A Heuristic-Based Approach for Reducing the Power Consumption of Real-Time Embedded Systems

Vlad Rădulescu
A. I. Cuza University of Iaşi
Department of Computer Science
Iaşi, Romania
rvlad@info.uaic.ro

Ştefan Andrei
Lamar University
Department of Computer Science
Beaumont, TX, U.S.A.
Stefan.Andrei@lamar.edu

Albert M.K. Cheng
University of Houston
Computer Science Department
Houston, TX, U.S.A.
cheng@cs.uh.edu

Abstract—The current trend in designing power-efficient devices is concerning not only Personal Computer-like (PC) systems, but also real-time embedded systems. While a lot of research has been done on minimizing the total energy of a system, adapting the scheduling techniques for lower energy consumption has been less popular. Nevertheless, this can prove highly efficient, as the Central Processing Units (CPUs) are usually responsible for the largest part of the whole system’s energy consumption.

This paper presents an approach on improving the energy consumption of a real-time system. Starting with a given feasible schedule for a non-preemptive, single-instance, \( n \)-task set, power saving is achieved by reducing the CPU frequency whenever possible, without breaking the task deadlines. The goal can be described in analytical terms as a multivariate optimization problem. Due to the complexity of the resulting problem, the use of heuristic techniques, such as using genetic algorithms and differential evolution, provides good chances for finding the desired optimum. To the best of our knowledge, the use of these methods for the power-aware scheduling problem has not been attempted.

I. INTRODUCTION

The dramatic increase in the processing power of the computing devices has lead to an explosion of the real-time embedded systems, which are now capable of performing very complex tasks. Their applicability now covers, among other fields, telecommunications (with the smartphones as flagship), data transmission (fax machines, GPS receivers), transportation (cars, airplanes, boats), medicine (life-support and monitoring devices, blood pressure monitors), or entertainment (Xboxes, PSPs, Wii).

Mobility has been the key point to the commercial success of such devices - and is now becoming their main problem. The higher processing and storage capabilities have also brought higher energy consumption, while the capabilities of the batteries could not keep up with the new requirements. Therefore, until a new generation of power storage will emerge (if ever), the solution is the reduction of power consumption. Given the multitasking nature of today’s mobile devices, power-aware scheduling is an essential design issue for real-time embedded systems.

There are many ways to define a task in a real-time system. In this paper, we consider a simple model, where a task as defined by two parameters: the computation cost (also known as the worst-case execution time), denoted by \( c \), deadline, denoted by \( d \). The tasks are considered to be single-instance and their starting times are all zero. The notation \( T = (c, d) \) means that the task \( T \) requires an execution time equal to \( c \) and its execution must be completed by the time \( d \).

The two parameters mentioned above are usually positive integers. However, there is no theoretical restriction for treating them as rational or even real numbers, except for the fact that the systems becomes far more complex from the analytical point of view.

Given a task set \( T = \{T_1, ..., T_n\} \), \( T \) is called schedulable by a scheduling algorithm \( Sch \) if \( Sch \) ensures that the timing constraints of all jobs in \( T \) are met. An algorithm \( Sch \) is called optimal if whenever \( Sch \) fails in finding a schedule, all other scheduling algorithms also fail [9]. The definition of a schedule, also known as execution assignment, will be detailed in Section III for multiprocessors platforms.

Two traditional scheduling techniques are widely used: Earliest Deadline First (EDF) and Least-Laxity First (LLF). Given a task set \( T = \{T_1, ..., T_n\} \), where each job \( T_i \) is defined by \( (c_i, d_i) \forall i \in \{1, ..., n\} \), the EDF scheduling means that the task with the earliest deadline \( d_i \) has the highest priority in being assigned for execution on the processor. On the other hand, the LLF scheduling means that the task with the smallest laxity \( l_i = d_i - c_i \) has the highest priority.

The paper is organized as follows:

- Section II reviews the existing approaches and results in the field of power saving electronic design.
- In section III, the terminology and the definitions required for understanding the power-aware scheduling problem are introduced.
- Section IV provides the formulation of the optimization problem corresponding to power-aware scheduling. The heuristic techniques used to solve the problem are also described, together with the experimental results.
- Finally, section V draws the conclusions of the paper.

II. RELATED WORK

Power consumption in electronic systems has two main components: dynamic power and static power, which will be discussed in more detail later in the paper. Dynamic voltage scaling (DVS) has been an effective approach to reduce the
energy consumption by scaling the processor's voltage and frequency on the fly. Currently, the dynamic power is considered the predominant component in the overall power consumption. However, as the state-of-the-art processors have nanoscale-level Very Large Scale Integrated (VLSI) designs, working at very high frequencies (above 1 GHz), the static power generated by the leakage dissipation is becoming increasingly important.

Reducing the leakage dissipation has followed two main directions. The first one is the use of finer process technologies, which allow operating at lower power supply voltage and diminish the capacitance load.

The second direction is the use of multicore platforms, which has become dominant in the last years, due to their advantages [42]. On one hand, multicore systems are better in handling the heat dissipation problem, caused by the smaller sizes of the circuits and the higher operation frequencies. On the other hand, the so-called memory space and processor performance gap lead to the conclusion that multicore systems with large caches are superior to single-processor systems using a single memory cache. In [33], a system level dynamic scheduling algorithm was designed, which is capable of minimizing the energy consumption by the DVS processor and multiple non-DVS peripheral devices in a hard real-time system.

During the last decade, several power-aware scheduling techniques have been proposed [4], [14], [43]. Nevertheless, most of them were designed for regular real-time jobs, such as periodic jobs, and cannot deal with the complex timing characteristics specific to many real-life embedded systems. Such situations require more complex models for their specification, of which we mention the imprecise computation model [30], [37], the skip-over model [23], the \((m, l)-firm\) guarantee model [17], [28], or the reward-based model [5]. Using these models for a better description of the real-time systems, new power reduction techniques have been developed. The maximization of the total reward under timing and energy constraints has been studied in [7] and [36]. The correlation between the energy and the performance of \((m, l)-firm\) systems has been studied in [1], [35]. The optimization of both dynamic and static power consumption was approached in [34]. Real-time systems with probabilistic guarantees also consider power-aware scheduling [45].

The common point of all these approaches is the assumption that a task or a task instance has a single feasible interval in which it can successfully complete its execution. This is not the case with all real-time applications, especially when considering the computing resources, such as signal, bandwidth or power, which are not always available. An example is real-time packet delivery for mobile devices in ad-hoc networks [8]. An there may be periods of time when certain areas are not covered by any wireless network, the packets sent during such periods will be lost. The straightforward solution is to transmit packets only when the area is covered by some networks, which means that the ad-hoc networks have more than one feasible interval to send packets to the mobile devices.

The Global Positioning System (GPS) navigation device is a practical example of such devices. This system is equipped with a Frequency Modulation (FM) receiver for the automatic reception of traffic updates. The main function of a GPS navigator device is to provide voice and map-guided navigation from a starting point to a destination selected by the user. At a given time instant, the device may have received one or more traffic updates concerning the chosen path. Tasks may be initiated to compute new routes (re-routing) in order to avoid going through the congested paths. The lengths of the alternate paths and traffic congestions along those paths are also considered when re-routing is computed. Depending on the frequency of the traffic updates as well as the current position and speed of the vehicle in which the GPS device is installed, there are multiple feasible intervals for executing these re-routing jobs.

Another research direction is the finer estimation of computational costs. In most cases, the processor time required for executing a task is not constant. The safest approach is to consider always the worst case execution time (WCET) as the computational cost of the task. On the other hand, this may obviously lead to a waste of resources. Paper [20] proposes a method for achieving performance guarantees (within some reasonable limits) with reduced energy consumption, by considering smaller values for the computational cost.

An example is given by the multimedia and digital signal processing (DSP) applications, which consist in repetitive processing on periodically arriving inputs (e.g., voice samples or video frames). The processing deadlines, which are determined by the throughput of the input data streams, may occasionally be missed and still provide sufficient quality for the end user. For example, in packet audio applications, loss rates between 1% and 10% can be tolerated [6], while tolerance for losses in low bit-rate voice applications may be significantly lower [22]. Such tolerance gives rise to slacks that can be exploited when streamlining the embedded processing associated with such applications. For instance, when the embedded processing does not interact with a lossy communication channel, then the application's slacks can be used to reduce cost or power consumption. Typically, slacks arise from the run-time job execution time variation and can be exploited to improve real-time applications response time or reduce power.

The multi-core System-on-Chip (SoC) architectures raise the problem of overheating. An efficient runtime thermal-aware scheduler was proposed in [44], making use of task migration and power-gating techniques in order to avoid the thermal hotspots on a multi-core chip. Another power-aware category of embedded systems is made by the battery-operated systems, notably the biomedical systems. The scheduling of frame-based real-time jobs for multiprocessor systems powered by rechargeable batteries was introduced in [26], while [27] explore the job assignment problem on heterogeneous distributed systems with rechargeable batteries. These biomedical systems, including health monitors and assisting devices, are essential for enhancing the quality of life for the patients. These time-critical biomedical systems are mainly battery-operated, so they can be carried by the patients as implants or used in remote areas, which are not connected to the power grid. These are obviously hard-constrained real-time systems, with severe consequences if the deadlines are not met. In fact, if the battery does not supply sufficient voltage to a biomedical system, its service may deteriorate. Existing techniques are designed only to assess the feasibility of such
systems, but they cannot provide accurate assessments for working under improper conditions, such as dynamic voltage fluctuations in the batteries.

Although similar with [3], this paper is a continuation on a different path. Compared to the analytical formula from [3] which gives the minimum power consumption for only two tasks, this paper represents an extension of the previous work in three directions:

1) We designed two new techniques using genetic algorithms and differential evolution instead of a analytical formula;
2) The best power provided by these algorithms is found with a precision of $10^{-6}$ instead of integer values;
3) The algorithms are able to reduce the power consumption for sets with more than two tasks.

III. DEFINITIONS

We denote a time interval by $[s, e)$, where $s$ is the start time and $e$ is the finish time. As time intervals are used here to indicate the execution of tasks, the intervals are right-open. Moreover, a time interval has associated a processor, on which the task is executed. Thus, we say that task $T$ executes in the time interval $[s, e)$ if $T$ starts to execute on processor $p$ at time $s$ and finishes its execution just before time $e$, so that the next task will be able to start its execution on processor $p$ at time $e$.

Definition 3.1: Two time intervals with their associated processors, $[s_1, e_1](p_1)$ and $[s_2, e_2](p_2)$, are overlapping if $p_1 = p_2$ and $[s_1, e_1) \cap [s_2, e_2) \neq \emptyset$. The overlapping is denoted by $[s_1, e_1)(p_1) \cap [s_2, e_2)(p_2) \neq \emptyset$.

Another important concept is the execution assignment of a task (and by extension, of a task set), which is seen here in a similar manner to [2], [3], [38]. Considering a real-time system where computational costs and deadlines are known in advance [9] (thus allowing static scheduling) and context-switch times are negligible, execution assignment can be defined as follows:

Definition 3.2: Given a task set $\mathcal{T} = \{T_1, ..., T_n\}$, where $T_i = (c_i, d_i)$, an execution assignment of task $T_i$ on a multiprocessor platform $\mathcal{MP} = \{p_1, ..., p_m\}$, denoted by $EA(T_i)$, is the reunion of all time intervals and their associated processors, where the task is executed:

$$EA(T_i) = [s_i^{(1)}, e_i^{(1)})(p_{i, 1}) \cup ... \cup [s_i^{(n_i)}, e_i^{(n_i)})(p_{i, n_i})$$

where:

- $p_{i, 1}, ..., p_{i, n_i} \in \mathcal{MP}$
- $s_i^{(1)} < e_i^{(1)} \leq ... \leq s_i^{(n_i)} < e_i^{(n_i)}$
- $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$
- $s_i \leq s_i^{(1)}$ and $e_i^{(n_i)} \leq d_i$

At the same time, the execution assignment of the task set $\mathcal{T}$, denoted by $EA(\mathcal{T})$, is the set of the execution assignments of all tasks $T_i \in \mathcal{T}$:

$$EA(\mathcal{T}) = \{EA(T_1), ..., EA(T_n)\}$$

where $\forall i, j \in \{1, ..., n\}, i \neq j$, we have $EA(T_i) \cap EA(T_j) = \emptyset$.

With the notations of Definition 3.2, the particular case of non-preemptive scheduling corresponds to $n_i = 1, \forall i \in \{1, ..., n\}$, while the general case of preemptive scheduling means that $n_i \geq 1$ (the case $n_i = 1$ corresponds to non-preemptive jobs). We note that it is sometimes possible in preemptive scheduling for a task $T_i$ to have $n_i = 1$ (that is, it may run uninterrupted until its end), but in non-preemptive scheduling a task $T_i$ will never get interrupted ($n_i > 1$).

Another important concept in scheduling theory is the execution rate, defined next.

Definition 3.3: Given a task $T = (c, d)$, where $c$ is its computation time and $d$ is its deadline, executed without interruption on a processor in the time interval $[s, e)$, where $e \leq d$, the execution rate, denoted by $r$, is given by the formula $r = \frac{c}{e-s}$.

For example, let us consider a task $T = (5, 16)$. If executed from time 2 to time 7, then the execution rate is $r = 1$, as the execution time is equal to the computational cost of the task. On the other hand, if the task is executed from time 4 to time 14, then the execution rate is $r = \frac{10}{10} = 1$. The lowest possible rate is achieved by executing the task from time 0 to time 16 and will have the value $r = \frac{16}{16} = 0.3125$. Any value of the execution rate below that threshold will result in failing to meet the deadline.

We make the observation that the execution rate cannot exceed 1. This is because, by definition, the computational cost of a task is the amount of time necessary for executing the task at processor’s highest speed. Thus, it is not possible to execute the task in less time than its computational cost. On the other hand, as we will see in the sequel, a lower processor speed (and thus a lower execution rate) leads to lower energy consumption.

Given a real-time embedded system, the power consumption, denoted as $P$, can be expressed as $P = P_{\text{leak}} + P_{\text{dyn}}$, where $P_{\text{leak}}$ is the static power and $P_{\text{dyn}}$ is the dynamic power. The static power is responsible for most of the energy consumption while the processor is idle [19], but it is considered negligible during task execution. The means that energy saving has to be achieved mainly by reducing the dynamic power.

The dynamic power function is expressed as $P_{\text{dyn}}(s) = C \cdot V^2 \cdot s$, where $C$ denotes the effective switch capacitance, $V$ is the supply voltage, and $s$ represents the processor speed.

The value of the effective switch capacitance is highly related to the software implementations and the execution path of each job (which could be usually derived by profiling). Thus, the power consumption function $P_{\text{dyn}}(s)$ can be considered as $h \cdot s^\alpha$, where $\alpha$ is a hardware-dependent factor, and $h$ is a parameter related to the corresponding job execution [7], [29]. According to [7], $\alpha$ is a positive real number, and $h$ is usually a real number between between 2 and 3. Obviously, the dynamic power consumption function is a strictly convex and increasing function of the processor speed, as processor speeds are non-negative numbers.

As multiple execution assignments are possible for a certain task set, this raises the question of which one is more energy-efficient. The next definition introduces the power-aware scheduling problem.
Definition 3.4: Let us consider a task set $T = \{T_1, \ldots, T_n\}$, where $T_i = (c_i, d_i)$, $\forall i \in \{1, \ldots, n\}$, schedulable on the multiprocessor platform $MP = \{P_1, \ldots, P_m\}$. The power-aware scheduling problem is to determine the execution assignment $EA_{min}$, such that the energy consumption $E(EA)$ is minimized:

$$E(EA_{min}) = \min \{ E(EA) \mid EA \in EA(T) \}$$

where:

$$E(EA) = \sum_{i=1}^{n} P_{dyn}(T_i) \cdot c(T_i)$$

$P_{dyn}(T_i) = r_i^2$ is the power consumed by the execution of $T_i$

$r_i = \frac{c_i}{c_i - s_i}$ is the execution rate of $T_i$

$c(T_i) = c_i - s_i$ is the computation time at rate $r_i$

The $h$ parameter has been omitted from the formula above because, as a constant positive value, it has no influence on the optimum problem. In this paper we consider $P_{dyn}(T_i) = r_i^2$ instead of $P_{dyn}(s) = h \cdot s^2$, as the execution rate of $T_i$ is in fact a measurement for the processor speed $s$ when $\alpha = 2$.

Example 3.1: Let us consider a 1-processor platform and a 4-task set, given by $T = \{T_1, T_2, T_3, T_4\}$, where $T_1 = (3, 6)$, $T_2 = (5, 11)$, $T_3 = (3, 15)$, and $T_4 = (4, 20)$.

It is straightforward that an execution assignment can be obtained by scheduling each task as soon as possible and executing it as fast as possible. Thus, a first execution assignment $EA_1$ is given by: $EA_1(T_1) = [0, 2]$, $EA_1(T_2) = [2, 5]$, $EA_1(T_3) = [5, 8]$, and $EA_1(T_4) = [8, 10]$. By computing the execution rate as in Definition 3.3, we get $r_1 = 3/3 = 1$, $r_2 = 5/5 = 1$, $r_3 = 3/3 = 1$, and $r_4 = 4/4 = 1$. The computation times corresponding to $EA_1$ are identical the computational costs, that is, $c(T_1) = 3$, $c(T_2) = 5$, $c(T_3) = 3$, and $c(T_4) = 4$. The energy consumption corresponding to $EA_1$ is then $E(EA_1) = 1^2 \cdot 3 + 1^2 \cdot 5 + 1^2 \cdot 3 + 1^2 \cdot 4 = 15$.

Let us now make a slight change, by adding 1 time unit to the computation time of each task. The new execution assignment $EA_2$ is then given by: $EA_2(T_1) = [0, 4]$, $EA_2(T_2) = [4, 10]$, $EA_2(T_3) = [10, 14]$, and $EA_2(T_4) = [14, 19]$. One can see that all deadlines are still met, so the execution assignment is feasible. The new execution rates for each task are $r_1 = 3/4$, $r_2 = 5/6$, $r_3 = 3/4$, and $r_4 = 4/5$. As stated above, the computation times are now $c(T_1) = 4$, $c(T_2) = 6$, $c(T_3) = 4$, and $c(T_4) = 5$. The energy consumption corresponding to $EA_2$ is $E(EA_2) \approx (3/4)^2 \cdot 4 + (5/6)^2 \cdot 6 + (3/4)^2 \cdot 4 + (4/5)^2 \cdot 5 \approx 2.25 + 4.17 + 2.25 + 3.2 \approx 11.87$.

The second execution assignment has brought a significant reduction of the energy consumption. Nevertheless, this does not mean that $EA_2$ is necessarily the best possible execution assignment. There may be other execution assignments which require even less energy. The challenge is to develop a method for finding the lowest energy consumption. One hint already emerges from the example above: using the processors as much time as possible (i.e., reducing idle times) allows longer execution times and thus lower processor speed.

IV. IMPROVING THE POWER CONSUMPTION

As we saw in the previous section, the CPU power consumption depends on both the hardware configuration and the software implementation of the tasks. We will consider only the dynamic power, as it represents the dominant power component. We remind that the formal description is given by $P(s) = h \cdot s^\alpha$, where $\alpha$ is a hardware-dependent constant, $h$ is a parameter related to software implementation, and $s$ is the processor speed (frequency). In this paper, we will consider $\alpha = 2$.

As $h$ and $\alpha$ can be regarded as constants for a certain hardware/software environment, the power consumption can only be controlled by reducing the processor speed. Scheduling algorithms by themselves cannot provide power-aware solutions, as their aim is strictly to meet the deadlines for all tasks - they simply consider the processor working at its highest speed. However, in most cases, some of the tasks are finished before their deadlines. This means that, as soon as a feasible schedule has been determined, a lower power consumption can be achieved by allowing the processor the execute some of the tasks at a lower speed, while still meeting all deadlines. In other words, while the scheduling algorithms consider the value of 1 for the execution rates of all tasks, it is possible to allow lower execution rates for some tasks.

We make two observations:

- If two different feasible schedules are found for the same task set, they will exhibit the same power consumption. This is because a schedule is actually an ordering of the tasks in the set, while the total amount of CPU time required is the same (that is, the sum of the computational costs of all tasks). However, the potential of reducing the power consumption may be different for the two schedules.

- On a non-preemptive, multiprocessor system, reducing the power consumption on any processor can be done independently from the other processors. This is true because the amount of time spent with the execution of a certain task will only affect the execution of the other tasks scheduled on the same processor.

A. The optimization problem

As shown above, starting from a feasible schedule of a task set, it is possible to improve the power consumption by reducing the processor speed for some tasks, thus allowing the CPU to execute them in for longer periods of time than their computational costs. One must consider that, by allowing one task to finish later, the execution of the next task may also have to start later, and so on.

We consider here single-instance, non-preemptive scheduling and the EDF scheduling algorithm. We can then state the corresponding optimization problem:

Let $T = \{ T_1, \ldots, T_n \}$ be a set of non-preemptive tasks, where $T_i = (c_i, d_i)$, $\forall i \in \{1, \ldots, n\}$. The task set has been scheduled using the EDF algorithm, so $d_1 \leq d_2 \leq \ldots \leq d_n$. As shown in [3], the lowest energy consumption is achieved for the following execution assignment:

$EA(T_1) = [0, x_1]$

$EA(T_i) = [x_{i-1}, x_i]$, $\forall i \in \{2, \ldots, n-1\}$

$EA(T_n) = [x_{n-1}, d_n]$

where $(x_1, x_2, \ldots, x_{n-1})$ is the minimum point of the multivariate function $P$, defined as

$P(x_1, x_2, \ldots, x_{n-1}) = \frac{c_1^2}{x_1} + \frac{c_2^2}{x_2 - x_1} + \ldots + \frac{c_n^2}{x_n - x_{n-1}}$.
\[ \ldots + \frac{c_i}{x_{n-1} - x_{n-2}} + \frac{c_i}{x_n - x_{n-1}}. \]

The feasible schedule, as provided by the EDF algorithm (or any other scheduling technique), may have periods of time when the processor is idle, if all tasks terminate before the latest deadline (i.e., \( d_n \)). On the other hand, the power reduction technique uses the CPU continuously between 0 and \( d_n \), thus allowing lower execution speed.

Usually, in real-time systems only integer values are considered as solutions. This is due to the wide popularity of preemptive systems, where each time period is expressed as a multiple of the CPU time-slice allowed by the operating system. However, in non-preemptive systems, the concept of time-slice has little meaning, so the search for solutions expressed as real numbers is entirely justified.

Unfortunately, the complexity of the optimization problem makes it very hard to find a mathematical solution expressed as an analytical formula. For the simplest case, that is, a set consisting in only two tasks, an analytical formula has been derived [3]; however, for the general case, heuristic techniques may provide the best chances of finding the optimum. In this paper, we consider two such approaches:

- genetic algorithms, which are well-known for their adaptability to various classes of problems
- differential evolution, a meta-heuristic used mostly for numerical optimization problems

B. The genetic algorithm

A genetic algorithm [11], [15] works with a population of candidate solutions (individuals), where each candidate encodes a potential solution of the problem, usually as bit-strings. The population is initialized at random and then it is made to evolve throughout the generations, by applying the genetic operators (selection, crossover, mutation). The quality of each candidate solution is measured by the fitness function, which is defined to express the adequacy of the solution to the problem to be solved. The evolution of the population leads to the emergence of better candidate solutions (i.e., with better fitness values) determined by selection.

The algorithm designed for solving the current problem has the following characteristics:

**Representation.** Each candidate solution is represented as a vector of \( n - 1 \) values, where \( n \) is the number of tasks. The vector is denoted as \((y_1, y_2, \ldots, y_{n-1})\) and corresponds to the arguments of the multivariate function \( P \), defined above. Each gene (vector component) \( y_i \) encodes a value between a maximum and a minimum limit allowed for \( x_i \), defined as below:

\[
\begin{align*}
x_i &\leq X_{\text{max}_i} = \min_{j \in \{1, \ldots, n\}} (d_j - \sum_{k=i+1}^{3} c_k) \quad (x_0 = 0) \\
x_i &\geq X_{\text{min}_i} = \max(x_{i-1}, c_1 + c_2 + \ldots + c_i) 
\end{align*}
\]

Because floating-point representations are difficult to handle in genetic algorithms, each gene \( y_i \) stores an integer number between 0 and 1000, which locates \( x_i \) between \( X_{\text{min}_i} \) and \( X_{\text{max}_i} \):

\[
x_i = \frac{X_{\text{min}_i} + y_i \cdot (X_{\text{max}_i} - X_{\text{min}_i})}{1000}
\]

**Fitness function.** The fitness function is identical to function \( P \). Due to the formulation of the problem, the aim of the algorithm is to minimize the fitness values of the candidate solutions.

**Selection.** A certain (predefined) number of individuals are selected for crossover during each generation, using the technique called roulette wheel. Each time an individual has to be selected, a random variable is generated, and each individual in the population has a chance of being selected which depends on its fitness value; because in this problem better individuals have lower fitness values, the probability of selecting an individual is inversely proportional to its fitness value. This process is repeated independently until the desired number of selected individuals is achieved, which means that some individuals may be selected more than once.

**Crossover.** The crossover operator uses a single cutting point. Given two individuals selected for crossover, a position inside both individuals (the cutting point) is generated randomly. All genes before the cutting point are preserved, while the genes after the cutting point are switched between the two individuals. As a result, two new individuals (offspring) emerge, which are different from their parents.

**Mutation.** The mutation operator consists in choosing a gene of the individual and replacing its value with a new, randomly generated value. The mutation operator may affect all individuals in the population, but it is applied with a predefined probability, usually low, called mutation rate.

**Replacement.** After new individuals have been created by crossover, they replace some of the individuals from the population, whose size remains unchanged. This is the step leading to a new generation. The individuals to be replaced are selected randomly from the current population. If the algorithm uses elitism, then the individual with the best fitness value is always preserved in the population (i.e., it cannot be replaced).

C. Differential evolution

Differential evolution [40], [41] also works with a population of candidate solutions (individuals), whose quality is improving over the generations. Unlike genetic algorithms, differential evolution works naturally with floating-point values and is specifically designed for exploring multi-dimensional search spaces.

**Representation.** A candidate solution is represented as a vector of \( n - 1 \) values, where \( n \) is the number of tasks, corresponding to the arguments of function \( P \). Each item in the vector is a floating-point value between 0 and 1, and is also used to locate \( x_i \) between the same X\text{min}_i and X\text{max}_i limits, as in the case of the genetic algorithm:

\[
x_i = X_{\text{min}_i} + y_i \cdot (X_{\text{max}_i} - X_{\text{min}_i})
\]

**Fitness function.** The fitness function is identical to function \( P \). Again, the aim is to get candidate solutions with lower fitness values.

**Crossover.** The crossover operator creates a new individual for each existing one in the current population. This is done by performing the following steps:
Select at random three other individuals $U$, $V$, and $W$ (different from each other and from the current individual).

Select at random a position $R$ in the vector.

For each position $i$ in the vector, compute the value $x_i = U_i + F \cdot [(V_i - W_i)]/(F \cdot 2 + 1)$, where $F \in [0, 2]$ is a parameter of the algorithm. This value is used in the new individual with a predefined crossover probability (except for position $R$, where it is always used); otherwise, the value from the current individual is used.

**Replacement.** Each individual in the current population is compared with the corresponding new individual, created by crossover. If the new individual has a better fitness value, it replaces the existing individual in the population; otherwise, the existing individual is preserved.

### D. Experimental results

Four task sets have been used for testing, consisting in 4, 6, 8, and 10 tasks, respectively. Each task is represented by the pair $(c_i, d_i)$, as below:

1. $TS_1 = \{(20, 50), (30, 70), (30, 110), (20, 140)\}$
2. $TS_2 = \{(80, 90), (50, 140), (20, 200), (60, 270), (90, 330), (50, 410)\}$
3. $TS_3 = \{(50, 80), (60, 120), (50, 190), (90, 270), (40, 340), (100, 400), (30, 490), (90, 540)\}$
4. $TS_4 = \{(90, 160), (70, 240), (100, 290), (90, 410), (50, 550), (180, 630), (140, 750), (30, 820), (80, 900), (100, 1010)\}$

The genetic algorithm was run with the following parameters: population size 30, crossover rate 70%, mutation rate 10%, number of generations 100. In addition, the algorithm was run with and without elitism, in order to determine whether this can improve the behaviour of the algorithm.

The differential evolution was run with the same population size and number of generations. The value of parameter $F$ was set to 1, while the crossover rate was 90%.

Tables I to IV show the best results obtained by each algorithm (GA-E - genetic algorithm with elitism, GA-OUT - genetic algorithm without elitism, DE - differential evolution) after running 20 times. The best fitness from each run is inserted in the corresponding table. Because the optimal value is not available, the initial energy consumption corresponding to the EDF scheduling algorithm (i.e., with no optimization) is also indicated, in order to show the improvement brought by each approach.

The first observation derived from the experiments is that the use of elitism brings little improvement to the genetic algorithm. Although the non-elitist algorithm can loose some good solutions and has to start the search over again, it also exhibits a higher variability in the population, which leads to similar results. In most cases, the evolution of the average fitness in the population throughout the generations is the same for the elitist and the non-elitist algorithm.

The second observation is that differential evolution is surpassed by the genetic algorithms. Because replacement is
done in the population only when better individuals are found, good solutions are never lost. However, the technique proved less capable of efficiently exploring the search space. This is probably due to the fact that the crossover operator is designed less capable of efficiently exploring the search space. This is good solutions are never lost. However, the technique proved done in the population only when better individuals are found, good solutions are never lost. However, the technique proved less capable of efficiently exploring the search space. This is probably due to the fact that the crossover operator is designed less capable of efficiently exploring the search space. This is done in the population only when better individuals are found, good solutions are never lost. However, the technique proved

V. CONCLUSIONS

This paper approaches the problem of power saving in single instance, non-preemptive real-time systems. As consumption depends on the processor frequency, the tasks are executed for a longer time than strictly necessary, thus allowing the processor to slow down. However, the tasks have specific constraints, given by their deadlines, so the problem is to find the appropriate working frequency for each task in order to minimize the global consumption.

The multivariate optimization problem emerging from these requirements does not allow a general analytical solution, which leads to the use of heuristic techniques. Experimental results show that genetic algorithms have the potential to achieve significant improvement in energy consumption. To a lesser extent, a specialized meta-heuristic like differential evolution is also capable of providing good results.

REFERENCES

[27] J. D. Lin, A. M. Cheng, and R. Kumar. Real-time task assignment in heterogeneous distributed systems with rechargeable batteries. In


