Power-conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-time Embedded Systems

Jiong Luo and Niraj K. Jha
Department of Electrical Engineering
Princeton University, Princeton, NJ, 08544
{jiongluo, jha}@ee.princeton.edu

Abstract

In this paper , we present a power-conscious algorithm for jointly scheduling multi-rate periodic task graphs and aperiodic tasks in distributed real-time embedded systems. While the periodic task graphs have hard deadlines, the aperiodic tasks can have either hard or soft deadlines. Periodic task graphs are first scheduled statically. Slots are created in this static schedule to accommodate hard aperiodic tasks. Soft aperiodic tasks are scheduled dynamically with an on-line scheduler. Flexibility is introduced into the static schedule and optimized to allow the on-line scheduler to make dynamic modifications to the static schedule. This helps minimize the response times of soft aperiodic tasks through both resource reclaiming and slack stealing. Of course, the validity of the static schedule is maintained. The on-line scheduler also employs dynamic voltage scaling and power management to obtain a power-efficient schedule. Experimental results show that the flexibility introduced into the static schedule helps improve the response times of soft aperiodic tasks by up to 43%. Dynamic voltage scaling and power management reduce power by up to 68%. The scheme in which the static schedule is allowed to be flexible achieves up to 32% more power saving compared to the scheme in which no flexibility is allowed, when both schemes are power-conscious. Our work gives an average architecture price saving of 30% over a previous approach for embedded system architectures synthesized with execution slots for hard aperiodic tasks present.

1. Introduction

High-performance distributed embedded systems [1] are generally composed of a heterogeneous network of processing elements (PEs), where a PE can be a general-purpose processor, an application-specific integrated circuit, or a field-programmable gate array. The input specification of such a system is typically in the form of task graphs. A task graph is a directed acyclic graph in which each node is associated with a task and each edge is associated with the amount of data that must be transferred between the two connected tasks. The period associated with a task graph indicates the time interval after which it executes again. A hard deadline, the time by which the task associated with the node must complete its execution, exists for every sink node and some intermediate nodes. An embedded system may contain multiple task graphs with different periods. Such a system is called a multi-rate system. Besides periodic task graphs, the embedded system may also contain aperiodic tasks. An aperiodic task is invoked for execution at any time and may have a hard deadline or a soft deadline. All hard deadlines must be met. However, one only needs to minimize the response times of soft aperiodic tasks. For hard aperiodic tasks, generally a minimum inter-instance arrival time is specified.

Power consumption is a big concern in the design of portable battery-powered embedded systems. Dynamic voltage scaling and power management represent two powerful system-level techniques to reduce power consumption. Dynamic voltage scaling refers to dynamically varying the speed of a processor by changing the clock frequency along with the supply voltage. Dynamic power management refers to the use of power-down modes when the processor is idle to reduce processor power. Many modern embedded processors support software-controlled sleep modes [20, 21, 22]. The Crusoe processor [21], which is a new generation embedded processor targeting mobile computing, allows its clock frequency to be adjusted on the fly. It also provides the capability for software to adjust the processor's voltage on the fly correspondingly.

The problem of jointly scheduling both periodic task graphs and aperiodic tasks is an important issue in many real-time embedded systems. The goal of real-time scheduling algorithms is to guarantee the deadlines of periodic task graphs and hard aperiodic tasks while providing good response times for soft aperiodic tasks. Given the importance of power consumption, the scheduling algorithm should be power-conscious as well. For example, it should be able to vary the voltage of PEs and manage power dynamically, while maintaining the validity of the schedule.

1.1 Previous Work

There have been extensive studies in the literature on the scheduling of periodic tasks, aperiodic tasks, and their combinations. The work in [6, 25] gives several bandwidth-preserving polling server approaches. The slack stealing algorithm presented in [3] attempts to make time for servicing aperiodic task by stealing all the processing time it can from the periodic tasks. The adaptable fixed-priority algorithm in [4] employs variants of a scheme where tasks are assigned a fixed priority. All the above approaches target only a single processor and are applicable to only independent task sets, in which no precedence is defined among the different tasks. The method in [2] performs resource reclaiming in shared-memory real-time multiprocessor systems, where resource reclaiming refers to exploiting a PE at run-time when the actual execution time of a task is less than its specified worst-case execution time. The work in [23] performs joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. The algorithm in [7] performs concurrent hardware-software co-synthesis of hard real-time aperiodic and periodic task graphs. This involves allocating the required number of PEs and communication links of different types, assigning tasks and communication events to them and scheduling these tasks and events to meet real-time constraints while minimizing overall system price.

Dynamic voltage scaling has also been studied in the literature. The work in [14] proposes methods to vary the clock speed dynamically along with the supply voltage under the control of the operating system. The work in [9] proposes a synthesis technique for variable-voltage core-based systems containing a set of independent tasks with arbitrary arrival times. The work in [15] presents a power-conscious fixed-priority scheduling algorithm for hard real-time systems using rate-monotonic scheduling.

For dynamic power management, a policy is needed to achieve a good trade-off between latency and savings in system power. The fixed time-out policy shuts down the processor after a fixed amount of idle time. The predictive system shutdown methods [8, 10, 12] predict the upcoming idle period based on previous user behavior. The stochastic modeling approaches in [11, 13, 24] use a discrete-time or continuous Markov chain to model the system and workload.

1.2 Our Approach and Contributions

In this paper, we give an algorithm to jointly schedule multi-rate periodic task graphs along with hard and soft aperiodic tasks. The

Acknowledgments: This work was supported in part by Army CECOM and in part by DARPA under contract no. DAAB07-00-C-L516

power issue is addressed through both dynamic voltage scaling and power management. The algorithm allocates resources to accommodate both periodic and hard aperiodic tasks. It then generates a feasible schedule based on the maximum supply voltage of each PE, which satisfies all the timing and communication constraints and has enough capacity reserved to serve the hard aperiodic workload, given minimum inter-instance arrival times of such tasks. Flexibility is introduced into the static schedule and optimized to allow the on-line scheduler to make local changes to PE schedules, without interfering with the validity of the global schedule of the distributed embedded system.

Our work makes several contributions: (1) We combine resource reclaiming and slack stealing in a distributed embedded system to improve the response time of soft aperiodic tasks. Although the concepts of resource reclaiming and slack stealing have been used in single-processor systems, there is only limited work in simultaneously addressing them in distributed systems which execute tasks with precedence constraints. The work presented in [2] only targets resource reclaiming in a multi-processor shared-memory system. The work in [23] exploits unused resources and leeway in statically scheduled distributed real-time systems. As opposed to the work in [23], we perform a global optimization to achieve a better distribution of the flexibility in the static schedule. Moreover, our on-line scheme is simpler and more powerful in exploiting the slack time. (2) The combination of resource reclaiming and slack stealing also facilitates dynamic voltage scaling and power management. (3) Schedule slot reservation for hard aperiodic tasks is more efficient than the previous method given in [7], while still guaranteeing that all deadlines are met. (4) We provide a unified framework in which the response times of aperiodic tasks and power consumption are dynamically optimized simultaneously, which has not been done before.

This paper is organized as follows: Section 2 presents a motivational example. Section 3, 4, 5, 6 and 7 elaborate upon several aspects of our approach in detail. These include static resource allocation, assignment and scheduling, handling of hard aperiodic tasks, on-line scheduling for soft aperiodic tasks, dynamic voltage scaling and power management, as well as global optimization of the static schedule, respectively. Section 8 presents experimental results. Section 9 provides the summary.

2. Motivational Example

In this section, we present an example, which motivates the various problems we address in this paper. The calculation of clock period and power consumption in this example is based on Equations (1) and (2) respectively, which are presented below.

The processor clock period, T, can be expressed in terms of the supply voltage, V_{dd}, and threshold voltage, V_t, as follows:

$$T = kV_{dd}/(V_{dd} - V_t)^2 \qquad (1)$$

where k is a constant. We assume $V_t = 0.8$V. The processor power, P, can be expressed in terms of the frequency, f, switched capacitance, N, and supply voltage as:

$$P = \frac{1}{2} fNV_{dd}^2 \qquad (2)$$

Example 1: Fig. 1 gives an embedded system specification consisting of two task graphs. Assume for simplicity that both have a period of 9.5 time units.

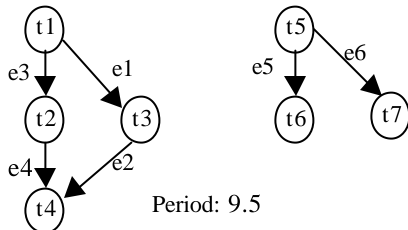


Fig.1 : Task Graphs

Figs. 2 and 3 give two feasible schedules on a distributed system consisting of PEs PE1 and PE2 connected by a link. These are as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules, respectively. The schedules are based on the worst-case execution times of tasks and communication times given in Tables 1 and 2, respectively, assuming a supply voltage of 3.3V. Based on the traditional assumption in distributed computing, we assume intra-PE communications, e3, e4, e5 and e6, all take zero time. We assume PE2 does not have a communication buffer. Therefore, the communication edges also need to be scheduled on it.

Tasks	Worst-case exec. time	Actual exec. time	Deadline	Assignment
t1	2.0	1.5	/	PE1
t2	2.5	2.0	/	PE1
t3	3.0	/	/	PE2
t4	2.0	1.5	9.0	PE1
t5	1.5	/	/	PE2
t6	1.0	/	9.5	PE2
t7	1.5	/	9.5	PE2

Table 1: Execution Times, Deadlines and Task Assignments

Edges	Worst-case comm. time	Actual comm. time
e1	0.5	0.5
e2	0.5	0.5

Table 2: Communication Times

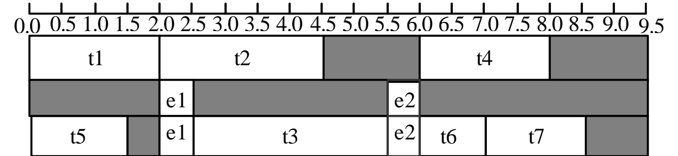


Fig. 2: An ASAP Schedule
(top: PE1, middle: link, bottom: PE2)

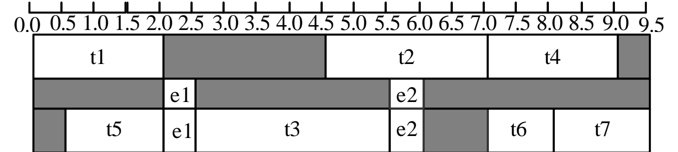


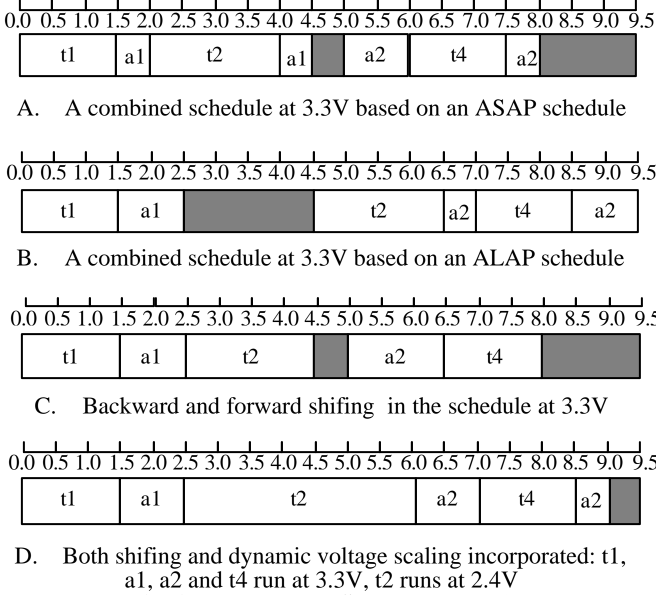
Fig. 3: An ALAP Schedule with the Communication Schedule Fixed as in Fig. 2 (top: PE1, middle: link, bottom: PE2)

Fig. 4 shows the post-run schedule with respect to the *actual execution times* of tasks on PE1. Two soft aperiodic tasks, a1 and a2, arrive at the PE at times 1.5 and 5.0, respectively. Their execution times are 1.0 and 1.5 time units, respectively. Four different schemes are presented in this Figure. Scheme A (B) incorporates the aperiodic tasks into the ASAP (ALAP) schedule, and uses the actual execution times of the periodic tasks. Schemes C and D allow forward and backward shifting of the static schedule. Scheme D also incorporates dynamic voltage scaling. PE1 is assumed to be shut off in the shaded parts of the schedule. Schemes C and D still meet the underlying real-time and precedence constraints provided that the communication edges are always scheduled in the fixed slots as in Fig. 2 or Fig. 3. For simplicity, we assume that the power consumption in the shut-off state is zero and that there is no overhead in entering and leaving this state. We also assume that there is no preemption cost in this example. Note, that our algorithm, which is presented later, does not need to make the above assumptions about the preemption cost and shut-off state.

In Scheme C, t2 is scheduled at time 2.5 in order to service a1, and t4 is scheduled at time 6.5 to service a2. In Scheme D, when t2 is scheduled at time 2.5, since its latest finish time is 7.0, the processor

speed can be scaled down by a ratio of (7.0 – 2.5) / 2.5. Correspondingly, the supply voltage can be scaled down from 3.3V to 2.4V, extending the actual running length of t2 from 2.0 to 3.6.

Using Equations (1) and (2) to compute the schedule length and power consumption, the performance of the different schemes is compared in Table 3.



A. A combined schedule at 3.3V based on an ASAP schedule

B. A combined schedule at 3.3V based on an ALAP schedule

C. Backward and forward shifing in the schedule at 3.3V

D. Both shifing and dynamic voltage scaling incorporated: t1, a1, a2 and t4 run at 3.3V, t2 runs at 2.4V

Fig. 4: The Post-run Schedule on PE1

From Table 3, we observe that the introduction of flexibilty in the schedule improves the response times of the aperiodic tasks. Comparing the results for Schemes C and D, we observe that voltage scaling achieves a better power saving than PE shut-off. The power reduction is at the cost of extra latency for one of the soft aperodic tasks. □

Scheme	A	B	C	D
Response time of a1	3.0	1.0	1.0	1.0
Response time of a2	3.0	4.5	1.5	4.1
Average response time	3.0	2.75	1.25	2.55
Power	1	1	1	0.87

Table 3: Aperiodic Task Response Time and Power Consumption for the Different Schemes

3. Static Resource Allocation, Assignment and Scheduling

The static resource allocation, task/communication assignment and scheduling algorithm we use is from a system synthesis tool [18]. It uses a slack-based list scheduling algorithm to generate static PE and communication link schedules for each task and communication event along the hyperperiod, which is the least common multiple of all the task graph periods in a multi-rate system specification. It is well known that there exists a feasible schedule for the periodic task graphs if and only if there exists a feasible schedule for the hyperperiod [19]. Static scheduling makes it possible to guarantee that hard real-time constraints of periodic task graphs will be met. The static schedule from [18] is modified with a post-processing stage, as explained later in Section 7.

4. Handling of Hard Aperiodic Tasks

One approach to handle hard aperiodic tasks is to reserve execution slots for them at regular intervals throughout the hyperperiod [7]. The hard aperiodic tasks are always executed at the next available execution slot.

Suppose that a hard aperiodic task *j*, with a deadline d_j, *is* assigned to PE *k*, and the execution time of task *j* on PE *k* is $\mathbf{m}_j$. We assume that $\mathbf{m}_j \leq d_j$. Suppose that the minimum arrival time between any two consecutive instances of task *j* is Y_j. We assume that $Y_j \geq d_j$, or equivalently, in any time frame of length d_j, only one instance of task *j* is assumed to be present.

In [7], the length of each reserved execution slot for the aperiodic task in the hyperperiod is $\geq \mathbf{m}_j$, and the interval between the reserved execution slots is $\leq d_j - 2\mathbf{m}_j$. This should be satisfied in a cyclic way along the hyperperiod. The allocation of execution slots in this way is illustrated in Example 2.

Example 2: Assume a hard aperiodic task has an execution time of 1 on the PE it is assigned to and a deadline of 5. Suppose the hyperperiod of the embedded system is 10. Then one feasible allocation of execution slots following the above condition is given in Fig. 5. The total allocated schedule length is 3. □

In our approach too, we reserve execution slots for hard aperiodic tasks throughout the hyperperiod. However, for individual aperiodic tasks, our approach allows preemption of hard aperiodic task instances. This results in a smaller schedule and, hence, a less costly distributed embedded system architecture. Under the assumption that $Y_j \geq d_j$, it is obvious that the optimal condition under which the deadlines of hard aperiodic task instances can be guaranteed irrespective of their arrival times is: for any time frame of length d_j, there is enough capacity to service one instance of the hard aperiodic task. This optimal condition leads to Theorem 1, which is applied in our approach.

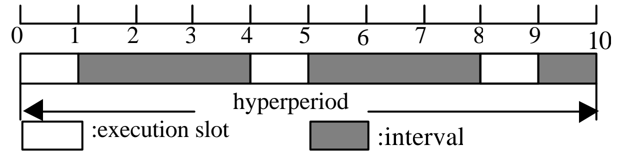


Fig. 5: Allocation of Execution Slots

Theorem 1: Under the assumption that $Y_j \geq d_j$, and that each reserved execution slot is at least of length $\mathbf{m}_j$, the necessary and sufficient condition for ensuring that the deadline is met is that the interval between two reserved execution slots is $\leq d_j - (\mathbf{m}_j + p_p + p_r)$, where p_p (p_r) is the worst-case context switch cost in preempting (resuming) the hard aperiodic task. We assume $p_p + p_r$ is always less than $\mathbf{m}_j$.

Proof:

Necessity: Assume that an interval between two reserved execution slots is greater than $d_j - (\mathbf{m}_j + p_p + p_r)$. Consider a time frame of length d_j which contains this interval. Also, assume the execution region of this time frame covered by the first execution slot is $p_p + p_r + \mathbf{d}$, where $\mathbf{d}$ is a positive value less than $\mathbf{m}_j - (p_p + p_r)$. Then the region covered by the second execution slot is less than $\mathbf{m}_j - \mathbf{d}$. Assume an instance of the aperiodic task arrives at the beginning of this time frame. Since neither of the two execution regions is enough to finish the execution, the task has to be scheduled using the combination of the two execution regions. Hence, it has to be preempted once. However, the total execution length of this time frame is less than $\mathbf{m}_j + p_p + p_r$. Hence, the finish time of the task will exceed this time frame, i.e., it cannot meet the deadline.

Sufficiency: Consider any time frame of length d_j. If it contains a complete execution slot, then this time frame has enough capacity to finish the task. If it does not contain a complete execution slot, then it must contain only one interval between two execution slots while not crossing any other intervals. Since the interval is $\leq d_j - (\mathbf{m}_j + p_p + p_r)$, the execution region covered by this time frame must be $\geq d_j - (d_j - (\mathbf{m}_j + p_p + p_r)) = \mathbf{m}_j + p_p + p_r$, which is enough to serve the task even if preemption occurs, since

obviously only one preemption can occur in this case. □

In Example 2, assume $p_p = p_r = 0.05$. One feasible allocation of execution slots following the condition in Theorem 1 is given in Fig. 6. The total length of execution slots is 2.2, which is a 27% improvement compared to the scheme in Fig. 5.

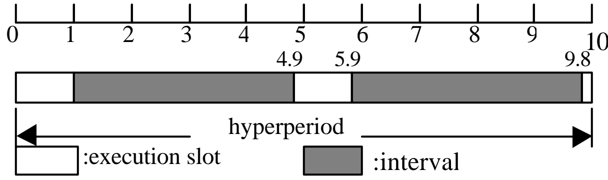


Fig. 6: An Improved Allocation of Execution Slots

5. On-line Scheduling Algorithm

The job of the on-line scheduling algorithm is to dispatch the periodic tasks according to the static schedule, serve the hard aperiodic task with the reserved execution slots, and serve the soft aperiodic tasks with the execution time left unused by the hard tasks. An on-line scheduling algorithm should be correct, inexpensive and of bounded complexity [2]. We intend to introduce flexibility into the static schedule which the on-line scheduling algorithm can take advantage of. To maintain the simplicity of the on-line scheduling algorithm, local flexibility is introduced into the static feasible schedule in the following way:

(1) The schedule of all the communication events is kept fixed. This helps to make the flexibility introduced in the static schedule local to each PE. Hence, no global re-scheduling is required. Also, the order of all the events scheduled on every PE and link is kept fixed. This helps to maintain the precedence constraint among the tasks assigned to the same PE.

(2) A table generated off-line provides the *earliest start* and *latest finish* times (correspondingly the *latest start* time) for each scheduled event. This shows how much slack is available for the event. This can be exploited at run-time.

Fig. 7 presents Algorithm 1 for computing the *earliest start* and *latest finish* times. In this algorithm, event*_list* is a list of statically scheduled events in the order of their start times on each PE for one hyperperiod. The scheduled event is a periodic task, or a hard aperiodic task execution slot, or a communication event (*comm* in the algorithm). In the static schedule, every event is characterized by a *start* time and a *finish* time. For a task, *in-edges* (*out-edges*) refers to all the *inter-PE* communication edges entering (coming out of) the task, where *inter-PE* communication edges refer to those edges for which the parent task and child task are assigned to different PEs. A *deadlin*e may be associated with a task. It is ∞ if it is undefined for the task. A task graph may also have a start time requirement which is characterized as *required_start.*

The statically scheduled events are dispatched following the order in *event_list* in a cyclic way. The current event refers to the scheduled event currently being processed. Soft aperiodic tasks arrive and leave each PE in a first-in first-out (FIFO) order. The on-line scheduler works in the following way: whenever there are soft aperiodic tasks pending, and the latest start time of the current event has not been reached, the scheduler dispatches the soft aperiodic task. If there are no soft aperiodic tasks pending, and the earliest start time of the current event is reached, the scheduler dispatches the current event. If the current event is running and a soft aperiodic task has arrived, the scheduler detects if there is enough slack time in the current event to serve the soft aperiodic task. If so, the current event gets preempted and the incoming aperiodic task gets dispatched.

The capacity of the hard aperiodic task slots can be fully or partially released based on the previous history and minimum inter-instance arrival time of the hard aperiodic task. When the schedule arrives at the reserved execution slot, the slot is used to either serve the pending hard aperiodic task, or wait for the arrival of the corresponding hard aperiodic task.

The dispatch procedure of the on-line scheduler is illustrated through Example 3.

Example 3: Assume there are five events in the static schedule for a PE and the hyperperiod is 700. The characteristics of the scheduled events and arriving events are shown in Tables 4 and 5, respectively. In Table 4, the column entries *Start* and *Finish* indicate the start and finish time for that event in the static schedule, while the column entries *Earliest-start* and *Latest-finish* indicate the off-line flexibility of the static schedule. The minimum inter-instance arrival time of hard aperiodic tasks is 650. The preemption and resumption cost is 10.

```
Algorithm 1: compute_flexibility_offline(event_list){
   latest_start = hyperperiod;
   for( i = event_list → last; i != event_list → begin; i-- ){
      start = i → start;
      finish = i → finish;
      if (type(i) is comm || type(i) is a hard aperiodic task slot) {
         i → earliest_start = start;
         i → latest_finish = finish;
         i → latest_start = start;
      }
      else if (type(i) is task){
         i → earliest_start = max( required_start,
                                   max_{j ∈ in_edges(i)} (j → finish));
         i → latest_finish = min(i → deadline, latest_start,
                                 min_{j ∈ out_edges(i)} (j → start));
         i → latest_start = i → latest_finish − i → worst_execution_time;
      }
      latest_start = i → latest_start;
   }
}
```

Fig. 7: Computation of the Earliest Start and Latest Finish Times

Event	Type	Earliest -start	Latest-finish	Start	Finish	Actual exec. time
p1	P	0	100	0	100	100
e2	Ap	100	250	100	250	/
p3	P	0	450	250	350	80
p4	P	400	550	400	500	90
e5	Ap	550	700	550	700	/

Table 4: Characteristics of Statically Scheduled Events (P: periodic task; Ap: hard aperiodic task execution slot)

Task	Type	Arrival time	Worst-case exec. time	Actual exec. time
a1	Hard	0	150	80
a2	Soft	200	100	100
a3	Soft	360	80	60
a4	Soft	510	100	100

Table 5: Arrival Times and Execution Times of Aperiodic Tasks

The execution of tasks in the dispatched order is shown in the following list (each element is of the format: event (start, end)):

{p1(0, 100); a1(100, 180); p3(180, 200); preemption of p3(200, 210); a2(210, 310); resumption of p3(310, 380); a3(380, 440); p4(440, 530); a4 (530, 630) }

p1 is dispatched first. Then e2 is used to serve a1. The remaining capacity of e2 is released and p3 gets dispatched earlier than its specified start time. Then a2 arrives and preempts p3 since it is detected that p3 has enough slack time. p3 is resumed later after a2 finishes. Note that the resumption cost is included for p3 at this stage. After that, a3 gets dispatched and finishes since the latest start time of p4, which is 550 – 100 = 450, has not been reached. p4 is then dispatched. Following this, a4 gets executed with the released

capacity of e5. □

6. Dynamic Voltage Scaling and Power Management

The flexibility introduced in a static schedule, as discussed in Section 5, can also be used to facilitate dynamic voltage scaling and power management. However, there are energy and delay penalties involved in shutting a system down. Also, as mentioned before, reducing the voltage always slows down the currently scheduled task. There is also extra system overhead involved in adjusting the system power supply and clock frequency. The delay penalties mentioned above conflict with the goal of minimizing the response times of soft aperiodic tasks. Hence, the on-line scheduler needs to intelligently decide when to enter the sleep mode, which level of sleep mode to enter, as well as when to reduce the supply voltage to extend the task schedule, without violating the feasibility of the guaranteed tasks, and without causing too much increase in the response times of the soft aperiodic tasks.

The power-conscious dispatching procedure is similar to the one in Section 5. The major difference is that the on-line scheduler needs to predict the next idle period for the PE or predict the actual available processing time for a scheduled event when there are no soft aperiodic tasks pending. Periodic and hard aperiodic tasks are dispatched according to the static schedule and thus have good predictability. The dynamic nature of the schedule lies in the unknown arrival times of the soft aperiodic tasks. To manage power consumption efficiently, it is important to predict when the next soft aperiodic task will arrive.

If we assume that aperiodic task arrivals can be modeled as a Poisson process [25], then the inter-instance arrival times of aperiodic tasks assume an exponential distribution. In such a case, the best estimation, I_k', for the inter-instance arrival time I_k should be $I_k' = E\{I_k\} = 1/l$, where l is the arrival rate of the Poisson process. In order to capture the possible time-varying dynamics of the arrival rate, we use the average of previous n values to approximate the mean: $I_k' = \frac{1}{n}\sum_{i=1}^{n} I_{k-i}$. If the predicted inter-instance arrival time has elapsed, but a new soft aperiodic task has not arrived, we need to make a new prediction. Based on the property of an exponential distribution, we need to wait for the same period of time again before expecting to see a new soft aperiodic task. I_k' can be used to compute *predicted_next_arrival*, which is the predicted arrival time of the next soft aperiodic task.

To guarantee a prompt response to soft aperiodic tasks, dynamic voltage scaling and power management are performed only if there are no soft aperiodic tasks pending. Since dynamic voltage scaling has more potential in saving power than dynamic power management [9], the former is always performed first.

If we are dispatching a statically scheduled event k and there are no soft aperiodic tasks pending, then the actual available running time for k is:

$$available_time = \min(predicted_next_arrival, k \rightarrow latest_finish) - timestamp$$

where $timestamp$ is the current time point. The worst-case remaining time for executing k is:

$$worst_remaining = worst_exec_time - executed$$

where *executed* is the amount of execution time that has already been spent for event k.

The ratio of *available_time* to *worst_remaining* is used to determine the scale for reducing the clock frequency and supply voltage, using a scheme similar to the one in [15].

If there are no soft aperiodic tasks pending and the PE is currently idle, then the next idle period IP can be predicted as:

$$IP = \min(k \rightarrow earliest_start, predicted_next_arrival) - timestamp$$

where k refers to the statically scheduled event currently being processed.

IP can be used to justify which sleep state to enter such that its energy consumption in IP is minimized. The energy consumption for assuming sleep state i in idle period IP is

$$EC_i = E_i * P_{E_i} + W_i * P_{W_i} + (IP - E_i - W_i) * P_i$$

where E_i (W_i) is the delay overhead and P_{E_i} (P_{W_i}) is the power consumption in entering (leaving) sleep state i. P_i is the power consumption in this state. We choose the sleep state which minimizes the energy consumption [12].

After entering sleep state i, the processor wakes up either when a soft aperiodic task arrives, or when a hard task is ready to be scheduled.

7. Global Optimization of the Static Schedule

The static schedule should be such that it provides the maximum flexibility to the on-line scheduler. As discussed in Section 5, the schedule of all the communication events is kept fixed. Therefore, how the communication events are distributed throughout the static schedule affects the flexibility that can be exploited on-line. The original static schedule is generated with an objective that the cost of the system architecture is optimized (in case of hardware-software co-synthesis). It is based on a list scheduling algorithm. The schedule slot is often positioned as early as possible when an event is ready to be scheduled. Hence, the distribution of communication events is not optimized.

We add a post-processing stage to the static scheduler which tries to globally re-position the schedule slots while still meeting the real-time constraints and precedence relationships of the original schedule. The basic heuristics in our algorithm is to distribute the flexibility more evenly along the hyperperiod. As far as voltage scaling is concerned, intuitively, more evenly distributed flexibility in the static schedule can lead to more even voltage scaling, and thus higher power savings. Furthermore, under the assumption that the arrival times of soft aperiodic tasks are randomly distributed, more evenly distributed flexibility should also help reduce the average response time of soft aperiodic tasks. The major part of the schedule optimization procedure, which tries to shift forward the scheduled events in the original schedule in order to distribute the slack time more evenly, is illustrated in Algorithm 2 in Fig. 8.

In Algorithm 2, *pe_sched* and *link_sched* are arrays of *event_list* and *comm_list* for PEs and links, respectively. *event_list* is defined in Algorithm 1, and *comm_list* is the list of statically scheduled communication events in the order of their start times on each link for one hyperperiod. In Algorithm 2, we maintain two queues for processing, i.e., *task_queue* and *comm_queue,* which are the processing queues for the statically scheduled tasks on PEs and communication events on links, respectively. For a scheduled event, *next_event* is the next scheduled event in the same *event_list* or *comm_list*. A task is inserted into the *task_queue* only if all its *out-edge*s and its following tasks in the same *event_list* have finished shifting. A communication edge is inserted into *comm_queue* only if its child task has finished shifting. Both *task_queue* and *comm_queue* are in the reverse order of the start times of the scheduled events in the queue. The *slack* time of a task is defined as the difference between the minimum of its *latest finish* time and the *start* time of its *next_event*, and its *finish* time. The *average slack ratio* for each PE is calculated as the ratio of the total *slack* time over total worst-case execution time for all the tasks on that PE. The sub-function *shift_forward_task* (*event_list ,i, constraint, slack_amount*) shifts forward the scheduled task i in *event_list*, while maintaining a *slack* time of task i no greater than *slack_amount* and a *finish* time of i not exceeding *constraint*. The sub-function

shift_forward_edge (*comm_list ,e ,constraint, slack_amount*)

simultaneously shifts forward the communication event e in *comm_list*, as well as all the corresponding communication events on any non-buffered PE the parent task or the child task is assigned to, in a manner similar to

shift_forward_task (*event_list ,i, constraint, slack_amount*) .

The shifting procedure presented in Algorithm 2 is illustrated through Example 4.

Example 4: Fig. 9 shows an embedded system specification consisting of two task graphs. Fig. 10 gives a feasible static schedule

on a distributed system consisting of PEs PE1 and PE2 connected by a link. The worst-case execution times of t1 and t3 on PE1 are both 1 time unit. The worst-case execution times of t2 and t4 on PE2, and t5 on PE1 are all 2 time units. The communication times of inter-PE communication edges e1 and e3 are both 1 time unit. We assume both PEs are buffered. Fig. 11 gives the optimized static schedule after shifting.

```
Algorithm 2: shift_static_sched( pe_sched, link_sched ){
  for( pe = 0; pe < num_of_pes; pe++ ){
    total_slack = total_execution = 0;
    for( i = pe_sched[pe] → begin; i != pe_sched[pe] → end; i-- ){
      next_start = (i → next_event) → start;
      slack = min(i → latest_finish, next_start) − i → finish;
      total_slack += slack;
      total_execution += i → worst_execution_time;
    }
    average_slack_ratio[pe] = total_slack / total_execution;
  }
  initialize comm_queue as empty;
  initialize task_queue;
  while( task_queue != empty || comm_queue != empty ){
    while( task_queue != empty ){
      i = task_queue → head;
      task_queue.delete( i );
      constraint = min( i → deadline,
                   min_{j ∈ out_edges(i)} (j → start) );
      pe = i → pe_assignment;
      shift_forward_task( pe_sched[pe], i, constraint,
             average_slack_ratio[pe] * i → worst_execution_time );
      for all j ∈ in_edges( i )
        comm_queue.insert( j );
    }
    while( comm_queue != empty ){
      e = comm_queue → head;
      comm_queue.delete(e);
      link = e → link_assignment;
      shift_forward_edge( link_sched[link], e, (e → child) → start, 0 );
    }
    insert newly available tasks into task_queue;
  }
}
shift_forward_task( event_list, event, constraint, slack_amount ){
  next_start = (event → next_event) → start;
  actual_slack = min( constraint, next_start ) − event → finish;
  if( actual_slack > slack_amount )
    event → start = min( event → start + actual_slack − slack_amount,
                         next_start − event → worst_execution_time )
}
```

Fig. 8: Static Schedule Optimization

In Fig. 10, For the tasks assigned to PE1, the *slack* times are 0, 1 and 3 for t1, t3 and t5, respectively. The *average slack ratio* for PE1 is (0+1+3)/(1+1+2) = 1. For the tasks assigned to PE2, the *slack* times are 0 and 4 for t4 and t2, respectively. The *average slack ratio* for PE2 is (0+4)/(2+2) = 1. We notice the slack times in Fig. 10 are concentrated in the latter part of the schedule. Fig. 11 presents the new static schedule after shifting. In the shifting procedure, t5 is first shifted forward by 1 time unit, in order to keep a *slack* time no greater than *average slack ratio* * *worst-case execution time*, which is 1 * 2 time units for t5. Then t2 is shifted forward 2 time units in a similar way. Now e1 and e3 are ready for shifting. They are both shifted forward 1 time unit such that their finish times do not exceed the start times of their child tasks t2 and t5, respectively. After e1 and e3 finish shifting, the distribution of flexibility in the schedule is determined. We observe the slack times are distributed more evenly along the hyperperiod in Fig. 11. □

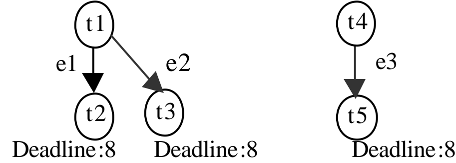


Fig. 9: Task Graphs for Example 4

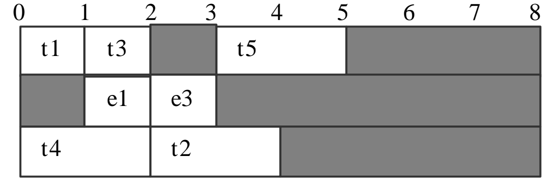


Fig. 10: Original Static Schedule (top: PE1, middle: link, bottom: PE2)

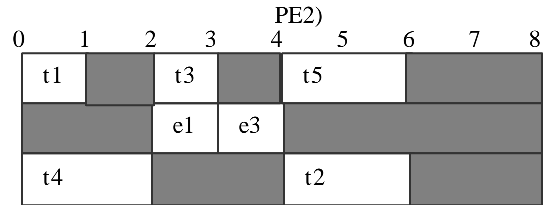


Fig. 11: The Static Schedule after Shifting (top: PE1, middle: link, bottom: PE2)

8. Experimental Results

In this section, we present the experimental results. The task graphs in our example are generated with the aid of TGFF [5], a randomized task graph generator.

The first experiment set compares the cost of the embedded system architecture synthesized with execution slots for hard aperiodic tasks present. In Table 6, our approach is compared with Dave's approach from [7]. The system-on-a-chip synthesis tool discussed in [18] is used for this purpose. Our approach gives an average of 30% architecture price saving over Dave's approach.

The second experiment set deals with the response times of soft aperiodic tasks and system power consumption. For simplicity, we assume that all the processors in the distributed system are PowerPc603e processors [22]. Note that our scheduling algorithm is applicable to heterogeneous distributed systems as well. PowerPC603e provides four power modes [17]: full-power mode, doze mode, nap mode and sleep mode. The power consumption in the doze mode, nap mode and sleep mode with the phase locked loop (PLL) disabled are 16%, 6% and 0.2% of the full-power mode power consumption, respectively. The transition time from the doze mode or nap mode to the full-power mode only takes a few clock cycles, while the transition from the sleep mode with PLL disabled takes several thousand cycles (up to 200 μsec) [17]. The voltage can be varied from 3.3V to 1.6V. We assume that changing the clock frequency and supply voltage takes a worst-case delay of 10 μs [16]. We assume the computation itself can continue during the voltage and frequency change [15].

The static schedule is generated using the tool in [18]. The synthesized system architecture has six PEs with an average utilization factor of 51% with respect to the worst-case execution time of tasks. The actual execution of hard periodic tasks is uniformly distributed in the range of 60% to 100% of their worst-case execution times. The total number of tasks in the periodic task

graphs is 183, and the total number of communication edges is 296.

The soft aperiodic task arrivals are modeled as a mixed Poisson process. The soft aperiodic task workload is characterized by the average arrival rate of such tasks. In the experiment, we compare results for four different arrival rates. They are 0.02, 0.04, 0.06 and 0.08 arrivals/ms, respectively. We also compute the results for power assumption for the case when there are no soft aperiodic tasks present, which is characterized as zero arrival rate in the curves. We use the average response time of soft aperiodic tasks to characterize the performance with respect to response times. The number of soft aperiodic tasks we average over for the different arrival rates is shown in Table 7. The average execution time of soft aperiodic tasks is approximately 2.04 ms. The power consumption of the schemes considered ahead is normalized over the power consumption of the corresponding non-power-conscious scheme with the same arrival rate of soft aperiodic tasks.

No. of task graphs	No. of tasks	Dave's approach			Our approach		
		No. PEs	No. links	Price	No. PEs	No. links	Price
6	61	4	6	395	2	1	216
6	62	3	0	273	2	1	246
6	63	6	8	907	4	6	440
7	69	4	6	459	2	1	277
7	69	4	6	542	4	6	456
8	76	4	6	666	5	8	629

Table 6: Comparison of System Architecture Cost

Average arrival rate (No. of arrivals/ms)	0.02	0.04	0.06	0.08
No. of soft aperiodic tasks	2088	4344	6480	8802

Table 7: Characteristics of Soft Aperiodic Tasks

We compare four different schemes. In the first three schemes, the static schedule used is generated by a list scheduling algorithm and not globally optimized for flexibility, while in the last scheme, the static schedule is globally optimized. These four schemes are: (i) completely static schedule without flexibility, (ii) partially flexible static schedule which incorporates resource reclaiming but not slack stealing, (iii) fully flexible static schedule that incorporates both resource reclaiming and slack stealing, and (iv) fully flexible static schedule as in (iii), but the static schedule is globally optimized. We present results for both the power-conscious case and the non power-conscious case. For the power-conscious full flexibility case with global optimization, we compare the results with and without prediction of future soft aperiodic task arrival times. Recall that this prediction helps determine whether or not to slow down or shut off the processor. When no prediction is done, the processor is slowed down or shut off whenever there is no soft aperiodic task pending, without any consideration for future soft aperiodic task arrivals. Curves for the average response time of soft aperiodic tasks and power consumption vs. average arrival rate of soft aperiodic tasks are plotted.

Fig. 12 presents the results for the non-power-conscious case. We can see that the full flexibility scheme with global optimization (no flexibility scheme) performs the best (worst) in terms of the response times of soft aperiodic tasks. The full flexibility scheme with global optimization performs better than the scheme without optimization. The full flexibility + global optimization scheme achieves as high as 43% improvement in the average response time compared to the no flexibility scheme, and as high as 23% improvement compared to the full flexibility scheme without global optimization.

Fig. 13 shows the results for power-conscious scheduling. The prediction scheme estimates the arrival time of soft aperiodic tasks. Again, we see that the full flexibility + global optimization scheme outperforms other schemes in both the power consumption and the response times of soft aperiodic tasks. As the soft aperiodic task workload increases, the power reduction decreases as shown in Fig. 13(b). Power consumption is computed based on Equation (2). We also observe that there is a tradeoff between power reduction and improvement of response times of soft aperiodic tasks. The highest power saving is achieved when the workload of the soft aperiodic tasks is the lowest, while the highest response time improvement is achieved when the workload is the highest. The full flexibility scheme with global optimization achieves 32% power saving compared to the no flexibility case, when the workload of the soft aperiodic tasks is zero, while it achieves a 38% improvement in the average response time of soft aperiodic tasks compared to the no flexibility case, when the workload of soft aperiodic tasks is the highest. The full flexibility scheme with global optimization achieves as high as 17% improvement in the average response time of soft aperiodic tasks, and 14% power saving, compared to the full flexibility scheme without global optimization.

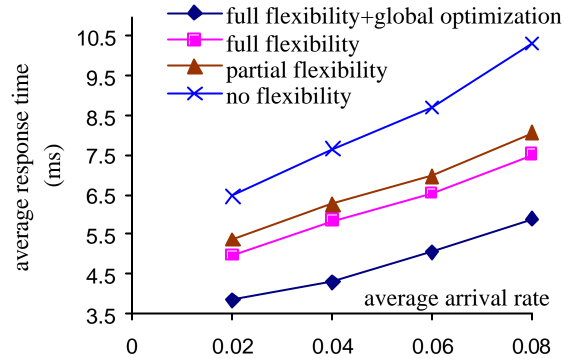
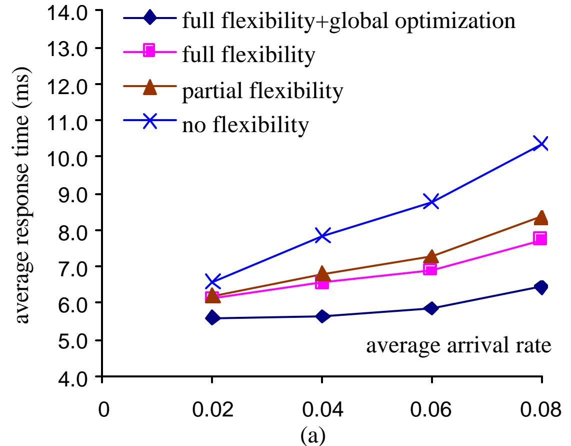


Fig. 12: Non-power-conscious Scheme



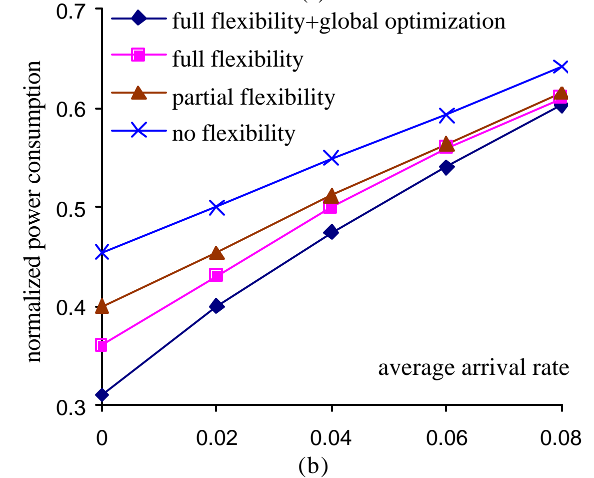


Fig. 13: Power-conscious Schemes (prediction incorporated)

Fig. 14 explicitly shows the tradeoff between the response time of soft aperiodic tasks and power consumption. Three different cases for the full-flexibility scheme with global optimization are compared. These include the non-power-conscious scheme, power-conscious prediction scheme, and power-conscious non-prediction scheme. We observe that the non-power-conscious (non-prediction) scheme achieves the best (worst) aperiodic task response time and the worst (best) power consumption. The prediction scheme performs in between. The prediction scheme achieves a very close performance on power consumption and as high as 30% improvement in the average response time of soft aperiodic tasks compared to the non-prediction scheme. We also observe that the power-conscious prediction scheme reduces power consumption from 40% to 68% compared to the non-power-conscious scheme.

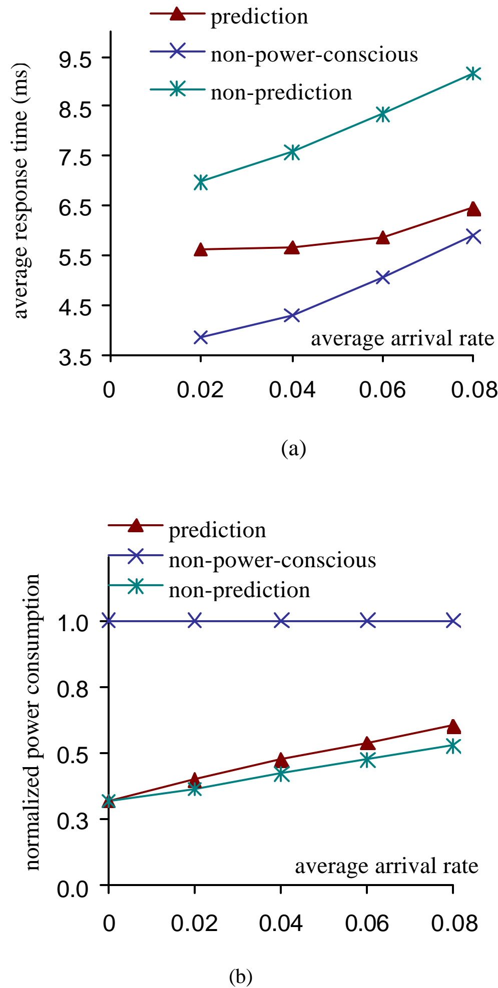


Fig. 14: Comparison of Non-Power-Conscious, Prediction and Non-prediction Schemes

9. Conclusions

We presented a combined static and dynamic approach to facilitate power-conscious joint scheduling of periodic and aperiodic tasks. The deadlines of hard aperiodic tasks are guaranteed through execution slot reservation in the static schedule. Flexibility is introduced into the static schedule and is exploited by the on-line scheduler to improve the response times of the soft aperiodic tasks. Dynamic voltage scaling and power management are also incorporated into the on-line scheduler. Power savings are balanced against the response times of the soft aperiodic tasks, while maintaining a valid static schedule.

References

[1] W. H. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967-989, July 1994.

[2] C. Shen and K. Ramamritham, "Resource reclaiming in multiprocessor real-time systems," *IEEE Trans. Parallel & Distributed Systems*, vol. 4, no. 4, pp. 382-397, Apr. 1993.

[3] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Proc. Real-time Systems Symp.*, pp. 110-123, Dec. 1992.

[4] J. Lee, S. Lee, and H. Kim, "Scheduling soft aperiodic tasks in adaptable fixed-priority systems," *Operating Systems Review*, pp. 17-28, Oct. 1996.

[5] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop Hardware/Software Codesign*, pp. 97-101, Mar. 1998.

[6] J. P. Lehoczyky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. Real-time Systems Symp.*, pp. 261-270, Dec. 1987.

[7] B. P. Dave and N. K. Jha, "CASPER: Concurrent hardware-software co-synthesis of hard real-time aperiodic and periodic specifications of distributed embedded systems," in *Proc. Design Automation & Test in Europe Conf.*, pp. 118-124, Feb. 1998.

[8] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. VLSI Systems*, vol. 4, no. 1, pp. 42-55, Mar. 1996.

[9] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable-voltage core-based systems," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, pp. 1702-1714, Dec. 1999.

[10] C.-H. Hwang and A. C. Wu, "A predictive system shutdown method for energy saving of event-driven computation," in *Proc. Int. Conf. Computer-Aided Design*, pp. 28-32, Nov. 1997.

[11] L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 6, pp. 813-833, June 1999.

[12] E. Y. Chung, L. Benini, and G. De Micheli, "Dynamic power management using adaptive learning tree," in *Proc. Int. Conf. Computer-Aided Design*, pp. 274-279, Nov. 1999.

[13] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," in *Proc. Design Automation Conf.*, pp. 555-561, June 1999.

[14] W. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. USENIX Symp. Operating Systems Design & Implementation*, pp. 13-23, Nov. 1994.

[15] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Design Automation Conf.*, pp. 134-139, June 1999.

[16] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics and Design*, pp. 76-81, Aug. 1998.

[17] T. Burd and R. Brodersen, "Processor design for portable systems," *J. VLSI Signal Processing*, vol. 13, pp. 203-222, Aug. 1996.

[18] R. P. Dick and N. K. Jha, "MOCSYN: Multiobjective core-based single-chip system synthesis," in *Proc. Design Automation & Test in Europe Conf.*, pp. 263-270, Mar. 1999.

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

[20] http://www.arm.com/Pro+Peripherals/

[21] http://www.transmeta.com/

[22]http://www.chips.ibm.com:80/products/powerpc/chips/

[23] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic Tasks in statically scheduled systems," in *Proc. Real-time Systems Symp.*, pp. 152-161, Dec. 1995.

[24] E. Y. Chung, L. Benini, A. Bogliolo, and G. De Micheli, "Dynamic power management for non-stationary service requests," in *Proc. Design Automation & Test in Europe Conf.*, pp. 77-81, Mar. 1999.

[25] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Computers*, vol. 44, no. 1, pp. 73-91, Jan. 1995.