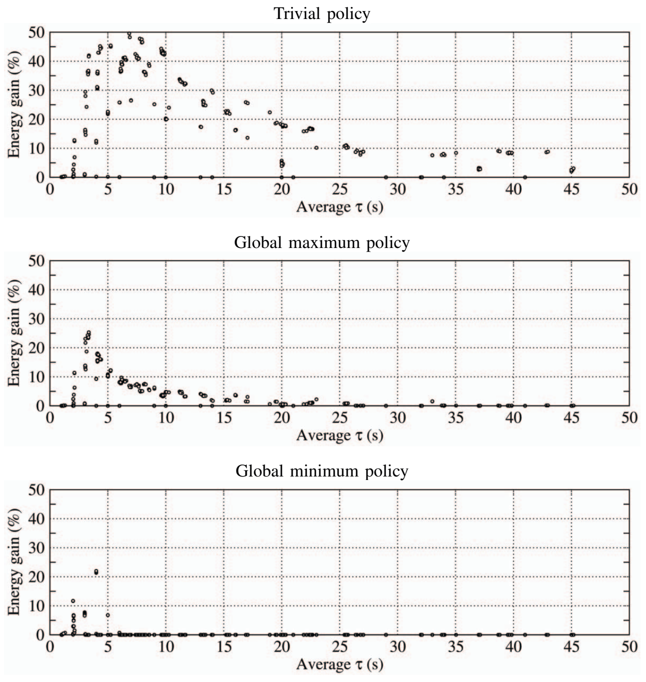

Fig. 5. Optimal versus sub-optimal policies energy gains

the external MR25H10 magnetoresistive RAM and the ultra low-power nRF24L01+ transceiver. The software infrastructure is based on the Miosix operating system, enabling standard C/C++ programming and libraries, process execution and support for process isolation through memory protection. The operating system has been provided with an hibernation mechanism, according to the architecture presented in Section IV. Most of the experiments reported in the following refer to the implementation based on the abovementioned components, while others have been performed by perturbing the ratio E_{active}/P_{sleep}, the most significant parameter affecting hibernation choices[8]. In the experiments we assumed a maximum available memory of 128 KB and a pool of 10 periodic processes, with memory image sizes randomly chosen between 1 KB and 32 KB. The RAM memory footprint of the specific operating system is about 32 KB, but, as discussed in Section IV, it does not need to be swapped-out in the external MRAM since it performs a complete reboot each time the system is resumed. Recalling the feasibility condition of Equation (14), and considering an average operating system boot time of approximately 1.5 ms[9], the value of $||S_{k+1}||$ is about 110 KB, which practically equals the maximum available memory for processes. We will thus no longer consider this constraint in our experiments, since it is assumed to be always satisfied. The following sections summarize the results obtained, in particular energy gains and the robustness and stability of the proposed policy.

[8]The operating energies and timings of the non-volatile memory have not been perturbed since they are typical for a wide range of similar devices.

[9]Embedded operating systems boot times range from hundreds of μs up to a few ms for the most complex ones. These data refer to microcontroller with operating frequency around 60 MHz. In our specific implementation, the boot time of Miosix kernel is approximately 450 μs.

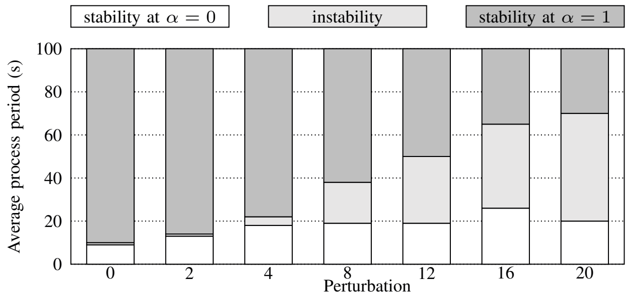

Fig. 6. Policy robustness and “gray zones”

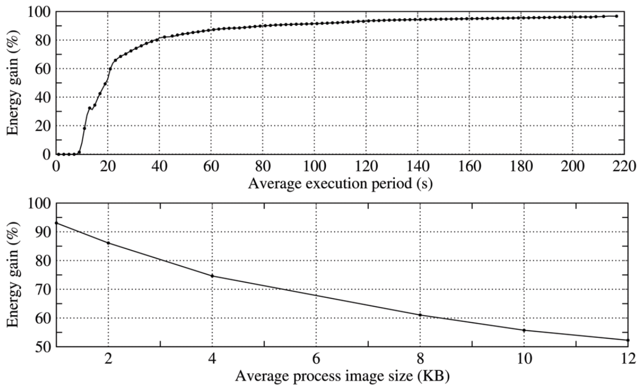

Fig. 7. Energy gain with respect to process periods and image size

The first set of simulations studies the energy gain obtained applying the optimal policy with respect to three sub-optimal policies, namely: a *trivial* policy assuming to swap-in and out the entire memory, the *global maximum* and the *global minimum* policies defined by the conditions $\mathcal{C}_M$ and $\mathcal{C}_L$. Such gains are reported in of Figure 5 against the average time interval τ between subsequent states. Each point corresponds to different combination of process periods and sizes and summarizes the average gain over 1,000 randomly generated test instances. As it can be noted, the gains with respect to the global minimum policy are smaller than those for the other policies, meaning that the global minimum policy tends to dominate over the others, i.e. the optimal value of the α parameter tends to 0. On the other hand, the trivial policy, often adopted for small sensor nodes, resulted by far the least efficient one.

As just observed, the value of the parameter α strongly affects the gain. In particular, the results suggest that the values 0 and 1 are often those maximizing the gain, with some exceptions where an intermediate value leads to better results. Since the exact optimal value of α cannot be determined explicitly, a desirable property of the adopted policy would be its robustness against small variations of the parameters—periods and sizes—of the application. To this purpose, we performed more than one million tests, by randomly changing the periods and sizes of the processes and by perturbing the ratio E_{active}/P_{sleep} of a factor 2, 4, 8, 12, 16 and 20.

As Figure 6 shows, the policy has a good stability, converging towards the two attractors $\alpha = 0$, before a certain critical average period $\overline{p}_{min}$, and $\alpha = 1$, after a critical average period $\overline{p}_{max}$. Amidst these two thresholds, the systems is in a sort of *gray zone*, delimited by the two dashed lines in the figure, where specific simulations are needed to determine the optimal α. It should be noted, however, that for energy values in the same order of magnitude of the considered microcontroller the policy shows an almost stepwise behavior. The change of the optimal value of α from 0 to 1, though, rarely abrupt but rather passes through intermediate optimal α values. For example, in the case of the system without perturbation, simulations have identified intermediate values of α ranging from 0.3 to 0.7 for average process periods around the critical value of 8s, and in particular from 7s to 11s. In such cases, energy savings up to 20% can be obtained. A similar trend can be observed for $2\times$ and $4\times$ perturbations, which embraces the majority of real micrcontrollers of the upper-mid range. For perturbations greater than 8, which however account for extreme cases, the *gray zone* is more extended, but it is still possible to note a general prevalence of $\alpha = 1$. Concluding, the prevalence of the two attractors $\alpha = \{0, 1\}$ confirms a good robustness of the policy. It does not, in fact, need expensive run-time calibration of the α parameter for small variations of task periods, except in the very proximity of the gray zone. The optimal α can in fact be determined before deployment, based on the average task period, and hard-coded into the application.

Finally, the energy gain of the hibernation approach (HA) with respect to classical operation (CO), in which the system is brought in sleep mode during idle periods, an thus suffers from static leakage current problems, has been studied and estimated through several hundreds of tests. The relative gain $E_{gain} = (E_{CO} - E_{HA})/E_{CO}$ reported in the following refers to the specific architecture proposed in [1]. The test conditions and application parameters used for this analysis are the same as those before. The executions periods, in particular, have been varied between 4 and 250 seconds[10], while the process image sizes was selected randomly between between 1 and 32 KB. Figure 7 shows how the energy gain increases when the average execution period of the processes augments: this behavior points out that with higher intervals between subsequent executions the time spent in hibernation state augments and the frequency of swap-in and swap-out operations decreases. For very short execution periods, on the contrary, the gain tends to zero, since hibernating is no longer convenient. Concerning process image sizes, it can be noted that the energy gain is maximum for smaller sizes, because smaller processes require less energy for the swap-in and swap-out operations, making hibernation more convenient.

VII. Conclusions

This paper presented the achievements obtained by our research on the static leakage power mitigation in high and mid-range microcontrollers. A formal model, a general mechanism and a sample architecture have been studied and implemented to exploit hibernation during idle periods of application processes. Results obtained in several stress tests and simulation scenarios considering the electrical characteristics of a real implementation demonstrated the validity of the studied approach. Energy gains up to 50% compared to widely used trivial hibernation policies have been obtained. Moreover, thanks to the conciseness of the closed-form model, the proposed policy can be easily implemented in a broad range of embedded systems. Furthermore, since it only depends on the electrical parameters of the target hardware platform, the hibernation mechanism can adapt to different applications and/or to run-time variations of the application properties (number, periods and sizes of processes).

[10]Periods in that order, and even longer, are typical, for example, of sensing and monitoring applications, e.g. WSNs.

References

[1] Brandolese,C., Fornaciari,W., Rucco,L. and Terraneo, F. Enabling ultra-low power operation in high-end wireless sensor networks nodes. In Proc. of CODES+ISSS '12. ACM, New York, NY, USA, 433-442.2012.

[2] G. Anastasi, M. Conti, M. Di Francesco and A. Passarella, Energy conservation in wireless sensor networks: A survey. Ad Hoc Netw. 7, 3 (May 2009), 537-568. 2009.

[3] Schurgers,C., Tsiatsis,V., Srivastava,M. B. STEM: Topology Management for Energy Efficient Sensor Networks. In IEEE Aerospace Conference '02, Big Sky, MT, March 10-15, 2002.

[4] Schurgers,C., Tsiatsis,V. , Ganeriwal,S., Srivastava, M. B. Optimizing Sensor Networks in the Energy-Latency-Density Design Space. IEEE Trans. on Mobile Computing, Vol.1, No.1, pp. 70-80. 2002.

[5] Tseng,Y., Hsu,C. , Hsieh,T. Power Saving Protocols for IEEE 802.11 Ad Hoc Networks. In Proc. Infocom'02, New York (USA).2002.

[6] Keshavarzian, A., Lee, H., Venkatraman,L. Wakeup Scheduling in Wireless Sensor Networks, In Proc. of MobiHoc' 06. pp. 322-333, Florence, Italy.2006.

[7] Hohlt, B., Doherty, L., Brewer, E. Flexible Power Scheduling for Sensor Networks. In Proc. of ISPN'04, Berkeley (USA). 2004.

[8] Lu, G., Sadagopan, N., Krishnamachari, B., Goel, A. Delay Efficient Sleep Scheduling in Wireless Sensor Networks. In Proc. of Infocom'05. Miami, Florida, USA. 2005.

[9] L. Benini, A. Bogliolo and G. De Micheli, A survey of design techniques for system-level dynamic power management, *IEEE Trans. on VLSI*, 8(3):299–316, June 2000.

[10] Y.H. Lu, T. Šimunić, G. De Micheli, Software controlled power management, *In Proc. of CODES'99,* pp. 157-161.

[11] Y.H. Lu, L. Benini, G. De Micheli, Requester-aware power reduction, *In Proc. of ISSS'00,* pp. 18-23.

[12] Z. Hu, S. Kaxiras, M. Martonosi, Let caches decay: Reducing leakage energy via exploitation of cache generational behavior. *ACM Trans. on Computer Systems*, 20(2):161–190, May 2002.

[13] L. Li et al., Leakage energy management in cache hierarchies, *In Proc. of PACT'02*, pp. 131–140.

[14] K. Meng, R. Joseph, Process variation aware cache leakage management, *In Proc. of ISLPED'06*, pp. 262–267.

[15] F. Menichelli, M. Olivieri, Static Minimization of Total Energy Consumption in Memory Subsystem for Scratchpad-Based Systems-on-Chips. IEEE Trans. VLSI Syst. 17(2): 161-171,2009.

[16] M. Verma, P. Marwedel, Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans. VLSI Syst. 14(8): 802-815, 2006.