1

Power Aware Mapping of Real-Time Tasks to Multiprocessors

1.1 Introduction

The performance of modern processors has increased at the expense of drastically increased power consumption. The increased power consumption not only reduces the operation time for battery powered embedded systems (such as PDAs and laptops) but also increases the sophistication/cost of the cooling infrastructures for dense clusters (such as web servers). Therefore, energy has been promoted to be a first-class resource in a system [45] and power aware computing has emerged to be a major research area, which aims at “using the right amount of energy in the right place at the right time” [23]. Various studies have been conducted to manage power consumption of different components in a system (such as CPU, memory, disk and network interfaces), interested readers are referred to [16, 20] for more comprehensive information. In this chapter, we focus on CPU power consumption when executing a set of real-time tasks on multiprocessor platforms.

As the simplest scheme, a system can finish the required computation as fast as possible and turn off the processors when the system is (and likely to stay) idle to save energy.
However, this shutdown technique is sub-optimal even with the perfect knowledge of idle intervals in a schedule. The reason comes from the convex relation between CPU speeds and its power consumption [7, 10], which implies that a better approach should uniformly scale down the processing speed during the computation across all processors [4]. Voltage scaling (VS), as one powerful and energy efficient technique, scales down the CPU speed and supply voltage simultaneously for lower performance requirements to get more energy savings [40, 43]. However, with scaled processing speeds, an application takes more time to complete. For real-time tasks, which normally have timing constraints and have to finish the execution before their deadlines, special cares are needed when scaling down their processing speed for energy savings. Based on the voltage scaling technique, various power aware scheduling schemes have been studies for different task models on uniprocessor systems [2, 21, 36]. However, much less work has been done for power awareness in multiprocessor systems.

Multiprocessor real-time scheduling is one of the most extensively studied areas. There are two major approaches for scheduling/mapping real-time tasks to processors: partition and global scheduling [9, 11]. In partition scheduling, each task is assigned to a specific processor and, each processor has its own scheduler and fetches tasks for execution from its own queue. In global scheduling, all tasks are put in a global queue and processors select from the queue the task with the highest priority for execution.

For real-time systems, to ensure that tasks can meet their deadlines in the worst case scenario, scheduling decisions are normally based the worst case execution time (WCET) of tasks. For different partitions or different priority assignments to tasks (therefore different orders of tasks in the global queue), the mappings of tasks to processors are different, which result in different schedule length (that is, the total time needed for executing the tasks is different). It is well-known that the optimal partition as well as optimal priority assignment in global scheduling for minimizing the schedule length are NP-hard in the strong sense [11]. Therefore, many heuristics (such as First Fit, Best Fit [6] and Longest Task First [35]) have been studied to efficiently obtain close-to-optimal schedules. Moreover, considering the run-time behaviors (which normally vary substantially [13]) of real-time tasks, optimal priority assignment in terms of schedule length does not necessarily lead to minimum energy consumption [47].

Despite the simplicity of exploiting the well-studied uniprocessor energy management schemes on individual processors, the fixed mapping of tasks to processors in partition scheduling limits the power management opportunity across different processors. Hence, the power aware mapping introduced in this chapter will base on global scheduling. Moreover, instead of designing new heuristics on priority assignment, we adopt Longest Task First (LTF) heuristic when assigning priorities to tasks. First, more slack may be expected from the execution of longer tasks and thus can be reclaimed by remaining tasks for more energy savings. Second, the ratio of the schedule length under LTF heuristic over the one under optimal priority assignment is bounded by a constant [35].

For the LTF priority assignment, if the resulting schedule completes all tasks well before the deadline in the worst case, static slack exists. Following the idea of static power management (SPM) for uniprocessors, we can uniformly stretch the schedule and scale down the processing speed to finish all tasks just-in-time and obtain energy savings [17]. However, for parallel systems, due to the possible dependencies among tasks, gaps exist in the schedule and result in different parallelism for different sections in a schedule. A better approach should take such parallelism into consideration when exploits the static slack for more energy savings [32].

In addition, dynamic slack can be expected online due to the run-time behaviors of real-time tasks, which normally only use a small fraction of their WCETs [13]. The efficient
power aware mapping should also take such slack into consideration and adjust the mapping of tasks to processors at run-time for better performance on energy savings.

In this chapter, we first introduce an efficient static scheme for parallel applications, especially the ones with dependent tasks. The scheme explores different degrees of parallelism in a schedule and generates an energy efficient canonical schedule, which is defined as the one corresponding to the case where all tasks take their worst case execution time. Then, based on global scheduling, we focus on the online schemes that exploit the run-time behaviors of tasks and dynamic slack for more energy savings. Specifically, for independent and dependent real-time tasks, the slack sharing power aware scheduling dynamically adjusts the mapping of tasks to processors at run-time. By sharing the dynamic slack across processors, the scheme intends to scale down the processors more uniformly with balanced actual workload and thus achieves better energy savings. For any given task priority assignment heuristic (e.g., LTF), if the canonical schedule can finish all tasks in time, the power aware mapping with slack sharing will meet tasks’ deadlines as well. For applications with multiple execution paths consisting of different real-time tasks, slack shifting further explores slack time from the execution paths other than the longest one. Speculation schemes that exploit statistical characteristics about applications are discussed with the intention to scale down all tasks more evenly for better energy savings. Practical issues related to power management are addressed in the end and the idea of slack reservation is introduced to incorporate CPU speed adjustment overhead.

1.1.1 Overview of Power Aware Scheduling for Parallel Systems

The research on power aware scheduling in real-time system can be traced back to the important seminar work of Weiser et al. [40] and Yao et al. [43], where the authors introduced the concept of voltage scaling for interval-based adjustment as well as task-based speed adjustment. For systems with processors running at different fixed speeds (and thus with different power profiles), several task assignment and scheduling schemes have been proposed to minimize system energy consumption while still meeting applications’ deadlines, where applications are usually represented by directed acyclic graphs (DAG) [18, 27, 42].

For systems with variable speed processors, static power management (SPM) can be accomplished by deciding beforehand the best speed for each processor given the fixed task sets and predictable execution times [17]. Gruian et al. proposed a priority based energy sensitive list scheduling heuristic to determine the amount of time allocated to each task, considering energy consumption and critical path timing requirement in the priority function [19]. In [46], Zhang et al. proposed a mapping heuristic for fixed task graphs to maximize the opportunities for voltage scaling algorithms, where the voltage scaling problem was formulated as an integer programming problem.

For independent periodic hard real-time tasks to be executed on identical multiprocessors, Aydin et al. address the problem of energy minimization for partition-based scheduling. The authors investigate the joint effects of partitioning heuristics on energy consumption and feasibility based on EDF scheduling [4] and rate-monotonic scheduling (RMS) [1], respectively. The system synthesis problem using global EDF is addressed by Baruah et al. [5], where utilization based approach is used to determine the speed for each processor. When considering heterogeneous multiprocessors, the problem of power aware resource allocation is formulated as an Generalized Assignment Problem (GAP) by Yu et al [44]. Based on integer linear programming (ILP), the authors propose a LR-heuristic based solution with earliest deadline first (EDF) scheduling.

For distributed systems where communication cost is significant and task migration is pro-
hibitive, Luo et al. proposed a static optimization algorithm by shifting the static schedule to re-distribute static slack according to the average slack ratio on each processor element for periodic task graphs and aperiodic tasks. The scheme reduces energy consumption and response time for soft aperiodic tasks at run time [29]. The authors improved the static optimization by using critical path analysis and task execution order refinement to get the maximal static slow down factor for each task [30], where a dynamic voltage scaling scheme is also proposed.

For web servers, where static power from other components may be significant, Elnozahy et al. presented cluster-wide power management and evaluated policies that combine voltage scaling and turning on/off individual server nodes [12]. The policies are further extended and evaluated by Xu et al. [41]. Sharma et al. investigated adaptive algorithms for dynamic voltage scaling in QoS-enabled web servers to minimize the energy consumption subject to service delay constraints [38]. For surveillance systems based on sensor network, Maniezzo et al. further explore the tradeoff between the energy consumption for computation and communication components [31].

1.2 Variable Speed Processors and Power Management

In this chapter, we consider a frame-based application to be executed on a shared memory multiprocessor system. The processors in the system are assumed to be homogeneous with identical performance-power characteristics and tasks in the application can be executed on any processor.

The power consumption of a processor when executing a task is dominated by its dynamic power dissipation $P_d$, which is quadratically related to the processor supply voltage and linearly related to its processing speed [7, 10]. With the knowledge of almost linear relation between processing speed and supply voltage, voltage scaling techniques reduce processor supply voltage for lower processing speeds [40, 43]. From now on, we refer to speed adjustment as changing both CPU supply voltage and frequency. Therefore, the dynamic power dissipation of a processor can be simply modeled as:

$$P_d = kf^3$$  \hspace{1cm} (1.1)

where $k$ is a system dependent constant and $f$ is the processor clock frequency (i.e., the processor speed). The maximum speed of processors is denoted by $f_{\text{max}}$, and, without loss of generality, the processing speeds are normalized with respect to $f_{\text{max}}$ (e.g., $f_{\text{max}} = 1.0$).

Note that energy is the integration of power over time. For a task with fixed amount of computation $c$, the energy consumption of executing the task at speed $f$ is $E(f) = P_d \cdot \frac{c}{f} = k \cdot c \cdot f^2$. That is, although scaling down the processing speed linearly increases the processing time (i.e., $\frac{1}{f}$) of the task, it saves power as well as energy. The lower the speed is, the more energy will be saved. However, with lower speeds, a task takes more time to finish. For real-time tasks, the extended execution may cause deadline misses and utmost care is needed to select the appropriate processing speed to meet the timing constraints and obtain the maximum energy savings.

For example, consider a task that, with the maximum speed $f_{\text{max}}$, needs 20 time units to complete its work. If we have 40 time units allocated to this task (e.g., the task has a deadline 40 time units away), we can reduce the processor speed and supply voltage by half while still finishing the task on time. The new power when executing the task would be: $P'_d = k \cdot (\frac{f_{\text{max}}}{2})^3 = \frac{1}{8} \cdot P_d$ and the new energy consumption would be: $E' = P'_d \cdot 40 = \frac{1}{8} \cdot k \cdot f_{\text{max}} \cdot 20 = \frac{1}{4} \cdot E$, where $P_d$ is the power and $E$ is the energy consumption of the task with normal execution at the maximum processor speed $f_{\text{max}}$. 
When no task is available for execution, processors become idle and are put into a power savings sleep state. For simplicity, it is assumed that no power is consumed when processors are in sleep state. That is, we consider only dynamic power dissipation in this chapter. However, in addition to the dominated dynamic power, static leakage power consumption becomes more significant as the technology size shrinks [39]. Recent research shows that, the effect of static leakage power on energy management is equivalent to imposing a minimum energy efficient speed limitation. That is, if the processing speed is not scaled down below this limit, a task will consume less energy at lower processing speeds. Interested readers are referred to [14, 26, 37, 48] for more details.

1.3 Frame-based RT-Applications and Canonical Schedule

1.3.1 Frame-based Real-Time Applications

A frame-based real-time application consists of a set of tasks $\Gamma = \{T_1, \ldots, T_n\}$ and has a frame length $D$. Within each frame, the tasks are mapped to processors and the execution of all tasks needs to complete before the end of the frame (i.e., after the tasks are released at the beginning of one frame, they have a relative deadline $D$). The mapping of tasks to processors and the schedule within a frame are repeated for the following frames [28]. Because of the periodicity, we consider only the problem of scheduling the tasks in a single frame with the deadline $D$

For each task $T_i \in \Gamma$, the estimated worst case execution time (WCET) is denoted by $c_i$. Note that the execution time of a task depends on the processing speed of the processor that executes the task. As variable speed processors are considered in the systems, we assume that the WCETs of tasks are based on the maximum processor speed $f_{\text{max}}$. When a task is executed at a scaled speed (e.g., $f$), its execution time (e.g., $c_i f_{\text{max}}$) will increase linearly. Moreover, we consider only non-preemptive scheduling schemes; that is, a task will run-to-completion whenever it begins to execute.

The execution of one task may or may not depend on the results of other tasks, which corresponds to dependent and independent tasks, respectively. For independent tasks, all tasks are ready for execution at the beginning of each frame. While the readiness of dependent tasks relies on the completion time of their predecessor(s).

In general, the dependencies among tasks are represented by a directed acyclic graphic
(DAG), where the nodes represent tasks and the edges represent dependencies. There is an edge, $T_i \rightarrow T_j$, if and only if $T_i$ is a direct predecessor of $T_j$, which means that $T_j$ needs the data produced by $T_i$. The tasks without predecessors are referred to as root tasks and are ready for execution at the beginning of each frame. The communication between different tasks is achieved through accessing shared memory. Therefore, it is assumed that there is no communication cost even if tasks $T_i$ and $T_j$ are mapped onto different processors. After $T_i$ finishes execution, the produced data is available immediately and $T_j$ is ready to execute.

Consider an example with three dependent tasks as shown in Figure 1.1a, where tasks $T_1$ and $T_2$ are root tasks that will be ready for execution at the beginning of each frame. Task $T_3$ has two direct predecessors and depends on both task $T_1$ and task $T_2$. Only after $T_1$ and $T_2$ finish their execution, can task $T_3$ begin to run. For the two numbers associated with each node, the first one is the worst case execution time (WCET) of the corresponding task, and the second one is the task’s average case execution time (ACET), which will be used in the speculation schemes as discussed in Section 1.6.3.

1.3.2 Task Scheduling for Parallel Systems

Two approaches have been used traditionally to schedule a set of tasks on multiple processors: partition scheduling and global scheduling [9, 11]. The mapping of tasks to processors is fixed in partition scheduling and there is a task queue for each processor. In contrast, global scheduling only has a global queue and the mapping of tasks to processors will depend on run-time information.

For partition based scheduling, the well developed uniprocessor power management schemes may be applied directly on each processor after assigning tasks to processors. However, it has been shown that, due to the convex relation between CPU speed and its power consumption, the optimal energy efficient schedule can only be obtained with perfect load balancing among all processors [4]. Considering the run-time behaviors of the tasks [13], the actual workload on each processor under partition scheduling may vary substantially. Therefore, the fixed assignment of tasks to processors under partition scheduling limits the opportunities of managing power consumption across different processors, which may lead to sub-optimal energy efficient schemes. On the contrary, global scheduling has the merit of automatically balancing the workload among processors at run-time. Moreover, the non-constant mapping of tasks to processors makes the problem of global queue based power aware scheduling more interesting and we will focus on global scheduling in this chapter.

When tasks are available at the same time, the order of tasks entering the global queue (i.e., the priority assignment of tasks) affects which task goes where (i.e., task’s mapping) as well as the workload on each processor. It is well-known that the optimal priority assignment to balance the workload and minimize the schedule length is NP-hard [11]. Therefore, many heuristics (e.g., Longest Task First [35]) have been studied to efficiently obtain close-to-optimal schedules.

Our research has shown that, the optimal priority assignment for tasks that minimizes static schedule length based on tasks’ WCET may not result in the minimum energy consumption due to the run-time behaviors of tasks [47]. Instead of seeking another priority assignment heuristic, we focus on in this chapter exploring the available slack time for energy savings. For illustration purpose, we use the Longest Task First (LTF) heuristic (based on tasks’ WCETs) when assigning task’s priority. Intuitively, large tasks are likely to generate more slack at run-time and thus more energy savings may be expected. Moreover, the ratio of schedule length with LTF over the one with optimal priority assignment is bounded [35] and it is likely to obtain a feasible schedule with LTF. However, the power management schemes discussed next do not rely on specific heuristic and can work with any priority
1.3.3 Canonical Schedule and Static Power Management

In order to simplify the discussion, a canonical schedule is defined as the mapping/schedule of tasks to processors when all tasks use their WCETs. To focus on power management, for the application considered, it is assumed that the canonical schedule under the LTF priority assignment could finish all tasks in time. Otherwise, a better priority assignment heuristic may be used to obtain a feasible canonical schedule.

For the example shown in Figure 1.1a, the canonical schedule for the tasks executing on a dual-processor system is illustrated in Figure 1.1b. In the figure, the X-axis represents time, the Y-axis represents processing speed, and the area of a task box defines the amount of work needed for executing the task.

Initially, both task \( T_1 \) and task \( T_2 \) are ready for execution and, under LTF heuristic, \( T_2 \) has higher priority. After processor \( P_1 \) fetches and executes \( T_2 \), the second processor \( P_2 \) gets task \( T_1 \) for execution. However, when processor \( P_2 \) finishes the execution of task \( T_1 \) at time 1, no task is ready and it becomes idle. When processor \( P_1 \) finishes task \( T_2 \), task \( T_3 \) becomes ready. Figure 1.1b shows that processor \( P_2 \) obtains task \( T_3 \) and finishes at time 3.

Suppose that the frame length of the application considered is \( D = 5 \). There are 2 units of slack available. Following the idea of static power management (SPM) for uniprocessor systems, where the slack is proportionally distributed to tasks according to their worst case execution time [25], static slack can be uniformly distributed over the length of a schedule for energy savings [17]. Figure 1.2 shows the case of uniformly scaling down the processing of all tasks, where every task runs at speed \( \frac{3}{5} f_{\text{max}} \). Compared to the case where no power management (NPM) is applied (i.e., tasks are executed at speed \( f_{\text{max}} \) and finish before the deadline), the simple approach of scaling down task’s processing uniformly could save \( 0.64E \), where \( E \) is the energy consumed under NPM.

However, despite the simplicity and its optimality for uniprocessor systems, SPM is not optimal in terms of energy savings for multiprocessor systems. The reason comes from the fact that the degree of parallelism in a schedule is different for different schedule sections. For example, as shown in Figure 1.3, due to the dependency among tasks, processor \( P_2 \) is idle from time 1 to time 2 and a gap exists in the schedule. If we partition the schedule along the time-line at the time points when processors change their states from busy/idle
to idle/busy, we will get a sequence of schedule sections with different degrees of parallelism (i.e., the number of processors that are active and execute tasks).

Define $t_{ij}$ as the length of the $j^{th}$ section of a schedule with parallelism of $i$. The aggregated length for the schedule sections with degree of parallelism as $i$ is denoted as $t_i = \sum_{j} t_{ij}$. For the example, the schedule section from time 0 to time 1 is referred to as $t_{21} = 1$ and has the degree of parallelism 2. The schedule sections from time 1 to 2 and from time 2 to time 3 have the degree of parallelism 1 and are referred to as $t_{11} = 1$ and $t_{12} = 1$, respectively. Hence, there are $t_1 = t_{11} + t_{12} = 2$ and $t_2 = t_{21} = 1$.

Instead of distributing the static slack over the schedule and scaling down the processing of tasks uniformly, considering the different degrees of parallelism in a schedule, it will be more energy efficient to allocate more slack for schedule sections with higher degrees of parallelism. The intuition is that, the same amount of slack can be used to scale down more computation for the schedule sections with higher degrees of parallelism and thus obtain more energy savings.

For a system with $N$ processors, there are at most $N$ processors that can concurrently execute tasks for any schedule sections and the degree of parallelism in a schedule ranges from 1 to $N$. Suppose that the static slack in the system is $L_0$. Moreover, the sections in a schedule with degree of parallelism $i$ has total length of $t_i$ (which may consist of several sections $t_{ij}$, $j = 1, \ldots, u_i$, where $u_i$ is the total number of sections with parallelism $i$) and the amount of static slack allocated to them is denoted by $l_i$ ($i = 1, \ldots, N$). After scaling down the schedule sections accordingly, the total energy consumption $E_{total}$ will be:

$$E_{total} = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} (k \times i \times f_i^3 \times (t_i + l_i))$$

$$= k \times \sum_{i=1}^{N} \left( i \times \left( \frac{t_i}{t_i + l_i} \times f_{\text{max}} \right)^3 \times (t_i + l_i) \right)$$

$$= k \times f_{\text{max}}^3 \times \sum_{i=1}^{N} \left( i \times \frac{t_i^3}{(t_i + l_i)^2} \right)$$

(1.2)

where $k$ is the system dependent constant and $f_i$ is the speed for sections with parallelism $i$. As illustrated, different allocation for the slack will result in different amount of energy savings. The problem of finding an optimal allocation of $L_0$ in terms of energy savings will be to find $l_1, \ldots, l_N$ so as to

$$\text{minimize}(E_{total})$$
subject to:

\[ l_i \geq 0, \quad i = 1, \ldots, N \]

\[ \sum_{i=1}^{N} l_i \leq L_0 \]

The constraints put limitations on how to allocate the static slack. Solving the above problem is similar to solving the constrained optimization problem presented in [3]. Interested readers are referred to [3] for more details.

For the special case of dual-processor systems, by differentiating Equation (1.2) with respect to \( l_1 \) and \( l_2 \), we can get the following optimal solutions:

\[ l_1 = \frac{T_1 \times (L_0 - (2^{1/3} - 1) \times T_2)}{T_1 + 2^{1/3} \times T_2}; \tag{1.3} \]

\[ l_2 = \frac{T_2 \times (2^{1/3} \times L_0 + (2^{1/3} - 1) \times T_1)}{T_1 + 2^{1/3} \times T_2}; \tag{1.4} \]

From the above equations, if \( L_0 \leq (2^{1/3} - 1)T_2 \) then \( l_1 = 0 \) (since there is a constraint that \( l_i \geq 0 \)) and \( l_2 = L_0 \), that is, all the static slack will be allocated to the sections of schedule with parallelism 2. For the example in Figure 1.1, there will be \( l_1 = 1.0676 \) and \( l_2 = 0.9324 \). The minimum energy consumption is \( 0.3464E \) and the amount of energy saving is \( 0.6536E \), which is better than the case of SPM (where the energy savings is \( 0.64E \)).

Note that the above optimal solution has an assumption that the schedule sections with the same degree of parallelism are executed at the same speed. However, the scheduling unit in our system is a task. After the static slack is distributed to schedule sections according to their parallelism, tasks should collect the slack from different sections that involve them and run at a uniform speed. For the example, task \( T_2 \) gets \( l_2 = 0.9324 \) from the section \( t_{21} \) and \( \frac{1}{2}l_1 = 0.5338 \) from the section \( t_{12} \). Therefore, task \( T_2 \) will obtain 1.4662 units of slack in total and could run at a reduced speed of \( 0.577f_{\text{max}} \). In fact, with each task collecting its slack and running at a uniform scaled speed, more energy savings could be obtained. As an exercise, interested readers are encouraged to find out the amount of energy savings for the example after \( T_2 \) collects its slack and run at a single speed.

When a task finishes early, the difference between the actual time taken and the one allocated to it is referred to as dynamic slack that can be used by the remaining task(s) to scale down their execution and save energy. In what follows, we will focus on online power aware mapping schemes that exploit dynamic slack for energy savings while guarantee to meet all timing constraints. To simplify the discussion, we assume that the canonical schedule of the applications considered finish all tasks just-in-time (i.e., no static slack exists). Otherwise, the schedule with scaled execution of tasks that finishes just-in-time is referred to as canonical schedule and used as a based for the online schemes.

### 1.4 Power Aware Mapping with Slack Sharing

As a first step, we assume that tasks are independent. That is, there is no data dependency between tasks and all tasks are ready for execution at the beginning of each frame. Sections 1.5 and 1.6 further discuss the cases for dependent tasks.

With LTF heuristic, larger tasks have higher priorities and the order of tasks in the global queue follows the decreasing order of tasks’ WCETs, where larger tasks are in the front of the queue. When a task uses less time than its WCET and finishes early at run time, the processor on which the task just finished will fetch the next ready tasks from the head of
the global queue for execution. The task may have been supposed to run on a different processor, and thus, the mapping/schedule of tasks to processors could be different from the canonical schedule.

For example, consider an application consisting of six tasks $\Gamma = \{T_1(5), T_2(4), T_3(3), T_4(2), T_5(2), T_6(2)\}$ to be executed on a dual-processor system, where the number associated with each task is its WCET. With LTF priority assignment, Figure 1.4a shows the order of tasks in the global queue and the canonical schedule. Here, the canonical schedule finishes all tasks just-in-time with the frame length of $D = 9$. Suppose that, during one execution, tasks $T_1$ to $T_6$ take $2, 4, 3, 2, 2, 2$ time units respectively, Figure 1.4b shows that the actual schedule also meets the deadline. However, it can also be seen that the mapping of tasks to processors, where task $T_3$ is mapped to processor $P_1$ and task $T_4$ is mapped to processor $P_2$, is different from the canonical schedule. Without slack reclamation for power management, it can be proven that, if tasks use less than or equal to their WCETs, the actual schedule will take less time than the canonical schedule. That is, all tasks can finish their executions before the deadline.

1.4.1 Greedy Slack Reclamation

The dynamic slack generated from the early completion of tasks can be reclaimed by the remaining tasks to scale down their execution and save energy. In greedy slack reclamation, all the slack generated from the early completion of one task on a processor will be allocated to the next task (if any) to be executed on this processor. It has been shown that the greedy scheme is an effective power management technique in uniprocessor systems [2, 33]. However, as the mapping of tasks to processors may change in multiprocessor systems due to the run-time behaviors of tasks, such slack reclamation scheme is not guaranteed to meet the timing constraints of tasks as illustrated below.

For the above example, Figure 1.5a shows that task $T_1$ finishes early at time 2 and 3 units of slack time are available on processor $P_1$. Under global scheduling, the next task to be run on processor $P_1$ is $T_3$. With greedy slack reclamation, all the available slack on $P_1$ will be given to task $T_3$. Therefore, including its WCET, $T_3$ will have 6 units of time and the processor speed can be reduced to $\frac{3}{2} \cdot f_{\text{max}}$ accordingly. When $T_3$ uses up all its allocated time, task $T_6$ will miss the deadline $D$ as illustrated in Figure 1.5b. Hence, even if the canonical schedule can finish before $D$, greedy slack reclamation under global scheduling...
cannot guarantee that all tasks will finish in time.

1.4.2 Slack Sharing Among Processors

Compared with the canonical schedule, we can see that task \( T_3 \) should finish its execution no later than time 7. However, when processor \( P_1 \) finishes task \( T_1 \) early, with greedy slack reclamation, task \( T_3 \) obtains all the 3 units of slack on processor \( P_1 \), which makes it possible for task \( T_3 \) to finish after time 7 with the scaled processing speed, thus leading to a deadline miss. Note that, only 2 units of the slack on processor \( P_1 \) are available before time 4, which is the time point when task \( T_3 \) is supposed to start its execution following the completion of task \( T_2 \) on processor \( P_2 \) in the canonical schedule. Therefore, we may divide the slack on processor \( P_1 \) into two parts: part one with 2 units is available before time 4 that can be reclaimed by task \( T_3 \); part two with 1 unit that will be shared with processor \( P_2 \) (see Figure 1.6a).

With \textit{slack sharing}, task \( T_3 \) reclaims the 2 units of slack on processor \( P_1 \) and emulates the timing as in the canonical schedule. That is, without considering these 2 units of slack, it looks like \( T_3 \) starts at time 4 and finishes at time 7, the same as in the canonical schedule.
Algorithm 1 Slack Sharing Mapping Algorithm

1: if ((T_k = Dequeue(Ready-Q)) != NULL) then
2:   Find P_r such that:
3:     STNT_r = min{STNT_1, ..., STNT_n};
4:   if (STNT_id > STNT_r) then
5:     STNT_id ↔ STNT_r;
6:     EF T_k = STNT_id + c_k;
7:     STNT_id = EF T_k;
8:     f_k = f_{max} \cdot \frac{c_k}{EF T_k - t};
9:     Execute T_k at speed f_k;
10: else
11:     wait();
12: end if

With the slack, T_3 starts at time 2, runs for 5 time units at the speed of \frac{2}{3} \cdot f_{max} and also finishes its execution at time 7. Task T_4 starts at time 4 on processor P_2, reclaims the shared 1 unit of slack, executes for 3 time units at the speed of \frac{2}{3} \cdot f_{max} and ends at time 7. As illustrated in Figures 1.6b, all tasks complete before the deadline. Therefore, if a processor is supposed to finish executing its task later than other processors in the worst case, but finishes early and obtains some dynamic slack, such slack should be shared with other processors appropriately before reclaiming it for energy savings.

From a different perspective, sharing the slack emulates the mapping of tasks to processors as in the canonical schedule. For the example, it can be looked at as task T_1 being allocated 4 time units on processor P_1 instead of 5, while task T_2 is allocated 5 time units on processor P_2 instead of 4. The next task T_3 will run the processor P_1 where the current running task on it supposes to complete early. Hence, when task T_1 finishes early, processor P_1 only gets 2 units of slack and P_2 gets 1 unit. So, for the sake of the remaining tasks, the situation looks like task T_1 is assigned to processor P_2 that is supposed to finish at time 5 and task T_2 is assigned to processor P_1 that is supposed to finish at time 4. All the remaining tasks that are assigned to P_2 in the canonical schedule will now be assigned to P_1 and visa versa.

Following the slack sharing idea illustrated in the example, the online slack sharing mapping for power management can be generalized for systems with N (\geq 2) processors. Before formally presenting the algorithm, we first define two notations for ease of discussion. The estimated finish time (EFT) for a task executing on a processor is defined as the time at which the task is expected to finish its execution if it consumes all the time allocated to it. The start time of the next task (STNT) for a processor is defined as the time at which the next task can begin its execution on that processor, which is actually the EFT of the task currently running on the processor.

The slack sharing mapping algorithm is presented in Algorithm 1. It is assumed that all tasks are put into the global queue Ready-Q at the beginning of each frame following the LTF priority assignment. Each processor invokes the algorithm individually. The shared memory holds the control information, such as the Ready-Q and STNT values, which must be updated within a critical section (not shown in the algorithm for simplicity). At the beginning of each frame, the value of STNTs for processors are initialized to 0. The current processor invoking the algorithm is P_id, t is the current time and f_k is the processing speed of P_id to execute task T_k.

When processor P_id finishes a task at time t, it will try to fetch the next ready task from
Ready-Q for execution. If there is no more tasks in Ready-Q, processor $P_id$ will stall and sleep until the next frame. Here, we use the function wait() to put one processor to sleep (line 11). Otherwise, The processor gets the next ready task $T_k$ from Ready-Q. To emulate the mapping of tasks to processors as in the canonical schedule, where task $T_k$ supposes to run on the processor with the smallest $STNT$, we exchange the value of $STNT_{id}$ with that of the minimum $STNT_r$ (lines 2, 3 and 4). By exchanging $STNT_{id}$ with $STNT_r$, processor $P_id$ shares part of its slack (specifically, $STNT_{id} − STNT_r$) with processor $P_r$, and task $T_k$ reclaims other part (with amount of $STNT_r − t$) to scale down its execution for energy savings (lines 6, 7 and 8). Therefore, the slack sharing scheme is still a greedy-oriented approach. That is, after sharing the slack appropriately, all the remaining slack is allocated to the next task to be executed.

From the algorithm, we notice that at any time (except when Ready-Q is empty) the values of $STNT$ for processors are always equal to the $N$ biggest values of $EFT$ of the tasks running on the processors. One of these tasks is the most recently started task (from line 6 to 7). The task that starts next will be executed by the processor with the smallest $STNT$ (after exchanging $STNT$ values between processors). Therefore, by sharing the slack among the processors, the slack sharing mapping algorithm emulates the mapping of tasks to processors as in the canonical schedule and uses no more time than the canonical schedule while reclaiming the slack for energy savings. The result is further summarized in the following theorem. Interested readers are referred to [47] for the proof of this theorem.

**THEOREM 1.1** For any given heuristic of assigning task’s priority, if the canonical schedule for the independent tasks of a frame-based real-time application finishes within the frame length $D$, any execution with the same priority assignment under the slack sharing mapping algorithm will complete all tasks in time.

**1.5 Fixed Priority Slack Sharing for Dependent Tasks**

List scheduling is a standard scheduling technique for dependent tasks [11], where a task becomes ready for execution when all of its predecessors finish execution. The root tasks
that have no predecessors are ready at the beginning of each frame. List scheduling puts tasks into a global ready queue as soon as they become ready and processors fetch tasks from the front of the ready queue for execution. When more than one task are ready at the same time (e.g., have the same predecessors), finding the optimal order (i.e., priority) of tasks being added to the global queue that minimizes execution time is NP-hard [11]. Here, we use the same LTF heuristic as for independent tasks and put into the ready queue first the longest task (based on WCET) among the tasks that become ready simultaneously.

Hence, in list scheduling, the order of tasks entering the global queue (i.e., the order of tasks being fetched and executed by processors) depends on both tasks’ ready time and the priority assignment heuristics. Note that, due to the dependencies among tasks, the ready time of a task depends on the actual execution time of its predecessor(s). Therefore, the run time behaviors of tasks will result in different orders of tasks in the global queue, which in turn leads to different mapping of tasks to processors. The different mapping may cause the application to take more time than the canonical schedule, even if tasks use no more time than their WCETs as shown below.

For example, consider a dependent task set with six tasks ($\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$), where the data dependencies is represented by the DAG as shown in Figure 1.7a. The canonical schedule is shown in Figure 1.7b and the frame length of the application is $D = 12$. Task nodes are labeled with the tuple ($c_i, a_i$), where $c_i$ is the WCET for task $T_i$ and $a_i$ is the average case execution time (ACET). From the canonical schedule, we can see that $T_1$ and $T_2$ are root tasks and ready at time $0$. $T_3$ and $T_4$ are ready at time $2$ when their predecessor $T_1$ finishes execution. $T_5$ is ready at time $3$ and $T_6$ is ready at time $6$.

![FIGURE 1.8: The execution where tasks take their ACET](image)

For the execution where tasks take their ACET $a_i$, Figure 1.8 shows the order of tasks in the global queue and the schedule for the execution. It is clear that the actual schedule takes more time than the canonical schedule and misses the deadline. The reason is that the ready time of task $T_5$ becomes earlier than that of $T_3$ and $T_4$ due to early completion of task $T_1$. Thus, the order of tasks in the global queue changes, which in turn leads to a different mapping of tasks to processors and thus to the deadline miss.

To guarantee that the actual execution takes no more time than the canonical schedule, we need to prevent $T_5$ from executing before $T_3$ and $T_4$ and keep the order of tasks in the global queue the same as in the canonical schedule. In the beginning of each frame, regardless of whether tasks are ready or not, all tasks are put into the global queue in the
same order as in the canonical schedule. Processors fetch tasks for execution from the head of the global queue. Note that, due to the run-time behavior of tasks, it is possible that ready tasks exist in the global queue while the header task is not ready. In this case, the processor that tries to fetch the header task for execution can not fetch the ready tasks from the middle of the global queue and is blocked until the header task becomes ready.

By keeping the same execution order of tasks as in the canonical schedule, the fixed priority list scheduling ensures that any execution will take no more time than the canonical schedule. Moreover, the power aware mapping with slack sharing can work with the fixed priority list scheduling for energy savings. The scheme is illustrated by the above example as in Figure 1.9. As the header task may block processors due to not being ready, slack time may be wasted as shown in the figure. However, by enforcing the execution order, the same as for independent tasks, the slack sharing with fixed priority list scheduling ensures the same timing constrains as in the canonical schedule while achieving energy savings.

1.6 Slack Shifting for Applications with Different Execution Paths

We have presented the slack sharing mapping algorithm for applications where all tasks will be executed within every frame. However, for applications that have different execution paths following different branches, only a subset of the tasks on a specific execution path will be executed within one frame. For these applications, in addition to task’s less than WCET execution, dynamic slack is also expected at the task set level (e.g., shorter execution paths other than the longest one).

1.6.1 AND/OR Application Model

The AND/OR model that extends the one in [15] is used to represent branches as well as parallelism within such applications. In addition to computation nodes that represent normal tasks, two other different kinds of nodes are defined in the model: AND nodes and OR nodes. For the vertices in a directed acyclic graph (DAG), computation nodes are represented by a circle, AND nodes are represented by diamond and OR synchronization nodes are represented by dual-circle. The computation requirement for AND/OR is assumed to be 0. That is, they are considered as dummy tasks. For a synchronization node with
non-zero computation requirement, it is easy to transform it to a synchronization node and a computation node.

For an OR node, it depends on only one of its predecessors. In other words, it is ready for execution when any one of its predecessors completes. Moreover, only one of its successors depends on it. That is, exactly one of its successors will be executed after its execution. The OR nodes are used to model branches and explore the different execution paths in the applications. To represent the probability of taking each branch after an OR synchronization node, a number is associated with each successor of an OR node, which will be used by speculation schemes for more energy savings as discussed in Section 1.6.3.

An AND node depends on all its predecessors. Thus, only after all its predecessors finish execution can it be executed. All successors of an AND node depend on it, which implies that all its successors are executed after the completion of an AND node. The AND nodes are used to explore the parallelism in the applications.

For simplicity, we only consider the case where an OR node cannot be processed concurrently with other paths. All processors will synchronize at an OR node. We define application segment as a set of tasks that are separated by two adjacent OR nodes. There will be several application segments, one for each of the branches, between two corresponding OR nodes.

For example, Figure 1.10 shows one AND/OR application. Recall that the AND/OR synchronization nodes are dummy tasks with computation time as 0. In this example, there are two execution paths after the OR node $T_5$. The upper branch has the computation nodes $T_7$ and $T_8$, and the lower branch has the computation nodes $T_9$ and $T_{10}$. The longest path of the application is shown bold.

1.6.2 Shifting Schedule Sections in Canonical Schedule

To find out the amount of slack from different paths after a branch point, we need to generate the canonical schedule for the application segments between adjacent OR nodes. For the above example running on a dual-processor system, the canonical schedule is shown in Figure 1.11a.

For the synchronization nodes, which are considered as dummy tasks with 0 computation time...
requirement, they are shown as bold vertical lines. The dotted rectangles represent the application segments that, as a whole, will or will not be executed (integrated segments). For this example, if execution follows the upper branch after task $T_5$, it takes 2 time units less compared with the path along the lower branch. To utilize such possible slack online, we can shift the canonical schedule for each application segment toward the deadline as late as possible. The shifted canonical schedule for the example is shown in Figure 1.11b.

Here $L_1$ comprises the possible slack that can be reclaimed when the execution follows the upper branch. Notice that the tasks in one application segment are shifted together for simplicity. We do not consider shifting individual tasks in a segment. For example, for the segment consisting tasks $T_7$ and $T_8$, we may shift task $T_8$ one time unit more without violating the timing requirement. But shifting single tasks increases the complexity of the algorithm and hopefully that one unit of slack could be claimed by the subsequent tasks at run-time.

When there are nested OR nodes, different segments may obtain different amounts of slack from shifting. For the section of canonical schedule corresponding to each application segment, the power aware mapping with slack sharing can be applied at run-time, which will guarantee the timing constraints as explained in previous sections. For reclaiming slack from different execution paths at run-time, additional timing information regarding the tasks in the shifted canonical schedule need to be collected offline. Please refer to [49] for
detailed algorithms and explanations.

1.6.3 Speculation for Better Performance

Note that, the above discussed slack sharing/shifting schemes are greedy-oriented. That is, all available slack will be allocated to the next ready task after sharing/shifting the slack appropriately. Each task would run at a different speed after reclaiming the slack and many speed adjustments are expected.

As mentioned earlier, due to the convex relation between CPU speed and its power consumption, the energy consumption will be minimized when all tasks run at the same scaled speed across all processors [4]. Considering the speed adjustment overhead as discussed in the next Section, the single speed setting is even more attractive. From this intuition, we can speculate one processing speed at which tasks should, on average, be executed by exploiting the statistical information (e.g., the ACET of tasks and the probability for each branch to be taken, which can be obtained from profiling) about an application.

As the simplest approach, one processing speed will be speculated at the beginning of each frame by considering the average behaviors of all tasks and paths. More specifically, for a given application with set of tasks to be executed on a multiprocessor system, a static schedule may be generated assuming that each task takes its ACET. Thus, the average schedule length $\Pi_a$ could be obtained from the average schedule length for the application segments between corresponding branch points and the probability of the segments being executed. The single speculated speed will be:

$$f_{ss} = \frac{\Pi_a}{D} f_{max}$$

(1.5)

For tasks that may receive excessive amount of dynamic slack, the processing speed calculated from the greedy-oriented slack sharing scheme (line 8 in Algorithm 1) could be less than $f_{ss}$. In this case, we would rather execute the task at speed $f_{ss}$ and save some slack for future tasks with the expectation of running all tasks at the same speed. However, the speculative speed $f_{ss}$ is optimistic and does not take into consideration the worst-case behaviors of tasks. That is, it is possible that there is no enough slack for a task and the speed calculated from Algorithm 1 is higher than $f_{ss}$. In this case, the task will be executed at the higher speed calculated from the slack sharing scheme to guarantee that all future tasks meet their timing constraints.

Therefore, the speed at which a task is executed will be the higher speed between the speculation speed $f_{ss}$, and the one calculated from the greedy-oriented slack sharing schemes. By executing tasks at the speed no lower than the one from the greedy slack sharing schemes, the speculation guarantees that all tasks will finish no later than their completion times under slack sharing schemes. Thus, if all tasks could finish in time in canonical schedule, slack sharing schemes can meet all timing constraints, so does the speculation scheme.

If the statistical characteristics of tasks in an application vary substantially (e.g., the tasks at the beginning of one application have the average/maximum execution time ratio as 0.9 while for tasks at the end of the application the ratio is 0.1), it may be better to re-speculate the speed while the application execution progresses based on the statistical information about the remaining tasks. Considering the difficulty of speculating a speed after each task’s completion for multiprocessor systems and significant amount of slack may be expected after an OR node, the adaptive scheme speculates one new speed after each OR node. That is, the speculative speed would be:

$$f_{as} = \frac{\Pi_a}{D - t}$$

(1.6)
where $t$ is the current time (when the OR node is processed) and $\Pi_i'$ is the average remaining schedule length when branch $b_i$ is taken after that OR node. Again, to guarantee the timing constraints to be met, the speed for executing a task will be the higher speed between $f_{as}$ and the one from the slack sharing/shifting scheme. Please refer to [49] for more detailed discussion regarding how the speculative speeds are calculated and used.

### 1.7 Practical Considerations

In the previous discussion, we have assumed that tasks can run at any arbitrary speed and there is no overhead for speed adjustment. However, for the available commodity variable voltage/speed processors, such as Transmeta [24] and Intel XScale [22], only a few speed levels are available and changing speed takes time and energy. We will address the issues with discrete speed levels and present the idea of slack reservation to incorporate the overhead of speed adjustment.

For the processors that only have several working speeds, the power aware mapping algorithms can be easily adapted. Specifically, after reclaiming the slack and obtaining the new processing speed $f$ for the next task, if $f$ falls between two speed levels ($f_k < f \leq f_{k+1}$), setting $f$ to the next higher speed level $f_{k+1}$ will always guarantee that the task meets its deadline.

With the higher speed, the task does not need all the slack initially allocated to it and part of the slack is saved for future use. By sharing the slack with future tasks, the processing speed for all tasks may get more close to each other, which results in more energy savings. The simulation results show that the power aware mapping with discrete speed levels may achieve better performance, in terms of energy savings, than the case where processors have continuous speeds [47].

#### 1.7.1 Slack Reservation for Speed Adjustment Overhead

Depending on different techniques, the timing overhead of speed adjustment varies substantially, ranging from tens of microseconds [8] to a few milliseconds [34]. Moreover, such overhead may also depend on the scope of speed adjustment. For a general speed adjustment time overhead model, we assume that the overhead consists of two parts: a constant part that imitates the set-up time and a variable part that is proportional to the scope of speed adjustment. Hence:

$$O_{adj} = O_c + \lambda \cdot |f_1 - f_2|$$

where $O_c$ and $\lambda$ are constants; $f_1$ and $f_2$ are the speeds before and after speed adjustment. Here, the choice of $\lambda = 0$ results in a constant time overhead, which could be the maximum amount of time needed to adjust the speed with the full scope from/to the highest speed to/from the lowest one.

As the power aware mapping with slack sharing is a non-preemptive scheme, processors will at most change speed twice for each task. Therefore, as the simple approach, the time overhead can be incorporated by adding twice the maximum overhead to the worst case execution time for all tasks. This ensures that the processor has enough time to scale down the processing speed for each task and speed up afterward (which may be necessary for the remaining tasks finishing in time). However, this approach is conservative and the inflated WCET of tasks could lead to false deadline misses in the canonical schedule, which is used as a baseline.
Instead of statically adding the maximum speed adjustment overhead to task’s WCET, we could reserve slack for speed adjustment whenever needed. Specifically, when there is available slack, before using it to scale down the processing speed for the next task, we set aside enough slack for scaling down and speeding up the processing speed, which make sure that future tasks could run at the appropriate speed level and meet the timing constraints. The idea is further illustrated in Figure 1.12.

![Figure 1.12: Slack Reservation for Speed Adjustment Overhead](image)

When task $T_i$ finishes early and leaves the amount of slack $S_i$, part of $S_i$ will be used to change the processing speed for task $T_{i+1}$. Moreover, enough slack is also reserved for changing the processing speed back to $f_{\text{max}}$ in the future. In case task $T_{i+1}$ uses up its allocated time, the reserved slack makes sure that the remaining tasks could run at the maximum speed $f_{\text{max}}$ and meet their deadlines. The rest of the slack is used as additional time for task $T_{i+1}$ to scale down its processing speed.

Suppose that the current speed for task $T_i$ is $f_i$ and assume that the speed for task $T_{i+1}$ is $f_{i+1}$ (to be computed). The overhead, $O_i$, to change speed from $f_i$ to $f_{i+1}$, and the overhead, $R_i$, to change speed from $f_{i+1}$ back to $f_{\text{max}}$ are:

$$O_i = O_c + \lambda \cdot |f_{i+1} - f_i|$$  \hspace{1cm} (1.8)

$$R_i = O_c + \lambda \cdot (f_{\text{max}} - f_{i+1})$$  \hspace{1cm} (1.9)

Hence, $f_{i+1}$ can be calculated by giving additional time, $(S_i - O_i - R_i)$, to task $T_{i+1}$ as:

$$f_{i+1} = f_{\text{max}} \cdot \frac{c_{i+1}}{c_{i+1} + S_i - O_i - R_i}$$  \hspace{1cm} (1.10)

Assuming that $f_{i+1} < f_i$, the above equation is a quadratic equation in $f_{i+1}$:

$$2 \cdot \lambda \cdot f_{i+1}^2 + [c_{i+1} + S_i - 2 \cdot O_c - \lambda \cdot (f_{\text{max}} + f_i)] \cdot f_{i+1} - f_{\text{max}} \cdot c_{i+1} = 0$$  \hspace{1cm} (1.11)

If no solution is obtained with $f_{i+1} < f_i$ from the above equation, the assumption is wrong; that is, we should have $f_{i+1} \geq f_i$. It is possible to set $f_{i+1} = f_i$ if the slack $S_i - R_i$ is enough for task $T_{i+1}$ to scale down the speed from $f_{\text{max}}$ to $f_i$, that is, if $f_{\text{max}} \cdot \frac{c_{i+1}}{c_{i+1} + S_i - R_i} \leq f_i$, we can set $f_{i+1} = f_i$. If it is not possible to set $f_{i+1} \leq f_i$, we have $f_{i+1} > f_i$ and $f_{i+1}$ can be solved as:

$$f_{i+1} = f_{\text{max}} \cdot \frac{c_{i+1}}{c_{i+1} + S_i - 2 \cdot O_c - \lambda \cdot (f_{\text{max}} + f_i)}$$  \hspace{1cm} (1.12)
Finally, if $f_{i+1}$ computed from the above equation is larger than the maximum speed $f_{\text{max}}$, the available slack is not enough to make two speed adjustments and we will set $f_{i+1} = f_{\text{max}}$.

In most cases, the reserved slack, $R_i$, will not be used and becomes part of the slack $S_{i+1}$ from the early completion of task $T_{i+1}$. However, in some cases, the useful slack, $S_{i+1} - R_i$, is not enough for task $T_{i+2}$ to use it. In this case, $R_i$ will be used to change the speed back to $f_{\text{max}}$ and task $T_{i+2}$ will run at $f_{\text{max}}$. Moreover, because of the speed adjustment overhead, special care is needed before sharing the slack among processors. That is, if there is not enough time for the current processor to change its speed back to the maximum speed $f_{\text{max}}$ after sharing the slack, the slack sharing process should not be invoked. Instead, the current processor needs to change its speed back to $f_{\text{max}}$ first and share the slack after that (if possible). Please refer to [47] for more detailed discussions.

1.8 Summary

Power awareness has become an important system requirement not only for portable devices, which are normally battery powered, but also for systems with extremely high computation density (such as server clusters). Although many efforts have been devoted to uniprocessor systems, the research for parallel system is quite limited.

In this chapter, we addressed the problem of power aware mapping of a set of real-time tasks on a multiprocessor system. Voltage scaling, which scales down the processing speed for tasks by exploring slack time in the system, is explored for power management. Despite the simplicity of partition scheduling, the fixed mapping of tasks to processors limits the run-time mapping flexibility and thus the opportunity of managing power across processors. In global scheduling, the optimal priority assignment for tasks to minimize the schedule length is known to be NP-hard, and many heuristics have been proposed based on tasks’ worst case execution time (e.g., Longest Task First, LTF). Instead of working on another heuristic to get the close-to-optimal static schedule, we focus on the mapping of tasks to processors and explore the slack (static and dynamic) for energy savings.

Considering the different degrees of parallelism in a schedule due to dependencies among tasks, one scheme that aims at more efficiently allocating static slack is first discussed. As a reference, the canonical schedule is defined as the schedule/mapping of tasks to processors in which all tasks take their worst case execution time (WCET). It is assumed that the canonical schedule meets the timing constraints under a given priority assignment heuristic (e.g., LTF).

Based on global scheduling, for a set of independent tasks, power aware mapping with slack sharing shares slack among processors at run-time, scales down the processing speed of tasks for energy savings while ensuring that the actual schedule takes no more time than the canonical schedule. Therefore, for the same priority assignment heuristic, if the tasks can finish in time in the canonical schedule, the power aware mapping with slack sharing will meet all tasks’ timing constraints.

For dependent tasks, the readiness of a task depends on the run-time behaviors of its predecessors, which may result in different execution order of tasks and different mapping of tasks to processors as the ones in the canonical schedule. To ensure the timeliness of tasks, the fixed order list scheduling executes all tasks in the same order as in the canonical schedule. The slack sharing is further combined with the fixed order list scheduling for energy savings.

For applications with different execution paths, the AND/OR application model is extended to represent the branches within an application. For such applications, only a subset of tasks along a certain execution path will be executed. In addition to the slack sharing
that is applied between two adjacent branch points for reclaiming the slack resulting from task’s less than WCET execution, slack shifting is used to reclaim the additional slack resulting from executing paths other than the longest one.

Motivated by the fact that minimum energy consumption is obtained by executing all tasks with a single speed across all processors, one speculation scheme is discussed. The scheme speculates one speed at which tasks should be executed by exploiting the statistical information of an application (e.g., the ACET of tasks and the probability for each branch to be taken). Considering the possible uneven distribution of dynamic slack in an application, an adaptive scheme is further addressed which considers only remaining tasks for speculation.

Finally, some practical issues are discussed. For discrete speed levels, after obtaining an arbitrary speed, running at the next higher discrete speed always guarantees the task finish before its deadline. Slack reservation sets aside part of the available slack whenever needed for incorporating the speed adjustment overhead.

References

Power Aware Mapping of Real-Time Tasks to Multiprocessors


