Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 2, No. 1, April 2016.

PERFORMANCE EVALUATION OF META-HEURISTICS IN ENERGY AWARE REAL-TIME SCHEDULING PROBLEMS

Ashraf Suyyagh[1], Jason G. Tong[2] and Zeljko Zilic[3]

Department of Electrical and Computer Engineering,
McGill University, Montreal, Canada
ashraf.suyyagh@mail.mcgill.ca[1], jason.tong@mail.mcgill.ca[2],
zeljko.zilic@mcgill.ca[3]

(Received: 15-Dec.-2015, Revised: 03-Feb.-2016, Accepted: 10-Feb.-2016)

ABSTRACT

Energy efficient real-time systems have been a prime concern in the past few years. Techniques at all levels of system design are being developed to reduce energy consumption. At the physical level, new fabrication technologies attempt to minimize overall chipset power. At the system design level, technologies such as Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) allow for changing the processor frequency on-the-fly or go into sleep modes to minimize operational power. At the operating system level, energy-efficient scheduling utilizes DVFS and DPM at the task level to achieve further energy savings. Most energy-efficient scheduling research efforts focused on reducing processor power. Recently, system-wide solutions have been investigated. In this work, we extend on the previous work by adapting two evolutionary algorithms for system-wide energy minimization. We analyse the performance of our algorithms under variable initial conditions. We further show that our meta-heuristics statistically provide energy minimizations that are closer to the optimum 85% of the time compared to about 30% of those achieved by simulated annealing over 500 unique test sets. Our results further demonstrate that in over 95% of the cases, meta-heuristics provide more minimizations than the CS-DVS static method.

KEYWORDS

Real-time systems, Embedded systems, Energy-aware scheduling, Meta-heuristics, DVFS, DPM.

1. INTRODUCTION

Embedded systems are evolving into cyber-physical systems; highly interconnected, tightly-coupled systems with the physical world. The consolidation of the physical world into the interconnected virtual world requires the use of many devices. Cyber-physical systems heavily rely on sensors, communication devices and even the cloud. Such devices claim significant portion of the system power profile and their share can no longer be excluded in energy minimization. As a result, system-wide power reduction becomes a significant challenge.

Traditionally, research efforts have focused on processor power optimizations. The processor dynamic power consumption highly depends on operational voltage and frequency as given by Equation 1:

$$P_{\,dynamic} = C_{eff}.V_{dd}^{2}.f \tag{1}$$

where C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage and f is the operational frequency [21]. With the advancement of fabrication technologies and the shrinking of transistors' size into the nanometer scale, effects of sub-threshold leakage current become more prominent. Transistor miniaturization allows for the reduction of the supply voltage. However, sub-threshold leakage current I_{sub} is inversely proportional to V_{dd}. This becomes more evident in Equation 2:

$$I_{sub} = K_1 e^{\frac{-V_{th}}{nV_\theta}} (1 - e^{\frac{-V_{dd}}{V_\theta}}) \tag{2}$$

where K_1 and n are experimentally derived, V_θ is the thermal voltage which is 25mV at room temperature and increases linearly with temperature. Increasing V_{th} to reduce I_{sub} is not a viable option, as any increase in V_{th} will reduce the maximum processor frequency and therefore affect the performance. The relationship between threshold voltage V_{th} and frequency f is given by Equation 3:

$$f \propto \frac{(V_{dd} - V_{th})^\alpha}{V_{dd}} \tag{3}$$

where α is a technology parameter. To reduce processor dynamic power, Dynamic Voltage and Frequency Scaling (DVFS) was introduced [1]-[2]. DVFS allows for the run-time change of operating voltages/frequencies. Modern operating systems can access and operate processor DVFS hardware. On the other hand, to reduce leakage current, Dynamic Power Management (DPM) techniques are employed to turn off the processor [3] or inactive devices [4]-[5] when idle.

Previous research has targeted system-level optimizations. It was shown that when system devices are involved, the interplay between processor DVFS/DPM and device DPM techniques is complex [6]-[8]. DVFS techniques could lower processor energy but increase device energy. Many devices are expected to remain in the active state for the entire duration the task is running on the processor [9]. Devices are allowed to switch into their low power states if they are no longer used by successive scheduled tasks and only if it is energy efficient to do so. As task execution times are scaled using DVFS techniques, the time the associated devices are expected to be powered on is scaled as well, and more energy is consumed [9]-[10]. Moreover, even though employing DVFS reduces dynamic power, the reduction of V_{dd} increases leakage current, and the prolonged execution of the task due to frequency scale down increases the overall leakage power consumption. Therefore, an optimal frequency scaling assignment needs to balance and minimize the overall system energy; that is of the processor and the devices combined. This problem is known to be an NP-Hard problem [11]. As such, design time algorithms developed specialized heuristics [11]-[12] or were based on mathematical optimizations such as integer linear programming [9].

This work extends on previous research by adapting different meta-heuristics to minimize system-wide energy and evaluating their performance. Meta-heuristics have been used in scheduling problems related to energy minimization, load balancing [13], [22] or makespan minimization. The genetic algorithm (GA) is one the earliest meta-heuristic evolutionary algorithms employed for task scheduling and partitioning and for test set generation [14]-[16]. Differential Evolution (DE) is one of the newer evolutionary algorithms which differs in that it is not biologically inspired, but rather relies on stochastic approaches [17]. Simulated annealing (SA) is another meta-heuristic which approximates a global optimum of an NP-hard problem. Simulated annealing was employed in [10] to minimize power consumption of a periodic hard-real time system. We summarize our contributions in this work as follows:

- We propose and adapt two meta-heuristic evolutionary algorithms to find the DVFS configurations which minimize system energy. We analyse and compare the performance proposed algorithms against each other and against previous work.
- We investigate the optimal parameters for these algorithms and attempt to establish confidence in the ability of such algorithms in producing near-optimal energy savings.

The organization of the paper is as follows: Section 2 introduces the system's task and energy models. Section 3 describes the proposed algorithms as well as the reference algorithms. The simulation methodology is presented in Section 4 and our analysis and results are summarized in Section 5.

2. SYSTEM MODELS

In this section, the power and task models used in this paper are presented. The notations used to describe task and power parameters are listed in Table 1.

Table 1. List of notations used in the paper.

τ_i	Task i, i = 1, 2, … N
D_i	Deadline of Task i
C_i	Worst Case Execution Time (WCET) of i under maximum system frequency F1
T_i	Period of task τ_i
F_i	Normalized discrete CPU frequency levels, $F_i \in F_1, F_2 \dots F_Q$ and $F_1 > F_2 > .. > F_Q$
d_k	Device k, k = 1, 2, …K
s	The deep sleep mode where the least power is consumed
$t_{sw}^{()}$	Total switching time overhead when the processor or device d_k is switched down from active state to the low power state $t_{sd}^{()}$ and up from the low power state to the active state $t_{su}^{()}$
$E_{sw}^{()}$	Total energy dissipated when the processor (CPU) or device d_k is switched down from active state to the low power state $E_{sd}^{()}$ and back up from the low power state to the active state $E_{su}^{()}$, respectively
$P_s^{()}$	Device d_k or processor (CPU) power consumed while the device is in the low power state s
$P_a^{d_k}$	Device d_k power consumed in the active state
$P_{F_i}^{CPU}$	Processor power consumed in the active state while running at frequency level F_i
$t_{BET}^{()}$	Break even time of device d_k or the processor

2.1 Task Model

The system is a hard real-time system based on a set of N independent, periodic and fully pre-emptible tasks with implicit deadline model (deadline equals the period). Tasks are assumed schedulable under EDF policy when executed with no frequency or voltage scaling techniques employed. Each task τ_i is represented by the tuple (C_i, D_i, T_i) denoting task worst case execution time, deadline and period, respectively. Each task τ_i is assigned a number of devices d_k and a frequency scaling factor F_i. Similar to previous work [10], [23], we assume an inter-task device scheduling model. That is, devices are available, active and running throughout the associated task run time. Devices can be switched into an inactive state only at the end of the associated task run-time. Switching the device to low power state takes place if the device is no longer needed for a subsequent task and if it is energy-efficient to go to low-power state. The hyper-period (HP) is the least common multiple of all task periods and represents the period in which the task scheduling pattern repeats. Utilization is measured by $\sum_1^N \frac{C_i}{T_i}$ and must be less than or equal to one for a feasible EDF schedule. In accordance with previous research [9]-[10], [7], the relationship between task execution time and frequency scaling is assumed to be linear and the execution time for task τ_i after scaling is measured by C_i/F_i. Slack time (processor / device idle time) is only utilized for the possibility of switching the processor / device to low-power mode.

2.2 System Power Model

We consider a processor and devices which have one active state and one low power (deep sleep) state s. The deep sleep state s is the state where most of the processor/device components are turned off. This model is used to keep the design space exploration for the algorithms proposed in this paper manageable. Yet, the system could be readily extended to support multiple low power states s_i. In the active state, the processor is capable of executing tasks at one of Q-discrete frequencies. F_i is the normalized frequency corresponding to frequency f_i; where $f_i > f_{i+1} > \ldots > f_Q$. The frequencies are normalized to the highest system frequency f_1. Switching from the processor active state to the lower power states s entails switching time and energy overheads, defined as t_{sw}^{CPU} and E_{sw}^{CPU} , respectively. The switching overheads include both the switching overheads from active to low power state and vice versa. The power consumed while the processor is in the active state depends on the currently selected frequency and is denoted $P_{F_i}^{CPU}$. Power consumed while the processor is in deep sleep state s is denoted as P_s^{CPU}.

Similar to the processor model, the device power consumed in active and low power states are denoted as $P_a^{d_k}$ and $P_s^{d_k}$, respectively. And device time and energy switching overheads between the active and low power state are represented by $t_{sw}^{d_k}$ and $E_{sw}^{d_k}$, respectively.

Processors' and devices' transition from their active states to a lower power state occurs only when the transition is power-efficient. Switching states is considered power-efficient if the total switching power between active, low-power to active states is less than the power consumed if the processor/device is kept idle in the active state. The decision to switch to a lower power processor or device state is based on the break-even time t_{BET} . t_{BET} represents the minimum idle time threshold required to switch to a lower power state to satisfy the power-efficiency condition. Equation 4 computes the break-even time for the processor [18].

$$t_{BET}\ (CPU) = max\left(t_{sw}^{CPU}, \frac{E_{sw}^{CPU} - P_s^{CPU} \times t_{sw}^{CPU}}{P_{F_i}^{CPU} - P_s^{CPU}}\right) \tag{4}$$

Similarly, Equation 5 computes the break-even time for any device d_k.

$$t_{BET}\ (d_k) = max(t_{sw}^{d_k}, \frac{E_{sw}^{d_k} - P_s^{d_k} \times t_{sw}^{d_k}}{P_a^{d_k} - P_s^{d_k}}) \tag{5}$$

3. ENERGY AWARE SCHEDULING ALGORITHMS

In this section, we propose and adapt two meta-heuristics for system wide energy minimization. One is based on a discrete implementation of genetic algorithm for frequency scale assignment. The other is based on the newer differential evolution algorithm. In section 5, we compare these algorithms against each other and against the simulated annealing meta-heuristic based on the recent work of [10]. We also compare the results to the famed heuristic algorithm CS-DVS [7] which is one of the most powerful and regularly cited algorithms. We summarize the simulated annealing algorithm and present the CS-DVS for completeness at the end of this section.

3.1 Definitions

Before introducing the proposed energy aware scheduling algorithms, a few definitions need to be presented, as they are frequently encountered in the subsequent sections.

Definition 1: A power configuration is defined as a permutation of DVFS frequency assignments of dimension N (mapped to each task τ_i), a power cost variable and a set of status flags. Flags convey information on the feasibility of the configuration or control decision paths within the algorithm.

Definition 2: The quality of a configuration (and algorithm) is defined as how well it is able to minimize system energy to near optimal values (within 1% of optimal).

Definition 3: A feasible power configuration is one which satisfies EDF scheduling feasibility condition when DVFS is employed; that is $\sum_{1}^{N} \frac{C_i}{T_i \times F_i} \leq 1$.

3.2 Genetic Algorithm Frequency Scaling (GAFS)

The genetic algorithm is an optimization algorithm based on the principles of genetics. The genetic algorithm mimics the process of natural selection and the survival of the fittest. The algorithm starts with a population of random solutions of a certain size that is allowed to evolve through time towards global optimum. Each member of this population is called a chromosome. Each chromosome consists of a set of variables which are called genes.

The genetic algorithm passes through multiple iterations. In each iteration, the cost of each chromosome is evaluated by applying the chromosome genes (variables) into the objective function. The objective function represents the problem which we aim to minimize or maximize. The chromosomes are then sorted in terms of their cost. Half of the chromosomes which have costs closer to the optimum are maintained, while the other half is discarded. This set of preserved chromosomes is called parents. Parents are used to generate the other missing half of the population which is termed offspring or children. Parent chromosomes are paired amongst each other. Genes are exchanged between paired chromosomes in an operation called crossover. There exist many techniques and strategies for pairing parents and gene crossover [20]. The current population of parent and child chromosomes is subjected to a mutation operation. A certain percentage of genes across all chromosomes is randomly selected and altered. This is to mimic genetic mutations which occur in nature. This new population is now processed in the very same manner in the next iteration. The algorithm iterates until it converges to a solution.

In our work, a discrete non-binary version of the genetic algorithm is implemented to find a power configuration that minimizes system power. The power configuration used in GAFS is comprised of a chromosome *c* of *N* genes and a set of configuration flags, where *N* is the system tasks count. Each gene represents a possible task frequency assignment F_i for task τ_i and is initialized by the index *i* of the frequency scale level F_i assigned to the task. This is due to the fact that we are using a discrete and integer genetic algorithm. The flags specify whether the configuration is feasible, a parent chromosome or in mutated state. Mutated chromosomes are the ones with one or more genes randomly changed through the mutation operation. Crossover and mutation operations are detailed below. The genetic algorithm for frequency assignment (GAFS) is listed in Algorithm 1.

Initially, a set of power configurations of size *NP* are initialized with random frequency scaling factors such that each configuration is feasible. All configurations are set to be parents and in an unmodified state. Each initial configuration is run for one hyper-period and the system power cost is computed and assigned to the configuration. The lowest power *NP/2* chromosomes are selected as parents and sorted from lowest to highest power. A top-bottom pairing approach is used to pair parents from the parent pool. A one point crossover operation is performed, where genes are exchanged between the paired chromosomes. No minimum less than 25% and no maximum more than 40% of chromosome genes take part in the crossover process. After the exchange, a mutation operator is applied on the whole population except for the elite chromosome, which is the one that yielded the overall minimum system energy. The elite chromosome carries from one generation to another and is only replaced if another chromosome yields lower system power. The number of mutations applied is calculated according to Equation 6.

Algorithm 1 GAFS

```
Initialization:
iteration ← 0
for $i \leftarrow 1, NP$ do                                  ▷ NP: Population Size
    $c_i$ ← Random and feasible frequency scales
    Set $c_i$ as parent, feasible, and unmodified
    Run $c_i$ in next HP - measure and store power
    iteration ← iteration + 1
end for
while iteration $< max\_hp$ do
    Produce Next Generation:
    Sort NP configurations from min to max power
    Select best $\frac{NP}{2}$ configurations for crossover and mutation
    Pair best configurations from lowest to highest power
    Perform one point cross over and replace discarded configurations
    Set state for new configurations as child
    for $n \leftarrow 1, \#mutations$ do
        Randomly choose a configuration from new generation (exclude best/elite)
        Randomly mutate frequency assignment to new value
        if chosen configuration is parent then
            Change parent state to modified
        end if
    end for
    for $j \leftarrow 1, NP$ do
        if new $c_i$ is unfeasible then
            Set power of unfeasible configuration to ∞
        end if
    end for
    for $n \leftarrow 1, NP$ do
        if new $c_i$ is feasible configuration then
            if new $c_i$ is child or modified parent then
                Load new configuration $c_i$, measure power in HP
                iteration ← iteration + 1
            end if
        end if
    end for
    if No configuration is feasible in current generation then
        Terminate. Choose elite chromosome as solution
    end if
end while
```

$$Mutations = \mu \times (NP - 1) \times N; \tag{6}$$

where *NP* and *μ* are the population size and the mutation factor, respectively.

Finally, each new chromosome configuration is checked for feasibility. If it fails, then its cost is set to ∞. Parent chromosomes which have undergone mutation are marked as such.

Only feasible child and modified parent configurations are run in subsequent hyper-periods. This eliminates redundant computation for the elite and unmodified parents. The whole process repeats for each generation of power configurations until the maximum hyper-periods specified are reached or the algorithm converges. Ideally, if each generated chromosome is feasible, then the lower bound of the number of generations produced is $\left\lceil \frac{HP}{NP} \right\rceil$, where HP is the number of test hyper-periods. The upper bound case will be when the algorithm is only able to produce one feasible configuration per generation. The upper bound will be equal to 1 + HP - NP. If the algorithm returns no feasible chromosomes to be run, then it stops and the elite chromosome is used as a solution.

3.3 Differential Evolution Frequency Scaling (DEFS)

Differential evolution is an optimization algorithm which belongs to the same group of evolutionary algorithms as that of the genetic algorithm. Differential evolution is founded on stochastic principles to find a solution. The algorithm maintains a set of solutions called candidate vectors which evolve through iterative operations. Each vector is comprised of a set of variables which are applied to the objective function to calculate the cost of a solution.

The differential evolution algorithm passes through multiple iterations. In each iteration, for each vector in the population (called base or parent vector), a new vector is created. Each new vector is generated from the addition of a scaled difference of two different candidates to another third candidate. The set of new candidates are called donor vectors. A new vector called the trial vector is generated from each base vector and its corresponding donor vector based on a certain probability. The cost of the trial vector is measured. It replaces its corresponding base vector only if it is closer to the optimum. Otherwise, it is discarded.

In our adaptation of the differential evolution algorithm which is listed in Algorithm 2, a configuration in DEFS is comprised of an N-dimensional vector v, a feasibility flag and a cost variable. N is the number of system tasks. In a population of a size of NP configurations, each configuration is initialised with random and feasible frequency scales (i.e., the index i of the frequency scale level F_i assigned to the task) and run in subsequent hyper-periods. System power consumption of initial configurations is recorded for each configuration in each hyper-period.

To produce the next set of candidate configurations, for each base vector in the population, three different vectors are randomly chosen. A donor vector t_v is computed from these three vectors on an element-by-element basis using a mutation formula as shown in Equation 7:

$$t_{v_i} = round\ (t_{v_1} + \varphi.\left(t_{v_2} - t_{v_3}\right)); \tag{7}$$

where $i \neq i_1 \neq i_2 \neq i_3$ and Φ is the vector difference scaling factor (mutation factor) and should not be confused with frequency scaling factors. Rounding to integer is one form of discretising the continuous version of DE algorithm. A boundary check follows to constrain the frequency scales to fall within the supported processor frequency levels according to Equation 8:

$$t_{v_i} = \min(f_{max}, \max(f_{min}, t_{v_i})); \tag{8}$$

where f_{min} and f_{max} are the lowest and highest frequency scales supported by the processor, respectively. Finally, the donor vector t_{v_i} is crossed over on an element-by-element basis with its parent (base) v_i, the i^{th} vector of the population using Equation 9:

$$u_i[j] = \begin{cases} t_{v_i}[j] & if\ r_j > CR \\ v_i[j] & otherwise \end{cases} \tag{9}$$

where j is the j^{th} element of vectors v_i, t_{v_i} and $j \in [1, N]$. r_j is a randomly generated number for each element j, where $r_j \in [0; 1]$. CR is the crossover probability used as a control element for the differential evolution algorithm, $CR \in [0, 1]$.

Algorithm 2 DEFS

```
1: Initialization:
2: iteration ← 0
3: for $i ← 1, NP$ do                                          ▷ NP: Population Size
4:     $v_i$ ← Random and feasible frequency scaling
           configuration
5:     Set $v_i$ as feasible
6:     Run $v_i$ in next HP - measure and store power
7:     iteration ← iteration + 1
8: end for
9: while iteration < $max\_hp$ do
10:    Produce Next Generation:
11:    for i ← 1, NP do
12:        Get donor vector $t_{v_i}$ for parent (base) $v_i$
13:        Perform boundary checking and correction on $t_{v_i}$
14:        Trial vector $u_i$ ← crossover between donor vector $t_{v_i}$ and base vector $v_i$
15:        if $u_i$ is feasible then
16:            Run trial $u_i$ in next HP and measure power
17:            iteration ← iteration + 1
18:        else
19:            Set power of $u_i$ to $\infty$
20:        end if
21:        if Power($u_i$) < Power($v_i$) then
22:            Replace $v_i$ with new candidate configuration $u_i$
23:        end if
24:    end for
25: end while
```

Each of the trial vectors undergoes a schedulability check and its feasibility flag is set accordingly. If it is unfeasible, then its cost is set to ∞ ensuring that it will never replace its parent (base vector). Only feasible configurations are allowed to execute in the next hyper-period. Once the trial vector power is measured, a replacement check is conducted according to Equation 10:

$$v_i = \begin{cases} u_i & if\ Power(u_i) < Power(v_i) \\ v_i & otherwise \end{cases} \tag{10}$$

where $Power(u_i)$ is the trial configuration power within the hyper-period and $Power(v_i)$ is the power of its parent (base) configuration. Candidate configurations will be generated until the number of iterations hits the specified maximum or the algorithm converges.

3.4 Critical Speed – Dynamic Voltage Scaling (CS-DVS)

Each running task consumes both processor dynamic and leakage power, as well as power that is related to all associated system devices. Increasing the voltage/frequency scale of the processor leads for the task to have a shorter execution period on the expense of consuming higher dynamic power. Since the task execution time has been reduced, the leakage power and all associated devices' power is also reduced. This is due to the fact that associated devices are kept in an ON/wait state for a shorter time. The converse is equally true when the voltage/frequency scales are reduced.

As such, using the lowest frequency/voltage scales does not necessarily lead to minimum system power due to increased leakage power effects and prolonged device execution/wait time. Critical speed is defined as the speed which minimizes the overall dynamic, leakage and device power. Due to the different set of associated devices that a task uses, each task may have a different critical speed which minimises its power consumption. To find the critical speed of a task τ_i, the power consumed by task τ_i is measured at each frequency scale F_i. The frequency scale F_i which results in the minimum task power is thus selected as the one corresponding to its critical speed.

Setting all system tasks to run at their critical speeds does not ensure feasibility, as the critical speed scales might cause system tasks to miss their deadlines. To maintain a schedulable system, it is necessary to select and run certain tasks at higher speeds than the critical speed.

Algorithm 3 shows the description of the CS-DVS algorithm which was presented in [7]. The algorithm runs in two stages. Initially, it computes the critical speed of each system task. Then, to maintain feasibility, it determines which tasks and by which factor their frequency/voltage scales need be increased while maintaining minimum power consumption. As long as the system is not in a feasible state, all tasks which are not already assigned the maximum system speed are candidates to have their frequency scales increased to the next scale. The power difference between having a task run at its current scale and the next scale is computed. The task with the minimum power consumption penalty is chosen and its scale is adjusted to the next one. The process repeats until a feasible schedule is achieved.

Algorithm 3 CS-DVS [7]

```
1:  Initialisation:
2:  Compute critical speed of each task $\tau_i$
3:  Set frequency scale $F_i$ of task $\tau_i$ to that of the critical speed
4:  while (configuration not feasible) do
5:      for All tasks not running at max processor speed $F_1$ do
6:          Compute task associated power at next higher frequency scale
7:          Compute task power consumption difference between current
            and next higher frequency scale
8:      end for
9:      Choose task with lowest increase in power
10:     Set chosen task frequency scale $F_i$ to $F_{i-1}$
11: end while
```

3.5 Simulated Annealing (SA)

Simulated annealing is a meta-heuristic that approximates a global optimum of an NP-hard problem. The algorithm starts with a solution and explores neighbouring solutions that move towards a global optimum. A neighbouring configuration is defined as that which differs from the current configuration by one value in the set of solutions (in our case one frequency scale). A neighbouring solution that is closer to the global optimum always replaces the current solution. To avoid falling into a local minimum, a worse solution could be accepted based on a certain acceptance probability. The rationale behind this is that even though a worse solution is accepted, the neighbouring solutions of the worse solution could potentially move us toward the global optimum. In our work, we follow the adaptation of simulated annealing meta-heuristic for system energy minimization as presented in [10] with minor modifications.

Similar to our proposed algorithms, SA starts with a feasible configuration *J* which is a vector of size *N* tasks. The frequency scales assigned to the initial vector are set to those resulting from applying the CS-DVS algorithm. The initial configuration is executed in one hyper-period and its power consumption cost is recorded. One frequency scale in the current configuration *J* is randomly changed to another supported scale to generate a neighbouring configuration *J**. *J** is checked for feasibility. If feasible, the cost of this neighbour configuration *J** is measured and recorded in a subsequent hyper-period; otherwise, another neighbour *J** is generated from *J* until a feasible neighbour is found. If the newly found neighbour configuration minimizes the system energy more than the current configuration, the optimized configuration replaces the current configuration *J*. However, if the neighbour configuration results in more system power consumption, it still can replace the current configuration *J*. For this case, a random probability $\rho \in [0, 1]$ is generated and acceptance probability α is computed according to Equation 11:

$$\alpha = e^{\frac{Power(J)-Power(J*)}{Power(J)*K}} \tag{11}$$

where *K* is the annealing factor to be decided through experimentation. If α > ρ, then the worst solution replaces the current solution. The algorithm stops when the number of test hyper-periods is reached. The SA algorithm is listed in Algorithm 4.

Algorithm 4 Simulated Annealing

```
1: Initialisation:
2: Set initial configuration $J$ to the output of CS-DVS [7]
3: Run $J$ in next HP - measure and store power
4: iteration ← iteration + 1
5: Produce Next Neighbour:
6: while iteration < $max\_hp$ do
7:     Generate neighbouring configuration $J*$
8:     if $J*$ is feasible then
9:         Run $J$ in next HP - measure and store power
10:        if Power ($J*$) < Power ($J$) then
11:            $J = J*$
12:        else
13:            Generate random probability $\rho$
14:            Compute acceptance probability $\alpha$
15:            if $\alpha > \rho$ then
16:                $J = J*$
17:            end if
18:        end if
19:        iteration ← iteration + 1
20:    end if
21: end while
```

4. Simulation

To analyse the performance of the proposed algorithms for frequency scaling and system wide power reduction, we devised a set of experiments. We developed an event-driven simulator using SystemC 2.3.0 and TLM. Processor and device power models are consistent with previous work [10], [18]. They are based on the Intel XScale processor power profile and the device set shown in Tables 2 and 3, respectively.

Table 2. Intel XScale processor power model.

Frequency Steps (MHz)	1000	800	600	400	150
$P_{F_i}^{CPU}$ (Watt)	1.6	0.9	0.4	0.17	0.08
Voltage (V)	1.55	1.45	1.35	1.25	1.15
E_{sw}^{CPU} = 0.5 mJ		t_{sw}^{CPU} = 85ms			

For each task set of size *N*, where $N \in [1 - 9]$, a total of 500 random and unique task sets are generated. The upper limit for task set size is limited by the exhaustive search time for an optimal solution. Each task τ_i is randomly assigned a unique device set, where the number of different devices per task ∈ [0 - 2]. Each device is randomly chosen from the device set shown in Table 3. The periods of the tasks are randomly and uniformly chosen from the range of [0.5 - 100] ms. We assume that the periods of tasks in real-time systems are harmonic [10], so that they can help in reducing the simulation time. The algorithm in [19] is employed for this purpose. Tasks WCETs are randomly selected to be between 2% and 40% of the original unmodified task period. Simulations are run ten times per test case for a total of 5000 simulations for each task set size.

Table 3. Devices power model.

Device	$P_a^{d_k}$ (W)	$P_s^{d_k}$ (W)	$P_{su}^{d_k}$ (W)	$P_{sd}^{d_k}$ (W)	$t_{su}^{d_k}$ (W)	$t_{sd}^{d_k}$ (W)
Realtek Ethernet Chip	0.187	0.085	0.125	0.125	0.01	0.01
IBM Microdrive	1.3	0.1	0.5	0.5	0.12	0.12
SST Flash SST39LF020	0.125	0.001	0.05	0.05	0.001	0.001
SimpleTech Flash Card	0.225	0.02	0.1	0.1	0.002	0.002
MaxStream Wireless Module	0.75	0.005	0.1	0.1	0.04	0.04
$P_{su}^{d_k} = E_{su}^{d_k} \times t_{su}^{d_k}$			$P_{sd}^{d_k} = E_{sd}^{d_k} \times t_{sd}^{d_k}$			

The next step is to find the best tuning parameters to initialize the meta-heuristics algorithms; namely, population size, mutation factors and crossover probabilities. Population size for both GAFS and DEFS is set to 8, 16 and 24 with an additional 32 test case for GAFS. GAFS mutation rates are set to 0.1 and 0.2. Larger mutation rates could in theory make it harder to converge as the algorithm will keep jumping between search points. Lower values could possibly lead to premature convergence and produce non-optimal results [20]. In DEFS, we chose crossover probabilities CR of 0.3, 0.5 and 0.7. We also chose the same range of mutation factors (scaling factors). We chose these values as uniform probability samples in the range [0-1]. All simulations are investigated over hyper-period sizes of [50, 100, 200 and 400]. We use two additional hyper-periods 1000 and 2500 with task sizes of 7 and 9. We assume the scheduling overhead to be low and therefore neglected.

An exhaustive search with all possible task DVFS permutations is carried out to obtain the optimal value with minimum system power. The optimal configuration serves as a reference for testing the quality of the configuration found by the algorithms under investigation. The performance of the algorithm is measured by how often the near optimal results are produced in every single case. This gives confidence in the ability of the algorithm performance to minimize system power. Finally, we simulate the CS-DVS [7] algorithm and the SA algorithm from [10] for comparison purposes.

5. Results and Discussion

In this section, we report the sensitivity results of the proposed algorithms as well as the simulated annealing algorithm to the different tuning parameters. We also compare the proposed algorithms to previous work in terms of how close they are to optimal energy savings, and how much they yield better results than the well-established CS-DVS algorithm. The base energy savings from DVFS assignments are shown for the static CS-DVS and optimal search in Figure 1. These results serve as a baseline for comparing the quality of the proposed and previous algorithms.

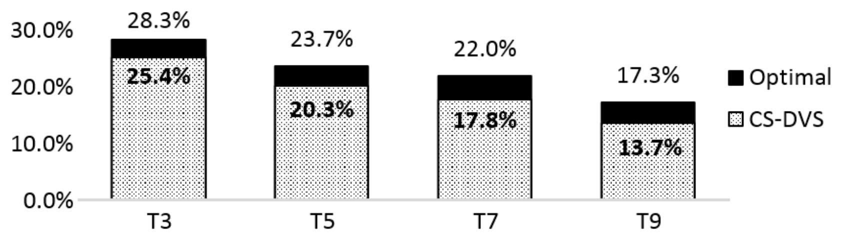

Figure 1. Average CS-DVS and optimal DVFS power savings over 500 unique sets. Tx denotes a task set with x tasks.

5.1 Sensitivity Analysis

In this sub-section, we present the performance of the proposed algorithms when their initial parameters are changed. Mainly, we vary the parameter under investigation, while the values of the other parameters remain fixed. Our experiments include varying the number of hyper-period iterations for which the algorithm is simulated. This is to analyse the convergence of the algorithms and their possible early termination effects (i.e., GAFS no longer has a pool of feasible chromosomes). We also vary mutation rates and crossover probabilities and report our findings. The results in Table 4 through Table 11 are shown for the cases where one variable is studied, while the others are fixed at the values which gave the best overall results.

The effects of running the algorithms over more hyper-periods (generations) is shown in Table 4 and Table 5 for a sample of the tasks for both GAFS and DEFS algorithms. Since the assignment space for *N = 3* is small, an exhaustive search will always guarantee an optimal result in fewer HPs than running either GAFS or DEFS (5^3 compared to 400). The base five corresponds to the number of frequency levels supported by our model processor as shown in Table 2. However, for larger task sets, meta-heuristics deliver near-optimal energy savings in much less time (within 2500 HPs compared to 5^9 HPs for N = 9). The majority of the results for GAFS are near-optimal. Our observations show that setting the HP test limit to around 3% of the search space 5^N yields good results. The more hyper-periods the algorithm runs through, the better the overall results. This allows for more time for the algorithm to converge towards a near optimal solution as it is testing more chromosomes as potential solutions.

Table 4. GAFS near-optimal power savings sensitivity to hyper-period.

Tasks	Hyper-period					
	50	100	200	400	1000	2500
T3	43.3%	67.1%	92.8%	**97.5%**	-	-
T5	22.9%	36.2%	67.7%	**84.6%**	-	-
T7	13.2%	17.7%	32.8%	57.5%	70.1%	**78%**
T9	13.9%	17%	27%	46.9%	68.6%	**85.3%**
	GAFS population size = 32					

Table 5. DEFS near-optimal power savings sensitivity to hyper-period.

Tasks	Hyper-period					
	50	100	200	400	1000	2500
T3	35.8%	42.2%	69.4%	**93.6%**	-	-
T5	19.5%	22.6%	33.4%	**67.1%**	-	-
T7	12.2%	13.1%	17.1%	36%	70.9%	**82.8%**
T9	12.5%	13.1%	16.5%	28.8%	62.4%	**86.5%**
	DEFS population size = 24, CR = 0.3 and φ = 0.5					

The next step is conducting analysis on varying the initial population size. That is, we change the number of initial feasible chromosomes for the GAFS algorithm, as well as the initial candidate vector pool for DEFS. In Table 6, we observe that larger population sizes in GAFS yield better results with wider margins for larger task sets. Larger population sizes make it possible to have more pairings and crossover possibilities that in turn allow for better exploration of the search space. A population size of eight only has two pairs of parents to generate an offspring compared to 16 pairs in a population of 32. The difference between a population size of 24 and 32 is insignificant in most of the cases.

Table 6. GAFS near-optimal power savings sensitivity to population size, μ = 0.1.

Population size		8	16	24	32
Task No.	HP	Percentage of near-optimal power saving configurations			
T3	400	95.9%	96.9%	96.9%	**97.5%**
T5		78.3%	82.1%	**84.6%**	**84.6%**
T7	2500	67.7%	75.3%	**78.0%**	**78.0%**
T9		77.1%	82.4%	84.6%	**85.3%**
Task No.	HP	Percentage of configurations better than CS-DVS			
T3	400	94.9%	95.0%	**95.0%**	**95.0%**
T5		94.8%	96.7%	97.6%	**98.0%**
T7	2500	95.8%	97.5%	**98.2%**	98.0%
T9		97.4%	98.3%	98.4%	**98.7%**

In GAFS, we limited our study of mutation rates to rates of μ = 0.1 and μ = 0.2. This is to keep by the suggestion in [20] of using low mutation rates to ensure algorithm convergence. Table 7 shows power savings sensitivity to GAFS mutation rates and task set size when the population size is fixed at 32. As seen from the table, we observe that lower μ= 0.1 gives overall better results and only in few cases μ = 0.2 results in marginal gains. Given the population size and the chromosome size of our problem, lower mutation rates were expected to give better results. Higher mutation rates would entail exploring further away from our current best results. As μ increases, the closer the genetic algorithm gets to a random search. GAFS outperforms CS-DVS in most cases, especially with larger task set sizes.

Table 7. GAFS near-optimal power savings sensitivity to mutation factors, population size = 32.

		Percentage of near-optimal power saving configurations		Percentage of operations better than CS-DVS	
Task No.	HP	μ = 0.1	μ = 0.2	μ = 0.1	μ = 0.2
T3	400	**97.5%**	96.7%	**95.0%**	94.9%
T5		**84.6%**	78.6%	**98.0%**	95.7%
T7	2500	78.0%	**81.8%**	98.0%	**99.0%**
T9		**85.3%**	73.2%	**98.7%**	97.3%

Table 8 shows the results of varying the population size for the DEFS algorithm when CR = 0.3 and the mutation rate (scaling factor) φ = 0.5. The reported results are best when each of the values of CR is tuned over the set [0.3, 0.5, 0.7]. We see that a population size of 16 provides better results for smaller task sets; whereas a population size of 24 gives better results for larger task sets. Similar to GAFS, large population sizes allow for richer selection of candidate vectors, as well as for more variance in the crossover and mutation operations. Since larger task sizes entail larger dimensions, larger initial population is expected to achieve convergence compared to smaller task sizes.

Table 8. DEFS near-optimal power savings sensitivity to population size, CR = 0.3 and φ =0.5.

		Percentage of near-optimal power saving configurations			Percentage of configurations better than CS-DVS		
Task No.	HP	Population Size					
		8	16	24	8	16	24
T3	400	90.8%	**95.8%**	93.6%	91.0%	**94.6%**	94.2%
T5		71.2%	**80.4%**	67.1%	90.5%	**96.7%**	94.6%
T7	2500	34.3%	77.9%	**82.8%**	86.8%	97.4%	**98.8%**
T9		40.6%	81.4%	**86.5%**	88.1%	98%	**98.6%**

The sensitivity analysis findings of DEFS crossover probability (CR) and mutation (scaling) factor φ parameters is summarized in Table 9. For larger task sizes of 7 and 9, we find that crossover probability and mutation factor carry no statistical differences in yielding better results across different combinations of CR and φ. However, for smaller task sizes, a CR of 0.3 and φ = 0.5 provide better results by a wide margin (i.e., up to 14% better results than those at CR = 0.7 and $\varphi = 0.7$ for a system with five tasks).

Table 9. DEFS sensitivity to crossover probability (CR) and mutation factor φ at best results of fixed population size and hyper-period.

Population size		16		24	
Hyper-period		400		2500	
CR	φ	T3	T5	T7	T8
0.3	0.3	94.4%	76.1%	77.1%	81.8%
	0.5	**95.8%**	**80.4%**	82.8%	86.5%
	0.7	95.3%	78.2%	83.3%	85.6%
0.5	0.3	92.9%	75.2%	80.7%	84.9%
	0.5	94.3%	78.2%	84%	87%
	0.7	93.7%	76.3%	83.7%	87.0%
0.7	0.3	87.9%	68.0%	80.1%	83.8%
	0.5	89.7%	69.2%	82.5%	84.8%
	0.7	87.6%	66.4%	82.6%	83.9%

Table 10. Simulated Annealing (SA) near-optimal power savings sensitivity to hyper-period.

Tasks	Hyper-period					
	50	100	200	400	1000	2500
T3	36.8%	43.5%	51.5%	**58.8%**	-	-
T5	21.5%	23.3%	26.5%	**32.3%**	-	-
T7	13.9%	14.7%	15.6%	17.3%	19.6%	**31.5%**
T9	20.6%	23.0%	25.3%	**27.9%**	17.3%	25.6%

Table 11. Simulated Annealing (SA) percentage of configurations better than CS-DVS.

Tasks	Hyper-period					
	50	100	200	400	1000	2500
T3	63.3%	70.4%	74.9%	**75.7%**	-	-
T5	54.7%	60.8%	65.9%	**72.8%**	-	-
T7	48.1%	54.4%	59.8%	66.5%	70.3%	**78.6%**
T9	40.5%	44.9%	48.1%	52.0%	63.4%	**72.6%**

The reference values for the simulated annealing (SA) algorithm implementation are summarized in Table 10 and Table 11. One major observation is that the results of the SA algorithm do not provide substantial gains as the number of hyper-periods is increased when it comes to near-optimal results. This is more obvious at the larger task set size of T9. In fact, lower number of hyper-periods could provide better results. This is due to the algorithm design as implemented by [10], where even though the algorithm can escape a local minimum, there is no guarantee that it will converge to a better solution and the best-yet found values are not preserved.

5.2 Algorithm Comparison and Discussion

Figure 2 provides a performance summary of the proposed and reference algorithms. We observe that the proposed algorithms outperform the SA algorithm in terms of their ability to consistently provide near-optimal power savings. As the task set size increases, the quality performance of SA decreases, while the proposed algorithms consistently maintain their quality performance. Figure 3 shows that the SA algorithm fails 25% of the time to yield quality configurations better than CS-DVS. In effect, this could lead to high system-wide energy consumption. Both GAFS and DEFS are superior to SA, as they almost always deliver better power configurations than CS-DVS.

GAFS slightly outperforms DEFS for smaller task sets and the converse is true for larger task sets. The weakness point of GAFS is that a new generation of test quality configurations can only be generated when the current population has been fully examined. DEFS does not suffer from this issue, as candidate test configurations are generated randomly from a list of the so-far best found quality configurations which are readily available. The SA algorithm suffers from the possibility of replacing an elite solution by a non-optimal one. This is due to the inherent design of the algorithm, where it stochastically accepts worse solutions as means to escape a local minimum.

Finally, even though GAFS maintains an elite solution through the generations (which only gets updated if better solution is found), GAFS suffers from the possibility of producing a whole generation of non-feasible solutions aside from the elite. DEFS, on the other hand, does not suffer from these issues as it maintains a population of best found feasible solutions at any time.

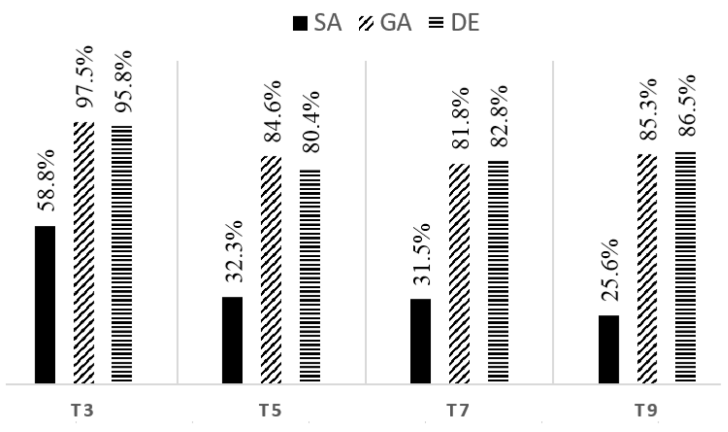

Figure 2. Percentage of near-optimal results of the three meta-heuristics over 500 unique sets.

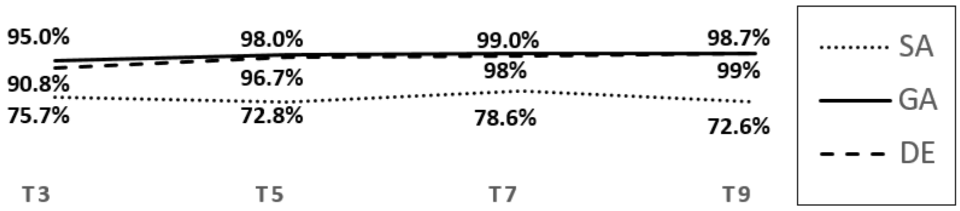

Figure 3. Percentage of the three meta-heuristics that are better than the CS-DVS heuristic over 500 unique sets.

Furthermore, since each configuration is randomly produced from this set based on mutation probabilities, even if a pass generates a set of unfeasible power configurations, new feasible power configurations can still be produced in subsequent hyper-periods.

6. Conclusions

System wide energy minimization is of paramount importance in modern embedded system design. We specifically adapted the use of genetic (GAFS) and evolutionary (DEFS) algorithms with the goal of reducing the overall energy consumption. We have investigated the performance of our developed meta-heuristic algorithms that assign frequency scaling (DVFS) to tasks in a hard-real-time system. We measure energy consumption at the system level; that is that of the processor and the devices. We have conducted a sensitivity analysis over a wide range of initial values of the proposed algorithms. We have found that setting the algorithm search space to 12% of the available search space for small task sizes (i.e., T5) yields a majority of near-optimal results. A much smaller search space of < 3% of the total exploration space works well for larger task sizes of 7 and 9. For the genetic algorithm, an initial chromosome set of 32 performs better than all other initial set sizes of 8, 16 or 24. There are marginal result differences in changing the mutation factor μ from 0.1 to 0.2. In DEFS, the size of the initial vector set affects smaller task sets differently than larger task ones. An initial size of 24 vectors provides a majority of near-optimal results for task set size of 7 and 9. The same results are obtained for smaller task sets of 3 and 5 with a smaller initial vector set size of 16. Finally, a crossover probability of 0.3 and mutation (scaling) factor $\varphi = 0.5$ provide the best overall results regardless of the task set size.

The proposed algorithms outperformed the simulating annealing (SA) algorithm by an approximate factor of 2.75 to 1 for finding a near-optimal configuration when the system task set is comprised of 5 to 9 tasks. Furthermore, based on 500 unique sets of tasks, our proposed algorithms deliver near-optimal results in over 95% of the cases compared to the CS-DVS algorithm. Simulated annealing is better than CS-DVS by an average of 75% of the time. The proposed techniques we put forth have allowed for additional energy optimizations, which is favourable for the quest in the design of low power embedded systems.

References

[1] P. Pillai and K. G. Shin, "Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems," in Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP ’01), New York, NY, USA , pp. 89-102, 2001.

[2] S. Saewong and R. Rajkumar, "Practical Voltage-scaling for Fixed-priority RT-Systems," in the 9th IEEE Proc. on Real-Time and Embedded Technology and Applications, pp. 106–114, 2003.

[3] L. Benini et al. "A Survey of Design Techniques for System-level Dynamic Power Management," Proc. in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 8, no. 3: pp. 299–316, 2000.

[4] H. Cheng and S. Goddard, "Online Energy-aware I/O Device Scheduling for Hard Real-time

Systems," in Proc. of Design, Automation and Test in Europe (DATE '06), volume 1, pp. 1055-1060, Munich, 6-10 March 2006.

[5] V. Swaminathan and K. Chakrabarty, "Pruning-based Energy-optimal Device Scheduling for Hard Real-time Systems," in Proc. of the 10th International Symp. on Hardware/Software Co-design (CODES 2002), pp. 175–180, 2002.

[6] V. Devadas and H. Aydin, "On the Interplay of Voltage/frequency Scaling and Device Power Management for Frame-based Real-time Embedded Applications," Proc. of the IEEE Transaction on Comput., vol. 61, no. 1, pp. 31–44, 2012.

[7] R. Jejurikar and R. Gupta, "Dynamic Voltage Scaling for System Wide Energy Minimization in Real-time Embedded Systems," Proc. of the 2004 International Symp. on Low Power Electronics and Design (ISLPED '04), pp. 78–81, 2004.

[8] W. Wang et al. "System-wide Energy Optimization with DVS and DCR," Proc. of Dynamic Reconfiguration in Real-Time Systems, no. 4, pp. 129–163, Springer, New York, 2013.

[9] F. Kong et al. "Minimizing Multi-resource Energy for Real-time Systems with Discrete Operation Modes," in Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS '10), Washington, DC, USA, pp. 113–122, 2010

[10] D. He and W. Mueller, "Online Energy-efficient Hard Real-time Scheduling for Component Oriented Systems," Proc.of the IEEE 15th International Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 56–63, 2012.

[11] V. Devadas and H. Aydin, "DFR-EDF: A Unified Energy Management Framework for Real-time Systems," Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), pp. 121– 130, 2010.

[12] L. Niu, "System-level Energy-efficient Scheduling for Hard Real-time Embedded Systems," in Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1–4, 2011.

[13] B. A. Mahafzah and B. A. Jaradat, "The Hybrid Dynamic Parallel Scheduling Algorithm for Load Balancing on Chained-cubic Tree Interconnection Networks," in Journal of Supercomputing, vol. 52, no. 3, pp.224– 252, 2010.

[14] J. Zhao and H. Qiu, "Genetic Algorithm and Ant Colony Algorithm Based Energy-efficient Task Scheduling," in International Conference on Inform. Science and Technology (ICIST), pp. 946–950, 2013.

[15] S. G. Ahmad et al. "PEGA: A Performance Effective Genetic Algorithm for Task Scheduling in Heterogeneous Systems," in IEEE 14th Intl. Conference on High Performance Computing and Commun. IEEE 9th Intl. Conference on Embedded Software and Systems (HPCC-ICESS), pp. 1082–1087, 2012.

[16] M. A Alshraideh et al. "Using Genetic Algorithm as Test Data Generator for Stored pl/sql Program Units," in Journal of Software Eng. and Applicat, vol. 6, no. 2, p. 65, 2013.

[17] D. Simon, Evolutionary Optimization Algorithms, Wiley, 2013.

[18] G. Chen et al. "Effective Online Power Management with Adaptive Interplay of DVS and DPM for Embedded Real-time System," in Euromicro Conference on Digital System Design (DSD), pp. 881–889, 2013.

[19] J. Xu, "A Method for Adjusting the Periods of Periodic Processes to Reduce the Least Common Multiple of the Period Lengths in Real-time Embedded Systems," in IEEE/ASME Intl. Conference on Mechatronics and Embedded Syst. and Applicat. (MESA), pp. 288–294, 2010.

[20] R. L. Haupt and S. E. Haupt, Practical Genetic Algorithms, Wiley InterScience Electronic Collection, Wiley, 2004.

[21] R. Jejurikar and R. Gupta, "Optimized Slowdown in Real-time Task Systems," Proc. in IEEE Computer Trans., vol. 55, no. 12, pp. 1588–1598, 2006.

[22] B. A. Mahafzah and B. A. Jaradat, "The Load Balancing Problem in OTIS-Hypercube Interconnection Networks," in Journal of Supercomputing, vol. 46, no. 3, pp. 276-297, 2008.

[23] V. Devadas and H. Aydin, "On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-Based Real-Time Embedded Applications," Proc. in IEEE Transactions on Computers, vol. 61, no. 1, pp. 31–44, 2012.

ملخص البحث:

تصاعدت أهمية أنظمة الزمن الحقيقي الفعّالة من حيث الطاقة كثيراً في السنوات القليلة الماضية. فقد تم تطوير تقنياتٍ على جميع مستويات تصميم الأنظمة للتقليل من استهلاك الطاقة. فعلى المستوى المادّي، تحاول تقنيات التصنيع التقليل من الطاقة التي تستهلكها مجموعة الرقاقات على وجه الإجمال. وعلى مستوى تصميم النظام، تسمح تقنيات مثل التدريج الديناميكي للفولتية والتردد (DVFS) الإدارة الديناميكية للطاقة (DPM) بتغيير تردد المعالج على نحوٍ فوريّ أو الانتقال إلى أنماط "النّوم" لتقليل القدرة التشغيلية إلى أدنى حدّ ممكن. وعلى مستوى أنظمة التشغيل، تستفيد الجدولة الفعالة من حيث الطاقة من خلال البحوث والدراسات التي أُجريت في هذا المجال من التدريج الديناميكي للفولتية والتردد، ومن الإدارة الديناميكية للطاقة على مستوى المهامّ لتحقيق وُفورات إضافية في الطاقة. وقد تركز معظم جهود البحث في مجال الجدولة الفعّالة من حيث الطاقة على تقليل قدرة المعالج. وحديثاً، تم استقصاء حلولٍ على مستوى النظام بمفهومه الواسع. في هذه الورقة البحثية، نتوسع في أعمالٍ سابقة عبر تكييف خوارزميتين تطوُّريتين لتقليل طاقة النظام ككل إلى أدنى حدّ ممكن، ونحلّل أداء خوارزمياتنا تحت ظروف ابتدائية متغيرة. إضافة إلى ذلك، نبيّن أن تقنياتنا ما وراء الموجّهة تعطي وُفورات في الطاقة أقرب إلى الوضع المثالي لما نسبته 85% من الوقت مقارنة بنسبة 30% تقريباً، التي تم تحقيقها باستخدام التقوية بالمحاكاة لما يزيد على 500 مجموعة فريدة من الفحوصات.

كذلك، تُبين نتائجنا أنّهُ في ما يزيد على 95% من الحالات، تُعطي التقنيات ما وراء الموجّهة وُفورات أكثر في الطاقة المستهلكة مقارنةً بالطريقة الستاتيكية (CS-DVS).