Static and Dynamic Variable Voltage Scheduling Algorithms for Real-Time Heterogeneous Distributed Embedded Systems

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

Abstract

This paper addresses the problem of static and dynamic variable voltage scheduling of multi-rate periodic task graphs (i.e., tasks with precedence relationships) and aperiodic tasks in heterogeneous distributed real-time embedded systems. Such an embedded system may contain general-purpose processors, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). Variable voltage scheduling is performed only on general-purpose processors. The static scheduling algorithm constructs a variable voltage schedule via heuristics based on critical path analysis and task execution order refinement. The algorithm redistributes the slack in the initial schedule and refines task execution order in an efficient manner. The variable voltage schedule guarantees all the hard deadlines and precedence relationships of periodic tasks. The dynamic scheduling algorithm is also based on an initially valid static schedule. The objective of the on-line scheduling algorithm is to provide best-effort service to soft aperiodic tasks, as well as to reduce the system power consumption by determining clock frequencies (and correspondingly supply voltages) for different tasks at run-time, while still guaranteeing the deadlines and precedence relationships of hard real-time periodic tasks.

1. Introduction

This paper addresses the problem of static and dynamic variable voltage scheduling of hard and soft real-time tasks in heterogeneous distributed real-time embedded systems [1], in which processing elements (PEs) can be general-purpose processors, FPGAs or ASICs. The embedded system may have both periodic tasks with hard deadlines and precedence relationships and aperiodic tasks with soft deadlines. The goal of our scheduling algorithms is to provide good response times for soft aperiodic tasks and reduce the power consumption of the system, under the constraints that the deadlines of hard real-time tasks and their precedence relationships are guaranteed. It is well-known that variable voltage scaling, which refers to varying the speed of a processor by changing the clock frequency along with the supply voltage, has a high potential for reducing both energy and power consumption. Hence, in this paper, we focus on developing a power-efficient variable voltage scheduling algorithm for heterogeneous distributed embedded systems.

There have been extensive studies in the literature on scheduling of periodic tasks, aperiodic tasks, and their combinations. The algorithm given in [3] uses slack stealing, which serves aperiodic tasks by stealing all the processing time it can from the periodic tasks. The method in [2] studies 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. Some other work addressing joint scheduling of hard periodic tasks and soft aperiodic tasks can be found in [10, 11].

There is some work addressing variable voltage scheduling as well. The work in [12] gives an off-line algorithm, which generates a minimum-energy preemptive schedule for a set of independent tasks. The work in [14] provides a heuristic for a similar problem as in [12] for fixed-priority static scheduling. The work in [5] proposes a heuristic scheduling algorithm for non-preemptive scheduling of a set of independent tasks with arbitrary arrival times and deadlines on a variable voltage processor, which is an NP-complete problem. The work in [13] uses an energy priority heuristic for non-preemptive scheduling. The work in [6] presents a power-conscious fixed-priority scheduling algorithm. Other works can be found in [7, 8, 18, 19]. All the above approaches target only a single processor and are applicable to only independent tasks.

In this paper, first, we address the issue of variable voltage static scheduling in a heterogeneous distributed embedded system for a set of periodic tasks with precedence relationships and hard deadlines. A valid power-efficient variable voltage schedule is constructed using heuristics based on critical path analysis and task execution order refinement. Second, we address the issue of variable voltage joint scheduling of hard periodic tasks with precedence relationships along with soft aperiodic tasks. We take a combined static and dynamic approach. The static scheduling algorithm discussed above is used to construct a valid schedule for periodic tasks with precedence relationships. The static schedule is only partially fixed such that the on-line scheduler can schedule soft aperiodic tasks with best effort. The on-line scheduler also dynamically determines the speed-reduction ratios for scheduled events whenever there are no soft aperiodic tasks pending.

The new contributions of our approach are as follows. (1) Although a lot of previous work has been done to optimize power consumption through variable voltage scheduling of independent real-time tasks, there is only very limited work addressing variable voltage scheduling for distributed real-time embedded systems, in which precedence relationships exist among tasks [15,16]. In this paper, we develop an efficient heuristic for this problem motivated by the fact that the processor power consumption is normally a convex function of the clock period. Our algorithm is optimal if the problem is reduced to non-preemptive static scheduling with fixed-priority assignment on a single processor. (2) For the on-line variable voltage scheduling algorithm, we develop a unified framework, which incorporates slack stealing and resource reclaiming to provide best effort service to soft aperiodic tasks. It performs run-time analysis of processor clock speeds and voltages assigned to statically scheduled periodic tasks, by considering dynamic execution time variations.

2. Energy Consumption Model

This section discusses the relationship between the total energy consumption for a set of hard real-time tasks with precedence relationships implemented on multiple processors

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-00-C-L516 .

and their processor execution speeds. Similar discussions can be found in [12] for independent tasks on a single processor.

The periodic tasks are specified 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. An arrival time (deadline), the time by which the task associated with the node can begin (must complete) its execution, exists for every source (sink) node. Deadlines may exist for some intermediate nodes as well. Fig. 1 shows two task graphs, where for simplicity both are assumed to have the same period.

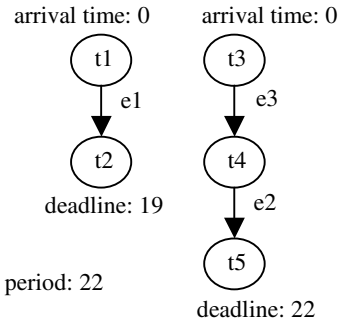


Fig. 1: Task graphs

The processor clock frequency, *f*, can be expressed in terms of the supply voltage, V_{dd}, and threshold voltage, V_t, as follows (*k* is a constant):

$$f = k(V_{dd} - V_t)^2 / V_{dd} \quad (1)$$

From (1), we can derive V_{dd} as a function of *f*, $F(f)$,

$$V_{dd} = F(f) = (V_t + \frac{f}{2k}) + \sqrt{(V_t + \frac{f}{2k})^2 - V_t^2} \quad (2)$$

The processor power, *p*, can be expressed in terms of the frequency, *f*, switched capacitance, *N*, and the supply voltage, V_{dd}, as:

$$p = \frac{1}{2} fNV_{dd}^2 = \frac{1}{2} fNF(f)^2 \quad (3)$$

which can be proved to be a convex function of *f*. The techniques presented in this paper are still valid even if Equation (1) is not accurate enough, as long as *p* is still a convex function of *f*.

Given the number of clock cycles, η_i, for executing task *i*, its energy consumption, E_i, under supply voltage V_i and clock frequency, f_i, is given by

$$E_i = (\eta_i / f_i) * p(f_i) \quad (4)$$

On any PE or link, all the tasks or communication events should be executed in non-overlapping intervals. Assume task *i* starts at $start_i$ and ends at $finish_i$, and its execution intervals are $[a_{i1}, b_{i1}]$, $[a_{i2}, b_{i2}]$, …, $[a_{ik}, b_{ik}]$, where

$$start_i = a_{i1} \le b_{i1} \le a_{i2} \le b_{i2} \ldots \le a_{ik} \le b_{ik} = finish_i .$$

Assume for inter-PE communication edge *j*, the execution time is $[start_j, finish_j]$. Based on the traditional assumption in distributed computing, the execution of intra-PE communication is assumed to take zero time. The total energy consumption for a set of tasks on different processors is:

$$Energy = \sum_{\forall processor\ j} \sum_{\forall i\ on\ processor\ j} \sum_{k \in (i1\ to\ ik)} (b_k - a_k) * p(f_k) \quad (5)$$

under the following constraints:

$$start_i \ge \max(arrival_i, \max_{j \in predecessors(i)} finish_j) \quad (6)$$

$$finish_i \le \min(deadline_i, \min_{j \in successors(i)} start_j) \quad (7)$$

for task or communication edge *i*.
Also,

$$\sum_{k \in (i1\ to\ ik)} (b_k - a_k) * f_k = \eta_i \quad (8)$$

for task *i*.

In the above equations, *predecessors*(*i*) (*successors*(*i*)) refers to all the predecessors (successors) of task or communication edge *i* in the task graphs, and $arrival_i$ ($deadline_i$) is its arrival time (deadline) specified in the task graphs. If unspecified, $arrival_i = \max_{j \in predecessors(i)} (arrival_j)$, and $deadline_i = \min_{j \in successors(i)} (deadline_j)$.

The objective of variable voltage scheduling is to assign different clock frequencies f_k and supply voltages V_k to different execution intervals $[a_k, b_k]$ on the processors which are voltage scalable, in order to reduce the system power consumption.

3. Static Variable Voltage Scheduling for Multi-rate Periodic Task Graphs

This section presents a variable voltage static scheduling algorithm for multi-rate periodic tasks in an embedded system consisting of a network of multiple heterogeneous PEs connected by communication links. An embedded system is a multi-rate system if it contains multiple task graphs with different periods. Given an embedded system specification, a hardware-software co-synthesis system [1] determines the number and type of PEs/communication links (i.e., allocation), and the assignment/scheduling of tasks/communications on different PEs/links.

Allocation/assignment and scheduling are each NP-complete for distributed systems [1]. To reduce the problem complexity, assume we start with a valid PE/link allocation and task/communication assignment, as well as a valid static schedule under maximum supply voltage and processor frequency $f_{\max}$. In this paper, the static schedule is generated based on a list-scheduling algorithm using the inverse of slack time as the task priority [20]. The static schedule consists of a set of scheduled events, which can be a task, a communication event, or a preemption event. First, we discuss the critical path analysis algorithm, which redistributes the slack time in the initial valid schedule. Second, we discuss a task execution order refinement algorithm, which refines the execution order imposed by the slack-based priority assignment in the initial schedule. We want to construct a new valid variable voltage schedule in which a processor's clock frequency can be varied along with the supply voltage for different time intervals. The new schedule still guarantees all the hard deadlines and precedence relationships. 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, which is the least common multiple of all the task graph periods in a multi-rate system specification [9]. Hence, the validity of the schedule can be determined along one hyperperiod.

3.1 The critical path analysis algorithm

In this section, we present the critical path analysis algorithm for variable voltage scheduling based on an initially valid schedule. In the generated variable voltage schedule, all

the scheduled events on a PE or a link maintain the same execution order as in the initial schedule (the execution order is modified later).

We first create a directed graph $G(V, E)$, where V is the set of vertices, containing all the scheduled events in the initial schedule, and E is the set of directed edges between vertices. An edge is inserted from one event to another if one is a direct predecessor of another in the task graphs, or if one is scheduled just ahead of another on the same PE or link. Therefore, these edges can represent all the precedence relationships in the original task graphs as well as execution ordering information in the initial schedule. Every event i can be associated with a start constraint r_i or a finish constraint d_i, which is initialized as $arrival_i$ or $deadline_i$, respectively, as defined in Section 2. Each node is associated with a weight, which equals its worst-case execution time. The creation of $G(V, E)$ can be illustrated through Example 1.

Example 1: Consider the embedded system specification given in the form of two task graphs in Fig. 1. Fig. 2 shows the corresponding directed $G(V, E)$ derived for the feasible schedule shown in Fig. 3. The distributed system consists of two PEs, PE1 and PE2, connected by a link. The schedule is based on the worst-case execution times of tasks and communication times, assuming a supply voltage of 3.3V. We assume both PE1 and PE2 have communication buffers. □

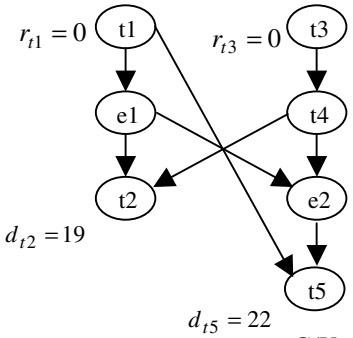


Fig. 2: Directed graph $G(V, E)$

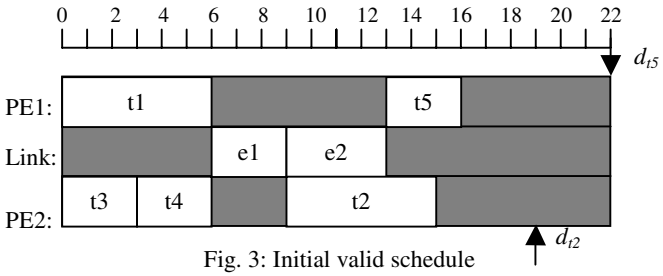


Fig. 3: Initial valid schedule

The variable voltage schedule is constructed by determining the processor speed reduction *ratio* for every scheduled event on processors, with *ratio* initialized to one. The algorithm is presented as Algorithm 1 in Fig. 4. The algorithm evaluates all the paths in graph $G(V, E)$ and locates the most critical one that minimizes the ratio of the total slack time on that path to the total worst-case execution time on that path. For a path j, assume that it begins with node *source* and ends with node *destination*. Then the scaling factor of path j, $scale_j$, is defined as

$$scale_j = (d_{destination} - r_{source}) / \sum_i wst\_exec_i \qquad (9)$$

where wst_exec_i is the worst-case execution time of scheduled event i on path j. The critical path is the one that minimizes the scaling factor.

The scaling factor of path j for tasks on processors, $task\_scale_j$, is defined as

$$task\_scale_j = ((d_{destination} - r_{source} - \sum_i wst\_exec_i) / \sum_{i\ on\ processor} wst\_exec_i) + 1 \qquad (10)$$

$task\_scale_j$ is the scaling factor by which the execution time of all the tasks scheduled on variable voltage processors can be extended without violating the start constraint and finish constraint of path j. $task\_scale_j$ is normally larger than $scale_j$.

For an event scheduled on a processor, we have $wst\_exec_i = \eta_i / (f_{\max} / ratio_i)$, where η_i is the worst-case execution cycle count of task i.

The critical path can be located in the following way. First, we locate the longest path, evaluated using the total worst-case execution times, between any *source* and *destination* pair. Then we pick the path that has the smallest scaling factor. Next, we multiply the speed reduction *ratio* for all the events scheduled on processors by a *multiplying ratio*, which can be set to be equal to $scale_j$ of critical path j. The speed reduction ratio for communication events and tasks implemented on FPGAs or ASICs remains unchanged (as they are assumed to be not variable voltage scalable). When $scale_j$ is below a *threshold*, we delete all the vertices on critical path j as well as their incoming and outgoing edges in $G(V, E)$, and update all the start and finish constraints of other vertices in a manner restricted by the execution length of the critical path, evaluated under new clock frequencies. In Algorithm 1, *threshold* is a value which is near or equal to 1.0 and is defined in a way so as to control the convergence rate with which the critical path approaches the state of being deleted, as well as to reduce the overhead of unnecessarily extending the execution time (with a ratio $\leq threshold$) of tasks allocated to hardware and communication events. The above process is repeated until $G(V, E)$ is empty. For an acyclic graph, the algorithm to locate the single-source longest paths has a complexity of $O(|V| + |E|)$, and the overall algorithm has a complexity of $O(k*|V|(|V| + |E|))$, where k is the number of times the above process is repeated until $G(V, E)$ is empty. k is normally much smaller than $|V|$. Since the speed reduction ratios need to be multiplied by *multiplying ratio* for events scheduled on processors only, we can store scaling factors for a set of critical paths

$$\{critical\_path_1, critical\_path_2, ..., critical\_path_n\},$$

where $critical\_path_i$ refers to the i-th most critical path. Let

$$multiplying\_ratio = \min(\min_{i=1\ to\ n} task\_scale_{critical\_path_i}, scale_{critical\_path_n}).$$

In the case of scheduling on a single processor, $G(V, E)$ is a set of vertices with an edge between any two adjacently scheduled events. We can compute the longest path length between any pair of vertices in the beginning. The *threshold* should be set to 1.0. The overall complexity is $O(|V|^2)$.

The rationale behind Algorithm 1 includes:

1. For each critical path, distributing the free slack time evenly is optimal due to the fact that power consumption is a convex function of processor speed, as discussed in Section 1.
2. For the overall algorithm, guaranteeing an equal speed reduction ratio on the most critical path first is a step in the right direction for reducing the power

consumption because it guarantees that all the other events can achieve a speed reduction ratio at least as high as on the most critical path. This is helpful in reducing the variance of the speed reduction ratios for various time intervals on other paths, therefore, reducing power consumption as well, again due to characteristics of the convex function.

```
Algorithm 1 : find_speed_reduction_ratio(G(V,E)){
  while |V| is not zero{
    (min_scale, critical_path, multiplying_ratio)
        = find_critical_path(G(V,E));
    for all i∈V and pe(i) is a variable voltage processor
      ratio_i = ratio_i * multiplying_ratio;
    (source, destination) = (critical_path_source,
                             critical_path_destination);
    if (min_scale ≤ threshold){
      release_time = r_source;
      for all i from source to destination on critical_path{
        d_i = release_time + wst_exec_i;
        release_time = d_i;
        for all n∈ direct successors of i and
              n∉ node on critical_path
          r_n = max(r_n, d_i);
        for all m∈ direct predecessors of i and
              m∉ node on critical_path
          d_m = min(d_m, r_i);
        for all e ∈ incoming or outgoing edges of i
          delete(e);
        delete(i);
      }
    }
  }
}
find_critical_path(G(V,E)){
  min_scale = infinity;
  ∀i, j∈V{
    l = longest_path(i, j);
    scale_l = (d_j − r_i) / Σ_{∀k on l} wst_exec_k;
    if (scale_l < min_scale){
      critical_path = l;
      min_scale = scale_l;
    }
  }
  multiplying_ratio = min_scale;
  return(min_scale, critical_path, multiplying_ratio);
}
```

Fig. 4: Algorithm for determining speed reduction ratios for scheduled events on variable voltage processors

The overall algorithm is not optimal, however, because the updated start and finish constraints of other events are dependent on how the slack times are distributed on the critical path, when the critical path is deleted. However, if we reduce the problem to a non-preemptive scheduling algorithm on a single processor with fixed-priority assignment, since the execution order of all the jobs can be determined, Algorithm 1, which resembles the approach in [12], can be shown to be optimal for minimizing the power consumption. For the sake of brevity, we omit the proof. For the general case as well, Algorithm 1 performs very well, as evidenced by experimental results later.

3.2 Task execution order refinement

The scheduling priority assignment based on the inverse of slack may be a good heuristic for constructing a valid schedule to guarantee deadlines [20], but it may not be very efficient for fully exploiting the slack time for variable voltage scheduling. Therefore, we refine the task execution ordering based on the new variable voltage schedule, in which the execution time of every scheduled event is multiplied by its corresponding speed reduction ratio. The algorithm is shown in Fig. 5. In Algorithm 2, $sched_j$ is the list of scheduled events on processor j in the order of their execution. Two adjacent events are *interchangeable* if interchanging their execution order will not violate any timing and precedence relationships in the new variable voltage schedule. In Algorithm 2, *interchange*(i, j) interchanges the order of i and j in the list and returns a value that points to the second event after interchanging.

```
Algorithm 2 : execution_order_refinement{
  ∀processor j{
    prev = begin node of sched_j;
    current = prev → next_node;
    if (interchangable(prev, current){
      current = interchange(prev, current);
    }
    prev = current;
    if (prev ≠ end node of sched_j)
      current = prev → next_node;
    else
      return;
  }
}
```

Fig. 5: Algorithm for task execution order refinement

The new task execution order generated in this way can still guarantee schedule validity and has the potential to achieve larger power reduction because more flexibility is introduced in the schedule. Then we can apply Algorithm 1 based on the new execution order. Task execution order refinement can be repeated until the power reduction ratio between two iterations is less than some pre-specified threshold.

3.3 Illustrative example

Example 2 is used to illustrate the critical path analysis algorithm.

Example 2: We compare three different schemes. In Scheme 1, we apply Algorithm 1 to the directed graph in Fig. 2, assuming *threshold* in Algorithm 1 is defined as one. In Fig. 2, the first critical path is (t1→e1→t2), with a scaling factor of 19/15 calculated by Equation (9), and a *task-scaling* factor of $((19-15)/12)+1=16/12$, calculated by Equation (10). The second critical path is (t1→e1→e2→t5), with a scaling factor of 22/16. We take the minimum of 16/12 and 22/16, which is 16/12. Then all task execution times in the schedule are extended by a ratio of 16/12, as shown in Fig. 6. In the next step, t1, e1 and t2 are deleted from $G(V, E)$, the *start constraint* of e2 and *finish constraint* of t4 are both updated to 11. Next, (t3→t4) is identified as the critical path, and all undeleted task execution times are extended by a ratio of 11/8, as shown in Fig. 7. Finally, the execution time of t5 is extended by a ratio of 7/5.5, as shown in Fig. 8.

In Scheme 2, instead of starting from the critical path, we first distribute the slack time equally on path (t3→t4→t2), which is the path with the second smallest *task-scaling* factor. The corresponding variable voltage schedule is shown in Fig. 9. In Scheme 3, we compute the optimal speed reduction ratios for this small example, assuming the power consumption is computed based on Equation (3) and that V_t is 0.8V. The power consumption during idle time on processors is assumed to be zero. The speed reduction ratios for different tasks and the overall power consumption on processors are compared in Table 1. The power number is normalized to the power consumption under maximum supply voltage (3.3V). The deviation of the power consumption of Scheme 1 from the optimal solution is

only 0.1%, while for scheme 2 it is 5.8%. For larger examples, as shown in Section 5, our scheme has a greater impact. □

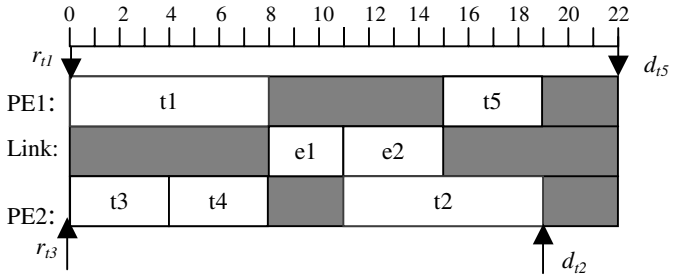


Fig. 6: Task execution times multiplied by a ratio of 16/12

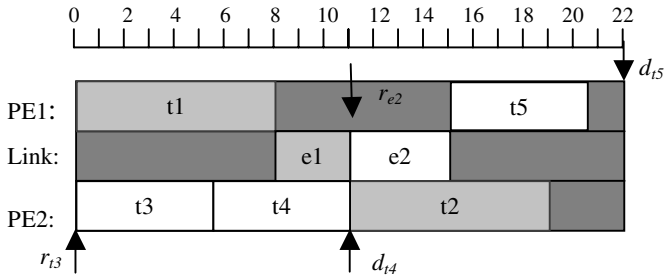


Fig. 7: Task execution times multiplied by a ratio of 11/8 (lightly shaded regions represent deleted events)

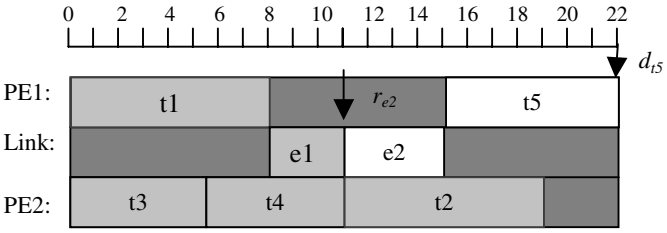


Fig. 8: Task execution times multiplied by a ratio of 7/5.5

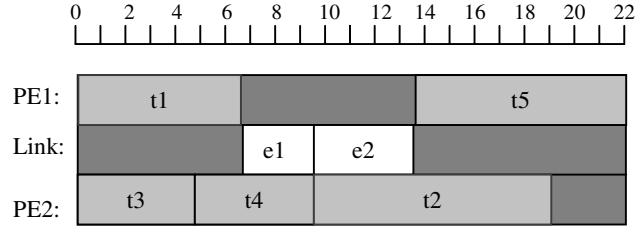


Fig. 9: Variable voltage schedule for Scheme 2

4. On-line Variable voltage Scheduling Algorithm

To maintain the simplicity and performance of the on-line scheduling algorithm, we use a combined static and dynamic approach. We start from an initially valid static schedule. The static schedule can be generated from critical path analysis and task execution order optimization techniques presented in the last section. This helps determine the task execution order and position the communication events such that the slack time is distributed more efficiently in a global manner for better variable voltage scheduling. However, processor clock frequencies as well as corresponding supply voltages are determined on-line in order to accommodate run-time execution time variations. The static schedule is partially fixed to guide the on-line scheduler in the following way:

(1) The schedule of all the communication events is kept fixed. This helps localize any dynamic decisions to each processor. Hence, no global re-scheduling is required at run-time. Also, the order of all the events scheduled on every processor and link is kept fixed. This helps maintain the precedence constraint among the tasks assigned to the same processor.

(2) A table generated off-line provides the *earliest start* and *latest finish time*s for each scheduled event. This guides the on-line scheduler in providing best-effort service to soft aperiodic tasks without violating the timing constraints of statically scheduled events.

(3) We also determine an off-line variable voltage schedule for each processor using Algorithms 1 and 2. In applying these algorithms, the vertices in $G(V, E)$ are all the scheduled events on a processor in one hyperperiod. The start constraint r_i or finish constraint d_i of a vertex is initialized as its *earliest start time* or its *latest finish time*, respectively. Recall that Algorithm 1 is optimal in the case of scheduling on a single processor with a fixed execution order of events. The off-line table also stores the *start time* and the speed reduction ratio of every scheduled event in the off-line variable voltage schedule.

The *earliest start time* is the earliest time by which an event can begin its execution without violating its arrival time and precedence relationships. The *latest finish time* is the latest time by which an event must complete its execution without violating the deadline and precedence relationships of itself as well as any other subsequent event scheduled on the same processor. They are computed in the following way. For the last event on a processor, we have

$$latest\_finish\_time_{last\_event} = \min(deadline_{last\_event}, hyperperiod) \quad (11)$$

For any other event i on a PE, the *latest finish time* is calculated by

$$latest\_finish\_time_i = \min(deadline_i, (latest\_finish\_time_{i\to next} - (\eta_{i\to next} / f_{\max})), \min_{j\in out\_edges(i)} start\_time_j) \quad (12)$$

where *out_edges(i)* refers to all the inter-PE out-going communication edges of *i*, *next* is the event scheduled just after *i* on the same processor, and η is the worst-case execution cycle count of an event.

For event *i* on a PE, the *earliest start time* is calculated by

$$earliest\_start\_time_i = \max(arrival_i, \max_{j\in in\_edges(i)} finish\_time_j) \quad (13)$$

where *in_edges(i)* refers to all the inter-PE in-coming communication edges of *i*.

Table 1: Comparison of power consumption of processors for different schemes

	Speed reduction ratio of statically scheduled tasks					Normalized power consumption
	t1	t2	t3	t4	t5	
Scheme 1	8/6	8/6	5.5/3	5.5/3	7/3	0.5856
Scheme 2	6.5/6	9.5/6	4.75/3	4.75/3	8.5/3	0.6190
Scheme 3	7.751/6	8.249/6	11.249/6	11.249/6	6.751/3	0.5849

4.1 On-line dispatching of soft aperiodic tasks

Soft aperiodic tasks are served in first-in-first-out (FIFO) order. In order to provide best effort service to aperiodic tasks, the on-line scheduler on each PE dispatches tasks and communication events in the following way. Whenever there are soft aperiodic tasks pending, and the *latest start time* (derived from the *latest finish time*) of the current statically scheduled event has not been reached, the scheduler dispatches the aperiodic task under maximum speed. If there are no aperiodic tasks pending, and the *earliest start time* of the current event has been reached, the scheduler dispatches the current event and determines the clock frequency and supply voltage for it at run-time. If the current event is running and an aperiodic task arrives, the scheduler detects whether or not servicing the aperiodic task will violate the *latest finish time* of the current event. If not, the current event gets preempted and the incoming aperiodic task gets dispatched under maximum speed.

```
Algorithm 3: find_online_ratio(current, end){
  for i from current to end{
      path_length = 0;
      for j from i to end{
          path_length_{i→j} = path_length + η_j / f_max;
          path_length = path_length_{i→j};
      }
  }
  while(1){
      min_ratio = infinity;
      for i from current to end{
          for j from i to end{
              ratio = (d_j − r_i) / path_length_{i→j};
              if (ratio < min_ratio){
                  min_ratio = ratio;
                  critical_path_source = i;
              }
          }
      }
      if (critical_path_source == current)
          return min_ratio;
      end = (critical_path_source) → prev;
      d_end = r_critical_path_source;
  }
}
```

Fig. 10: Determining the on-line speed reduction ratio for the current statically scheduled event

4.2 Variable voltage scheduling

Run-time variations can come from remaining execution times reclaimed on-line, or from servicing soft aperiodic tasks. This makes off-line analysis of the optimal speed reduction ratio for each statically scheduled event on every processor no longer valid. If there are no soft aperiodic tasks pending, the speed reduction ratio for the current statically scheduled event can be computed on-line by applying Algorithm 3 in Fig. 10 for a sequence of adjacent events (from *current* to *end*), including the current event and all subsequent events scheduled on the same processor in the hyperperiod. In Algorithm 3, the start constraint r_i or finish constraint d_i of an event i is initialized as its *earliest start time* or its *latest finish time*, respectively. For the current event being dispatched, its start constraint r is initialized as the current time. First, the critical path is located for the set of adjacent events from *current* to *end*. Then the *end* event is updated as the event scheduled just ahead of the source node of the critical path. The finish constraint d_{end} of event *end* is updated as well. The algorithm repeats until the source node of the critical path becomes the *current* event. The overall computation complexity of the algorithm is $O(m^2)$, where m is the number of events the algorithm needs to look at.

5. Experimental Results

This section presents experimental results to verify the efficacy of the proposed methods. The experiment is performed on five different embedded systems. Their characteristics are shown in Table 2. Task graphs in Tests 1 to 4 are generated using *TGFF* [4], which is a randomized task graph generator. The task graphs in Test 5 are based on a digital signal processing example taken from [17]. All the PEs in these systems are processors. However, our approach is general enough to be applied to embedded systems containing ASICs and FPGAs as well.

We first compare four different static scheduling algorithms in Fig. 11. A *lower bound* of power consumption with the given task/communication assignment is computed by evenly extending the execution length of every scheduled event on a processor to make utilization of each processor 1.0, under the assumption that no deadlines and precedence relationships exist. The *list-scheduling* algorithm just uses the initial valid schedule generated based on slack-based list scheduling. The execution length of every scheduled event is extended locally by reducing its voltage and clock frequency without violating its timing constraints. The *schedule-shifting* algorithm tries to evenly extend the execution length of every scheduled event on a processor by using global schedule slot shifting [15]. The *critical-path-analysis* algorithm uses Algorithm 1 to generate the speed reduction ratios for all the statically scheduled events. The *critical-path-analysis + execution-order-refinement* algorithm uses Algorithm 2 along with Algorithm 1 to refine the execution order of scheduled events. Since only processors are voltage scalable, all the different algorithms have the same power consumption on communication links. Hence, we only compare the power consumption on processors. The voltage can be varied from 3.3V to 1.4V. The power consumption is calculated based on Equation (2), where V_t equals 0.8V, and is normalized to the power consumption under maximum supply voltage. The *critical-path-analysis* algorithm reduces processor power consumption by an average (maximum) of 30% (44%) over the *list-scheduling* algorithm, and by an average (maximum) of 9% (14%) over the *schedule-shifting* algorithm. The *critical-path-analysis + execution-order-refinement* algorithm further reduces processor power consumption by an average (maximum) of 7% (15%) over the *critical-path-analysis* algorithm. In all the test cases, the average (minimum) deviation of the *critical-path-analysis + execution-order-refinement* algorithm from the lower bound power consumption is 11% (5%). The average (minimum) deviation of the *schedule-shifting* algorithm from the lower bound power consumption is 32% (9%). Note that the lower bound is a loose one since it is based on assumptions that are not valid (e.g., no precedence relationships and deadlines exist). The CPU times (933Mhz Pentium III with 258MB memory) for the five tests are 2.2s, 5s, 2s, 35s and 0.4s, respectively, for the *critical-path-analysis* algorithm, and 11s, 30s, 8s, 211s and 0.4s, respectively, for the *critical-path-analysis + execution-order-refinement* algorithm.

Table 2: Characteristics of different systems

Test	1	2	3	4	5
No. tasks	177	455	204	380	120
No. inter-PE edges	173	229	157	434	65
No. PEs/ No. links	3/3	4/4	3/3	5/4	3/2
Utilization factor for each processor	(0.48,0.57,0.56)	(0.40,0.41,0.69,0.78)	(0.31,0.51,0.57)	(0.42,0.44,0.45,0.64,0.62)	(0.81,0.8,0.8)

Table 3: Characteristics of soft aperiodic tasks

Test	1	2	3	4	5
Average arrival rate (No. arrivals/ms)	0.047	0.047	0.047	0.047	0.01
Average execution time (ms)	7	13	5	4	11.6
Total no. of soft tasks simulated	921	943	957	956	359

For on-line variable voltage scheme scheduling, in Fig. 12 we also compare four different scheduling algorithms, which are based on four different initial static schedules, optimized by scheduling algorithms *list-scheduling*, *schedule-shifting*, *critical-path-analysis* and *critical-path-analysis + execution-order-refinement*, respectively. All the four on-line algorithms re-compute the speed reduction ratio at run-time for the adjacent set of events, as discussed in Section 4.2. The aperiodic task arrivals are modeled as a Poisson process. The actual execution cycle counts of statically scheduled events are uniformly distributed in the range of 60% to 100% of their worst-case execution cycle counts. The characteristics of soft aperiodic tasks are shown in Table 3. Figs. 12(a) and (b) show the results for average response time of all soft aperiodic tasks and processor power consumption, respectively. The *critical-path-analysis + execution-order-refinement* algorithm performs the best in terms of both average power consumption and average response time for soft aperiodic tasks. It reduces processor power consumption (average response time of soft tasks) by an average of 28% (37%) over the *list-scheduling* algorithm, and by an average of 12% (13%) over the *schedule-shifting* algorithm. This indicates that the heuristics based on critical path analysis and task execution order refinement not only improve the efficiency of variable voltage scaling, but also improve the slack time distribution so that soft aperiodic tasks can get served more efficiently without violating the timing constraints of statically scheduled events.

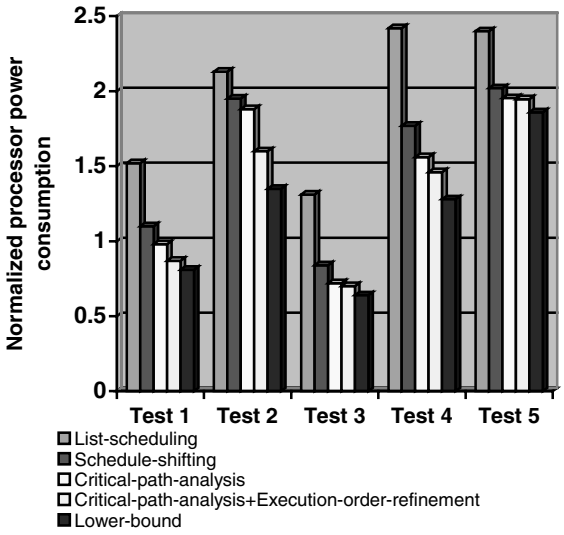
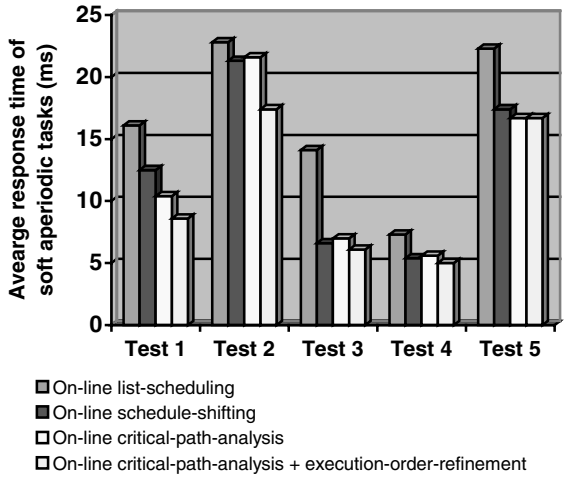
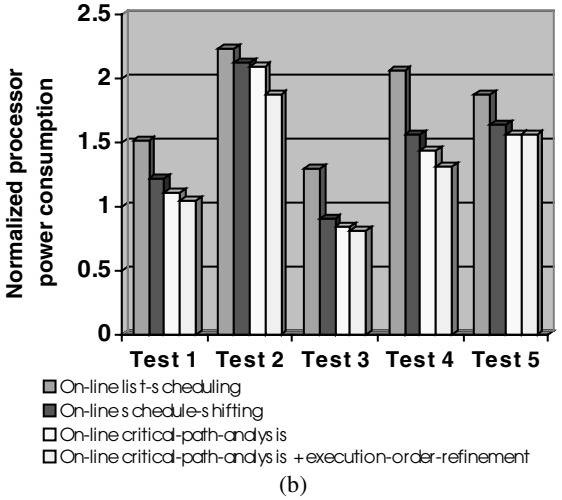


Fig. 11: Average processor power consumption for different static scheduling algorithms

Fig. 13 compares three different on-line schemes, all of which based on the static schedule generated by the *critical-path-analysis + execution-order-refinement* algorithm. For every statically scheduled event, on-line Scheme 1 (*latest-finish-time*) calculates the speed reduction ratio for the currently dispatched event by extending its execution length to its *latest finish time*. On-line Scheme 2 (*finish-time*) calculates the speed reduction ratio by extending its execution length to its *finish time* in the off-line variable voltage schedule on each processor. On-line Scheme 3 (*re-compute*) re-computes the scaling ratio at run-time as discussed in Algorithm 3. We can see that Schemes *latest-finish-time* and *finish-time* both achieve a very close power consumption reduction to Scheme *re-compute*, while sacrificing the response times of soft aperiodic tasks. Scheme *re-compute* achieves the best trade-off between processor power consumption and response times of soft aperiodic tasks.



(a)



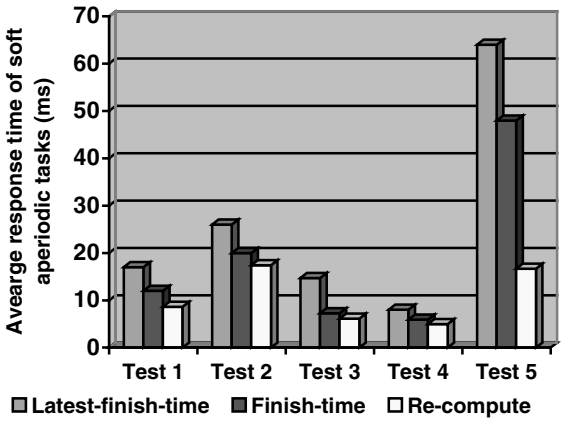
(b)

Fig. 12: Comparison of on-line algorithms based on four different static schedules

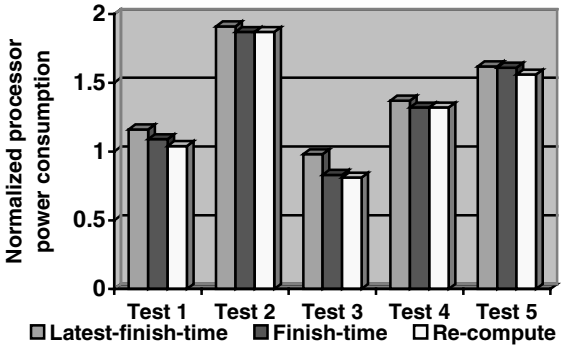
6. Conclusions

This paper discussed efficient static and on-line variable voltage scheduling algorithms for distributed real-time embedded systems. The static scheduling algorithm is based on

critical path analysis and task execution order refinement, motivated by the fact that power consumption is normally a convex function of processor clock frequency. The algorithm efficiently reduces power consumption of processors. The on-line algorithm utilizes a combined static and dynamic approach, with an objective of providing best-effort service to soft aperiodic tasks and reducing the system power consumption, under the constraint that hard deadline and precedence relationships of statically scheduled events are guaranteed. The static schedule optimized through critical path analysis and task execution order refinement, which the on-line algorithm is based on, also helps improve the slack distribution in the static schedule and results in a better service of soft aperiodic tasks. The on-line analysis of clock frequencies and supply voltages, incorporating both run-time variations and static hints, can achieve a better trade-off between average response time of soft aperiodic tasks and system power consumption.



(a)



(b)

Fig. 13: Comparison of three on-line schemes based on the same static schedule generated by the *critical-path-analysis+ execution-order-refinement* algorithm

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Manzak and C. Chakrabarti. "Variable voltage task scheduling algorithms for minimizing energy," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 2001.

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

[10] 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.

[11] S. Choi and A. K. Agrawala, "Scheduling aperiodic and sporadic tasks in hard real-time systems," Tech. Rep. CS-TR-3794, University of Maryland, College Park, Dept. of Computer Science, May 1997.

[12] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Symp. Foundations of Computer Science*, pp. 374-382, Oct. 1995.

[13] J. Pouwelse, K. Langendoen, and H. Sips, "Energy priority scheduling for variable voltage processors," in *Proc. Int. Symp. Low-Power Electronics and Design*, Aug. 2001.

[14] G. Quan and X. Hu, "Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors," in *Proc. Design Automation Conf.*, pp. 828-833, June 2001.

[15] J. Luo and N. K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 357-364, Nov. 2000.

[16] J. Luo and N. K. Jha, "Battery-aware static scheduling for distributed real-time embedded systems," in *Proc. Design Automation Conf.*, pp. 444-449, June 2001.

[17] C. M. Woodside and G. G. Monforton, "Fast allocation of processes in distributed and parallel systems," *IEEE Trans. Parallel & Distr. Syst.*, vol. 4, no. 2, pp. 164-174, Feb. 1993.

[18] C.M. Krishna and L.-H. Lee, "Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time Systems," in *Proc. Real Time Technology and Applications Symp.*, May 2000.

[19] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low Power Electronics and Design*, pp.197-202, Aug. 1998.

[20] 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.