Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems*

Dongkun Shin
School of Computer Science and Engineering
Seoul National University
sdk@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
jihong@davinci.snu.ac.kr

ABSTRACT

We propose a novel power-aware task scheduling algorithm for DVS-enabled real-time multiprocessor systems. Unlike the existing algorithms, the proposed DVS algorithm can handle conditional task graphs (CTGs) which model more complex precedence constraints. We first propose a condition-unaware task scheduling algorithm integrating the task ordering algorithm for CTGs and the task stretching algorithm for unconditional task graphs. We then describe a condition-aware task scheduling algorithm which assigns to each task the start time and the clock speed, taking account of the condition matching and task execution profiles. Experimental results show that the proposed condition-aware task scheduling algorithm can reduce the energy consumption by 50% on average over the non-DVS task scheduling algorithm.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Ap -plication-Based Systems—*Real-time and embedded systems*

General Terms

Design, Algorithms

Keywords

dynamic voltage scaling, conditional task graph, real-time systems, multiprocessor

1. INTRODUCTION

Energy consumption is a primary issue in designing battery operated systems. One of the most effective design techniques for low-power systems is dynamic voltage scaling (DVS), which adjusts processor's supply voltage and clock frequency according to the required performance. Many DVS algorithms have been proposed for uniprocessor-based real-time systems. For example, [4] evaluates the energy efficiency of state-of-the-art DVS algorithms for uniprocessor systems.

In this paper, we focus on DVS-enabled multiprocessor-based real-time systems where processing elements (PEs) can be a combination of general-purpose microprocessors, DSPs, FPGAs or ASICs. The general design flow for such multiprocessor systems is shown in Figure 1(a). Given a task graph with the design constraints (e.g., execution time and power consumption), we first assign each task to an appropriate PE (i.e., task assignment). Then, each task is scheduled for its execution within a PE (i.e., task scheduling). For DVS-enabled PEs, the task scheduling step should determine the execution speed (i.e., the voltage level) as well as the execution schedule. In this paper, we separate the task scheduling step into two sub-steps, task ordering and task stretching. The task ordering step decides the execution order of each task while the task stretching step determines the execution speed of each task[1].

Recently, several research groups had investigated the task stretching problem for DVS-enabled multiprocessor-based real-time systems [5, 6, 9]. For example, Luo *et al.* [5] proposed a heuristic algorithm for task stretching based on the critical path analysis. Schmitz *et al.* [6] presented an algorithm considering power variations of processing elements. Zhang *et al.* [9] formulated the task stretching problem as an Integer Linear Programming (ILP) problem, which can be solved by a fully polynomial time approximation scheme. While these proposed algorithms work well for many applications, they all assume that input task graphs are *unconditional*.

In this paper, we propose a task scheduling algorithm for *conditional* task graphs (CTGs) in *DVS-enabled* multiprocessor-based real-time systems[2]. A CTG G can model the conditional execution relationship between tasks based on whether a specific condition is satisfied or not. Figure 1(b) summarizes the current state-of-the-art for task scheduling in DVS-enabled multiprocessor systems. As shown in the figure, no existing work supports both task ordering and task stretching for CTGs in DVS-enabled multiprocessor systems. For *non-DVS* multiprocessor systems, however, the same problem had been previously investigated by Eles *et al.* [2] and Xie *et al.* [8].

We propose two task scheduling algorithms. The first one, called the condition-unaware algorithm, is largely based on the existing task ordering algorithm for conditional task graphs and task stretching algorithm for unconditional task graphs. Since the condition-unaware algorithm cannot fully take advantages of conditional executions in CTGs as well as their execution profiles, we propose an improved version of the condition-unaware algorithm. (We call this algorithm *condition-aware*.) Experimental results show that the proposed condition-aware task scheduling algorithm can reduce the energy consumption by 50% on average over the non-DVS task scheduling. Furthermore, the experiments show that the condition-aware algorithm reduces the energy consumption by 20% on average over the condition-unaware algorithm.

The rest of this paper is organized as follows. In Section 2, we describe the conditional task graph model in detail. The condition-unaware task scheduling algorithm for CTGs is presented in Section 3 while the condition-aware task scheduling algorithm for CTGs is described in Section 4. We present experimental results in Sec-

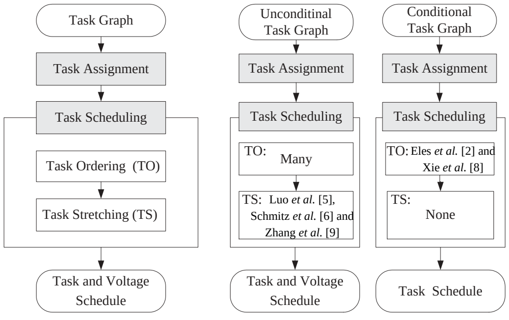
---

*This work was supported by grant No. R01-2001-00360 from the Korea Science & Engineering Foundation. The ICT at Seoul National University provides research facilities for this study.

ISLPED'03, August 25–27, 2003, Seoul, Korea.

[1] In the rest of the paper, we use the term *task scheduling* to include both task ordering and task stretching.

[2] The problem of task assignment is not discussed in this paper; we assume that tasks were already assigned to PEs. Obviously, different task assignments will change the efficiency of task scheduling. However, we leave an integrated approach to a future research topic because of its increased complexity.



(a) Design flow (b) Existing algorithms

Figure 1: System synthesis for DVS-enabled multiprocessor real-time systems.

tion 5. Section 6 concludes with a summary and directions for future works.

2. CONDITIONAL TASK GRAPH MODEL

We represent a periodic real-time application by a conditional task graph (CTG) $G =< V, E >$, which is a directed acyclic graph, where V is a set of tasks, E is a set of conditional directed edges between tasks. In a CTG, each directed edge $e = (\tau_i, \tau_j)$ represents that the task τ_i must complete its execution before the task τ_j can start its execution. Figure 2(a) shows an example CTG G along with its task mapping table. The task mapping table represents the result of the task assignment algorithm. Each entry of the mapping table includes the worst case execution time (WCET) and deadline of the corresponding task as well.

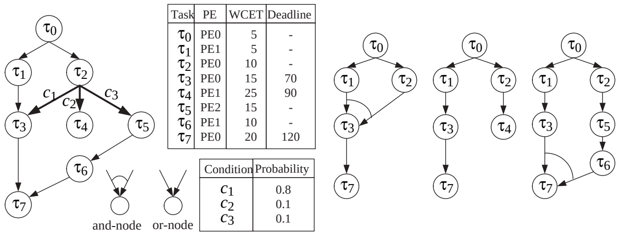
A conditional edge $e = (\tau_i, \tau_j) \in E$ is associated with a condition c and its activation probability $Prob(c)$. c and $Prob(c)$ have the following meaning: τ_i satisfies the condition c with the probability of $Prob(c)$. Activation probabilities for conditional edges can be obtained, for example, by profiling task executions. We denote the associated condition of an edge e as $C(e)$. In the given CTG representation, (unconditional) simple edges are the special case of conditional edges with their conditions and activation probabilities set by *true* and 1, respectively.

The head node of a conditional edge (which is not a simple edge) is called a branching node. A branching node satisfies only one condition out of several conditions associated with corresponding conditional edges. For example, in Figure 2(a), the task τ_2 is a branching node with three conditional edges. If the condition c_1 is true, τ_3 is executed after τ_2 is completed. The probability that c_1 becomes true is given by 0.8. Depending on the condition satisfied, the overall task execution is different. Figures 2(b)-(d) illustrate the corresponding subgraphs of G when the conditions c_1, c_2 and c_3 are satisfied, respectively.

Unlike the CTGs used in [2, 8], our CTG model is modified to represent more general control flows. The modified CTG model does not require control flows from a branching node to be merged at a single join node. For example, τ_4 does not join at the same node τ_7 where τ_3 and τ_5 join. Our CTGs also allow that a node can have multiple branching nodes as predecessor nodes. For nodes with multiple predecessor nodes, we define *and-node*s and *or-node*s. An and-node is activated when all its predecessor nodes are completed and the conditions of the corresponding edges are satisfied. On the other hand, an or-node is activated when one or more predecessors are completed and the conditions of the corresponding edges are satisfied. The edges to an and-node are tied with a round arch in a CTG as shown in Figure 2.

We denote by X_{τ_i} the necessary condition for τ_i to be activated in a CTG. (We call X_{τ_i} the guard of the task τ_i [2].) If a task τ_i has a set of predecessor nodes $Pred(\tau_i) = \{\tau_k | (\tau_k, \tau_i) \in E\}$, X_{τ_i} can be expressed by $\bigwedge_{\tau_k \in Pred(\tau_i)} (X_{\tau_k} \wedge C(\tau_k, \tau_i))$ if τ_i is an and-node, and by $\bigvee_{\tau_k \in Pred(\tau_i)} (X_{\tau_k} \wedge C(\tau_k, \tau_i))$ if τ_i is an or-node. We assume $X_{\tau_i} = \mathbf{1}$ if τ_i is a starting node (where $\mathbf{1}$ is a constant 'true' function).

We also denote by Ψ_{τ_i} the condition for τ_i to be executed at run time. In order to represent the execution precedence conditions modeled by edges, we define a boolean variable D_{τ_i} for each τ_i: D_{τ_i} becomes true when the execution of τ_i has completed. Using this notation, in Figure 2(a), Ψ_{τ_3} can be expressed by $(D_{\tau_1} \wedge D_{\tau_2}) \wedge (true \vee c_1)$.



Task	PE	WCET	Deadline
τ_0	PE0	5	-
τ_1	PE1	5	-
τ_2	PE0	10	-
τ_3	PE0	15	70
τ_4	PE1	25	90
τ_5	PE2	15	-
τ_6	PE1	10	-
τ_7	PE0	20	120

Condition	Probability
c_1	0.8
c_2	0.1
c_3	0.1

(a) Example conditional task graph G (b) g_{c_1} (c) g_{c_2} (d) g_{c_3}

Figure 2: Conditional task graph (CTG).

3. CONDITION-UNAWARE TASK SCHEDULING FOR CTGS

3.1 Condition-Unaware Task Scheduling Algorithm

We first propose a simple task scheduling algorithm for CTGs, essentially integrating the task ordering algorithm by Xie *et al.* [8] for conditional task graphs and the task stretching algorithm by Zhang *et al.* [9] for unconditional task graphs. We call this algorithm *condition-unaware* to emphasize that condition-dependent executions are not fully exploited in the algorithm.

To schedule each task of a given CTG, we first determine the execution order of the tasks which were allocated on the same PE using a task ordering algorithm. We use the task ordering algorithm by Xie *et al.* [8] for conditional task graphs, which considers the mutual exclusion relation. In ordering tasks on the same PE, τ_i and τ_j can share the same time slot if $X_{\tau_i} \wedge X_{\tau_j} = \mathbf{0}$ where $\mathbf{0}$ is a constant 'false' function[3]. For example, Figure 3(a) shows the task schedule for the conditional task graph G of Figure 2(a). We assume that tasks are not preemptive. τ_4 and τ_6 can overlap their execution schedules because they are mutually exclusive, i.e., $X_{\tau_4} \wedge X_{\tau_6} = \mathbf{0}$.

With the task schedule generated by the task ordering algorithm, we should determine the clock speed and the start time of each task by stretching the execution interval of the task. When the task stretching algorithm extends the execution interval of each task, it should satisfy the precedence constraints among tasks, because the execution interval of a task τ_i cannot be stretched beyond the start time of the task which has a precedence dependency with τ_i. The task schedule from Xie *et al.*'s algorithm, however, does not provide enough information on the task precedence constraints of the original CTG; it is not possible to extract the complete precedence dependencies from the start times of tasks only.

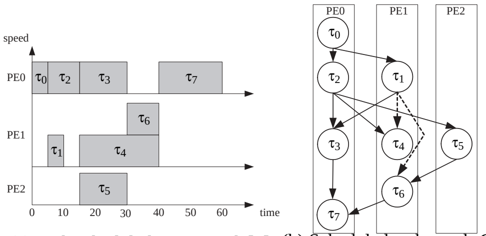
In order for Xie *et al.*'s algorithm to be used for task stretching, we added an extra step to Xie *et al.*'s algorithm. The extra step makes a scheduled task graph $G_S =< V, E \cup E_{PR} >$ which is a task graph modified from the original task graph $G =< V, E >$. The edge $(\tau_i, \tau_j) \in E_{PR}$, called as a precedence relation (PR) edge, indicates the precedence relation between τ_i and τ_j which are allocated on the same PE. For example, Figure 3(b) shows the scheduled task graph G_S. Since the tasks τ_1, τ_4 and τ_6 are assigned on PE1 and τ_1 is scheduled before τ_4 and τ_6, the PR edges (τ_1, τ_4) and (τ_1, τ_6) (represented by dashed lines) are added. Since the tasks τ_4 and τ_6 are mutually exclusive (i.e., there is no precedence dependency), there is no PR edge between them. Formally, the PR edge (τ_i, τ_j) is inserted when the following conditions are satisfied; (1) τ_i and τ_j are allocated on the same PE ($PE(\tau_i) = PE(\tau_j)$), (2) τ_i and τ_j are not mutually exclusive, (3) τ_i should be executed before τ_j, (4) there is no task τ_k which should be executed between τ_i and τ_j, and

---

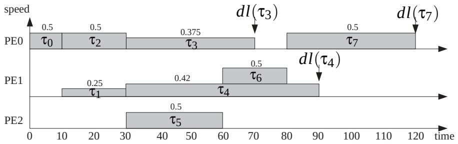
[3] Two tasks τ_i and τ_j on the same PE are said to be mutually exclusive if $X_{\tau_i} \wedge X_{\tau_j} = \mathbf{0}$. Otherwise, we call τ_i and τ_j non-exclusive tasks.

(5) there is no edge (τ_i, τ_j) in the original task graph G. (Conditions (4) and (5) remove redundant PR edges.)

With the task schedule generated from the modified task ordering algorithm, we can use the task stretching algorithm proposed for unconditional task graphs. Figure 3(c) shows the task schedule generated from the task stretching algorithm by Zhang *et al.* [9]. The relative execution speed of each task is specified over the corresponding task box. The schedule shown in Figure 3(c) consumes only 22% of the energy consumed by the task schedule in Figure 3(a).



(a) Task schedule by Xie *et al.* [8] (b) Scheduled task graph G_S



(c) DVS schedule S

Figure 3: Condition-unaware task scheduling of an example conditional task graph.

The task ordering algorithm by Xie *et al.*, which is based on the priority-based list scheduling, determines the start time $\sigma(\tau_i)$ of a task τ_i as the minimum value satisfying the following conditions; (1) $\sigma(\tau_i) \geq est(\tau_i)$ (where $est(\tau_i)$ is the earliest start time of τ_i) and (2) there is no other task τ_j such that $PE(\tau_i) = PE(\tau_j), X_{\tau_i} \wedge X_{\tau_j} \neq \mathbf{0}$ and $\delta(\tau_j) > \sigma(\tau_i)$ (where $\delta(\tau_j)$ is the end time of τ_j). For example, consider the tasks τ_1, τ_2, τ_3, τ_4 and τ_i (on the same PE) shown in Figure 4, where only τ_2 and τ_3 are mutually exclusive with τ_i. The original task ordering algorithm by Xie *et al.* determines $\sigma(\tau_i)$ as t_6. However, since τ_i will not be executed when either τ_2 or τ_3 is executed (and *vice versa*), τ_i can be scheduled earlier than t_6. We modified the original task ordering algorithm so that it finds the earliest time interval $[t_\alpha, t_\beta]$ for a task τ_i satisfying the following conditions; (1) $t_\alpha \geq est(\tau_i)$, (2) the WCET of τ_i is smaller than $t_\beta - t_\alpha$, and (3) there is no non-exclusive task τ_j whose execution interval is overlapping with the interval $[t_\alpha, t_\beta]$ (i.e., $t_\alpha < \delta(\tau_j)$ or $\sigma(\tau_j) < t_\beta$). The modified task ordering algorithm selects $[t_2, t_5]$ as the time interval for τ_i.

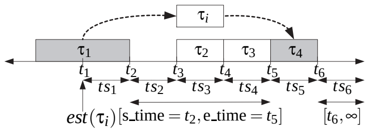


Figure 4: Task ordering for CTG.

Figure 5 summarizes the condition-unaware task scheduling algorithm. The function *Condition_Unaware_Task_Scheduling* calls the function *Find_AvailableTime* to compute the start time $\sigma(\tau_i)$ of a task τ_i which has the highest priority from the current ready list R. In this paper, we define the task priority as the task mobility which is computed as the difference between the latest start time and the earliest start time of a task[4].

To find the time slot [*s_time*, *e_time*] for τ_i, the function *Find_Available Time* examines time intervals on $PE(\tau_i)$ starting from $est(\tau_i)$. *s_time* and *e_time* are updated whenever each interval is examined. Initially, both are set to $est(\tau_i)$. When a slack interval is met, it changes *e_time* by the end time of the slack interval to include the slack interval into the time slot [*s_time*, *e_time*]. If the search step meets an interval $ts_{ex} = [t_k, t_{k+1}]$ which is assigned to a mutually exclusive task τ_j, it changes *e_time* by t_{k+1} to include the interval ts_{ex} into the time slot. However, when the search step encounters an interval $ts_{\widetilde{ex}} = [t_k, t_{k+1}]$ assigned to a non-exclusive task τ_j, it checks whether the task τ_i can be assigned to the current time slot [*s_time*, *e_time*]. If the current time slot [*s_time*, *e_time*] is too short for τ_i, it updates *s_time* and *e_time* by t_{k+1} to examine the following interval. Otherwise, the function *Find_AvailableTime* returns *s_time*.

For example, in Figure 4, the function *Find_AvailableTime* examines the time intervals from ts_1 to ts_6. When the search step meets the interval ts_1, both *s_time* and *e_time* are changed to t_2 from t_1. When the slack interval ts_2 is met, *e_time* is changed to t_3. Examining the intervals ts_3 and ts_4, *e_time* is changed to t_4 and t_5, respectively. When the interval ts_5 is met, the function stops and returns t_2 as the start time of τ_i.

In the function *Find_AvailableTime*, the maximal clock frequency f_{max} and the WCET $N_c(\tau_i)$ of the task τ_i are used in computing the execution interval of τ_i. The function *Mutex*(τ_i, τ_j) is used to check whether τ_i and τ_j are mutually exclusive.

To generate PR edges satisfying five conditions mentioned earlier, the function *Find_AvailableTime* computes $Pred_{PR}(\tau_i)$ and $Succ_{PR}(\tau_i)$ after the available time slot for τ_i is found (lines 11-26). $Pred_{PR}(\tau_i)$ is the set of tasks which should precede τ_i (i.e., $Pred_{PR}(\tau_i) = \{\tau_j | \ (\tau_j, \tau_i) \in E_{PR}\}$) while $Succ_{PR}(\tau_i)$ is the set of tasks which should follow τ_i (i.e., $Succ_{PR} = \{\tau_j | (\tau_i, \tau_j) \in E_{PR}\}$). For example, in Figure 4, since the task τ_i should be executed after the task τ_1 and before τ_4, the function *Find_Available Time* sets $Pred_{PR}(\tau_i)$ and $Succ_{PR}(\tau_i)$ to $\{\tau_1\}$ and $\{\tau_4\}$, respectively. This means that the task τ_i can be scheduled between the end time of τ_1 and the start time of τ_4. There is no need for τ_i to consider the schedule of τ_2 and τ_3 because they are mutually exclusive with τ_i. Using $Pred_{PR}(\tau_i)$ and $Succ_{PR}(\tau_i)$, the function *Condition_Unaware_Task_Scheduling* modifies the original task graph into the scheduled task graph (lines 12-13).

With the scheduled task graph G_S, we stretch tasks' time slots adjusting the voltage and clock speed. Zhang *et al.* [9] provides the formulation of task stretching problem. The task stretching problem is a constrained minimization problem which has an objective function and constraints. The objective function is the energy consumption while the constraints are the tasks' precedence dependencies and deadlines. The dependencies can be extracted from edges in the scheduled task graph G_S. We can formally define the task stretching problem as follows:

Task Stretching Problem

Given $G_S = < V, E \cup E_{PR} >, E_{\tau_i}(f(\tau_i)), N_c(\tau_i)$ and $dl(\tau_i)$,

find $\sigma(\tau_i)$ and $f(\tau_i)$ for each task τ_i such that

$$\sum_{\tau_i \in V} E_{\tau_i}(f(\tau_i)) \text{ is minimized}$$

subject to $\forall e = (\tau_i, \tau_j) \in E \cup E_{PR}, \ \ \sigma(\tau_i) + \dfrac{N_c(\tau_i)}{f(\tau_i)} \leq \sigma(\tau_j)$ and

$\forall \tau_i \in V$ with its deadline $dl(\tau_i), \sigma(\tau_i) + \dfrac{N_c(\tau_i)}{f(\tau_i)} \leq dl(\tau_i)$.

where $f(\tau_i)$, $N_c(\tau_i)$ and $dl(\tau_i)$ represent the clock frequency, the WCET and the deadline of the task τ_i, respectively.

The energy function $E_{\tau_i}(f(\tau_i))$, which represents the energy consumption during the execution of a task τ_i in the clock speed of $f(\tau_i)$, is given as follows:

$$E_{\tau_i}(f(\tau_i)) \quad = \quad C_L(\tau_i) \cdot N_c(\tau_i) \cdot S(f(\tau_i))^2 \qquad (1)$$

where $C_L(\tau_i)$ denotes the average load capacitance of the digital circuit in $PE(\tau_i)$. The function $S(f(\tau_i))$ indicates the supply voltage

---

[4] The proposed algorithms can work with other definitions of priority functions as well. That is, the correctness of the algorithms is orthogonal to the definition of a priority function.

```
Condition_Unaware_Task_Scheduling(CTG $G$)
for each task, calculate the priority of the task;
$R = R_0$; /* $R$ is the ready list and $R_0$ is the set of start nodes */
while ($R \neq \emptyset$) {
   select the task $\tau_i$ with the highest priority in $R$;
   $\sigma(\tau_i)$ = Find_AvailableTime($\tau_i$); /* $\sigma(\tau_i)$ is the start time of $\tau_i$ */
   $\delta(\tau_i) = \sigma(\tau_i) + N_c(\tau_i)/f_{max}$; /* $\delta(\tau_i)$ is the end time of $\tau_i$ */
   $R = R - \{\tau_i\}$;
   $D_{\tau_i}$ = true;
   for each task $\tau_j$, if ($\Psi_{\tau_j}$ == true) $R = R \cup \{\tau_j\}$;
}
$E = \emptyset$;
for each task $\tau_i$,
   $E = E \cup \{(\tau_i, \tau_j) | \tau_j \in Succ_{PR}(\tau_i)\} \cup \{(\tau_j, \tau_i) | \tau_j \in Pred_{PR}(\tau_i)\}$;
$G_S = G \cup E$;
Task_Stretching($G_S$);
```

```
Find_AvailableTime($\tau_i$)
s_time = e_time = est($\tau_i$); /* est($\tau_i$) is the earliest start time of $\tau_i$ */
interval = $N_c(\tau_i)/f_{max}$;
for each task $\tau_j$ scheduled in [s_time,$\infty$] of PE($\tau_i$) {
  /* with the increasing order for $\sigma(\tau_j)$ */
   if (Mutex($\tau_i, \tau_j$)==False) {
      e_time = $\sigma(\tau_j)$;
      if (e_time-s_time > interval) break;
      else s_time = e_time = $\delta(\tau_j)$;
   }
   else e_time = $\delta(\tau_j)$;
}
for each task $\tau_j$ scheduled in [0,s_time] of PE($\tau_i$)
   if (Mutex($\tau_i, \tau_j$)==False)
      if (there is no $\tau_k \in Succ_{PR}(\tau_j)$ scheduled in [0,s_time] and
                    Mutex($\tau_i, \tau_k$)==False)
            if (there is no edge ($\tau_i, \tau_j$) in $G$)
               {insert $\tau_j$ to $Pred_{PR}(\tau_i)$; insert $\tau_i$ to $Succ_{PR}(\tau_j)$; }
for each task $\tau_j$ scheduled in [e_time,$\infty$] of PE($\tau_i$)
   if (Mutex($\tau_i, \tau_j$)==False)
      if (there is no $\tau_k \in Pred_{PR}(\tau_j)$ scheduled in [e_time,$\infty$] and
                    Mutex($\tau_i, \tau_k$)==False)
            if (there is no edge ($\tau_j, \tau_i$) in $G$)
               {insert $\tau_j$ to $Succ_{PR}(\tau_i)$; insert $\tau_i$ to $Pred_{PR}(\tau_j)$; }
for each task $\tau_j \in Pred_{PR}(\tau_i)$,
   for each task $\tau_k \in Succ_{PR}(\tau_j)$,
      if ($\tau_k \in Succ_{PR}(\tau_i)$) remove $\tau_k$ from $Succ_{PR}(\tau_j)$;
for each task $\tau_j \in Succ_{PR}(\tau_i)$,
   for each task $\tau_k \in Pred_{PR}(\tau_j)$,
      if ($\tau_k \in Pred_{PR}(\tau_i)$) remove $\tau_k$ from $Pred_{PR}(\tau_j)$;
return s_time;
```

Figure 5: Condition-unaware task scheduling algorithm for CTGs.

$V_{dd}(\tau_i)$ of $PE(\tau_i)$ when the clock frequency is $f(\tau_i)$. This Non-Linear Program (NLP) formulation can be solved by a numerical technique such as the generalized reduced gradient method [3].

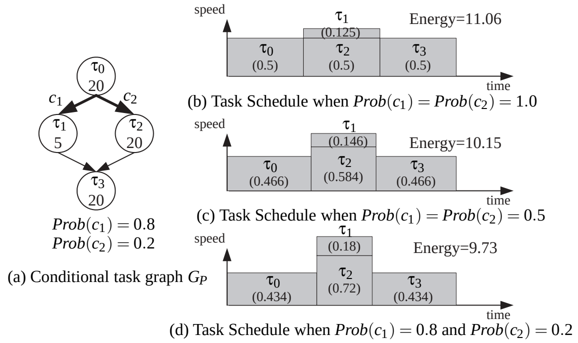
3.2 Problems of Condition-Unaware Approach

Although the condition-unaware task scheduling algorithm reduces the energy consumption over the non-DVS scheduling algorithm, it cannot effectively exploit the dynamic behavior of a CTG execution, limiting the energy efficiency level achieved. Consider the task schedule S shown in Figure 3(c) again. If the actual execution follows the subgraph g_{c3} shown in Figure 2(d), the task schedule S is very effective. However, if the actual execution follows g_{c1} instead, S is less efficient. In the subgraph g_{c1}, since the tasks τ_4, τ_5 and τ_6 are not executed, it is advantageous to start τ_7 earlier with a lower clock speed. However, in the condition-unaware algorithm, we cannot adapt the execution speeds depending on the conditions satisfied; The static task scheduling algorithm assigns a fixed start time and a fixed clock speed to each task. If we can assign different start times and clock speeds to each task depending on the conditions satisfied and select one of them at run time, more energy-efficient task schedules can be computed.

Another problem with the condition-unaware algorithm is that it cannot take advantages of the execution profiles of the given CTG when they are available. Though the task stretching minimizes the sum of energy values of all tasks, not all tasks in a CTG are executed with the same frequency during run time. For a typical program, about 80 percent of the program's execution occurs in only 20 percent of its code (which is called the hot paths). For a task scheduling algorithm to be energy-efficient, it should be energy-efficient when the hot paths are executed. If we assign more weight to the energy consumption of hot paths for task scheduling, the task schedule for the hot paths will be more energy-efficient. Therefore, when execution profiles are available, a DVS algorithm should take them into account.

Figure 6 illustrates the importance of the profile information for higher energy efficiency. For the CTG G_P in Figure 6(a), three task schedules are shown in Figure 6(b)-(d). G_P shows the WCET of each task in the corresponding node. For the conditions c_1 and c_2, $Prob(c_1)$ is larger than $Prob(c_2)$. If we assume the edges (τ_0, τ_1) and (τ_0, τ_2) are unconditional edges, the task schedule in Figure 6(b) is computed using the task stretching algorithm. However, if we assume the edges (τ_0, τ_1) and (τ_0, τ_2) are conditional edges having the same probability, the task schedule in Figure 6(c) is obtained. This task schedule is generated by multiplying the probability of the execution of each task to the energy function $E_{\tau_i}(f(\tau_i))$ in the task stretching problem. Though the clock speeds of τ_1 and τ_2 are higher than the clock speeds in Figure 6(b), the clock speeds of τ_0 and τ_3 are decreased. Since the task τ_1 is executed over τ_2 in most of cases, the energy consumption of the task schedule in Figure 6(c), which has a flatter schedule for τ_0, τ_1 and τ_3, is smaller than that of Figure 6(b).

Unlike Figures 6(b) and 6(c), if we use the available profile information of $Prob(c_1) = 0.8$ and $Prob(c_2) = 0.2$, a more energy-efficient task schedule can be computed as shown in Figure 6(d). Since the $Prob(c_2)$ is small, the influence of τ_2 is reduced and the time slots for the task τ_0 and τ_3 are increased. The task schedule in Figure 6(d) spends 12% and 4% less energy over that of Figures 6(b) and 6(c), respectively. (We call the task scheduling algorithm which utilizes the profile information in the task stretching as *profile-aware* algorithm.)



(a) Conditional task graph G_P

(b) Task Schedule when $Prob(c_1) = Prob(c_2) = 1.0$

(c) Task Schedule when $Prob(c_1) = Prob(c_2) = 0.5$

(d) Task Schedule when $Prob(c_1) = 0.8$ and $Prob(c_2) = 0.2$

Figure 6: Profile-aware task schedule.

4. CONDITION-AWARE TASK SCHEDULING FOR CTGS

4.1 Task Ordering Improvement

As discussed in Section 3.2, since the execution behavior of a CTG is different depending on the satisfied conditions, it is advantageous to have different start times and clock speeds for each task under all possible conditions using the schedule table technique proposed in [2]. Though the original schedule table determines only the start times of tasks, we extend it to have the clock speeds as well. Figure 7 shows an example of the schedule table. In the table, each row contains pairs of the start times and the clock speeds of the corresponding task under the different conditions. Each column in the table represents a condition expression. When the condition of an edge is satisfied during run time, the information is transferred to the run-time task scheduler. The task scheduler searches the schedule table with the condition satisfied and determines the start times and clock speeds of tasks. For example, in Figure 7, the task τ_7 starts at the time of 68.6 with the clock speed of 0.39 when the branching node satisfies the condition c_1 or c_2 while it starts at the time of 80 with the clock speed of 0.5 when the condition c_3 is true. The tasks τ_0, τ_1, and τ_2 have one start time and one clock speed in the column *true*, respectively, because they are the tasks executed initially. The tasks τ_4, τ_5, and τ_6 also have only one start time and one clock speed because they are activated by a single condition, respectively. As we can see from the task schedule table, the schedule for a task depends on the condition satisfied by the branching node. We call this task scheduling technique as *condition-aware* task scheduling.

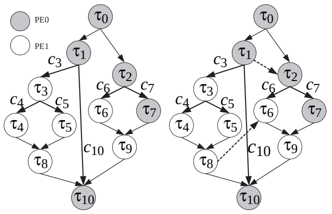
condition / task	$true$		c_1		c_2		c_3	
τ_0	0	0.5						
τ_1	10	0.25						
τ_2	10	0.5						
τ_3			30	0.39	30	0.39	30	0.38
τ_4					30	0.42		
τ_5							30	0.5
τ_6							60	0.5
τ_7			68.6	0.39	68.6	0.39	80	0.5

Figure 7: An example task schedule table.

For the condition-aware task scheduling, the task schedule table should be constructed. In particular, the appropriate columns should be decided for each task. To explain how to determine the appropriate columns for a task, we define a *minterm* of the conditional task graph G with branching nodes $\tau_1, \cdots, \tau_n$ as an expression $\bigwedge_1^n c_i$ where $c_i \in \Gamma(\tau_i)$. $\Gamma(\tau_i)$ for a branching node τ_i is the set of conditions which can be satisfied by τ_i (i.e., $\{C(e)|e = (\tau_i, \tau_j) \in G\}$). If a conditional task graph has branching nodes $\tau_1, \cdots, \tau_n$, there are $\prod_1^n |\Gamma(\tau_i)|$ number of minterms. A minterm of G represents one possible condition combination of G. By representing X_{τ_i} as the disjunction of minterms, we can enumerate all possible cases for τ_i to be executed. For example, in Figure 8(a), the task graph G has branching nodes τ_1, τ_2 and τ_3, where $\Gamma(\tau_1) = \{c_3, c_{10}\}$, $\Gamma(\tau_2) = \{c_6, c_7\}$, and $\Gamma(\tau_3) = \{c_4, c_5\}$. G has eight minterms and X_{τ_5} can be represented as $(c_3 \wedge c_5 \wedge c_6) \vee (c_3 \wedge c_5 \wedge c_7)$. This means there are two cases, when $(c_3 \wedge c_5 \wedge c_6) = true$ and when $(c_3 \wedge c_5 \wedge c_7) = true$, for the task τ_5 to be executed. For each case, we can make a column in the task schedule table and assign a different clock speed and start time for τ_5.

However, some conditions in a minterm cannot be determined before the start time of tasks. For example, if τ_2 is executed after τ_5, we cannot determine which value to use for τ_5 between the value in the column headed by $(c_3 \wedge c_5 \wedge c_6)$ and the value in the column headed by $(c_3 \wedge c_5 \wedge c_7)$ because we cannot know which condition is satisfied between c_6 and c_7 at the start time of τ_5. Therefore, we should represent each X_{τ_i} only with the conditions which are determined before the execution of the task τ_i. Such conditions are the elements of $\Gamma(\tau_k)$ where τ_k is a branching node executed before τ_i. We can decide that a branching node τ_k is executed before τ_i if there is a path from τ_k to τ_i in G. We denote the set of such branching nodes as B_{τ_i}. We define the *available minterm* of the task τ_i as an expression $\bigwedge_1^n c_k$ where $c_k \in \Gamma(\tau_k)$ and $\tau_k \in B_{\tau_i}$. By presenting X_{τ_i} as the disjunction of available minterms of τ_i, it is possible to enumerate all the cases before τ_i is executed. As we change the conditional task graph G by inserting PR edges at the task ordering step, the number of branching nodes in B_{τ_i} increases because the inserted edges can make a path between a new branching node and τ_i. For example, in the scheduled task graph G_S in Figure 8(b), which has PR edges (τ_1, τ_2) and (τ_8, τ_6), $B_{\tau_9} = \{\tau_1, \tau_2, \tau_3\}$ though $B_{\tau_9} = \{\tau_2\}$ in G.

Figure 9 describes the condition-aware task scheduling algorithm *Condition_Aware_Task_Scheduling*. The function *Condition_Aware_Task_Scheduling* selects the task τ_i which has the highest priority in the ready list R and transforms X_{τ_i} as the form of $\bigvee M_j^{\tau_i}$, where $M_j^{\tau_i}$ is an available minterm of τ_i. For each $M_j^{\tau_i}$, it adds a new task $\tau_{i,j}$ to G, where $\tau_{i,j}$ indicates the task τ_i executed when $M_j^{\tau_i}$ is true. In addition, the corresponding edges are inserted to G.



(a) Conditional task graph G (b) Scheduled task graph G_S

Figure 8: An example conditional task graph.

The corresponding edges can be generated by copying the edges in E=$\{(\tau_p, \tau_i)|(\tau_p, \tau_i) \in G$ and $M_j^{\tau_i} \Rightarrow X_{\tau_p} \wedge C(\tau_p, \tau_i)\} \cup \{(\tau_i, \tau_s)|(\tau_i, \tau_s) \in G\}$, where (τ_p, τ_i) is an in-edge activated when $M_j^{\tau_i}$ is true and (τ_i, τ_s) is an out-edge of τ_i. Depending on $M_j^{\tau_i}$, the function *Find_AvailableTime* returns different start times for $\tau_{i,j}$. After each $\sigma(\tau_{i,j})$ is determined, τ_i is removed from G and the PR edges are inserted to G.

With the modified task graph G after all tasks are scheduled, each $M_j^{\tau_i}$, should be re-examined because B_{τ_i} can be changed due to the inserted PR edges. If there is a task $\tau_{i,j}$ in G, where $M_j^{\tau_i}$ is not an available minterm of τ_i in G, we transform it into the disjunction of available minterms. If there is no such task, we terminate the task ordering step and move to the task stretching step. For example, in Figure 8, though X_{τ_7} can be represented as an available minterm c_7 in G, it should be represented as $(c_3 \wedge c_7) \vee (c_{10} \wedge c_7)$ after each task is scheduled at the first iteration of the loop (from line 4 to line 21) in the function *Condition_Aware_Task_Scheduling*.

```
Condition_Aware_Task_Scheduling(CTG $G$)
 1: for each task, calculate the priority of the task;
 2: for each task, transform $X_{\tau_i}$ to $\bigvee M_j^{\tau_i}$; /* $M_j^{\tau_i}$ is an available minterm of $\tau_i$ */
 3: $R = R_0$; /* $R$ is the ready list and $R_0$ is the set of start nodes */
 4: do {
 5:     stop = true;
 6:     while ($R \neq \emptyset$) {
 7:         select the task $\tau_i$ with the highest priority in $R$;
 8:         for each $M_j^{\tau_i}$,
 9:         { $G = G \cup \{\tau_{i,j}\}$;
10:             $\sigma(\tau_{i,j})$ = Find_AvailableTime($\tau_{i,j}$);
11:             $\delta(\tau_{i,j}) = \sigma(\tau_{i,j}) + N_c(\tau_i)/f_{max}$;}
12:         $G = G - \{\tau_i\}$;
13:         $G = G \cup \{(\tau_i, \tau_j)|\tau_j \in Succ_{PR}(\tau_i)\} \cup \{(\tau_j, \tau_i)|\tau_j \in Pred_{PR}(\tau_i)\}$;
14:         $R = R - \{\tau_i\}$;
15:         $D_{\tau_i}$ = true;
16:         for each task $\tau_j$, if ($\Psi_{\tau_j}$ == true) $R = R \cup \{\tau_j\}$;
17:     }
18:     for each task $\tau_{i,j} \in G$,
19:         if ($M_j^{\tau_i}$ can be transformed to $\bigvee M_k$ in $G$)
20:             {insert all $\tau_{i,j}$ into $R$; stop = false;}
21: } while (!stop)
22: $G_S = G$;
23: Task_Stretching($G_S$);
```

Figure 9: Condition-aware task scheduling algorithm for CTGs.

4.2 Task Stretching Improvement

To use the profile information of a CTG, we modified the objective function of the task stretching problem discussed in Section 3 as follows using the probability $Prob(\tau_{i,k})$ of the execution of the task $\tau_{i,k}$:

$$\sum_{\tau_{i,k} \in V} E_{\tau_{i,k}}(f(\tau_{i,k})) \cdot Prob(\tau_{i,k}).$$

$Prob(\tau_{i,k})$ can be computed by $\prod_{j=1}^{n} Prob(c_j)$ when $M_k^{\tau_i} = \bigwedge_{j=1}^{n} c_j$. (This problem also can be solved using numerical techniques mentioned in Section 3.1.)

Figure 10 shows the task schedule of the conditional task graph in Figure 2(a) after the task stretching. When the condition c_2 or c_3 is *true*, we can reduce the energy consumption by executing $\tau_{3,1}$ and $\tau_{7,1}$, or $\tau_{3,2}$ and $\tau_{7,2}$, instead of $\tau_{3,3}$ and $\tau_{7,3}$. Since $Prob(c_1)$ is larger than $Prob(c_2)$ and $Prob(c_3)$, the task schedule is optimized for the subgraphs g_{c1} rather than g_{c2} and g_{c3} in Figures 2(b)-(d). τ_0, τ_1, τ_2, $\tau_{7,1}$ and $\tau_{7,2}$ are assigned lower clock speeds while τ_3, τ_4, τ_5, τ_6 and $\tau_{7,3}$ are assigned higher clock speeds compared with the schedule in Figure 3(c).

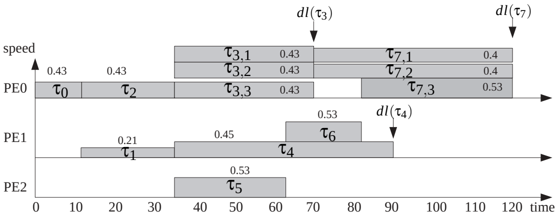


Figure 10: Condition-aware task schedule.

From the task schedule, we can see that it is not necessary to handle $\tau_{3,1}$ and $\tau_{3,2}$ (or $\tau_{7,1}$ and $\tau_{7,2}$) separately because they have the same start time and the same clock speed. Since they have same precedence dependencies (i.e., both $\tau_{3,1}$ and $\tau_{3,2}$ have two edges from τ_1 and τ_2, and both $\tau_{7,1}$ and $\tau_{7,2}$ have an edge from $\tau_{3,1}$ and $\tau_{3,2}$ in the scheduled task graph G_S), the same start time and the same clock speed are assigned to them. So, we can merge these tasks in G_S before the task stretching step because they would generate the same constraint for the task stretching problem.

5. EXPERIMENTS

We experimented the proposed task scheduling algorithms with a number of random conditional task graphs. We modified TGFF [1] to generate conditional task graphs. Using the modified TGFF program, we generated twelve CTGs, `ctg1`∼ `ctg12`. Each CTG is different in the number of nodes, the number of edges, the number of allocated PEs, and the number of branching nodes. The second column of Table 1 summarizes the characteristics of the 12 CTGs used for the experiments. For the task assignment step, we used the GA-based task assignment algorithm [7] to assign each task in a CTG to a PE.

We estimated the effectiveness of the condition-unaware task scheduling algorithm and the condition-aware task scheduling algorithm. For the condition-aware task scheduling algorithm, we experimented two versions, one without the profile information and the other with the profile information. Table 1 shows the experimental results for twelve CTGs. The third, fourth and fifth columns show the normalized energy consumption by task schedules generated from three task scheduling algorithms, the condition-unaware algorithm (denoted by CU), the condition-aware algorithm without the profile information (denoted by $\text{CA}_{no\_profile}$) and the condition-aware algorithm with the profile information (denoted by $\text{CA}_{profile}$). As a reference case, we used the energy consumption by task schedules generated from the task scheduling algorithm by Xie *et al.* [8] (such as Figure 3(a)), in which all tasks are executed at the full speed.

The condition-unaware algorithm reduced the energy consumption by 30% on average while the condition-aware algorithm without the profile information reduced the energy consumption by 45% on average. With the profile information, the condition-aware algorithm further reduced the energy consumption by 50% on average.

Table 1 also shows that the energy efficiency of the condition-aware algorithms varies significantly depending on the characteristic of a CTG. For example, the energy consumption in `ctg5` is reduced by 73%. On the other hand, the energy consumption in `ctg6` is reduced only by 32%. This large variation mainly depends on the location of branching nodes in CTGs. When the branching nodes are located near to the start node, the condition-aware task scheduling can generate a more energy-efficient schedule. Since the branching nodes are executed earlier, there are more opportunities for saving the energy consumption. The `ctg5` graph is such a case.

CTG	$a/b/c/d$*	task scheduling algorithms		
		CU	$\text{CA}_{no\_profile}$	$\text{CA}_{profile}$
ctg1	8/9/3/1	0.26	0.24	0.24
ctg2	26/43/2/4	0.64	0.46	0.44
ctg3	40/77/4/5	0.73	0.63	0.50
ctg4	40/77/3/5	0.86	0.62	0.47
ctg5	20/27/2/5	0.70	0.46	0.27
ctg6	16/21/2/5	0.80	0.72	0.68
ctg7	30/29/2/5	0.69	0.66	0.61
ctg8	40/63/3/5	0.59	0.39	0.34
ctg9	14/19/2/4	0.74	0.65	0.65
ctg10	19/25/2/5	0.62	0.51	0.49
ctg11	70/99/4/5	0.90	0.63	0.61
ctg12	49/92/3/4	0.87	0.66	0.66
average		0.70	0.55	0.50

* *a*= # of nodes, *b*= # of edges, *c*= # of PEs, and *d*= # of branches.

Table 1: Normalized energy consumption under three task scheduling algorithms.

6. CONCLUSIONS

We have presented the power-aware task scheduling techniques, which schedule the clock speeds and start times of tasks in the conditional task graph, for the DVS-enabled real-time multiprocessor systems. We first proposed the condition-unaware task scheduling by integrating the task ordering algorithm for conditional task graphs and the task stretching algorithm for unconditional task graphs. We also proposed the condition-aware task scheduling algorithm and the profile-aware task scheduling algorithm by considering the run-time behaviors and the profile information of conditional task graphs. Experimental results showed that the proposed technique can reduce the energy consumption by 50% on average.

The proposed algorithms can be further improved in several aspects. In this work, we assumed that the task assignment was given as a fixed input. However, for a higher energy efficiency, it will be necessary to investigate the task scheduling algorithm integrated with the task assignment algorithm. Furthermore, for a complete treatment of voltage scheduling in multiprocessor systems, we plan to develop on-line slack estimation and distribution techniques effective in DVS-enabled multiprocessor real-time systems.

7. REFERENCES

[1] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *Proc. of Workshop Hardware/Software Codesign*, pp. 97–101, 1998.

[2] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proc. of Design Automation, and Test in Europe*, pp. 23–26, 1998.

[3] G. A. Gabriele and K. M. Ragsdell. The generalized gradient method: a reliable tool for optimal design. *ASME Journal of Engineering and Industry, Series B*, Vol. 99, No. 2, pp. 394, 1977.

[4] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S. L. Min. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 219–228, 2002.

[5] J. Luo and N. K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proc. of Int. Conf. on VLSI Design*, pp. 719–726, 2002.

[6] M. T. Schmitz and B. M. Al-Hashimi. Considering power variations of DVS processing elements for energy minimisation in distributed systems. In *Proc. of Int. Symp. on System Synthesis*, pp. 250–255, 2001.

[7] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. In *Proc. of Design Automation, and Test in Europe*, pp. 514–521, 2002.

[8] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Proc. of Design, Automation, and Test in Europe,* pp. 620–625, 2001.

[9] Y. Zhang, X. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. of Design Automation Conference*, pp. 183–188, 2002.