IEEE Trans. on Parallel and Distributed Systems, Vol. 14, no. 7, pp. 686-700, 2003

Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems

Dakai Zhu, Rami Melhem and Bruce Childers
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
{zdk,melhem,childers}@cs.pitt.edu

Abstract

The high power consumption of modern processors becomes a major concern because it leads to decreased mission duration (for battery-operated systems), increased heat dissipation and decreased reliability. While many techniques have been proposed to reduce power consumption for uniprocessor systems, there has been considerably less work on multi-processor systems. In this paper, based on the concept of *slack sharing* among processors, we propose two novel power-aware scheduling algorithms for task sets with and without precedence constraints executing on multi-processor systems. These scheduling techniques reclaim the time unused by a task to reduce the execution speed of future tasks, and thus reduce the total energy consumption of the system. We also study the effect of discrete voltage/speed levels on the energy savings for multi-processor systems and propose a new scheme of *slack reservation* to incorporate voltage/speed adjustment overhead in the scheduling algorithms. Simulation and trace based results indicate that our algorithms achieve substantial energy savings on systems with variable voltage processors. Moreover, processors with a few discrete voltage/speed levels obtain nearly the same energy savings as processors with continuous voltage/speed, and the effect of voltage/speed adjustment overhead on the energy savings is relatively small.

Index Terms: Real-Time Systems; Multi-Processor; Scheduling; Slack Sharing.

1 Introduction

In recent years, processor performance has increased at the expense of drastically increased power consumption [15]. On the one hand, such increased power consumption decreases the lifetime of battery operated systems, such as hand-held mobile systems or remote solar explorers. On the other hand, increased power consumption generates more heat, which causes heat dissipation to be a problem since it requires more expensive packaging and cooling technology and decreases reliability, especially for systems that have many

processors.

To reduce processor power consumption, many hardware techniques have been proposed, such as shutting down unused parts or reducing the power level of non-fully utilized functional units [4, 7]. Processors that have multiple supply voltages (i.e., multiple power levels) have become available in recent years [16], making power management at the processor level possible. Using this feature, several software techniques have been proposed to adjust the supply voltage, especially for mobile or uniprocessor systems [1, 2, 6, 14, 17, 18, 20, 26]. However, much less work has been done for real-time multiprocessing applications [13, 24, 25], such as parallel signal processing, automated target recognition (ATR) and real-time MPEG encoding. For satellite-based parallel signal processing, the satellite may have multiple processing units and need to process the signals on-board in real-time [24]. ATR uses multiple processors to detect targets by comparing regions of interest (ROI) to templates in parallel. For mobile military systems (e.g., missiles), ATR is widely used and usually requires real-time processing [23]. Since such systems are battery operated, their power consumption needs to be managed to achieve maximum duration for minimal energy. For the applications of cable television and video conferencing, *real-time* performance of MPEG encoding is necessary and many processing units may be used to achieve real-time performance [12]. For such systems, power management can reduce energy consumption and associated costs.

In multi-processor real-time systems, power management that adjusts processor voltage/speed changes task execution time, which affects the scheduling of tasks on processors. This change may cause a violation of the timing requirements. This paper describes novel techniques that dynamically adjust processor voltage/speed while still meeting timing requirements. We propose scheduling algorithms that use *shared slack reclamation* on variable voltage/speed processors for task sets without precedence constraints (independent tasks) and task sets with precedence constraints (dependent tasks). All the algorithms are proven to meet timing constraints. We also discuss how to incorporate discrete voltage/speed levels into the algorithms, and propose a scheme to incorporate voltage/speed adjustment overheads into the scheduling algorithms with *slack reservation*. Simulation and trace (from real applications) based results show that our techniques save substantial energy compared to static power management.

1.1 Related Work

For uniprocessor systems, Yao et al. describe an optimal preemptive scheduling algorithm for independent tasks running with variable speed [26]. When deciding processor speed and supply voltage, Ishihara and Yasuura consider the requirement of completing a set of tasks within a fixed interval and the different switch activities for each task [17]. By assigning lower voltage to the tasks with more switch activities and higher

voltage to the tasks with less switch activities, their scheme can reduce energy consumption by 70%. Lee et al. proposed a power-aware scheduling technique using slack reclamation, but only in the context of systems with two voltage levels [18]. Hsu et al. described a performance model to determine a processor slow down factor under compiler control [14]. Based on the super-scalar architecture with similar power dissipation as the Transmeta Crusoe TM5400, their simulation results show the potential of their technique. Mossé et al. proposed and analyzed several techniques to dynamically adjust processor speed with slack reclamation [20]. The best scheme is the adaptive one that takes an aggressive approach while providing safeguards that avoid violating application deadlines [2]. For periodic tasks executing on uniprocessor systems, a few voltage/speed levels are sufficient to achieve the same energy saving as infinite voltage/speed levels [6]. AbouGhazaleh et al. have studied the effect of dynamic voltage/speed adjustment overhead on choosing the granularity of inserting power management points in a program [1].

For multiprocessor systems with fixed applications and predictable execution time, static power management (SPM) can be accomplished by deciding beforehand the best supply voltage/speed for each processor. Based on the idea of SPM, Gruian proposed two system-level designs for architectures with variable voltage processors. The simulation results show that both approaches can save 50% of the energy when the deadline is relaxed by 50% [13]. For system-on-chip designs with two processors running at two different fixed voltage levels, Yang et al. proposed a two-phase scheduling scheme that minimizes energy consumption under the time constraints by choosing different scheduling options determined at compile time [25]. Using the power aware multiprocessor architecture (PAMA), Shriver et al. proposed a power management scheme for satellite-based parallel signal processing based on different rate of the signals [24]. The work reported in this paper focused on *dynamic* power management for shared memory multi-processor systems, which is different from static power management [13], the selection of pre-determined scheduling options [25] and the master-slave architecture used in [24].

This paper is organized as follows. The task model, energy model and power management schemes are described in Section 2. Power-aware scheduling with dynamic processor voltage/speed adjustment using shared slack reclamation for independent tasks is addressed in Section 3. In Section 4, the algorithm for dependent tasks is proposed and proven to meet the timing requirements. Section 5 discusses how to incorporate voltage/speed adjustment overhead and discrete voltage/speed levels into the scheduling algorithms. Simulation and trace-based results are given and analyzed in Section 6 and Section 7 concludes the paper.

2 Models and Power Management

2.1 Energy Model

For processors based on CMOS technology, the power consumption is dominated by dynamic power dissipation P_d, which is given by: $P_d = C_{ef} \cdot V_{dd}^2 \cdot S$, where V_{dd} is the supply voltage, C_{ef} is the effective switched capacitance and S is the processor clock frequency, that is the processor speed. Processor speed is almost linearly related to the supply voltage: $S = k \cdot \frac{(V_{dd}-V_t)^2}{V_{dd}}$, where k is constant and V_t is the threshold voltage [4, 7]. Thus, P_d is almost cubically related to S: $P_d \approx C_{ef} \cdot \frac{S^3}{k^2}$. Since the time needed for a specific task is: $time = \frac{C}{S}$, where C is the number of cycles to execute the task, the energy consumption of the task, E, is $E = P_d \cdot time \approx C \cdot C_{ef} \cdot \frac{S^2}{k^2}$. When decreasing processor speed, we can also reduce the supply voltage. This reduces processor power cubically and energy quadratically at the expense of linearly increasing the task's latency. For example, consider a task that, with maximum speed S_{max}, needs 20 time units to finish execution. If we have 40 time units allocated to this task, 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: $P'_d = C_{ef} \cdot (\frac{V_{dd}}{2})^2 \cdot \frac{S_{max}}{2} = \frac{1}{8} \cdot P_d$ and the new energy consumption would be: $E' = P'_d \cdot 40 = C_{ef} \cdot (\frac{V_{dd}}{2})^2 \cdot \frac{S_{max}}{2} \cdot 40 = \frac{1}{4} \cdot C_{ef} \cdot V_{dd}^2 \cdot S_{max} \cdot 20 = \frac{1}{4} \cdot E$, where P_d is the power and E is the energy consumption with maximum processor speed.

2.2 Task Model

We assume a frame based real-time system in which a frame of length D is executed repeatedly [19]. A set of tasks $\Gamma = \{T_1, \ldots, T_n\}$ is to execute within each frame and is to complete before the end of the frame. The precedence constraints among the tasks in Γ are represented by a graph G. Because of the schedule's periodicity, we consider only the problem of scheduling Γ in a single frame with deadline D.

We assume a multi-processor system with N homogeneous processors sharing a common memory. Our goal is to develop a scheduling algorithm that minimizes energy consumption for all tasks while still meeting the deadline. In specifying the execution of a task T_i, we use the tuple (c'_i, a'_i), where c'_i is the estimated worst case execution time (WCET) and a'_i is the actual execution time (AET). Both are based on maximal processor speed. We assume that for a task T_i, the value of c'_i is known before execution, while a'_i is determined at run time. The precedence constraints are represented by $G = \{\Gamma, E\}$, where E is a set of edges. There is an edge, $T_i \rightarrow T_j \in E$, if and only if T_i is a direct predecessor of T_j, which means that T_j will be *ready* to execute only after T_i finishes execution.

2.3 Power Management Schemes

First, we consider the worst case scenario in which all tasks use their worst case execution time (referred to as *canonical execution*). In this case, if the tasks finish well before D at the maximal processor speed, S_{max}, we can reduce the processor's supply voltage and speed to finish the tasks *just-in-time*, and thus reduce energy consumption. The basic idea of static power management is to calculate beforehand the minimum processor speed that will ensure that the canonical execution of tasks finishes just-in-time. The tasks are then run with reduced supply voltage and speed to save energy [2, 13, 20]. In this paper, the minimal processor speed to ensure that all tasks finish just-in-time is referred to as S_{jit}.

In addition to static power management, we may reduce energy further by using dynamic voltage and speed adjustment. To simplify the discussion, we assume that the processor supply voltage and speed are always adjusted together, by setting the maximum speed under certain supply voltage. Since tasks exhibit a large variation in actual execution time, and in many cases, only consume a small fraction of their worst case execution time [11], any unused time can be considered as *slack* and can be reused by the remaining tasks to run slower while still finishing before D [2, 20]. In this case, power and energy consumption is reduced.

To get maximal energy savings, we combine static power management and dynamic voltage/speed adjustment. In the following algorithms, we assume that canonical execution is first checked to see whether a task set can finish before D or not. If not, the task set is rejected; otherwise, S_{jit} is calculated and used so that the canonical execution will finish at time D. Our algorithms then apply dynamic voltage/speed adjustment. In the rest of the paper, we normalize the worst case execution time and the actual case execution time of task T_i such that, $c_i = c'_i \cdot \frac{S_{max}}{S_{jit}}$ and $a_i = a'_i \cdot \frac{S_{max}}{S_{jit}}$. Task T_i will be characterized by (c_i, a_i).

Initially, to simplify the problem and our discussion, we assume that the processor supply voltage and frequency can be changed continuously, and ignore the overhead of voltage/speed adjustment. In Section 5, we discuss the effect of discrete speeds and overhead.

3 Power-Aware Scheduling for Independent Tasks

Without precedence constraints, all tasks are available at time 0 and are ready to execute. There are two major strategies to scheduling independent tasks in multi-processor systems: *global* and *partition* scheduling [10]. In global scheduling, all tasks are in a global queue and each processor selects from the queue the task with the highest priority for execution. In partition scheduling, each task is assigned to a specific processor and each processor fetches tasks for execution from its own queue. In this paper, we consider only the

non-preemptive scheduling scheme; that is, a task will *run-to-completion* whenever it begins to execute.

In global scheduling, the task priority assignment affects *which* task goes *where*, the workload of each processor, and the total time needed to finish the execution of all tasks. In general, the optimal solution of assigning task priority to get minimal execution time is NP-hard [10]. Furthermore, we show in Section 3.3 that the priority assignment that minimizes execution time may not lead to minimal energy consumption. Expecting that longer tasks generate more dynamic slack during execution, in this paper, we use the longest task first heuristic (LTF, based on the task's WCET) when determining task's priority. The difference between the total execution time using optimal priority assignment and that using longest task first priority assignment is small. Given a specific priority assignment, tasks are inserted into the global queue in the order of their priority, with the highest priority task at the front. For the examples, we number the tasks by their order in the global queue when using longest task first priority assignment. That is, the kth task in the global queue is identified as T_k.

To emphasize the importance of task priority on scheduling, we consider one simple example of a task set executing on a dual-processor system as shown in Figure 1. Here, $\Gamma = \{T_1(10, 7)$, $T_2(8, 4)$, $T_3(6, 6)$, $T_4(6, 6)$, $T_5(6, 6)\}$ and $D = 20$. In the figures, the X-axis represents time, the Y-axis represents processor speed (in cycles per time unit), and the area of the task box defines the number of CPU cycles needed to execute the task. Considering the canonical execution, from Figure 1(a) we see that the longest task first priority assignment meets the deadline D. But the optimal priority assignment in (b) results in less time. It is easy to see that some order, such as $T_3 \to T_4 \to T_5 \to T_2 \to T_1$, will miss the deadline.

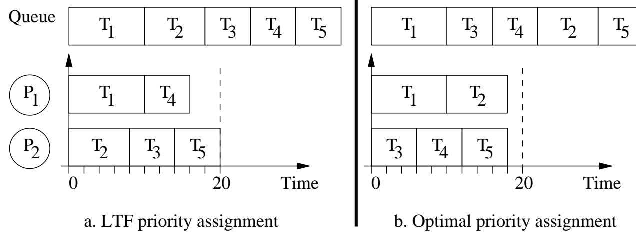


Figure 1: Global Scheduling for 2-Processor Systems

In what follows, we first extend the *greedy slack reclamation* scheme [20] to global scheduling, and we show that this scheme may fail to meet the deadline. Then we propose a novel slack reclamation scheme for global scheduling: *shared slack reclamation*.

3.1 Global Scheduling with Greedy Slack Reclamation

This scheme is an extension of the dynamic power management scheme for uniprocessor systems from Mossé et al. [20]. In the scheme of greedy slack reclamation, any slack on one processor is used to reduce the speed of the next task running on this processor.

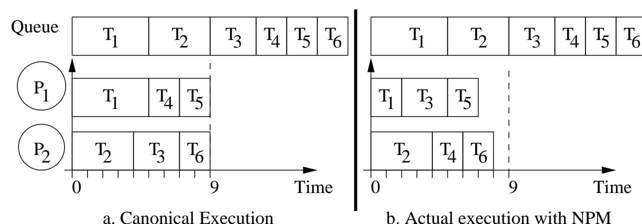


Figure 2: Global Scheduling with No Power Management

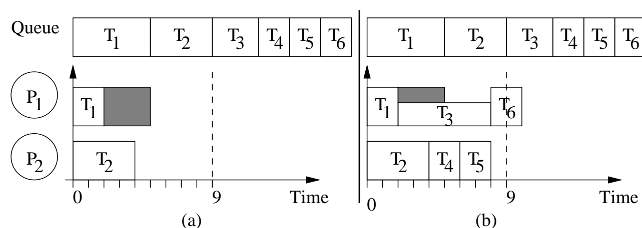


Figure 3: Global Scheduling with Greedy Slack Reclamation

Consider a task set: $\Gamma = \{T_1(5,2), T_2(4,4), T_3(3,3), T_4(2,2), T_5(2,2), T_6(2,2)\}$ and $D = 9$. Figure 2 (a) shows that the canonical execution can meet the deadline D. Figure 2 (b) shows that, with no power management and slack reclamation, actual execution can finish before D. Figure 3 (a) shows that in actual execution, T_1 finishes at time 2 with a slack of 3 time units. With greedy slack reclamation, this slack is given to the next task T_3 that runs on P_1. Thus, T_3 will execute in 6 units of time and the processor speed is reduced to $\frac{3}{6} \cdot S_{jit}$ accordingly. When T_3 uses up its time, T_6 misses the deadline D as shown in Figure 3 (b). Hence, even when canonical execution finishes before D, global scheduling with greedy slack reclamation cannot guarantee that all tasks finish before D.

3.2 Global Scheduling with Shared Slack Reclamation (GSSR)

For the example in Section 3.1, greedy slack reclamation gives all of the slack to T_3. This means that T_3 can start execution at time 2 at a speed of $\frac{3}{6} \cdot S_{jit}$ with 6 time units and finish execution at time 8. There is only 1 time unit left for T_6 which misses the deadline at time unit 9. In this case, it would be better to

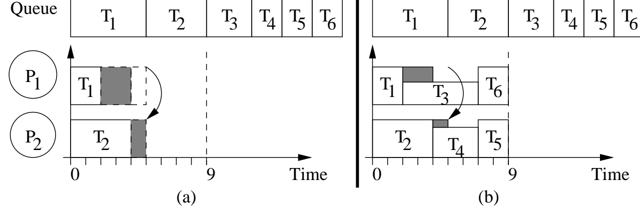


Figure 4: Global Scheduling with Shared Slack Reclamation

share the 3 units of slack by splitting it into two parts; i.e., give 2 units to T_3, and 1 unit to T_4. With *slack sharing*, T_3 starts at time 2, executes for 5 time units at the speed of $\frac{3}{5} \cdot S_{jit}$ and ends at time 7. T_4 starts at time 4, executes for 3 time units at the speed of $\frac{2}{3} \cdot S_{jit}$ and ends at time 7. Thus, both T_5 and T_6 meet the deadline. Figures 4 (a) and (b) demonstrate the operations of this scheme. When P_1 finishes T_1 at time 2, it finds that it has 3 units of slack. But only 2 of these time units are before P_2's expected finish time based on T_2's WCET. After fetching T_3, P_1 gives 2 units (the amount of slack before P_2's expected finish time) to T_3 and shares the remaining slack with P_2.

From a different point of view, sharing the slack may be looked at as T_1 being allocated 4 time units on P_1 instead of 5, with T_2 being allocated 5 time units on P_2 instead of 4. Here T_1 has 2 units of slack and T_2 has 1 unit of slack. So, in some sense, the situation is similar to T_1 being assigned to P_2 and T_2 being assigned to P_1, and all the tasks that are assigned to P_1 in canonical execution will now be assigned to P_2 and visa versa.

3.2.1 GSSR for N (≥ 2) Processor Systems (GSSR-N)

Following the idea described above, we propose the GSSR algorithm for N-processor systems. Before formally presenting the algorithm, we define the *estimated end time (EET)* for a task executing on a processor as the time at which the task is expected to finish execution if it consumes all of the time allocated for it. The *start time of the next task (STNT)* for a processor is the time at which the next task is estimated to begin execution on that processor. If no more tasks will execute on that processor within the current frame, the $STNT$ is defined as the finish time of the last task that executed on that processor.

The GSSR-N algorithm is presented in Algorithm 1. Each processor invokes the algorithm at the beginning of execution or when a processor finishes executing a task. A shared memory holds control information, which must be updated within a critical section (not shown in the algorithm). The shared memory has a common queue, *Ready-Q*, which contains all *ready* tasks and an array to record $STNT_p$ for processor P_p $(p = 1, \ldots, N)$. Initially, all tasks are put into *Ready-Q* in the order of their priorities, and the $STNTs$ of

Algorithm 1 The GSSR-N algorithm invoked by P_{id}

```
1:  while (1) do
2:    if (Ready-Q ≠ ∅) then
3:      T_k = Dequeue(Ready-Q);
4:      Find P_r such that:
        STNT_r = min{STNT_1, ..., STNT_n};
5:      if (STNT_id > STNT_r) then
6:        STNT_id ↔ STNT_r;
7:      end if
8:      EET_k = STNT_id + c_k;
9:      STNT_id = EET_k;
10:     S_id = S_jit · c_k / (EET_k − t);
11:     Execute T_k at speed S_id;
12:   else
13:     wait();
14:   end if
15: end while
```

processors are set to 0. In the algorithm, P_{id} represents the current processor, t is the current time, and S_{id} is the speed of P_{id}.

At the beginning of execution or when P_{id} finishes a task at time t, if there are no more tasks in *Ready-Q*, P_{id} will stall and sleep until it is waken up by the next frame. Here, we use the function *wait()* to put one processor to sleep (line 13). Otherwise, P_{id} will fetch the next task T_k from *Ready-Q* (line 3). Because T_k starts at the smallest $STNT$ in the canonical execution, we exchange $STNT_{id}$ with the minimum $STNT$ if $STNT_{id} > min\{STNT_1, \ldots, STNT_n\}$ (line $4, 5$ and 6). Here, we try to emulate the timing of the canonical execution. P_{id} then calculates its speed S_{id} to execute T_k based on the timing information and begins execution. By exchanging $STNT_{id}$ with $STNT_r$, P_{id} shares part of its slack (specifically, $STNT_{id} - STNT_r$) with P_r.

Reconsider the example from Figure 1 and suppose every task uses its actual execution time. Assuming that power consumption is equal to $C_{ef} \cdot \frac{S^3}{k^2}$, if no slack is reclaimed dynamically, the energy consumption is computed to be $29 \cdot \frac{C_{ef}}{k^2}$. Under global scheduling with shared slack reclamation and longest task first priority assignment, the energy consumption is computed to be $21.83 \cdot \frac{C_{ef}}{k^2}$. Note that if we use the optimal priority assignment as in Figure 1 (b) which optimizes the execution time, the energy consumption is computed to be $21.97 \cdot \frac{C_{ef}}{k^2}$. Hence, the optimal priority assignment in terms of execution time is not optimal for energy consumption when considering the dynamic behavior of tasks.

From the algorithm, we notice that at any time (except when *Ready-Q* is empty) the values of $STNT_p$ $(p = 1, \ldots, N)$ of the processors are always equal to the N biggest values of EET of the tasks running on the processors. One of these tasks is the most recently started task (from line 5 to 9). The task that starts next will follow the smallest $STNT$. These properties are used to prove the correctness of GSSR-N (in the

sense that, shared slack reclamation does not extend the finish time of the task set and execution with shared slack reclamation will use no more time than the canonical execution) as shown in next section.

3.2.2 Analysis of the GSSR-N Algorithm

For the canonical execution, we define the *canonical estimated end time*, EET_k^c, for each task T_k. From the definition, we know that EET_k is the latest time at which T_k will finish its execution. If $EET_k = EET_k^c$ for every task and the canonical execution can finish before time D, then any execution will finish before D. To prove that $EET_k = EET_k^c$ for every T_k, we define $max_N\{X_1, \ldots, X_n\}$ to be the set containing the N largest elements in the set $\{X_1, \ldots, X_n\}$[1]. We also define the history set $H(t)$ as the set of tasks that have started (and possibly finished) execution before or at time t.

Lemma 1 *For GSSR-N, at any time t, if T_k is the most recently started task, then $EET_k \in max_N\{EET_i|T_i \in H(t)\}$; moreover, $\{STNT_1, \ldots, STNT_N\} = max_N\{EET_i|T_i \in H(t)\}$.*

Proof If $n \leq N$, the result is trivial. Next, we consider the case where $n > N$. The proof is by induction on $T_k, k = 1, \ldots, n$.

Base case: Initially, after the first N tasks start execution and before any of them finish, at any time t, we have $H(t) = \{T_i, i = 1, \ldots, N\}$ and

$$EET_N \in max_N\{EET_i|T_i \in H(t)\};$$

$$\{STNT_1, \ldots, STNT_N\} = max_N\{EET_i|T_i \in H(t)\}.$$

Induction step: Assuming that, at any time t, T_{k-1} $(k-1 \geq N)$ is the most recently started task, we have $H(t) = \{T_1, \ldots, T_{k-1}\}$ and

$$EET_{k-1} \in max_N\{EET_i|T_i \in H(t)\};$$

$$\{STNT_1, \ldots, STNT_N\} = max_N\{EET_i|T_i \in H(t)\};$$

Without loss of generality, assume $EET_j = min\{STNT_1, \ldots, STNT_N\} = min\{max_N\{EET_i|T_i \in H(t)\}\}$ $(1 \leq j \leq k-1)$. After T_k started and before T_{k+1} starts, at any time t, T_k is the most recently

[1]If $n < N$, the remaining values are taken to be zero.

started task. Hence $H(t) = \{T_1, \dots, T_k\}$ and from line 5 to 9 of the algorithm:

$$\begin{aligned} EET_k &= min\{STNT_1, \dots, STNT_N\} + c_k \\ &= min\{max_N\{EET_i | T_i \in H(t)\}\} + c_k \\ &= EET_j + c_k \end{aligned}$$

Then, $EET_k \in max_N\{EET_i | T_i \in H(t)\}$. The new values of $STNT$s are thus given by:

$$\begin{aligned} \{STNT_1, \dots, STNT_N\} &= \{(\{STNT_1, \dots, STNT_N\} - \{EET_j\}) \cup \{EET_k\}\} \\ &= max_N\{EET_i | T_i \in H(t)\}; \qquad \diamondsuit \end{aligned}$$

Theorem 1 *For a fixed independent task set Γ with a common deadline executing on N-processor systems, if canonical execution with a priority assignment under global scheduling completes at a time D, any execution with the same priority assignment under GSSR-N will complete by time D.*

Proof

For a specific priority assignment, the canonical execution under global scheduling is the same as under GSSR-N and tasks can be identified by their orders in the ready queue during canonical execution. We prove this theorem by showing that, for any execution under GSSR-N: $EET_i = EET_i^c$ $(i = 1, \dots, n)$. If $n \leq N$, it is trivial. Next, we consider the case where $n > N$. The proof is by induction on T_k, $k = 1, \dots, n$.

Base case: Initially, GSSR-N sets EET_i at the beginning of execution without any consideration to the actual execution time of T_i $(i = 1, \dots, N)$. Hence, $EET_i = EET_i^c, i = 1, \dots, N$.

Induction step: Assume that $EET_i = EET_i^c$ for $i = 1, \dots, k-1$. At any time t before T_k starts, T_{k-1} is the most recently started task. Without loss of generality, assume that:

$$max_N\{EET_i | T_i \in H(t)\} = \{EET_{k-a_1}, \dots, EET_{k-a_{N-1}}, EET_{k-1}\}$$

$$EET_j = min\{max_N\{EET_i | T_i \in H(t)\}\}$$

Here, we have $a_1 > \dots > a_{N-1} > 1$ and $1 \leq j \leq k-1$. From Lemma 1:

$$\{STNT_1, \dots, STNT_N\} = \{EET_{k-a_1}, \dots, EET_{k-a_{N-1}}, EET_{k-1}\}$$

When T_k begins to run, from line 4 to 8 of Algorithm 1, we will have (for non-canonical and canonical execution, respectively):

$$
\begin{aligned}
EET_k &= min(STNT_1, \ldots, STNT_N) + c_k \\
&= min(EET_{k-a_1}, \ldots, EET_{k-a_{N-1}}, EET_{k-1}) + c_k \\
&= EET_j + c_k \\
EET_k^c &= min(STNT_1, \ldots, STNT_N) + c_k \\
&= min(EET_{k-a_1}^c, \ldots, EET_{k-a_{N-1}}^c, EET_{k-1}^c) + c_k \\
&= EET_j^c + c_k
\end{aligned}
$$

Notice that $EET_i = EET_i^c$ $(i = 1, \ldots, k-1)$. Thus, we have $EET_k = EET_k^c$. Finally, $EET_i = EET_i^c, i = 1, \ldots, n$. $\diamondsuit$

In the next section, we discuss scheduling with shared slack reclamation for dependent tasks. The idea of slack sharing is the same as that used for independent tasks. A new concern, however, is to maintain the execution order implied in the canonical execution of dependent tasks.

4 Power-Aware Scheduling for Dependent Tasks

List scheduling is a standard technique used to schedule tasks with precedence constraints [8, 10]. A task becomes *ready* for execution when all of its predecessors finish execution. The root tasks that have no predecessors are ready at time 0. List scheduling puts tasks into a ready queue as soon as they become ready and dispatches tasks from the front of the ready queue to processors. When more than one task is ready at the same time, finding the optimal task order that minimizes execution time is NP-hard [10]. In this section, we use the same heuristic as in global scheduling. We put into the ready queue first the longest task (based on WCET) among the tasks that become ready simultaneously. The tasks are numbered by the order at which they are added to the ready queue during canonical execution. That is, the kth task entering the ready queue in canonical execution is identified as T_k.

Consider a dependent task set with $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$ and $D = 12$. The precedence graph is shown in Figure 5a and the canonical execution is shown in Figure 5b. Task nodes are labeled with the tuple (c_i, a_i). For the canonical execution, we 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.

Due to dependencies among tasks, a task's readiness during non-canonical execution depends on the ac-

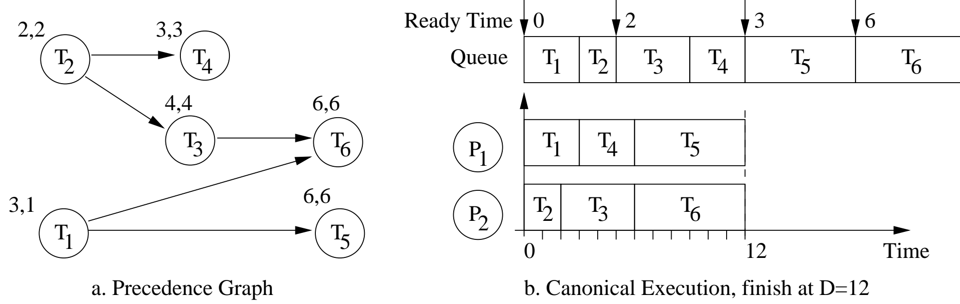


Figure 5: List Scheduling for Dual-Processor Systems

tual execution of its predecessors. From the discussion of independent tasks, we know that *greedy slack reclamation* cannot guarantee completion before D (i.e., the completion time of canonical execution). We next show that the straightforward application of *shared slack reclamation* to list scheduling cannot guarantee that timing constraints are met.

4.1 List Scheduling with Shared Slack Reclamation

Consider the example from Figure 5a and assume that every task uses its actual execution time. In Figure 6a, whenever one task is ready it is put into the queue. From the figure, it is clear that list scheduling with shared slack reclamation does not finish execution by time 12 (the completion time of canonical execution).

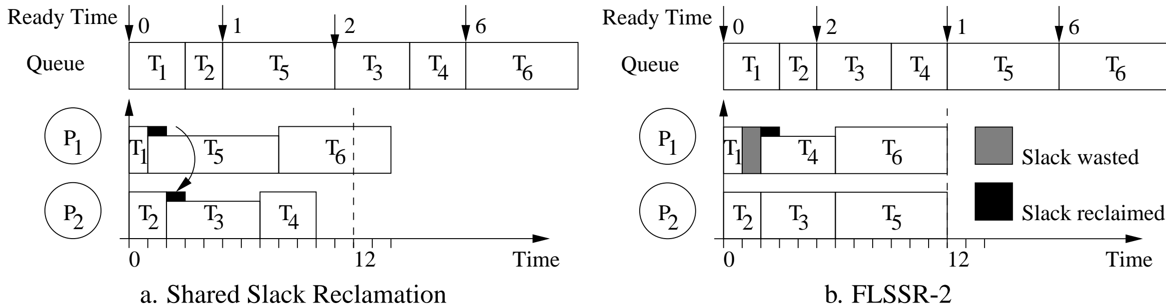


Figure 6: List Scheduling with Slack Reclamation; $D = 12$.

The reason list scheduling with shared slack reclamation takes longer than the canonical execution is that the tasks' ready time change. Thus, the order at which the tasks are added to the queue is different from the canonical execution order. In the example, T_5 is *ready* before T_3 and T_4, which leads to T_3 being assigned to P_2 rather than P_1. This in turn leads to the late completion of all tasks and the deadline being missed.

4.2 Fixed-order List Scheduling with Shared Slack Reclamation (FLSSR)

For the schedule in Figure 6a, we need to prevent T_5 from executing before T_3 and T_4 to guarantee that execution does not take longer than canonical execution; that is, we need to maintain the task execution order the same as in canonical execution. As discussed in Section 2, in the first step (which is not shown in the following algorithm), the canonical execution is emulated and S_{jit} is calculated. During the emulation, tasks' canonical execution order is collected and the ready time of task T_i is calculated as: $RT_i^c = max\{EET_k^c | T_k \rightarrow T_i \in E\}$ when all tasks run at S_{jit}.

To determine the *readiness* of tasks, we define the number of *unfinished immediate predecessors (UIP)* for each task. UIP_i will decrease by 1 when any predecessor of task T_i finishes execution. Task T_i is ready when $UIP_i = 0$. Whenever a processor is free, it will check the task at the head of *Global-Q* to see whether it is ready or not. If the task is ready, the processor will fetch and execute it; otherwise the processor goes to sleep. The details of the algorithm are described below.

4.2.1 FLSSR for N (≥ 2) Processor Systems (FLSSR-N)

As for independent tasks, we assume that the shared memory holds the control information. Algorithm 2 shows the FLSSR-N algorithm. Each processor (P_{id}) invokes the algorithm at the beginning of execution, when a task finishes execution on P_{id}, or when P_{id} is sleeping and signaled by another processor. We use the function *wait()* to put an idle processor to sleep and another function *signal(P)* to wake up processor P. Initially, all tasks are put in *Global-Q* in the canonical execution order (line 1; it is important for the algorithm to keep the canonical execution order to maintain temporal correctness). UIP_i $(i = 1, \ldots, n)$ are set to the number of predecessors of task T_i and $STNT_p$ $(p = 1, \ldots, N)$ are set to 0 (not shown in the algorithm).

If the algorithm is invoked by a signal from another processor, it will begin at the *'waiting for signal'* point (line 20). If the algorithm is invoked at the beginning or when P_{id} finishes a task, it begins at line 3. If the head of *Global-Q* is ready, P_{id} picks task T_k from the head of *Global-Q* (line 4). To claim the slack, P_{id} calculates EET_k as if T_k starts at the same time as in the canonical execution, which is RT_k^c or $STNT_{id}$ (whichever is bigger), and claims the difference between t and T_k's start time in the canonical execution as slack (line 9; notice that either $t \leq RT_k^c$ or $t \leq STNT_{id}$). Then P_{id} calculates the speed S_{id} to execute T_k and signals P_w if P_w is sleeping and the new head of *Global-Q* is ready (line 12 and 13). Finally, P_{id} runs T_k at the speed of S_{id} (line 15).

Reconsider the example shown in Figure 5, the execution on dual-processors for FLSSR-2 is shown in

Algorithm 2 The FLSSR-N algorithm invoked by P_{id}

```
1: Put the tasks in Global-Q in the order of their canonical execution.
2: while (1) do
3:   if ( Head(Global-Q) is ready ) then
4:     T_k = Dequeue(Global-Q);
5:     Find P_r such that:
       STNT_r = min{STNT_1, ..., STNT_n};
6:     if (STNT_id > STNT_r) then
7:       STNT_id <-> STNT_r;
8:     end if
9:     EET_k = max{RT_k^c, STNT_id, t} + c_k;
10:    STNT_id = EET_k;
11:    S_id = S_jit · c_k/(EET_k - t) ;
12:    if ( (Head(Global-Q) is ready) AND (P_w is sleep) ) then
13:      Signal(P_w);
14:    end if
15:    Execute T_k at speed S_id;
16:    for ( Each T_i such that T_k -> T_i ∈ E) do
17:      UIP_i = UIP_i - 1;
18:    end for
19:  else
20:    wait();
21:  end if
22: end while
```

Figure 6b. In order to wait for the readiness of T_3 and T_4, P_1 wastes part of its slack. By maintaining the same execution order as canonical schedule, all tasks finish on time.

4.2.2 Analysis of FLSSR-N Algorithm

Similar to GSSR-N, at any time (except when *Global-Q* is empty), the values of $STNT_p$ $(p = 1, \dots, N)$ are always equal to the N biggest values of EET of the tasks running on the processors. One of these tasks is the most recently started task. The task that starts next will follow the minimum $STNT$.

Lemma 2 *For FLSSR-N, at any time t, if T_k is the most recently started task, there will be $EET_k \in max_N\{EET_i | T_i \in H(t)\}$; moreover, $\{STNT_1, \dots, STNT_N\} = max_N\{EET_i | T_i \in H(t)\}$.*

Proof The proof is by induction on $T_k, k = 1, \dots, n$ and is similar to the proof of Lemma 1.

Base case: Initially, after T_i $(i = 1, \dots, m)$[2] start execution and before any of them finish, at any time t, we have $H(t) = \{T_1, \dots, T_m\}$ and

$$EET_m \in max_N\{EET_i | T_i \in H(t)\}$$

[2] If $m < N$, it means that there are only m tasks ready at time 0; otherwise, $m = N$, the number of ready tasks is greater than or equal to, N, the number of processors.

$$\{STNT_1, \ldots, STNT_N\} = max_N\{EET_i | T_i \in H(t)\}$$

Induction step: Assume that before T_k started execution, T_{k-1} is the most recently started task. At any time t, we have $H(t) = \{T_1, \ldots, T_{k-1}\}$ and

$$EET_{k-1} \in max_N\{EET_i | T_i \in H(t)\}$$

$$\{STNT_1, \ldots, STNT_N\} = max_N\{EET_i | T_i \in H(t)\}$$

Without loss of generality, assume $EET_j = min\{STNT_1, \ldots, STNT_N\} = min\{max_N\{EET_i | T_i \in H(t)\}\}$ $(1 \leq j \leq k-1)$. After T_k starts and before any more tasks finish, T_k is the most recently started task, and at any time t, $H(t) = \{T_1, \ldots, T_k\}$. From line 6 to 10 of Algorithm 2:

$$\begin{aligned} EET_k &= max\{min\{STNT_1, \ldots, STNT_N\}, RT_k^c, t\} + c_k \\ &= max\{EET_j, RT_k^c, t\} + c_k \end{aligned}$$

Notice that, when T_k starts, either $t \leq RT_k^c$ or $t \leq EET_j$. Then,

$$EET_k \in max_N\{EET_i | T_i \in H(t)\};$$

The new values of $STNT_p$ $(p = 1, \ldots, N)$ are thus given by:

$$\begin{aligned} \{STNT_1, \ldots, STNT_N\} &= \{(\{STNT_1, \ldots, STNT_N\} - \{STNT_q\}) \cup \{EET_k\}\} \\ &= max_N\{EET_i | T_i \in H(t)\}; \qquad \diamondsuit \end{aligned}$$

Theorem 2 *For a fixed dependent task set Γ with a common deadline executing on N-processor systems, if canonical execution with a priority assignment under list scheduling completes at time D, any execution will the same priority assignment under FLSSR-N will complete by time D*

Proof If all tasks use their WCET, canonical execution under list scheduling is the same as under FLSSR-N. For a specific priority assignment, the tasks are numbered by the order in which they entered *Global-Q* during canonical execution. We prove this theorem by showing that, for any execution of FLSSR-N: $EET_i = EET_i^c$ $(i = 1, \ldots, n)$. The proof is by induction on $T_k, k = 1, \ldots, n$.

Base case: Initially, FLSSR-N sets $EET_i, i = 1, \ldots, m(m \leq N)$ at the beginning of execution without any consideration to the actual execution time of T_i. Hence, $EET_i = EET_i^c, i = 1, \ldots, m(m \leq N)$.

Induction step: Assume that $EET_i = EET_i^c$ for $i = 1, \ldots, k-1$. At any time before T_k starts, T_{k-1} is the most recently started task. Without loss of generality, assume that:

$$EET_j = min\{max_N\{EET_i | T_i \in H(t)\}\}$$

$$max_N\{EET_i | T_i \in H(t)\} = \{EET_{k-a_1}, \ldots, EET_{k-a_{N-1}}, EET_{k-1}\}$$

Here, $a_1 > \ldots > a_{N-1} > 1$, $1 \leq j \leq k-1$. From Lemma 2:

$$\{STNT_1, \ldots, STNT_N\} = \{EET_{k-a_1}, \ldots, EET_{k-a_{N-1}}, EET_{k-1}\}$$

When T_k starts at time t (non-canonical execution) or t' (canonical execution), from line 5 to 9 of Algorithm 2, we will have:

$$\begin{aligned} EET_k &= max\{min\{STNT_1, \ldots, STNT_N\}, RT_k^c, t\} + c_k \\ &= max\{min\{EET_{k-a_1}, \ldots, EET_{k-a_{N-1}}, EET_{k-1}\}, RT_k^c, t\} + c_k \\ EET_i^c &= max\{min\{STNT_1, \ldots, STNT_N\}, RT_k^c, t'\} + c_k \\ &= max\{min\{EET_{k-a_1}^c, \ldots, EET_{k-a_{N-1}}^c, EET_{k-1}^c\}, RT_k^c, t'\} + c_k \end{aligned}$$

When T_k starts, either $t < RT_k^c$ and $t' < RT_k^c$, or
$t < min\{STNT_1, \ldots, STNT_N\}$ and $t' < min\{STNT_1, \ldots, STNT_N\}$.
Notice that, $EET_i = EET_i^c$ $(i = 1, \ldots, k-1)$, we will have:

$$\begin{aligned} & max\{min\{EET_{k-a_1}, \ldots, EET_{k-a_{N-1}}, EET_{k-1}\}, RT_k^c, t\} \\ = \; & max\{min\{EET_{k-a_1}^c, \ldots, EET_{k-a_{N-1}}^c, EET_{k-1}^c\}, RT_k^c, t'\} \end{aligned}$$

Thus $EET_k = EET_k^c$. Finally, $EET_i = EET_i^c, i = 1, \ldots, n$. $\diamondsuit$

In the above discussion, we assumed continuous voltage/speed and ignored the speed adjustment overhead. However current variable voltage processors have only discrete voltage/speed levels [16]. Moreover, there is time and energy overhead associated with voltage/speed adjustment. In the following section, we discuss how to incorporate these issues into the scheduling algorithms.

5 Accounting for Overhead and Discrete Voltage/Speed Levels

5.1 Voltage/Speed Adjustment Overhead

There are two kinds of overhead that have to be considered when changing processor voltage/speed: *time* and *energy*. The time overhead affects the feasibility of our algorithms; that is, whether the timing constraints can be met or not. We focus on time overhead first and discuss energy overhead later. When time overhead is considered, we need a model to calculate that overhead and a scheme to incorporate it into the algorithms. In the following, we propose a new scheme of *slack reservation* to incorporate time overhead into the dynamic speed adjustment algorithms.

5.1.1 Time Overhead

We model the time overhead as consisting of two parts: a constant part that is a set-up time and a variable part that is proportional to the degree of voltage/speed adjustment. Hence:

$$Time_{overhead} = C + K \cdot |S_1 - S_2|$$

where C and K are constants, and S_1 is the processor speed before adjustment and S_2 is the processor speed after the adjustment. Here, the choice of $K = 0$ results in a constant time overhead. In the simulations of Section 6, we set C and K to different values to see how they affect energy savings.

One conservative way to incorporate the time overhead is by adding the maximum time overhead of voltage/speed adjustment, $C + K \cdot (S_{max} - S_{min})$, to the worst-case execution time for all the tasks. In this case, there will be enough time to change speed for each task.

We propose the idea of slack reservation to incorporate the time overhead. Specifically, whenever we try to use slack to slow down processor speed, we reserve enough slack for the processor to change the voltage/speed back to the appropriate level in the future. In this way, we ensure that future tasks can be executed at the appropriate speed to meet the deadline. The idea is illustrated in Figure 7.

From the figure, when T_i finishes early with slack L_i, we use a portion of L_i to change the voltage/speed for T_{i+1}. We also reserve enough slack for changing the processor voltage/speed back to S_{jit} when T_{i+1} uses up its allocated time. The rest of the slack is used to slow down the speed of T_{i+1}.

Suppose that the current speed for T_i is S_i and assume that the speed for T_{i+1} is S_{i+1} (to be computed). The overhead, O_i, to change speed from S_i to S_{i+1}, and the overhead, R_i, to change speed from S_{i+1} back

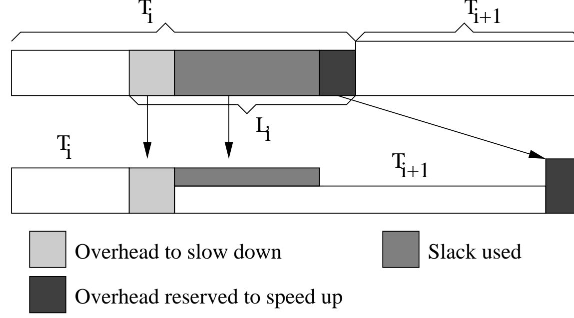


Figure 7: Slack Reservation for Overhead

to S_{jit} are:

$$O_i = C + K \cdot |S_{i+1} - S_i|$$

$$R_i = C + K \cdot (S_{jit} - S_{i+1})$$

Hence, S_{i+1} can be calculated by giving additional time, $(L_i - O_i - R_i)$, to task T_{i+1}, that is:

$$S_{i+1} = S_{jit} \cdot \frac{c_{i+1}}{c_{i+1} + L_i - O_i - R_i}$$

Assuming that $S_{i+1} < S_i$, then the above equation is a quadratic equation in S_{i+1}:

$$2 \cdot K \cdot S_{i+1}^2 + [c_{i+1} + L_i - 2 \cdot C - K \cdot (S_{jit} + S_i)] \cdot S_{i+1} - S_{jit} \cdot c_{i+1} = 0$$

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

$$S_{i+1} = S_{jit} \cdot \frac{c_{i+1}}{c_{i+1} + L_i - 2 \cdot C - K \cdot (S_{jit} - S_i)}$$

Finally, if S_{i+1} computed from the above equation is larger than S_{jit}, we set $S_{i+1} = S_{jit}$.

In most cases, the reserved slack, R_i, will not be used and becomes part of the reclaimed slack L_{i+1}. However, in some cases after T_{i+1} finishes, the useful slack, $L_{i+1} - R_i$, is not enough to use for T_{i+2}. In these cases, R_i will be used to change the speed back to S_{jit} and T_{i+2} will run at S_{jit} (see Figure 8).

When considering time overhead, slack sharing between processors needs to be modified. Referring to Figure 9, suppose processor P_i runs at S_i and finishes early. As described in Section 3, it would share its

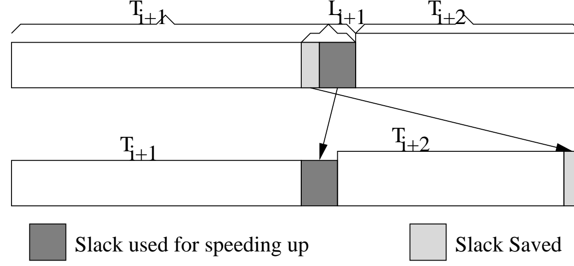


Figure 8: Slack not Enough to be Used by T_{i+2}

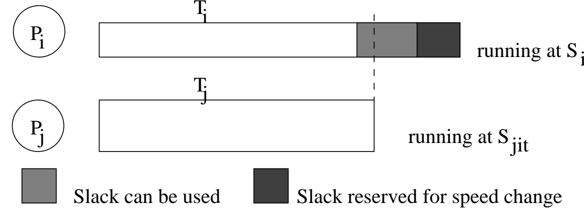


Figure 9: Slack Sharing with Overhead Considered

slack with processor P_j running at S_{jit} if $STNT_i > STNT_j$. But, if after slack sharing there is not enough time for P_i to change its speed back to S_{jit}, we should not share the slack. Processor P_i needs to change speed to S_{jit} first and share the slack later (if possible).

5.1.2 Energy Overhead

Besides the time overhead of voltage/speed adjustment, there is also energy overhead associated with the speed change. Suppose the energy overhead for changing speed from S_i to S_j is $E(S_i, S_j)$. Assuming that the energy consumption of T_{i+1} is E_{i+1} with S_{jit} and E'_{i+1} with S_{i+1}, then, it is not efficient to change the speed from S_i to S_{i+1} for T_{i+1} if $E(S_i, S_{i+1}) + E'_{i+1} + E(S_{i+1}, S_{jit}) > E_{i+1} + E(S_i, S_{jit})$. In other words, even if the timing constraints can be met with the time overhead, we may decide not to run T_{i+1} at a lower speed (if the energy overhead is larger than the energy saved by the speed change).

5.1.3 Setting the Processor to the Idle Speed

When no voltage/speed adjustment overhead is considered, we can always let the processor run at the slowest speed when it is *idle* (not executing a task). This speed achieves the least energy consumption for the idle state. With the overhead considered, for independent tasks, the idle state only appears at the very end of the execution and we can set the processor to idle if there is enough time to adjust the voltage/speed. For dependent tasks, however, the idle state may appear in the middle of execution. To ensure that future tasks finish on time, during idle state the processor needs to run at speed S_{jit} since the processor cannot predict

exactly when the next task will be available. We use this scheme to deal with the idle states appearing in the middle of execution.

We may put the processor to sleep when it is idle and wake it up before the next task is ready by predicting the ready time of the next task using the task's canonical ready time. This scheme will require a 'watchdog timer' to specify when the task is ready. However, it is possible that a task arrives before the timer expires, in this case, the processor needs to be activated and the timer deactivated. While this scheme can possibly achieve some additional energy savings, it makes the implementation more complex and for the purpose of this paper, will not be considered further.

Another way to deal with the idle state for dependent tasks is to be conservative and add the maximum overhead to each task's worst case execution time. In this case, we can always put the processor to sleep when it is idle and guarantee that there will be enough time to speed up the processor when the next task is ready to execute.

5.2 Discrete Voltage/Speed Levels

Currently available variable voltage processors have only several working voltage/speed settings [16]. Our algorithms can be easily adapted to discrete voltage/speed levels. Specifically, after calculating a given processor speed S, if S falls between two speed levels ($S_l < S \leq S_{l+1}$), setting S to S_{l+1} will always guarantee that the tasks finish on time and that the deadline is met.

With the higher discrete speed, some slack will not be used for the next task and thus will be available for future tasks. Our experimental results show that, when sharing slack with future tasks, scheduling with discrete voltage/speed levels sometimes have better performance, in terms of energy savings, than continuous voltage/speeds.

6 Performance Analysis

In this section, we empirically demonstrate how slack reclamation reduces energy consumption. Along with synthetic data, we also use several sets of trace data (from actual real-time multiprocessor applications) in the simulation. We compare the energy consumed when using the combination of static power management and dynamic supply voltage/speed adjustments with the energy consumed when using only static power management. Following the idea of the minimal energy scheduling technique for uniprocessor systems [17], we consider the clairvoyant (CLV) algorithm that uses the tasks' actual run time information to generate the schedule and to compute a single voltage/speed for all the tasks (the idle state may be still in the schedule).

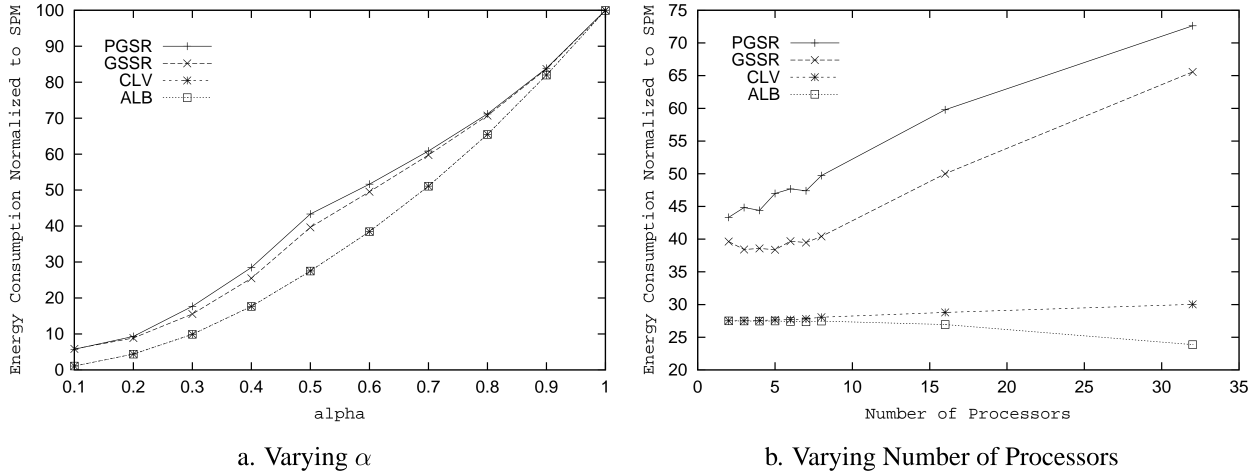
We also consider an absolute lower bound (ALB) scheme which assumes the application is fully parallel and is obtained by averaging the total actual workload on all processors with the speed being uniformly reduced (there is no idleness in this case, and pre-emption is needed to generate the schedule; e.g. P-fairness scheduling [3]). CLV and ALB are achievable only via post-mortem analysis and are impractical since they require knowledge of the future.

6.1 Experiments

First, we describe the simulation experiments. For the synthetic data, to get the actual execution time for each task, we define α_i as average/worst case ratio for T_i's execution time, and the actual execution time of T_i will be generated as a normal distribution around $\alpha_i \cdot c_i$. For the task sets, we specify the lower (c_{min}) and upper (c_{max}) bounds on the task's WCET and the average α for the tasks, which reflects the amount of dynamic slack in the system. The higher the value of α, the less the dynamic slack. A task's WCET is generated randomly between (c_{min}, c_{max}) and α_i is generated as a uniform distribution around α. For simplicity, power consumption is assumed to be proportional to S^3. In the following experiments, energy is normalized to the energy consumed when using only static power management. We also assume continuous voltage/speed scaling and no penalty for changing voltage/speed if not specified otherwise. The effects of discrete voltage/speed scaling and voltage/speed adjustment overhead are reported in Sections 6.5 and 6.6. When no overhead is considered, the processor speed in the idle state is set to $0.1 \cdot S_{jit}$; when overhead is considered, for the idle state appearing at the end of schedule, the processor speed is set to $0.1 \cdot S_{jit}$, while for the idle state in the middle of execution, the processor speed is set to S_{jit} as discussed earlier.

6.2 GSSR and Partition Scheduling with Greedy Slack Reclamation vs. SPM

The results in this section were obtained by running a synthetic independent task set with 100 tasks and the results are the average of 1000 runs. The WCET of tasks are generated by setting $c_{min} = 1$ and $c_{max} = 50$. In Figure 10a, the number of processors is 2, and α is varied from 0.1 to 1.0. We compare the global scheduling with shared slack reclamation (GSSR) with partition scheduling and greedy slack reclamation (PGSR). For PGSR, we use the longest task first partitioning to divide tasks among processors, and then apply greedy slack reclamation on each processor [20]. From the figure, we see that global scheduling with shared slack reclamation consumes less energy than partition scheduling with greedy slack reclamation. The reason is that the slack sharing scheme gives more slack to longer tasks and less to shorter tasks. This balances the speed of each task and reduces energy consumption. When the average/worst case ratio (α) is about 0.5 (that is, on the average we have 50% of time as dynamic slack), global scheduling with shared



a. Varying α b. Varying Number of Processors

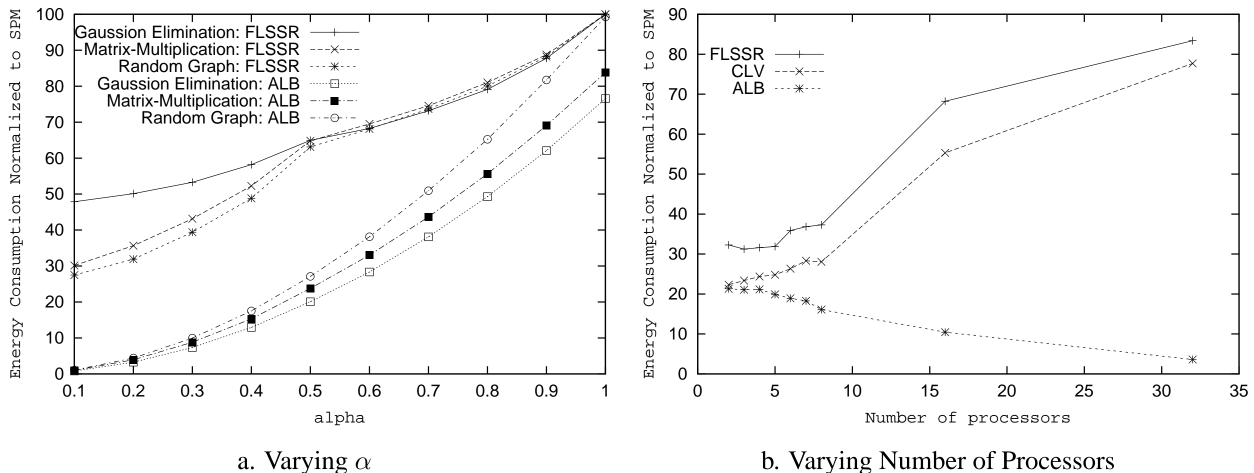
Figure 10: Energy Savings for Independent Tasks

slack reclamation results in energy saving of more than 60% versus static power management. When α increases, there is less dynamic slack and, compared to SPM, the energy saving of GSSR decreases. Note that, for independent tasks, only a little idle state appears at the very end of the schedule and CLV gets almost the same energy savings as ALB. Compared with these lower bounds, the performance of our algorithm is within 15% difference (when $\alpha = 0.5$).

To see the shared slack reclamation scheme's performance on systems with different number of processors, we run the synthetic independent task set by changing the number of processors and setting $\alpha = 0.5$. The results are shown in Figure 10b. Compared to SPM, the energy savings of GSSR is almost the same when the number of processors is less than or equal to 8. When the number of processors is more than 8, the energy savings of GSSR decreases sharply. The reason is that the first task on each processor is always executed at S_{jit} and the slack at the very end on each processor is wasted. Since there are only 100 tasks in the task set, with more processors, such as 16 or 32, the number of tasks running at S_{jit} and the total amount of slack wasted increases quickly. While compared with PGSR, our algorithm is always better. When the number of processors is less or equal to 8, our algorithm is within 13% of CLV and ALB. With more processors, such as 16 or 32, ALB performs better than CLV. The reason is that ALB assumes the actual workload is evenly balanced among all processors.

6.3 FLSSR vs. SPM

In this section, we consider the dependent task sets and compare the energy consumption used by FLSSR vs. SPM. First, we consider an example with 20 synthetic tasks. The tasks' WCET are generated randomly



a. Varying α

b. Varying Number of Processors

Figure 11: Energy Savings for Dependent Tasks

from 1 to 50 and we assume a 2-processor system. In Figure 11a, we vary α from 0.1 to 1.0. The energy saving of fixed-order list scheduling with shared slack reclamation (FLSSR) compared to that of static power management (SPM) varies from 0% when α is 1.0 to 72% when α is 0.1. When α increases, there is less dynamic slack and compared to SPM the energy savings of FLSSR decreases. On average, when α is 0.5, the energy savings is approximately 40%. Since there is more idle time for dependent tasks, compared with ALB, the performance of our algorithm is within 35% difference (when $\alpha = 0.5$).

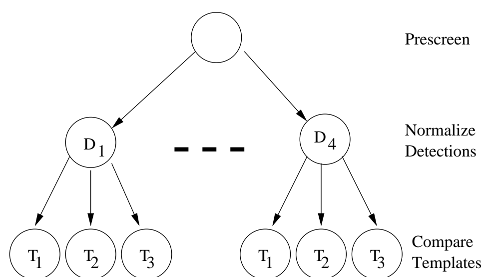
We next consider two matrix operations, matrix-multiplication and Gaussian-elimination (assuming a 5×5 matrix of 100×100 submatrices) [9], and measure the effectiveness of our techniques for these benchmarks. The worst case execution time of each task is determined by the operations involved. We conduct the same experiments as above, achieving similar energy savings for fixed-order list scheduling with shared slack reclamation. The results are shown in Figures 11a.

For Gaussian-elimination, we also considered a 20×20 matrix of 100×100 submatrices to allow more parallelism. With $\alpha = 0.5$, we vary the number of processors as shown in Figure 11b. For this application, when the number of processors is larger than 8, the energy consumption of FLSSR increases sharply compared to SPM. One reason is similar to what happen for GSSR: the number of tasks running at S_{jit} and the amount of slack wasted increases. Another reason is the idleness of the processors due to the dependence among tasks. Compared with CLV, our algorithm is within 15% difference. Note that ALB assumes a fully parallel application, which is not possible for Gaussian-elimination with a large number of processors.

6.4 FLSSR with Trace Data

In this section, we use several sets of trace data for different parallel applications to show the effectiveness of our algorithms. The trace data is gathered by instrumenting the applications to record their execution time for each parallel section. The applications are then run on a Pentium-III $500MHz$ with $128MB$ memory.

The first application we considered is automated target recognition (ATR). ATR searches regions of interest (ROI) in one frame and tries to match specific templates with each ROI. The dependence graph for ATR is shown in Figure 12a. Figure 12b shows the run time information about the tasks in ATR for processing 180 consecutive frames on our platform. Here, we assume that ATR can process up to four ROIs in one frame and that each ROI is compared with three different templates. If the number of ROIs is less than 4, the actual run time of the tasks corresponding to undetected ROIs (the first few ROIs) is set to 0.



a. Dependence Graph of ATR

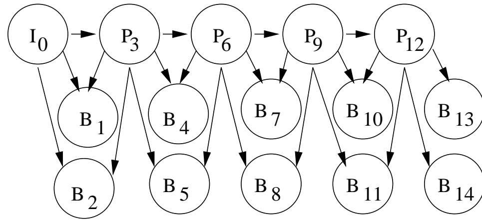
	min(μs)	max(μs)
Prescreen	1146	1299
Norm. Detection	429	748
Template 1	466	574
Template 2	466	520
Template 3	467	504

b. Execution Time for Tasks in ATR

Figure 12: The Dependence Graph of ATR to Process One Frame and The Execution Time for The Tasks of ATR. Assuming up to 4 detections in one frames and 3 templates.

Second, we consider the Berkeley real-time MPEG-1 encoder [12]. By setting the group of pictures (GOP) as 15 with the pattern of **IBBPBBPBBPBBPBB** and forcing it to encode the last frame, the dependence graph to process the frames in one GOP using decoded frame as reference is shown in Figure 13a. There are three different frames in the dependence graph. The **I** frame is the intra-frame that is encoded as a single image with no reference to any past or future frames. The **P** frame is the forward predicted frame that is encoded relative to the past reference frame. A **B** frame is a bi-directional predicted frames that is encoded relative to the past, the future or both reference frames. The reference frame is either an **I** or a **B** frame. For the *Flower-Garden* and *Tennis* movies with each having 150 frames, Figure 13b shows the run time information of processing different frames (the time is only for encoding and does not include I/O).

Using the trace data, we vary the number of processors and run these two applications (note that the maximum parallelism for Berkeley MPEG-1 encoder is 3 for one GOP) on our simulator. The results of



a. Dependence Graph of MPEG-1 Encoder

	Flower(ms)		Tennis(ms)	
	min	max	min	max
I	50	70	60	70
P	120	140	100	140
B	270	320	190	340

b. Execution Time for Different Frames

Figure 13: The Dependence Graph and Execution Time to Process Different Frames of MPEG-1 Encoder; assuming the encoding sequence is IBBPBBPBBPBBPBB, force to encode the last frame and use decoded frame as reference.

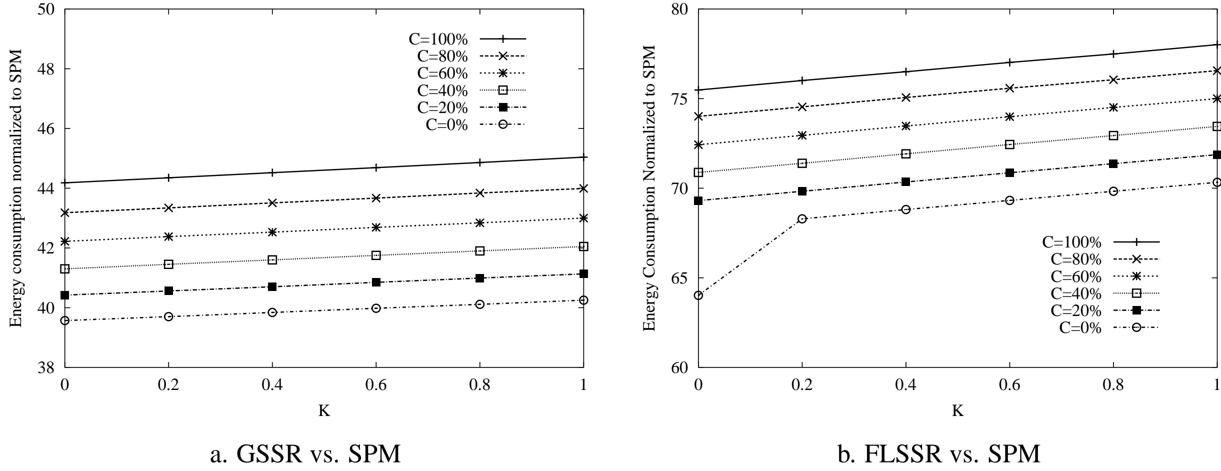
energy savings are shown in Table 1. There is more energy savings for *Tennis* than *Flower-Garden* from MPEG-1 encoder because the encoding time for *Tennis* varies more than *Flower-Garden* (see Figure 13b). CLV gets 7% -32% more energy savings than FLSSR and ALB gets 27%-44% more. Again, ALB assumes fully parallel application with preemption and an evenly balanced actual workload. It is impractical and is not a tight lower bound. The results are consistent with the earlier results from the synthetic data.

Table 1: Energy Savings vs. SPM using Trace Data

	ATR			MPEG-1 Encoder			
				Flower		Tennis	
	2-Proc	3-Proc	4-Proc	2-Proc	3-Proc	2-Proc	3-Proc
FLSSR	26.35%	38.65%	41.66%	17.42%	16.53%	25.16%	23.77%
CLV	58.83%	54.71%	52.14%	24.11%	26.43%	35.07%	36.92%
ALB	70.58%	78.19%	80.43%	44.33%	53.75%	52.65%	60.67%

6.5 Considering the Overhead

To observe how the time overhead affects the algorithms' performance in terms of energy savings, we set in the experiments the constant part of the overhead (C) to different values relative to the smallest task's worst case execution time. We also experiment with setting the co-efficient (K) to different values from 0 to 1. The maximum variable part of time overhead (changing speed between S_{max} and S_{min}) equals K times the smallest task's worst case execution time. Recall that the range of task's worst case execution time is from 1 to 50 and the smallest task has worst case execution time of 1. Figure 14a shows an independent task set with 100 tasks, and Figure 14b shows the synthetic dependent task set with 20 tasks. The results reported here do not include the energy overhead optimization discussed in Section 5. We expect better results when



a. GSSR vs. SPM b. FLSSR vs. SPM

Figure 14: Energy Savings with Varied Time Overhead of Voltage/Speed Adjustment

this optimization is considered.

From the figures, the constant part of the overhead affects the algorithms' performance the most. With the maximum overhead considered, for independent tasks there is a 6% difference in energy consumption from the case with no overhead. For dependent tasks, the difference is 12%. There is a big jump between the case with no overhead and with minimal overhead. The reason is that without overhead the idle state runs at $0.1 \cdot S_{jit}$, and with overhead, the idle state runs at $S_{jit} = S_{max}$ (load=100%) to ensure that future tasks finish on time (see Section 5).

Note that C and K are dependent on specific processor hardware and the tasks running on the processor. Suppose that the minimum task has the worst case execution time of 10 ms, and we are using a Transmeta processor that takes 5 ms to change voltage/speed [16]. Hence, $C = 50\%$ and $K = 0$. Similarly, the AMD K6-2+ was measured to have an overhead of 0.4 ms to change voltage and 40 μs to change frequency [22]. Thus for AMD, $C = 4\%$ and $K = 0$. For the lpARM processor that needs 70 μs to change voltage [5], $C = 0.7\%$ and $K = 0$.

6.6 The Effect of Discrete Voltage/Speed Levels

To see how discrete voltage/speed levels affect the algorithms' performance in terms of energy savings, we set different levels between $200MHz$ and $700MHz$ (the speed is from Transmeta TM5400 [16]) and their corresponding supply voltage. The levels are uniformly distributed at the same increment between two discrete speed levels. The idle state runs at the minimum speed and consumes the corresponding energy. For GSSR, we run the task set with 100 tasks, and for FLSSR we run the synthetic task set with 20 tasks.

Here we set the number of processors as 2 and fix $\alpha = 0.5$. The energy consumption of GSSR and FLSSR vs. SPM with different number of voltage/speed levels is shown in Figure 15, where '∞' means continuous voltage/speed adjustment.

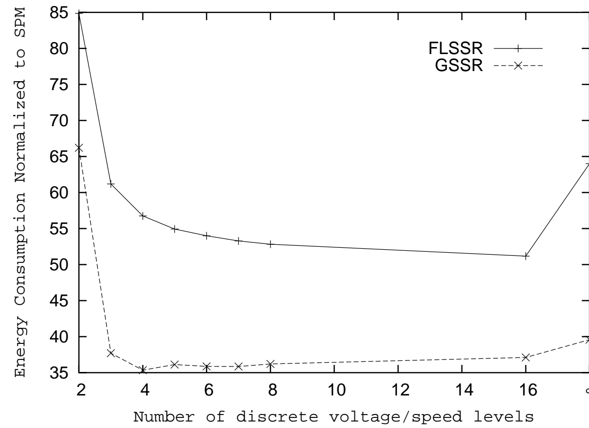


Figure 15: Energy Consumption of GSSR and FLSSR vs. SPM with different number of voltage/speed levels

Since the workload is 100%, there is no static slack and $S_{jit} = S_{max}$. For static power management, because the processors runs at either $200MHz$ or $700MHz$ for all speed configurations, the energy consumption is the same.

From Figure 15, we see that energy consumption of the algorithms with continuous adjustment is not always less than that with discrete voltage/speed levels, and more levels do not guarantee less energy consumption. The reason is that, with discrete voltage/speed levels, the processors set their speed to the next higher discrete level, which saves some slack for future tasks. When sharing the slack with future tasks, the energy consumption of the algorithms with discrete voltage/speed levels may be less than that with continuous adjustment, and a few levels may be better than many levels. In any case, 4-6 levels are sufficient to achieve the effect of continuous adjustment, which is the same observation as reported in [6] for uniprocessor with periodic tasks.

7 Summary

In this paper, we introduce the concept of *slack sharing* on multi-processor systems to reduce energy consumption. Based on this concept, we propose two novel power-aware scheduling algorithms for independent and dependent tasks. In both cases, we prove that scheduling with slack reclamation will not cause the execution of tasks to finish later than the completion time in canonical execution, where each task uses its worst

case execution time. Specifically, if canonical execution of a task set can finish before time D, then the two proposed algorithms, global scheduling with shared slack reclamation (GSSR) and fixed-order list scheduling with shared slack reclamation (FLSSR), will finish the execution of the tasks before D. Compared to static power management (SPM), Our simulation results show that GSSR and FLSSR achieve considerable energy saving when the task's execution time is smaller than their worst case execution time (which is true for most real applications). Using trace data from several real applications, such as automated target recognition [23] and Berkeley MPEG-1 encoder [12], the results show that our schemes can save up to 44% energy compared to SPM.

The effect of discrete voltage/speed on the performance of the algorithms is also studied. Our simulation results show that a few discrete voltage/speed levels are sufficient to achieve almost the same or better energy savings than continuous voltage/speed.

Finally, we propose a scheme to incorporate the voltage/speed adjustment overhead into our scheduling algorithms using *slack reservation*. Based on the assumption that it takes a few milliseconds to adjust processor supply voltage and speed [21], our simulation results show that the effect of the overhead on energy saving ranges from 6% to 12%.

Acknowledgments

This work has been supported by the Defense Advanced Research Projects Agency through the PARTS project (Contract F33615-00-C-1736). The authors would like to thank Dr. Daniel Mossé for useful discussion about the overhead in Section 5. We also thank the referees for their criticisms and suggestions that helped us in rewriting the paper in a better form.

References

[1] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proc. of Workshop on Compilers and Operating Systems for Low Power*, Barcelona, Spain, 2001.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of The* 22th *IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[4] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, pages 288–297, Maui, Hawaii, Jan. 1995.

[5] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35(11):1571–1580, 2000.

[6] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: An approach for energy efficient computing. In *Proc. Int'l Symposium on Low-Power Electronic Devices*, Monterey, CA, 1996.

[7] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuit*, 27(4):473–484, 1992.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. Technical report, Dept. of Computer Science, Rice University, Sep. 1998.

[9] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.

[10] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, 15(12):1497–1505, 1989.

[11] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of The International Conference on Computer-Aided Design*, pages 598–604, San Jose, CA, Nov. 1997.

[12] K. L. Gong and L. A. Rowe. Parallel mpeg-1 video encoding. In *Proc. of 1994 Picture Coding Symposium*, Sacramento, CA, Sep. 1994.

[13] F. Gruian. System-level design methods for low-energy architectures containing variable voltage processors. In *Proc. of The Workshop on Power-Aware Computing Systems (PACS)*, Cambridge, MA, Nov. 2000.

[14] C. H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Proc. of The Workshop on Power-Aware Computing Systems (PACS)*, Cambridge, MA, Nov. 2000.

[15] http://www.micorprocessor.sscc.ru.

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

[17] T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The International Symposium on Low Power Electronics and Design*, pages 197–202, Monterey, CA, Aug. 1998.

[18] C. M. Krishna and Y. H. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proc. of The* 6th *IEEE Real-Time Technology and Applications Symposium (RTAS00)*, Washington D.C., May. 2000.

[19] F. Liberato, S. Lauzac, R. Melhem, and D. Mossé. Fault-tolerant real-time global scheduling on multiprocessors. In *Proc. of The* 10th *IEEE Euromicro Workshop in Real-Time Systems*, York, UK, Jun. 1999.

[20] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Proc. of Workshop on Compiler and OS for Low Power*, Philadelphia, PA, Oct. 2000.

[21] W. Namgoong, M. Yu, and T. Meng. A high-efficiency variable-voltage cmos dynamic dc-dc switching regulator. In *Proc. of IEEE International Solid-State Circuit Conference*, pages 380–381, San Francisco, CA, Feb. 1997.

[22] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of* 18th *ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, Oct. 2001.

[23] J. A. Ratches, C. P. Walters, R. G. Buser, and B. D. Guenther. Aided and automatic target recognition based upon sensory inputs from image forming systems. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 19(9):1004–1019, 1997.

[24] P. M. Shriver, M. B. Gokhale, S. D. Briles, D. Kang, M. Cai, K. McCabe, S. P. Crago, and J. Suh. *A Power-Aware, Satellite-Based Parallel Signal Processing Scheme*, chapter 13, pages 243–259. Power Aware Computing. Plenum/Kluwer Publishers, 2002.

[25] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Design & Test of Computers*, 18(5):46–58, 2001.

[26] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The* 36th *Annual Symposium on Foundations of Computer Science*, pages 374–382, Milwaukee, WI, Oct. 1995.