Dynamic Runtime Re-Scheduling Allowing Multiple Implementations of a Task for Platform-based Designs

Tin-Man Lee
Princeton University

Jörg Henkel
NEC USA Inc.

Wayne Wolf
Princeton University

Abstract

This paper introduces an extension to the RMS scheduling technique that we call "Hot Swapping". Hot Swapping enables a system to choose between various selected implementations of one task on-the-fly and thus to optimize the system's cost (e.g. power savings). The on-the-fly swapping between those implementations requires extra time to save and/or transform states of a certain task implementation. Even if the two steady-state schedules before and after the swapping are feasible, the transient schedule with the additional swapping computation time may exceed the system's capacity. Our technique is an extension to Rate Monotonic Scheduling (RMS). While maintaining and meeting performance requirements, our technique shows an average reduction of 31% in power consumption compared to systems using a pure static scheduling approach (RMS) that cannot make use of task swapping. We have evaluated our algorithm through simulation of five real-world task sets and in addition by use of a large number of generated task sets.

1. Introduction

Most design methodologies for embedded systems require that a design is implemented with the highest workload in mind (e.g. highest possible performance the system should be capable of running at). Often, the highest workload is only required during a fraction of the whole run-time of a system. Consequently, many embedded systems tend to be *over-designed* i.e. throughout the largest time during the execution of an embedded system the system's capabilities are *not fully utilized*. An example is hardware/software partitioning of an embedded system: typically, the partitioning between hardware and software is determined upfront i.e. before implementation and after an analysis of requirements is conducted. Then, for the lifetime of the system the partitioning is fixed i.e. the mapping of tasks to hardware or software is inflexible and may lead to the above-mentioned over-design. The disadvantages can be manifold: for example, the power consumption might be very high when the system runs at higher as required performance. Many other disadvantages are associated with fixed partitions.

This paper introduces Hot Swapping, a technique for run-time re-scheduling of tasks. Hot Swapping assumes that potentially any task in a real-time system may have multiple implementations and that the system may switch between implementations on-the-fly. In addition, we take into consideration that the computational cost of switching between various implementations is greater than zero---additional computations may be necessary to complete the handoff from one implementation to another. Through a combination of design-time and run-time analysis, Hot Swapping determines whether a system's schedule will always be feasible for a) the initial implementation, b) during swapping, and c) after swapping is completed.

Good examples for deploying Hot Swapping are provided by mobile communication systems. A variety of error correction schemes can advantageously be used since they vary in power consumption (compare a 3-bit-detection-2-bit correction code against a 2-bit-detection-1-bit-correcton code, for example). Since the channel characteristics change spatially (e.g. mountainous and sparsely covered areas) and temporally (wireless backbone system running at overload, for example), a cell phone may want to use a less-costly scheme when the channel characteristics are good and switch to a more robust but also more power consuming scheme when the channel characteristics degrade. Note that switching between error correction schemes will require a computational overhead like the handshaking with the base station etc. In this example Hot Swapping would allow the system to ensure that the switch can be performed without violating any real-time deadlines in the system while saving power when operating the mobile phone.

The rest of the paper is structured as follows: Section 2 describes the related work and Section 3 discusses the model definitions and assumptions. In Section 4 our scheduling analysis for Hot Swapping is introduced whereas Section 5 shows the strategy for Hot Swapping. In Section 6 experimental results are conducted that show the advantages of Hot Swapping using real-world applications. Finally, Section 7 gives a conclusion.

2. Related Work And Focus of our Work

The Rate Monotonic Scheduling (RMS) theory was introduced by Liu and Layland[1] . The basic idea of RMS is to assign fixed priorities to tasks according to their periods: the smaller the period is, the higher is the assigned priority. At any time, the scheduler simply chooses to execute the task with the highest priority, with a lower priority task preempted if necessary. RMS can schedule a set of processes to meet their deadlines if the total utilization (workload) of all tasks is lower than the least upper bound. The RMS

schedule-ability analysis is based on the *critical instant theorem* that says that if a task meets its deadline, whenever the task is requested simultaneously with requests for all higher priority tasks, then the deadline will always be met for all task phasings.

Mode changes were first brought up in [3] [4] . The idea of mode changes is basically the deletion of an old task and the addition of a new task to fulfill functionality and performance requirements. The task set before and after a mode change is different, and each task has *one implementation only*. This is different from our approach. In [6] an offset value to a new mode task is introduced by stretching the period of the old mode task so that the new task can be released anytime after the period of the original old mode task.

We have considered the imprecise computations model as in [18] . However, that work targets on handling different task models with schedule-ability analysis and run time scheduling while our approach is to optimize the power consumption and thus imposing different constraints for scheduling.

A technique for reducing the time required to enforce schedule-ability is proposed by Shin and Choi [5] . In [7] the idea of reconfigure-ability at system-level by considering different task sets and using both hard and soft real-time tasks is proposed. They aim at reducing the execution time of a task set to improve schedule-ability. A dynamic '*shutdown'* and '*activate'* of a system is proposed in [8] [9] . They use techniques such as *frequency scaling* by making use of the slack time to facilitate schedule-ability.

When preemptively scheduling periodic tasks on a system of parallel processors Lawler and Martel[16] found that there exists a feasible schedule if and only if there exists a schedule that is cyclic with a period (also referred to as the hyper-period) equal to the LCM of the periods of all tasks

Our Hot Swapping (in the following we will mostly use the simple term *Swapping*) does not *predict* when it is a good time to consider the swapping Approaches addressing this problem already exist: for example, the system-level power management methodology of Bogliolo et al. [10] could be adapted to predict good times for swapping between implementations based on an observed behavior.

3. Model, Definitions and Assumptions

In the following we describe our model as well as various assumptions and definitions associated with it.

We assume that the timing parameters of a task τi are known a priori. Since our Hot Swapping uses RMS as a basis the hardware architecture has a single Processing element (e.g. CPU, ASIC, etc). We will use the term **phase** to describe the temporal distance between a time point and a deadline.

A task set **{τ1,… τN}** of N tasks runs on the CPU, with no data dependencies between the tasks. Each task has:

a) a **deadline D_i**
b) a **WCET** (worst case execution time**) C_i**
c) and a **period T_i**.

To refer to the deadline and period of a task we use the notion $\{D_i/T_i\}$ throughout the paper. The *initiation time* of a task is the time at which it actually starts executing.

Each task can have various implementations denoted as τi_j where 'i' refers to the task-id and j refers to the implementation. A task may or may not have more than one implementation.

The notion τi_a, τi_b etc. is used to indicate that task 'i' has the implementations 'a', 'b' etc. Let us define the **characteristics of various implementations**:

a) Different implementations of a task typically have different power consumption and/or performance.
b) Swapping from one implementation of a task to another implementation of the same task implies an *additional* computation time and power consumption.
c) The basic functionality of the various implementations of a certain task is the same. At least one implementation is a superset of the other implementations.
d) Swapping can be triggered by two requirements: **constraints** and **cost minimization.** In the example of a cell phone the channel characteristics might worsen due to environmental changes, hence requiring a higher correction capability. A task swap might be required. If more implementation meet the new constraints, the the goal is to minimize the cost.

3.1 Cost Definition with Example

Without loss of generality throughout this paper we define cost as power/energy consumption. Hence, the cost may by tendency inversely proportional to the idling time of the CPU. In other words, we want to find implementations of tasks that *do meet the constraints (functionality, timing)* but at the same time use the resource CPU as little as possible. As an example, assume there is a task set with one task having two implementations as shown in the following table:

Task	C [ms]	T [ms]	Power [mW]	Power to swap τ1_1 to τ1_2 [mW]
1_1	4	10	15	10
1_2	6	10	5	

Table 1: Example on power saving

Let the power consumed during the swap-over be 10mW, the swap time 2ms and the swap computation time 1ms. Thus, the total time needed to complete a swap is 2ms + 1ms. A swap request may arrive at t = 5 to swap from τ1_1 to τ1_2. Furthermore we know that the new implementation can meet the current performance requirement. We analyze the power consumption from t = 0 to t = 30 as shown below:

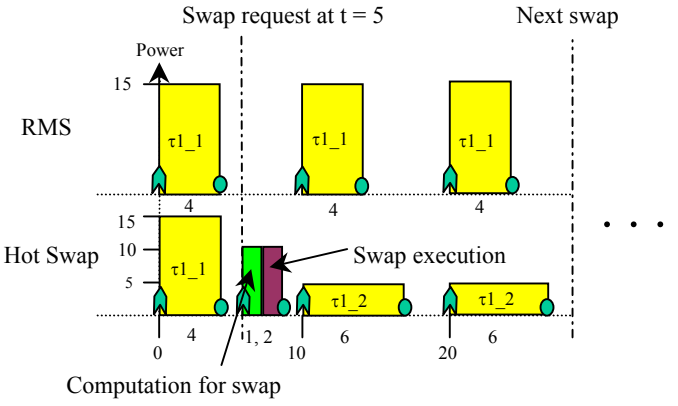

Figure 1: Power saving in a one-task set

In Figure 1, an arrow means the arrival of a request; a dot denotes the termination of an execution. The energy consumption is

$$E_{total} = N_{RMS} \text{ x } P_{RMS} \text{ x } t_{RMS} + N_{Swap} \text{x} P_{Swap} \text{ x } t_{Swap} + P_{overhead} \text{ x } t_{overhead}$$

where E_{total} represents the total energy, N_{RMS} is the number of executions carried out using RMS implementation, P_{RMS} is the power consumption of the task using an RMS implementation, t_{RMS} is the related execution time, N_{swap} is the number of executions carried out using Hot Swapping and P_{Swap} is the according power consumption and t_{Swap} is the execution time. Furthermore, $P_{overhead}$ is the power consumption to carry out the swap while $t_{overhead}$ gives the time needed.

As we can see in Figure 1 the RMS implementation (without Hot Swapping) has the following energy consumption:

$E_{total}(RMS) = N_{RMS} \text{ x } P_{RMS} \text{ x } t_{RMS} + 0 + 0$
$= 3 \text{ x } 15 \text{ x } 4 = 180uJ$

Using Hot Swapping we obtain:

$E_{total}(HotSwap) = N_{RMS} \text{ x } P_{RMS} \text{ x } t_{RMS} + N_{swap} \text{x} P_{Swap} \text{ x } t_{Swap} + P_{overhead} \text{ x } t_{overhead}$
$= 1 \text{ x } 15 \text{ x } 4 + 2 \text{ x } 5 \text{ x } 6 + 20 = 140uJ$

Obviously the yield is 22.2% of energy saving. If we can manage to keep the relationship between swap time and task execution times small and if there is time left to execute a swap when requested, we may be able to save a substantial amount of energy.

Throughout the paper we describe a system's task set by a list of timing and cost parameters as shown in the Table 2 for a specific example. '*Imp'* stands for implementation. '*C'* stands for worst-case execution time. *T* stands for period. *'Power'* gives the power consumption of each task.

4. Feasibility Analysis for Hot Swapping

The feasibility analysis in this section discusses whether the schedule-ability of the system is guaranteed when a swap request is triggered (either by changed constraints or due to the intend of cost minimization, as discussed earlier). We will distinguish the two cases "*Swap Request before Task Initiation"* and "*Swap Request after Task Initiation"*. Upon a swap request the criterion by Lehoczky et al. [2] must be checked:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \le N(2^{\frac{1}{N}} - 1) \qquad (1)$$

Ci and Ti are the worst case execution time and the period of task τi, respectively. All N tasks of the task set have to meet the above criterion. If a task has various implementations τi_j then only the task implementation we intend to swap *to* needs to meet this criterion. Note that it is not feasible to consider a swap request after all tasks have already received their *initiation request* at the *critical instant* (we use the same definition for *critical instant* as defined by Liu/Layland [1]). If, however, such a case occurs we have to wait until after the LCM of all tasks (i.e. least common multiple of the periods of all tasks) has arrived. Hence, it is practical to divide the following discussion into two cases.

4.1 Case 1: Swap Request Before Task Initiation

In this case, the task to be swapped has not started execution yet. We use the *critical instant theorem* and we check whether it is feasible to swap the task at **request time $t_{request}$**. This can be evaluated by iterative calculation as shown below:

$$t_{1b} = \sum_{\forall \tau_j \subset \{hp(i) \cup \tau_i\}} C \cdot \left\lceil \frac{t_{request} + 1}{T_i} \right\rceil$$

$$\vdots$$

$$t_{nb} = \sum_{\forall \tau_j \subset \{hp(i) \cup \tau_i\}} C \cdot \left\lceil \frac{t_{nb-1} + 1}{T_i} \right\rceil \qquad (2)$$

It is hp(i) the set of tasks that have a higher priority than the task τi which is requested to swap. The letter 'b' denotes that this is the feasibility analysis for the "*before case*". Finally, n gives the number of iterations. When tnb = tnb-1 holds, the process of iterative feasibility calculation is terminated. Then we have to deal with the following two cases:

A): $t_{request} < t_{nb}$:
Task τi has not yet started execution. Swapping is potentially possible if there is enough time left for conducting the swap (will be discussed later).

B): $t_{request} >= t_{nb}$:
Task τi has already started execution. This case tells us that the feasibility test from above is not sufficient.

4.2 Case 2: Swap Request After Task Initiation

In the previous section, "*Case 2*", we stated that Eq.2 cannot be applied when

$$t_{request} >= t_{nb}$$

holds. The only way to accomplish a swap is to wait until after all higher priority tasks have finished execution. The feasibility calculation is similar to Eq. 2 with τi included in

the calculation. Here is an example, assume the timing parameters are as follows,

5. Swapping Strategy

Hot Swapping Algorithm

At Design Time:

1. *Create database of timing parameters of each task with different implementation*
2. *Generate execution time, period of all task τi*

At Run Time:

```
1.  For each (swap_request received or generated) {
        method_1 := evaluate_equations_in_Eq(2)
        t_left(method_1) = tnb – t_request
        method_2 := evaluate_equations_in_Eq(3)
        t_left(method_2) =t(next_initiation_req) - tna
        do feasibility_ test (method_1)
            if (t_left(method_1) < 0) then {
                t_left = t_left(method_1)
                goto step 2}
            else do {
                feasibility_test (method_2)
                if (t_left(method_2) > 0) then {
                    t_left = t_left(method_2)
                    goto step 2}
                else
                    { // it is not feasible to swap
                      do ( ; new_request_recvd() ; )
                          { ; }
                      t_request=t_new_request
                      goto step 1
                    }
            }
    }
2.  if ( t_left >= t_swap_required() )
        do_swap()
    else {
        old_impl = curr_impl
        for all impl ∈ I {
            if (QoS(impl) ≥ QoS_req) {
                alt_impl = impl
            }
        }
        if (alt_impl ≠ old_impl) {
            t_swap_required() = t_swap(alt_impl)
            goto step 2
        }
        else {
            // it is not feasible to swap
            do ( ; new_request_recvd() ; )
                { ; }
            t_request=t_new_request
            goto step 1
        }
    }
```

The three main issues which have to be solved in our a real-time hot swapping scheduling algorithm are as follows:

a) *when* it is safe to swap
b) *when* to actually perform the swap (assumed that there has already been initiated a request to swap a.s.a.p.)
c) *how* to swap (e.g. using intermediate swaps etc.)

As stated earlier, the request for a swap can either be an environmental constraint to the system and/or cost minimization considerations. The latter is initiated by the system itself. Since swapping always incurs overheads such as swapping time (delay) and power consumption, and moreover, since a new implementation might have a longer execution time than the original implementation, our algorithm may not produce a solution with lower power consumption if the swapping is not feasible.

There are also some exceptions that we need to take into account. For example, the computation time to check the feasibility of swapping is 5ms, and the swap request arrives at a time where the deadline of a task is coming in 3ms. Since we are not able to know the swapping feasibility until we calculate it, then after the computation time of 5 units, that task may have missed its deadline. If this is the case, our algorithm may not be able to obtain a feasible solution. However, if we extend this issue into the future research of a heterogeneous processor system, this problem can be solved by deploying a specific processor that manages the calculation of the feasibility check. Since we are targeting on the alternative implementation of each task at a lower power consumption with a relatively lower performance and that all tasks would be running at its original implementation in the beginning, our algorithm will not swap a task to another implementation if the power consumed during the swap and the new power consumption is higher than the original setting. Upon the arrival of a swap request for τi_j -> τi_k, the algorithm first carries out a feasibility analysis to check whether τi_j has already started execution or not. It is safe to swap as long as τi_j has not yet started execution. If, however, τi_j it *has* started then it tries to determine when the task will have finished execution. Therefore, the second feasibility analysis is carried out. This could be a safe time for performing the swap because all other higher priority tasks have already finished execution at this time. In both cases t_left (the time remaining in the processor before the next request arrives) has to be calculated to compute whether there is sufficient time to actually perform the swapping. The swapping time is looked up in a database that has been created during design time. If there is not sufficient time, then the desired swap cannot be carried out. Then, the search for an alternative swap begins (else-case under 2.). Therefore, all possible implementations 'I' are evaluated.

6. Experiments and Results

Our Hot Swapping technique has currently been implemented as a C++ standalone scheduler (the next step will be the implementation into an RTOS) and it has tested throughout real-world task sets and in addition to a large number of generated task sets. The task sets have been chosen to investigate different scenarios and to evaluate the functionality and advantage of our technique within various contexts and scenarios. Some of the varying parameters are:

a) the number of tasks within a certain task set,
b) the number of tasks within a subset that have more than one possible implementation and

c) the number of implementations of a single task within a task set.

Besides the basic parameters of a task (execution time, period and cost etc) also the *swap times* may be different (note that a swap τi_a->τi_b can impose a different swapping time than what is required to swap back from τi_b->τi_a).

Table 2.1 shows the timing parameters of 7 periodic tasks and their implementations in simulation of the *EA-CSD* application running on *Palmpilot* (modified version from [12]). Table 2.2 shows the time and power needed for each swap for the same application.

Task	Description	C [ms]	T [ms]	Power [mW]
τ1_1	Parsing NMEA sentences+ receiving from GPS	5	100	90
τ2_1	Keeping serial line open to GPS	7	40	60
τ2_2		9	40	40
τ3_1	Downloading map from serial line	10	100	150
τ3_2		12	100	125
τ3_3		14	100	100
τ4_1	Searching map with memory-intensive computations	6	30	140
τ4_2		8	30	100
τ5_1	Background looping register-based computation job #1	6	50	125
τ5_2		8	50	80
τ6_1	Background looping register-based computation job #2	3	20	125
τ6_2		5	20	70
τ7	Keeping LCD display on	10	150	40

Table 2.1: Timing parameters of the Palmpilot application

The simulation results of the five real-world applications together with the results of a large number of generated task sets is shown in Figure 3 (the generated task sets are represented as one chart only since this chart gives the average savings over all these generated task sets).

By simulating each task set from t = 0 to a common multiple of all the tasks' periods and assigning a large number of different attempts of Hot Swapping at run time we obtained the power consumption shown in Figure 3: shown are the largest power savings found through our technique for each tasks set. **Thereby, all the deadlines are met i.e. the correct functionality is preserved.** The power reductions range from 8.3% to 47.0% depending on the specific task set and its implementations. The comparisons refer to the same respective task sets by applying RMS solely (without Hot Swapping). Figure 4 shows the relationship between the utilization factor and power savings. The power reduction ranges from 22.2% up to 47% when the utilization factor is in the range of 56% to 86%. The power reduction drops drastically when the utilization factor is about 93% in the *Olympus Satellite* application. That implies that the utilization factor is not linearly proportional to the resulting power reduction but serves as a limiting factor to power reduction when it reaches saturation. This has been expected since the idle time of the CPU is very small due to the high utilization factor, thus not allowing enough time gap for any Hot Swapping to carry out. In fact, the feasibility test may not even utilization factor, thus not allow the swap to take place because more tasks would miss their deadline during such a swap process.

Task	Swapping *from* ...	*to*...	Required Time [ms]	Power Consumption [mW]
τ2	τ2_1	τ2_2	1	10
	τ2_2	τ2_1	2	15
τ3	τ3_1	τ3_2	2	20
	τ3_1	τ3_3	3	25
	τ3_2	τ3_1	3	20
	τ3_2	τ3_3	2	30
	τ3_3	τ3_1	3	25
	τ3_3	τ3_2	2	30
τ4	τ4_1	τ4_2	2	15
	τ4_2	τ4_1	1	10
τ5	τ5_1	τ5_2	2	20
	τ5_2	τ5_1	3	30
τ6	τ6_1	τ6_2	1	30
	τ6_2	τ6_1	2	20

Table 2.2: Swap behavior of the Palmpilot application

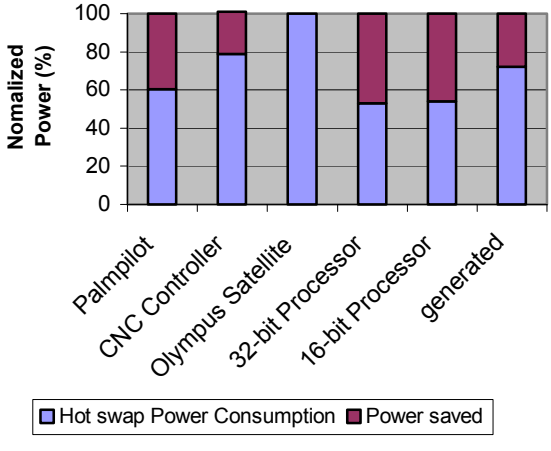

Figure 3: Final result of applying Hot Swap and RMS to five different task sets

7. Conclusions

We have presented a technique that we call Hot Swapping. It conducts scheduling feasibility analyses as well as the final scheduling for real-time systems. The technique conducts swappings between various implementations of tasks on-the-fly. Our technique is an extension to the RMS (Rate Monotonic Scheduling) theory that assumes only one implementation per task. We have currently implemented the technique as a standalone C++ scheduler and evaluated it using a variety of diverse real-world real-time task sets with differing constraints. The experiments and results show that Hot Swapping is capable of reducing power consumption between 22% and 47% for real-world real-time task sets compared to the original RMS scheduling technique.

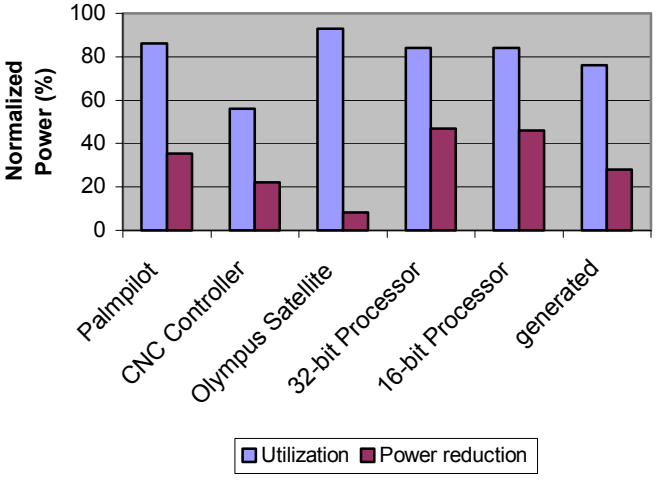

Figure 4: Utilization factor versus power reduction of the five different task sets

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *Journal of the ACM, vol. 20, no. 1*, pp. 46-61, Jan. 1973.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior,' in *Proc. IEEE Real-Time Systems Symposium*, pp. 166-171, Dec. 1989.

[3] L. Sha, R. Rajkumar, J. Lehoczky, K. Ramamritham, "Mode change protocols for priority-driven pre-emptive scheduling", in *Real Time Systems*, pp. 244-264, Dec. 1989.

[4] K. Tindell, A. Burns, A. Wellings, "Mode changes in priority pre-emptively scheduled systems", in *Proc. Real Time Systems Symposium*, 1992.

[5] Y. Shin and K. Choi, "Enforcing schedule-ability of multi-task systems by hardware-software codesign," in *Proc. International Workshop on hardware/Software Co-Design*, pp. 3-7, March 1997.

[6] P. Pedro and A. Burns, "Schedulability analysis for mode changes in flexible real-time systems," in *Proc. Euromicro Workshop on Real-Time Systems*, pp. 19-23, June 1998.

[7] Y. Shin, D. Kim, and K. Choi, "Schedule-ability-driven performance analysis of multiple mode embedded real-time systems," in *Proc. Design Automation Conf*erence, pp. 495-500, June 2000.

[8] *G. A. Paleologo, L. Benini, A. Bogliolo, G. De Micheli,"* Policy Optimization for Dynamic Power Management," in Design Automation Conference, pg 182-187, June 1998.

[9] L. Benini and G.D. Micheli. "Dynamic Power Management: Design Techniques and CAD tools" Kluwer, 1997.

[10] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Tr. on VLSI Syst.*,pp. 299-306,June 2000.

[11] N. Kim, M. Ryu and S. Hong, "Visual assessment of a real-time system design: a case study on a CNC controller," in Proc. IEEE Real-Time Systems Symp., pp. 13-21, Dec 1996.

[12] C. Ma, K. Shin, "A user-customizable energy-adaptive combined static/dynamic scheduler for mobile applications", in Proc. Real-Time Symposium, Nov 2000.

[13] A. Burns, K. Tindell, A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers", in *IEEE Tr. on Software Engineering*, pp. 475-480, May 1995.

[14] K.M. Zuberi, P. Pillai, K. G. Shin, "EMERALDS: a small memory real-time microkernel", in *Proc. of 17th ACM Symp. on Operating Systems Principles*, pp. 277-291, Dec. 1999.

[15] S. Srinivasan, D. Stewart, "High speed hardware-assisted real-time inter-process communication for embedded microcontrollers," in *Proc. IEEE Real-Time Systems Symposium, pp. 139-144, Nov. 2000.*

[16] E.L.Lawler and C.Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 7, pp. 9-12, Feb. 1981.

[17] C. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics platform in Ada: A case study," in *Proc. Real-Time System Symposium*, Dec. 1994.

[18] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, W. Zhao, "Algorithms for scheduling imprecise computations," in *Proc. IEEE Computer*, pp. 58-68, May 1991.