Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

A Hybrid DVS Scheduling Approach for Hard Real-Time Systems

Eduardo Tavares. Pedro Dallegrave, Bruno Silva. Gustavo Callou. Bruno Nogueira. Paulo Maciel
Center for Informatics
Federal University of Pernambuco
{eagt,pdf2,bs,grac,bcsn,prmm}@cin.ufpe.br

***Abstract*—Dynamic Voltage Scaling (DVS) has been largely adopted as an effective technology for reducing energy consumption in embedded systems. Since the usage of DVS may affect the timing constraints of a hard real-time system, over the last decade, several pre-runtime as well as runtime scheduling approaches have been developed to tackle such an issue. Nevertheless, both have drawbacks that can be mitigated using a joint approach. This paper proposes a hybrid DVS scheduling approach for energy-constrained hard real-time systems, taking into account overheads, precedence and exclusion relations. The proposed method adopts a formal model based on time Petri nets in order to provide feasible schedules that satisfy timing and energy constraints.**

***Index Terms*—Hard Real-Time Systems; Dynamic Voltage Scaling; Petri Nets; Formal Models**

I. Introduction

Energy consumption in embedded systems has received considerable attention over the last years due to the great proliferation of portable devices. In this context, DVS (Dynamic Voltage Scaling) has been one of the most studied technologies for providing energy saving in these systems. Adjusting CPU supply voltage has great impact on energy consumption, since the consumption is proportional to the square of supply voltage in CMOS microprocessors [10]. However, lowering the supply voltage linearly affects the maximum operating frequency.

Regarding hard real-time systems, several pre-runtime and runtime scheduling approaches have been developed to simultaneously cope with DVS and stringent timing constraints. However, those approaches have advantages as well as limitations. In the context of runtime scheduling, the advantages include flexibility and adaptability to changes in the environment. Nevertheless, runtime methods may provide infeasible results when considering arbitrary intertask relations [20] and may generate a significant overhead (time and energy) during system execution, for instance, due to calculations for managing task executions.

On the other hand, for a given specification, pre-runtime approaches can find a feasible schedule, if one exists, satisfying the specified constraints (e.g., intertask relations). Moreover, since schedules are generated at design-time, pre-runtime approaches also provide predictable executions with lower overheads than runtime counterparts. Despite the previous advantages, these methods are inflexible, in the sense that they cannot benefit from slack times that may occur due to earlier completion of tasks (at system runtime). The reader may refer to [20] for a comprehensive comparison between runtime as well as pre-runtime scheduling approaches.

To take advantage of both methods, this paper proposes a hybrid scheduling for hard real-time systems, considering DVS, intertask relations and overheads (such as preemption and voltage/frequency switching). As a broader view, the method begins applying the pre-runtime scheduling to generate a feasible schedule satisfying the specified constraints. At system runtime, a lightweight scheduler checks for earlier tasks' completions. If a task finishes its execution in lesser time than its respective worst-case execution time, and the constraints allow (e.g., release time), the scheduler adjusts the voltage/frequency level for executing the next task, such that energy consumption is improved. Additionally, the proposed method adopts time Petri nets for pre-runtime schedule generation as well as for qualitative analysis and verification.

This work is an extension of the software synthesis method described in [17], which aims to generate predictable code for hard real-time systems with energy constraints. New results regarding the pre-runtime scheduling algorithm are presented as well as results concerning the runtime scheduler.

II. Related Works

Many scheduling methods, for instance, [1], [4], [8], [15], [9], have been developed to cope with voltage scaling in time-critical systems. Works, such as [1], [15], are based on runtime scheduling policies, which can greatly improve energy consumption as shown by their experimental results. A subset of those runtime techniques applies a preprocessing for defining an initial voltage for each task before runtime. Indeed, this can be viewed as a hybrid approach, which mixes runtime and pre-runtime methods. However, some of these works do not properly tackle overheads related to voltage/frequency switching, preemption, and runtime calculations, as well as neglect precedence and exclusion relations.

Few works, for instance [2], [3], [5], have been developed to deal with intertask relations. Nevertheless, those scheduling methods either do not consider both precedence and exclusion relations or do not properly tackle runtime overheads.

As an alternative, this paper proposes a hybrid DVS scheduling method for hard real-time systems, taking into account runtime overheads (preemptions and voltage/frequency switching), intertask relations (precedence and exclusion), and, also, utilizes a formal model based on time Petri nets. The pre-runtime scheduling algorithm is a depth-first search method,

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

which seeks for a feasible schedule that satisfies stringent timing constraints and does not surpass an upper bound in terms of energy consumption. Different from other approaches (e.g., [21]), the algorithm does not necessarily generate an optimal solution due to the size of the state space (see Section V), but it looks for a solution that meets all non-functional requirements provided in the system specification. Besides, the adopted non-functional requirements allow the practical utilization of feasible schedules in the implementation of real systems. To take advantage of slack times that may occur at system runtime, a lightweight runtime scheduler is adopted to improve energy consumption without violating the constraints previously met by the pre-runtime schedule.

III. Preliminaries

This section presents fundamental concepts for a better understanding of the proposed method.

Specification Model. The specification model is composed of: (i) a set of periodic tasks with bounded discrete time constraints; (ii) intertask relations, such as precedence and exclusion relations; (iii) a discrete set of supply voltages and their respective CPU's maximum frequencies; and (iv) the system energy constraint.

Let $\mathcal{T}$ be the set of tasks in a system, A periodic task is defined by $\tau = (ph, r, c, d, p)$, where ph is the initial phase; r is its release time; c is the worst-case execution cycles (WCEC) required for the execution of task τ; d is its deadline; and p is its period. In this work, sporadic tasks are also considered by translating them into equivalent periodic tasks [20].

Tasks may have precedence and exclusion relations between them. A task τ_i *precedes* task τ_j, if τ_j can only start executing after τ_i has finished. A task τ_i *excludes* task τ_j, if no execution of τ_j can start while task τ_i is executing.

Let call $\mathcal{V}$ and $\mathcal{F}$ be two sets of discrete CPU supply voltage levels and frequencies, respectively, in which $|\mathcal{V}|$ = $|\mathcal{F}|$. $vff : \mathcal{V} \rightarrow \mathcal{F}$ (voltage-frequency function) is a bijective function that maps each voltage level to one, and only one, processor execution frequency, which is the maximum operating frequency at that supply voltage. In this work, voltage/frequency levels that do not provide energy saving due to the leakage current are not considered in the scheduling process.

In addition, the system energy constraint (e_{max}) needs to be defined, which sets an upper bound in terms of energy consumption that a schedule must not surpass. In other words, e_{max} is adopted by the scheduling algorithm to search for a feasible schedule, in which the energy consumption of all tasks executions does not violate such upper bound. Moreover, since the designer previously knows the schedule period (Section IV) and, for instance, may have some insights about the battery limitation in the final system, he can enforce a desired maximum energy consumption related to the execution of all tasks in a feasible schedule.

Computational Model. Computational model syntax is given by a time Petri net [11] , and its semantics by a timed labeled transition system. A time Petri net (TPN) is a bipartite directed graph represented by a tuple $\mathcal{N}= (P, T, F, W, m_0, I)$, where P (set of places) and T (set of transitions) are non-empty disjoint sets of nodes. The edges are represented by F, where $F \subseteq A = (P \times T) \cup (T \times P)$. $W : A \rightarrow \mathbb{N}$ represents the weight of the edges, such that $W(f) = \{(i)\ x \in \mathbb{N},\ if\ (f \in F),\ or\ (ii)\ 0,\ if\ (f \notin F)\}$. A TPN marking m_i is a vector ($m_i \in \mathbb{N}^{|P|}$), and m_0 is the initial marking. $I : T \rightarrow \mathbb{N} \times \mathbb{N}$ represents the timing constraints, where $I(t) = (EFT(t), LFT(t))\ \forall t \in T,\ EFT(t) \leq LFT(t)$. $EFT(t)$ is the Earliest Firing Time, and $LFT(t)$ is the Latest Firing Time. Due the lack of space, the reader is referred to [16], [17] for further information.

IV. Modeling Real-Time Systems

The proposed modeling method adopts a bottom-up approach, in which a set of composition rules are considered for combining basic building block models. The set of basic models have been conceived for automatic pre-runtime schedule generation, where the schedule period (P_S) corresponds to the least common multiple (LCM) of all tasks' periods. Within this period, several task instances (of the same task) might be carried out, such that $\mathcal{N}(\tau_i)$ $=P_S/p_i$ gives the number of instances for each task τ_i. Once a feasible schedule is generated, the same schedule will be infinitely often executed during system execution,

In order to present each building block, consider the model depicted in Fig. 1, which represents the following specification : T_1 = (0, 0, 240×10^6, 20, 20) and T_2 = (0, 5, 60×10^6, 15, 20). For this specification, the preemptive scheduling method is assumed, and the following voltage/frequency levels are considered: vff = {(1V,10MHZ),(2V,20MHZ)}. Moreover, an unavailable voltage/frequency level of 1.5V/15MHZ is also taken into account. In this case, the unavailable voltage can be "simulated" using the 2 immediately neighboring CPU voltages [4]. The building blocks are explained as follows:

a) Fork Block. Supposing that the system has n tasks, the fork block is responsible for starting all tasks in the system. This block models the creation of n concurrent tasks.

b) Periodic Task Arrival Block. This block models the periodic invocation for all task instances in the schedule period (P_S). A transition t_{ph_i} models the initial phase of the task first instance. Similarly, transition t_{a_i} models the periodic arrival (after the initial phase) for the remaining instances and transition t_{r_i} represents a task instance release. The reader should note the weight ($\alpha_i = \mathcal{N}(\tau_i){-}1$) of the arc (t_{ph_i}, p_{wa_i}), in which this weight models the invocation of all remaining instances after the first task instance. The timing intervals of transitions t_{ph_i}, t_{a_i} and t_{r_i} are the timing constraints depicted in the specification, in this case, ph_i (phase), p_i (period), and r_i (release) of task τ_i.

c) Voltage Selection Block. For each available voltage, this block represents every possible voltage selection for executing a task instance.

d) Non-preemptive Task Structure Block. Considering a non-preemptive scheduling method, the processor is just

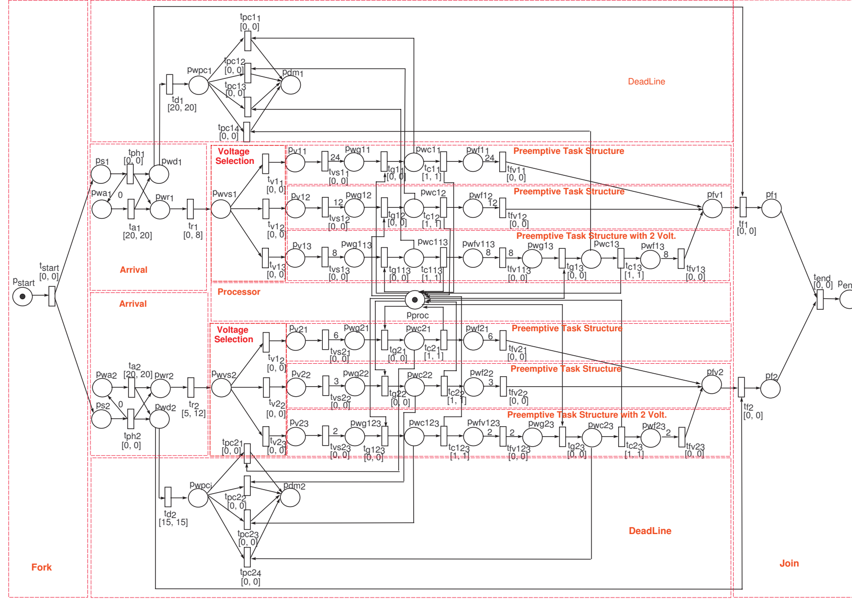

Fig. 1. Example Model

released after the entire computation is finished. This block (Fig. 2(a)) models a non-preemptive task computation adopting a specific voltage. Assuming a voltage $v \in \mathcal{V}$ and the respective maximum frequency $f = vff(v)$, task computation time (C) can be obtained by $C = \lceil c_i/f \rceil$, where c_i is the task (τ_i) WCEC. Fig. 2(a) shows that time interval of computation transition $t_{c_{i_n}}$ has bounds equal to the task computation time at a specific voltage ($[C.C]$).

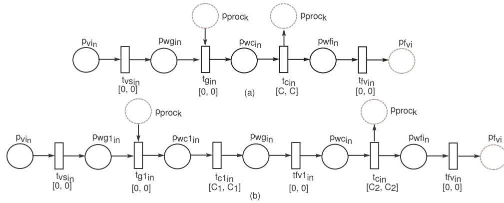

Fig. 2. Non-Preemptive Task Structure Blocks

e) Preemptive Task Structure Block. In this particular scheduling method, tasks are implicitly split into subtasks, in which the computation time of each subtask is exactly equal to one task time unit (TTU). This method allows running other conflicting tasks, in this case, meaning that one task may preempt another task,

f) Non-Preemptive Task Structure with 2 Voltages Block and **g) Preemptive Task Structure with 2 Voltages Block**. If the CPU provides a small number of discrete voltage levels and an ideal voltage is not available ($v_{ideal} \notin \mathcal{V}$), the two immediate neighbor voltages (v_{ideal_L},$v_{ideal_H} \in \mathcal{V}$) to the ideal one can be adopted for reducing energy consumption [4], For a better understanding, a task may be divided in two parts. The first part is executed at the immediate higher voltage in relation to the ideal one (v_{ideal_H}), and the second part is executed at the immediate lower voltage (v_{ideal_L}). These blocks model a task instance executing at two different voltages considering non-preemptive (Fig. 2(b)) and preemptive executions. C_1 represents the computation time of the first part of the task executing at v_{ideal_H}, and C_2 represents the computation time of the second part of the task executing at v_{ideal_L}. Without loss of generality, these blocks resemble the task structure blocks presented previously. For further information related to the time instants at which the voltage changes, the reader is referred to [4].

h) Deadline Checking Block. Deadline missing is an undesirable situation when considering hard real-time systems. Therefore, the scheduling algorithm should not reach deadline missing states, since those states do not allow finding out feasible schedules.

i) Join Block. Usually, concurrent activities need to synchronize with each other. The join block states that all tasks in the system have concluded their execution in the schedule period.

Due to lack of space, the reader is referred to [16] for information about the building blocks for modeling overheads (e.g., voltage/frequency switching and preemption) and inter-task relations (precedence and mutual exclusion).

V. Pre-runtime Scheduling

The proposed scheduling activity adopts a pre-runtime scheduling approach, in which a feasible schedule is generated at design-time. As stated in [20], a prime advantage of

pre-runtime scheduling methods over runtime approaches is predictability. This work conducts the pre-runtime scheduling through a depth-first search algorithm [16] that generates a subset of the state space related to the time Petri net. A partial state space is generated since the scheduling algorithm does not produce the subsequent states of a path whenever either timing or energy constraints are violated.

This section provides new results of the pre-runtime scheduling algorithm [16] regarding its complexity.

Complexity. In order to provide an estimation of the state space size generated when considering the proposed method, it is assumed n non-interacting tasks, each one with k local states. Hence, the respective state space size is $O(k^n)$ [18]. In general, the number of local states (k) of each task is somewhat affected by the following attributes: (i) the number of task instances; (ii) the number of available voltages for the task; (iii) the respective release interval; and (iv) when regarding preemptive tasks, the arc weights that represent the computation time at a specific voltage (which is affected by the adopted task time unit).

The complexity of the state space is not only related to the adopted formalism, but, primarily, due to the scheduling problem in question. For instance, other formal methods, such as process algebras and automata, face similar complexity to tackle this scheduling problem.

Regarding time complexity, the execution time of a depth-first search method is generally bounded by $O(e) = O(v^2)$, in which is e and v are the number of edges and nodes of a graph, respectively. Since the state space of Petri nets is usually represented by a reachability graph [12] and the estimated size of the proposed model's state space is $O(k^n)$, the execution time related to the pre-runtime scheduling algorithm is bounded by $O((k^n)^2) = O(k^{2n})$.

Section VII provides the execution time as well as the number of states actually reached by the adopted scheduling algorithm.

VI. Handling Dynamic Slack Times

During system runtime, slack times (CPU idle times) may appear due to tasks' early completion. In order to take advantage of such slacks for reducing even more energy consumption, a lightweight runtime scheduler is proposed for adjusting the starting times as well as the voltage/frequency levels associated to each task instance.

Before presenting the runtime scheduler algorithm, some concepts are required firstly. The pre-runtime schedule is partitioned in several *time slices* of the same size, in which each slice corresponds to one task time unit, and the total amount is equal to the LCM. These slices can be grouped into segments in such a way that represent task executions. Such segments are denominated *task segments*, and each one is represented by an interval ([start,end]). When a task is not completely executed within a segment, the task is preempted, in other words, it is carried out through more segments. Moreover, a global clock (`clock`) is adopted for tracking the current time (e.g., the accumulated number of time slices). Taking into account the

```
1  scheduler() {
2    nextSegment = retriveNextSegment();
3    taskInstance = nextSegment.taskInstance;
4
5    if(nextSegment not exists
        || taskInstance.release == nextSegment.start)
6       {return; //keep the pre-runtime schedule}
7
8    if(taskInstance.release > clock)
9       {nextSegment.start = taskInstance.release}
10   else {nextSegment.start = clock + schedulerWCET }
11
12   if (energy saving compensates the overhead)
13      {adjust voltage according to Ishihara's theorem [4]
         and the new execution window (nextSegment.end -
         nextSegment.start);
14       Prepare dispatcher to execute at
         nextSegment.start;}
15   else {return; //keep the pre-runtime schedule}
16 }
```

Fig. 3. Runtime Scheduler Algorithm

previous concepts, the runtime schedule algorithm is depicted in Fig. 3 using a C syntax notation,

In order to check the early completion of a task instance, the runtime scheduler is executed at the end of each segment, in such a way that its execution does not conflict with the dispatcher execution. Firstly, the scheduler verifies which is the next segment (line 2) in the pre-runtime schedule, since it is the candidate for adjusting the respective voltage/frequency level as well as the start time. If there is no segment to be executed - the remaining segments are returns from preemption of finished instances or the last segment was already executed - the original pre-runtime schedule is kept (line 6). Also, the pre-runtime schedule is not changed whether the start time of the next segment is equal to the release time assigned to the respective task instance. Considering that there is an available segment, the respective start time is set so that the release time is not violated (line 8 and 10). If the next segment can be promptly started, the start time is tuned to take into account the scheduler WCET (worst-case execution time). It is worth noting that the adjustment is only performed whenever the improvements compensate the scheduling overhead (line 12).

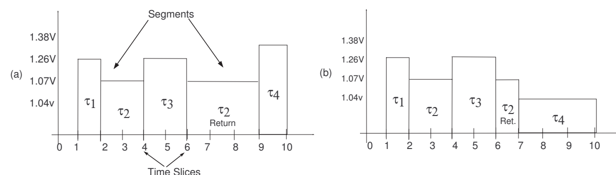

Fig. 4. Runtime Scheduler Example

For a better understanding, consider the schedule depicted in Fig.4(a), which is composed of the following segments: (i) $\tau_1 = [1, 2]$; (ii) $\tau_2^1 = [2, 4]$; (iii) $\tau_3 = [4, 6]$; (iv) $\tau_2^2 = [6, 9]$; and (v) $\tau_4 = [9, 10]$. A DVS platform based on [13] is adopted, considering the following voltage/frequency levels: vff = {(1.04V,20MHz), (1.07V,30MHz), (1.26V,50MHz), (1.38V,60MHz)}. Moreover, assume that the energy consumption is 0.54nJ/cycle at 60MHz, 0.45nJ/cycle at 50MHz, 0.34nJ/cycle at 30Mhz, and 0.31nJ/cycle at 20MHz. In this example, if task τ_2 completes its execution earlier at 7, the proposed scheduler attempts to adjust the voltage/frequency level as well as the start time of the next segment (τ_4).

Considering that τ_4 release time is equal to 6, τ_4 can start its execution earlier and, also, it can utilize a lower voltage/frequency level (Fig.4(b)). Assuming that WCEC of each task are $c_1 = 50 \times 10^6$, $c_2 = 150 \times 10^6$, $c_3 = 100 \times 10^6$, $c_4 = 60 \times 10^6$, the energy consumption is reduced from 0.1305J (early completion of task τ_2) to 0.1167J (Fig.4(b)).

VII. Experimental results

This work has conducted some experiments to evaluate the proposed pre-runtime and runtime scheduling algorithms. Firstly, experiments related to the pre-runtime scheduling are presented, and, next, results concerning the runtime scheduler are described.

Pre-Runtime Scheduling. Table I summarizes the experiments adopted to evaluate the pre-runtime scheduling algorithm. In this table, real-world applications as well as custom-built examples (that simulates real-world situations) are taken into account. The column *task* represents the number of tasks; *inst*. represents the number of tasks' instances; *sch*. is the number of states of the feasible schedule; *found* counts the number of states actually verified for finding a feasible schedule; *w/DVS* is the energy consumed (in joules) by the found feasible schedule using DVS; o/DVS is the energy consumed in joules by an alternative schedule that disregards DVS; *lpedf* is the energy consumed (in joules) by a schedule generated using the optimal scheduling mechanism proposed by Yao et al., considering discrete voltage/frequency levels [9]; and *time* expresses the algorithm execution time (in seconds) for finding the feasible schedule. All experiments were performed on a Pentium D 3GHz, 4Gb RAM. Linux, and compiler GCC 3.3.2. For a better comprehension, the following paragraph give an overview of each case study,

TABLE I
EXPERIMENTAL RESULTS SUMMARY

Case Study	task	inst.	sch.	found	w/DVS	o/DVS	lpedf	time(s)
1.Motiv. Ex.	4	4	48	141	0.2474	0.3132	0.17090.	0.001
2.Example 2	6	6	4377	518406	0.00069	0.00105	0.00048	35.200
3.Example 3	12	12	551	9906	267.84000	360.00000	254.11840	0.282
4.Kwon's Ex.	4	4	246	246	279.00000	371.00000	279.0000	0.003
5.CNC Control	8	289	235852	1884381	0.11900	0.34500	0.09440	291.221
6.Pulse Oximeter	3	10	83	4268	0.00021	0.00023	0.00014	0.234
7.MP3 & GSM	8	3604	381313	381313	3.86200	4.76600	3.85410	9.606

Experiments 1 and 2 are based on example 2 of [20], which demonstrates a situation where pre-runtime approaches can find feasible schedules. For those experiments, runtime methods may not provide feasible solutions due to exclusion relation between tasks. In the same way, experiment 3 is based on figure 2 of [19], which shows another situation where runtime approaches may not deal properly with. Experiment 4 is depicted on Table 1 in [9] and does not consider any intertask relation. In this case, the proposed approach finds a feasible schedule that consumes the same amount of energy as [9], which is the Yao's algorithm extended with discrete set of voltages. Experiments 5 [7], 6 [6], and 7 [14] are real-world applications, which have been utilized with the purpose of assisting the evaluation of the proposed approach. In their respective specifications, exclusion relations are considered in experiment 5, non-preemptable tasks and precedence relations are take into account in experiment 6, precedence relations are regarded in case study 7.

The adopted case studies demonstrate that the pre-runtime scheduling algorithm has provided meaningful results (Table I), since it has significantly reduced the number of visited states, found feasible schedules in which runtime counterparts may not, and, also, allowed energy saving by the adoption of DVS.

For a better visualization, Fig.5(a) depicts the estimated lower and upper bounds for each case study, regarding the state space size. Due to the large difference of sizes, this figure depicts all values as a power of ten (i.e., the exponents are presented). As the reader should note, the number of states visited by the scheduling algorithm is several orders of magnitude smaller than the whole state space in most experiments. Besides, the algorithm tends to visit a minimal number of states, mainly, due to the techniques adopted to reduce the state space (e.g., preprocessing). Taking account execution time, assume that $50\mu s$ is the average time to reach a single state, Such time has been obtained considering the execution time of each experiment as well as the number of visited states. In this context. Fig. 5(b) depicts the estimated lower and upper bounds in seconds for the algorithm execution in each case study. At first sight, the reader should observe that Fig. 5(b) resembles Fig. 5(a). Indeed, the execution time is associated with the number of visit states, and the time is also benefited by the techniques adopted to control the state space.

Regarding energy consumption, Fig. 5(c) depicts a comparison between feasible schedules generated by the proposed approach (with DVS), alternative schedules without DVS, and optimal solutions provided by Yao's Algorithm (LPEDF). In this figure, the energy consumption is normalized considering the highest value in each case study. Analyzing the results, the proposed approach generated feasible schedules that consume only 24% more energy (in average) than Yao's optimal solution and, in some experiments, provided the same consumption. Besides, it is important to emphasize that Yao's method does not consider precedence and exclusion relations. In other words, the values provided in column *lpedf* (Table I) assume a set of independent tasks, thus, the respective schedules are not feasible for most case studies. Nevertheless, those values still serve as an interesting parameter for comparison purposes,

Runtime Scheduler. An experiment based on case study 2 has been adopted to highlight some features related to the proposed runtime scheduler. In this experiment, 7 concurrent tasks with precedence and exclusion relations were analyzed executing on a real hardware platform based on [13]. In order to evaluate the runtime scheduler, 4 tasks varied their execution cycles at the same time from 100% to 10% in relation to each task WCEC. Fig. 5(d) depicts quantitative data, in which the energy consumption values are normalized,

Comparing the runtime scheduler (*scheduler*) with an approach based only on the dispatcher (*only dispatcher*), the results show a significant reduction in the energy consumption with the former (more than 40% in one scenario). The dispatcher only follows the schedule table and performs no

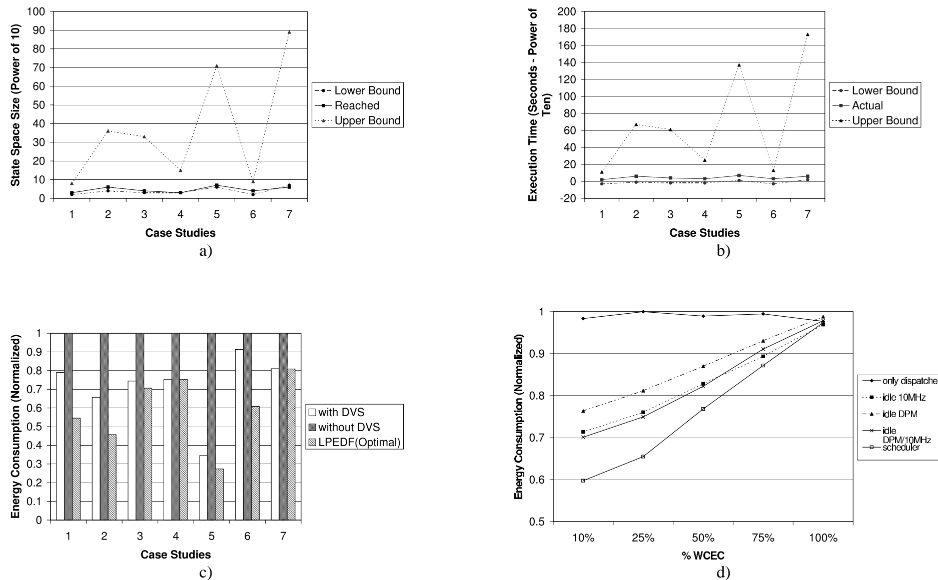

Fig. 5. Results Regarding the Pre-runtime and Runtime Scheduling Approaches

action to improve the consumption due to changes in the task executions. Additional savings can be obtained adjusting the dispatcher to reduce the voltage/frequency level to the minimum level during the idle periods (*idle 10MHz*). Nevertheless, the runtime scheduler still provides better results. Instead of switching to the minimum voltage/frequency level, consider the dispatcher turns off the microcontroller in the idle periods (*idle DPM*). Reducing to the minimum level seems more efficient, but the runtime scheduler provides greater savings. Besides, mixing DPM and the minimum voltage/frequency level (*idle DPM/10MHz*) slightly improves the consumption in some situations, regarding the idle periods. Nevertheless, the runtime scheduler can save 10% more energy in relation to such an approach,

VIII. Conclusion

This work presented a hybrid scheduling approach for energy-constrained hard real-time systems, considering DVS, overheads and intertask relations. The main idea is to mix the predictability of a pre-runtime scheduling method with the flexibility of a runtime counterpart, in such a way that energy consumption is minimized without affecting the specified constraints. Results demonstrated the feasibility of the proposed hybrid approach, in the sense that a significant amount of energy can be saved if tasks execute less than the respective WCET.

As future work, we are planning to extend the proposed scheduling mechanism in order to consider multiple processors.

References

[1] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. on Comp.*, 53(5):584–600, 2004.

[2] Y. Cai et al. Workload-ahead-driven online energy minimization techniques for battery-powered embedded systems with time-constraints. *ACM Trans. Des. Autom. Electron. Syst.*, 12(1):5, 2007.

[3] L. Cortés, P. Eles, and Z. Peng. Quasi-static assignment of voltages and optional cycles for maximizing rewards in real-time systems with energy constraints. In *DAC'05*, pages 889–894, 2005.

[4] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED'98*, pages 197–202, 1998.

[5] R. Jejurikar and R. Gupta. Energy-aware task scheduling with task synchronization for embedded real-time systems. *IEEE Trans. on Computer-Aided Des. of Integ. Circ. and Sys.*, 25:1024–1037,, 2006.

[6] M. O. Jr. *Desenvolvimento de Um Protótipo para a Medida Não Invasiva da Saturação Arterial de Oxigênio em Humanos - Oxímetro de Pulso (in portuguese)*. MSc Thesis, Departamento de Biofísica e Radiobiologia, Universidade Federal de Pernambuco, August 1998.

[7] N. Kim and et al. Visual assessment of a real-time systems design: A case study on a CNC controller. *RTSS'96*, pages 300–310, 1996.

[8] W. Kim and et al. Preemption-aware dynamic voltage scaling in hard real-time systems. *ISLPED'04*, pages 393–398, 2004.

[9] W. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM TECS*, 4(1):211–230, 2005.

[10] M. Liebelt et al. An energy efficient rate selection algorithm for voltage quantized dynamic voltage scaling. *ISSS '01*.

[11] P. Merlin and D. J. Faber. Recoverability of communication protocols: Implicatons of a theoretical study. *IEEE Trans. on Comm.*, 24(9):1036–1043, 1976.

[12] T. Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, April 1989.

[13] T. Phatrapornnant and M. Pont. Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling. *IEEE Trans. on Comp.*, 55(2):113–124, 2006.

[14] R. Prathipati. *Energy Efficient Scheduling Techniques for Real-Time Embedded Systems*. MSc Thesis, Texas A&M University, USA, 2004.

[15] G. Quan and X. Hu. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM TECS*, 6, 2007.

[16] E. Tavares and et al. Modeling hard real-time systems considering inter-task relations, dynamic voltage scaling and overheads. *Microprocessor and Microsystems*, 32(8), 2008.

[17] E. Tavares and et al. Software synthesis for hard real-time embedded systems with energy constraints. In *SBAC-PAD '08*, 2008.

[18] A. Valmari. The state explosion problem. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:429–528, June 1998.

[19] J. Xu. On inspection and verification of software with timing requirements. *IEEE Trans. on Soft. Eng.*, 29(8):705–720, 2003.

[20] J. Xu and D. Parnas. Priority scheduling versus pre-run-time scheduling. In *Real-Time Systems*, volume 18, pages 7–23. January 2000.

[21] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *36th Annual Symposium on Foundations of Computer Science*, 00:374, 1995.