

# 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

IBM Technical Contact - Patrick Bohrer, IBM Austin Research Laboratory

## Abstract

*The power consumption of modern high-performance processors is becoming a major concern because it leads to increased heat dissipation and decreased reliability. While many techniques have been proposed to reduce power consumption for uni-processors, there has been considerably less work on multi-processor systems. In this paper, we focus on power-aware scheduling for multi-processor real-time systems. Based on the idea of slack sharing among processors, we propose two novel scheduling algorithms for task sets with and without precedence constraints. 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. Simulation results indicate that our algorithms achieve up to 60% energy savings on multi-processor systems with variable voltage processors.*

## 1 Introduction

In recent years, processor performance has increased at the expense of drastically increased power consumption [14]. Thus heat dissipation has become a major problem because it requires more expensive packaging and cooling technology and decreases reliability [18], especially for multi-processor systems. In order to reduce heat dissipation and to increase reliability, many hardware and software techniques have been proposed to lower processor power consumption [1, 2, 4, 5, 7, 8, 10]. Processors running on multiple supply voltages (i.e., multiple power levels) have become available in recent years [15], making power management at the processor level possible. Using this feature, several software techniques have been proposed to adjust the supply voltage, especially in the area of mobile computing [6, 8, 11, 12], where devices are battery operated and have a restricted power budget. At the high end of computing systems, especially multi-processor systems, where

heat dissipation and reliability are major concerns, fewer techniques have been proposed for power management.

In uni-processor systems, several voltage/speed adjustment schemes have been explored and shown to dramatically save energy [4, 6, 8, 11, 12]. Optimal preemptive scheduling algorithms for independent tasks running on a single processor with variable speed are described in [4] and [11]. For multi-processor systems with fixed application sets and predictable execution time, static power management (SPM) can be accomplished by deciding beforehand the best supply voltage/speed for each processor [7]. However, to our knowledge, no results have been reported in the literature about dynamic adjustment of processor voltage and speed at run-time for real-time multi-processor systems.

Our work addresses dynamic processor supply voltage and speed adjustment for multi-processor real-time systems. Changing processor voltage/speed changes task execution time and affects the scheduling of tasks to processors. For real-time systems, this may cause a violation of 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. Simulation results show that our techniques save up to 60% of energy compared to static power management techniques.

This paper is organized in the following way. 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 algorithms for dependent tasks are proposed and proven to meet timing requirements. Simulation results are given in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

\*This work has been supported by the Defense Advanced Research Projects Agency through the PARTS project (Contract F33615-00-C-1736).

## 2 Models and Power Management

### 2.1 Energy Model

Processor power consumption is dominated by dynamic power dissipation  $P_d$ , which is given by:  $P_d = C_{ef} * V_{dd}^2 * f$ , where  $V_{dd}$  is the supply voltage,  $C_{ef}$  is the effective switched capacitance and  $f$  is the processor clock frequency. Processor speed  $S$ , represented by  $f$ , is almost linearly related to the supply voltage:  $f = k * (V_{dd} - V_t)^2 / V_{dd}$ , where  $k$  is constant and  $V_t$  is the threshold voltage. The energy consumption for a specific task is, thus, almost proportional to  $C_{ef} * V_{dd}^2$  [1, 2]. To decrease processor speed, we can reduce the supply voltage. This reduces processor power consumption cubically and reduces task energy consumption quadratically at the expense of linearly decreasing speed and increasing a task's latency.

### 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  $\Gamma = \{T_1, \dots, T_n\}$  is to execute within each frame and is to complete before the end of the frame. The precedence constraints among the tasks in  $\Gamma$  are represented by a graph  $G$ . Because of the periodicity of the schedule, we consider only the problem of scheduling  $\Gamma$  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 an 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  $(c_i^e, a_i^e)$ , where  $c_i^e$  is the estimated worst case execution time (WCET) and  $a_i^e$  is the actual execution time (AET), both based on maximal processor speed. We assume that for a task  $T_i$ , the value of  $c_i^e$  is known before execution, while  $a_i^e$  is determined at run time. The precedence constraints are represented by  $G = \{\Gamma, E\}$ , where  $E$  is a set of edges, such that 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 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 to 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 [6, 7]. 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 both dynamic supply voltage and speed

adjustment. 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 [9], any unused time can be considered as *slack* and can be reused by the remaining tasks to run slower while still finishing before  $D$  [6]. In this case, processor 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 just at time  $D$ . Our algorithms then apply dynamic voltage/speed adjustment. In the rest of the paper, we normalize worst case execution time and actual case execution time of task  $T_i$  such that,  $c_i = c_i^e * S_{max} / S_{jit}$  and  $a_i = a_i^e * S_{max} / S_{jit}$ . Task  $T_i$  will be characterized by  $(c_i, a_i)$ . In the following sections, we consider both independent and dependent tasks.

## 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 [3]. In the global scheduling strategy, all tasks are in a global queue and each processor selects from the queue the task with the highest priority for execution. In the partition scheduling strategy, each task is assigned to a specific processor and each processor selects a task for execution from its own queue.

In global scheduling, the priority of the tasks in the queue 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 [3]. 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 heuristics (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 very 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. We also number the tasks by their order in the global queue when using longest task first priority assignment. That is, the  $k^{th}$  task in the global queue is identified as  $T_k$ .

To emphasize the importance of task priority on scheduling, we consider the simple example of a task set executing on a dual-processor system shown as in Figure 1. Here,  $\Gamma = \{T_1, T_2, T_3, T_4, T_5\}$ ,  $T_1 = (10, 7)$ ,  $T_2 = (8, 4)$ ,  $T_3 = (6, 6)$ ,  $T_4 = (6, 6)$ ,  $T_5 = (6, 6)$ . Consider the canonical execution in

global scheduling and assume that  $D = 20$ . In the following 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. 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  $T_3 \rightarrow T_4 \rightarrow T_5 \rightarrow T_2 \rightarrow T_1$ , will miss the deadline.

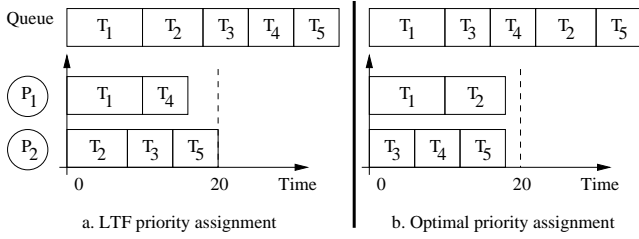


Figure 1. Global Scheduling for 2-Processor Systems

For partition scheduling, the assignment of tasks to processors to balance workload while achieving minimal execution time is also NP-hard [3] and must be handled by heuristics. In this paper, we use longest task first partitioning, which dispatches the longest task from the remaining tasks to the processor with the minimal workload (based on task’s WCET). For the above example, the optimal partitioning is  $\Gamma_1 = \{T_1, T_2\}$  and  $\Gamma_2 = \{T_3, T_4, T_5\}$ , where  $\Gamma_1$  is the sub-task set assigned to processor  $P_1$  and  $\Gamma_2$  is assigned to processor  $P_2$ . The LTF partition gives  $\Gamma_1 = \{T_1, T_4\}$  and  $\Gamma_2 = \{T_2, T_3, T_5\}$ . With partition scheduling, each processor can independently apply the slack reclamation and speed adjustment schemes discussed in [6].

First, we extend the *greedy slack reclamation (GSR)* scheme [6] 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 (SSR)*. To simplify the problem and our discussion, we assume that processor supply voltage and frequency can be changed continuously, and we do not consider the run-time overhead of changing processor supply voltage and speed. In the conclusion, we briefly discuss the implications of these two assumptions.

### 3.1 Global Scheduling with Greedy Slack Reclamation

This is an extension of the dynamic power management scheme for uni-processor systems from Mossé et al [6]. 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.

Consider the example:  $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$ ,  $D = 9$ ,  $T_1 = (5, 2)$ ,  $T_2 = (4, 4)$ ,  $T_3 = (3, 3)$ ,  $T_4 = (2, 2)$ ,

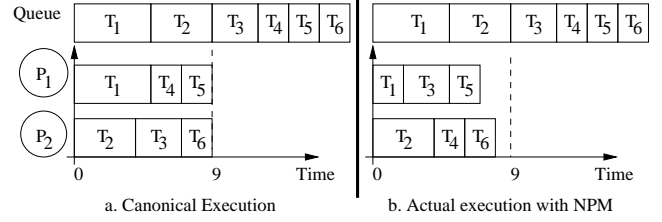


Figure 2. Global Scheduling with No Power Management

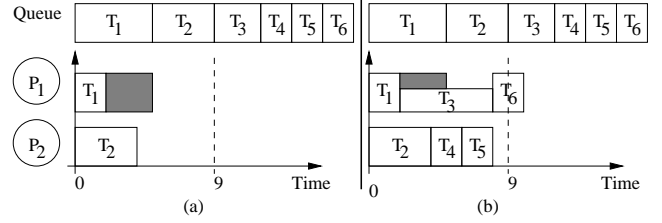


Figure 3. Global Scheduling with Greedy Slack Reclamation

$T_5 = (2, 2)$ ,  $T_6 = (2, 2)$ . 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, execution with actual execution time (AET) can finish before  $D$ . While Figure 3 shows that when tasks use their actual execution time,  $T_1$  finishes at time 2 with a slack of 3 time units. With GSR, 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  $3/6 * S_{jit}$  accordingly. When  $T_3$  uses up its time,  $T_6$  misses the deadline  $D$ . 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)

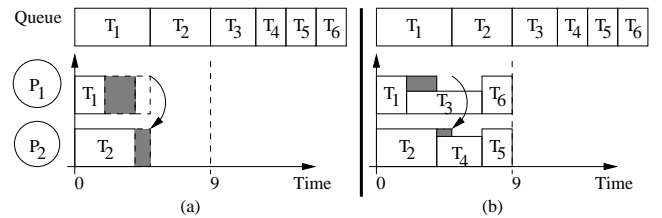


Figure 4. Global Scheduling with Shared Slack Reclamation

In the above example, 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  $3/6 * S_{jit}$  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 *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  $3/5 * S_{jit}$  and ends at time 7.  $T_4$  starts at time 4, executes for 3 time units at the speed of  $2/3 * S_{jit}$  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*.

Before formally presenting our 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 this processor.

### 3.2.1 GSSR for Dual-Processor Systems (GSSR-2)

Each processor invokes the scheduling algorithm in Figure 5 at the beginning of execution or when a processor finishes executing a task. A shared memory is used to hold control information, which must be updated within a critical section (not shown in the algorithm). The shared memory holds the common queue, *Ready-Q*, which contains all *ready* tasks and an array to record  $STNT_1$  and  $STNT_2$  for processor  $P_1$  and  $P_2$ , respectively. Initially, all tasks are put into *Ready-Q*, and the  $STNTs$  of the processors are set to 0. In the algorithm,  $id$  represents the current processor while  $\bar{id}$  represents the other 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; otherwise,  $P_{id}$  will select the next task  $T_k$  from *Ready-Q* (line 3). If  $STNT_{id}$  is larger than  $STNT_{\bar{id}}$ , it means that  $T_k$  should follow  $STNT_{\bar{id}}$  in the canonical execution, so  $P_{id}$  exchanges  $STNT_{id}$  with  $STNT_{\bar{id}}$  to make  $T_k$  still follow  $STNT_{\bar{id}}$  (line 4). 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  $STNT_{\bar{id}}$ ,  $P_{id}$  shares part of its slack (specifically,  $STNT_{id} - STNT_{\bar{id}}$ ) with  $P_{\bar{id}}$ .

From the algorithm, we notice that at any time (except when

```

1 While (Ready-Q ≠ ∅)
2 {
3    $T_k = Dequeue(Ready-Q)$ ;
4   if ( $STNT_{id} > STNT_{\bar{id}}$ )
5      $STNT_{id} \leftrightarrow STNT_{\bar{id}}$ ;
6    $EET_k = STNT_{id} + c_k$ ;
7    $STNT_{id} = EET_k$ ;
8    $S_{id} = S_{jit} * c_k / (EET_k - t)$ ;
9   Execute  $T_k$  at speed  $S_{id}$ ;

```

Figure 5. The GSSR-2 Algorithm invoked by  $P_{id}$

*Ready-Q* is empty), the value of  $STNT_1$  and  $STNT_2$  of the processors are always equal to the biggest two  $EETs$  of the tasks running on the two processors. One of these two tasks is the most recently started task (from line 4, 5, 6). The task that starts next will follow the relatively smaller  $STNT$ . These properties are used to prove the algorithm's correctness (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 in the canonical execution).

### 3.2.2 Analysis of the GSSR-2 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  $EET_k^c$  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 the function  $max_2\{X_1, \dots, X_n\} = \{X_a, X_b\}$ , where  $X_a$  and  $X_b$  are the two largest elements in the set  $\{X_1, \dots, X_n\}$ . 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-2, at any time  $t$ , if  $T_k$  is the most recently started task, then:

$$EET_k \in max_2\{EET_i | T_i \in H(t)\};$$

Moreover:

$$\{STNT_1, STNT_2\} = max_2\{EET_i | T_i \in H(t)\}.$$

**Proof** The proof is by induction on  $T_k$ ,  $k = 1, \dots, n$ .

Initially, after  $T_1$  and  $T_2$  start execution and before any of them finish, at any time  $t$ :

$$H(t) = \{T_1, T_2\};$$

$$EET_2 \in max_2\{EET_i | T_i \in H(t)\}; \text{ and}$$

$$\{STNT_1, STNT_2\} = max_2\{EET_i | T_i \in H(t)\}.$$

Assuming that, at any time  $t$ ,  $T_{k-1}$  is the most recently started task, we have:

$$\begin{aligned} H(t) &= \{T_1, \dots, T_{k-1}\}; \\ EET_{k-1} &\in \max_2\{EET_i | T_i \in H(t)\}; \\ \{STNT_1, STNT_2\} &= \max_2\{EET_i | T_i \in H(t)\}; \end{aligned}$$

And without loss of generality, assume  $STNT_1 \leq STNT_2$ .

After  $T_k$  started and before  $T_{k+1}$  starts, at any time  $t$ ,  $T_k$  is the most recently started task. Hence  $H(t) = \{T_1, \dots, T_k\}$  and from line 4, 5, 6 of the algorithm:

$$\begin{aligned} EET_k &= \min\{STNT_1, STNT_2\} + c_k \\ &= STNT_1 + c_k; \end{aligned}$$

Then,

$$EET_k \in \max_2\{EET_i | T_i \in H(t)\};$$

The new values of  $STNT_1$  and  $STNT_2$  are thus given by:

$$\begin{aligned} \{STNT_1, STNT_2\} &= \{STNT_2, EET_k\} \\ &= \max_2\{EET_i | T_i \in H(t)\}; \end{aligned}$$

□

**Theorem 1** *For global scheduling with shared slack reclamation in 2-processor systems (GSSR-2), if canonical execution completes at a time  $D$ , then any execution will complete by time  $D$ .*

**Proof** We prove this theorem by showing that, for any execution under GSSR-2:  $EET_i = EET_i^c$ . The proof is by induction on  $T_k$ ,  $k = 1, \dots, n$ .

Initially, GSSR-2 sets  $EET_1$  and  $EET_2$  at the beginning of execution without any consideration to the actual execution time of  $T_1$  and  $T_2$ . Hence,  $EET_1 = EET_1^c$  and  $EET_2 = EET_2^c$ .

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

$$\max_2\{EET_i | T_i \in H(t)\} = \{EET_{k-a}, EET_{k-1}\}, a > 1.$$

From Lemma 1:

$$\{STNT_1, STNT_2\} = \{EET_{k-a}, EET_{k-1}\}.$$

When  $T_k$  begins to run:

$$\begin{aligned} EET_k &= \min(STNT_1, STNT_2) + c_k \\ &= \min(EET_{k-a}, EET_{k-1}) + c_k; \\ EET_k^c &= \min(STNT_1, STNT_2) + c_k \\ &= \min(EET_{k-a}^c, EET_{k-1}^c) + c_k; \end{aligned}$$

Thus, whether  $EET_{k-a} > EET_{k-1}$  or  $EET_{k-a} < EET_{k-1}$ , we have:

$$EET_k = EET_k^c.$$

So,  $EET_i = EET_i^c$ ,  $i = 1, \dots, n$ .

□

Consider the example from Figure 1 when every task uses its actual execution time. Assuming that power consumption is equal to  $S^3 * C_{ef}$ , if no slack is reclaimed, the energy consumption is computed to be  $29 * C_{ef}$ . Under global scheduling with shared slack reclamation and longest task first priority assignment, the energy consumption is computed to be  $21.83 * C_{ef}$ . 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  $21.97 * C_{ef}$ . Hence, the optimal priority assignment in terms of execution time is not optimal for energy consumption.

### 3.2.3 GSSR for N (> 2) Processors Systems (GSSR-N)

Global scheduling with shared slack reclamation can be extended to  $N$ -processor systems as shown in Figure 6. The difference between this algorithm and GSSR-2 is how the minimum  $STNT$  is determined (line 4). The extended algorithm can be proved to meet the finish time of canonical execution by replacing  $\max_2$  with  $\max_N$  in the proof of Lemma 1 and Theorem 1, where  $\max_N\{H(t)\}$  is the set of the  $N$ -largest elements from the set  $H(t)$ . For brevity, the proof is omitted.

```

1 While (Ready-Q ≠ ∅)
2 {
3   Tk = Dequeue(Ready-Q);
4   Find Pr such that:
      STNTr = min{STNT1, ..., STNTn};
5   if(STNTid > STNTr)
      STNTid ↔ STNTr
6   EETk = STNTid + ck;
7   STNTid = EETk;
8   Sid = Sjit * ck / (EETk - t);
9   Execute Tk at speed Sid;
10 }
```

Figure 6. The GSSR-N Algorithm invoked by  $P_{id}$

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

## 4 Power-Aware Scheduling for Dependent Tasks

List scheduling is a standard technique used to schedule task sets with precedence constraints [3, 13]. 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 order that minimizes execution time is NP-hard [3]. In this section, we use the same heuristic as in global scheduling and put into the ready queue first the longest (based on WCET) among the tasks that become ready simultaneously. We number the tasks by the order at which they are added to the ready queue during canonical execution. That is, the  $k^{th}$  task entering the ready queue in canonical execution is identified as  $T_k$ .

An example of canonical execution with list scheduling is shown in Figure 7. Here  $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$ ,  $D = 12$ . The precedence graph is shown in Figure 7 (a) and the canonical execution is shown in Figure 7 (b). Task nodes are labeled with the tuple  $(c_i, a_i)$ .

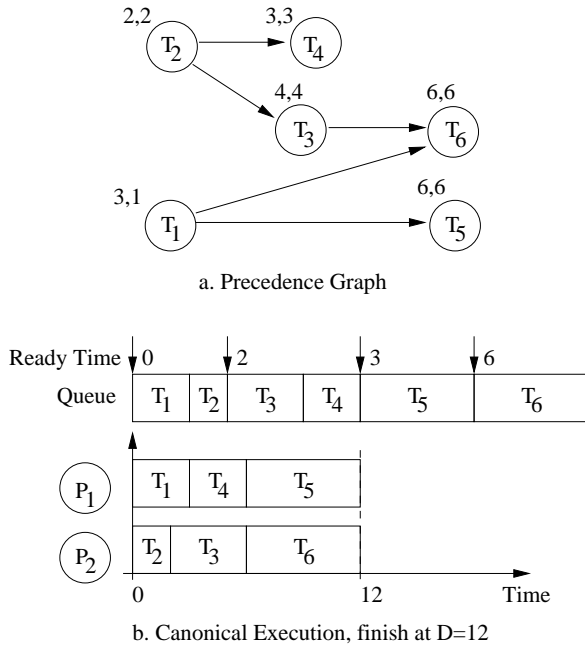


Figure 7. List Scheduling for Dual-Processor Systems

From the figure, we see that  $T_1$  and  $T_2$  are 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, the *readiness* of a task during non-canonical execution is dependent on the actual execution time 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 may not guarantee that timing constraints are met.

#### 4.1 List Scheduling with Shared Slack Reclamation

Consider the example from Figure 7 and assume that every task uses its actual execution time. In Figure 8, whenever one task is ready it is put into the queue. From Figure 8, it is clear that list scheduling with shared slack reclamation does not finish execution by time 12 (the finish time of canonical execution).

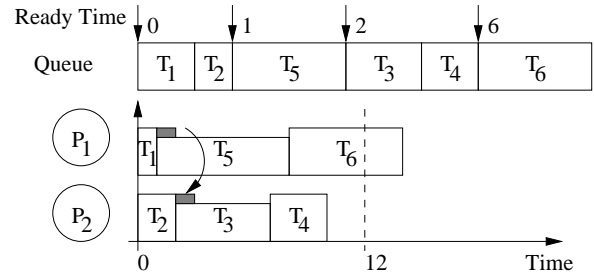


Figure 8. List Scheduling with Shared Slack Reclamation

The reason why list scheduling with shared slack reclamation takes longer than the canonical execution is that the ready time of the tasks change, and 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 (LSSR)

For the example in Figure 8, 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. Recall that the execution order of tasks has been known during the first step when checking the canonical execution, so we can put all tasks into *Global-Q* in the canonical execution order. Then 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 select it; otherwise the processor goes to sleep. The detail of the algorithm is described below.

##### 4.2.1 LSSR for Dual-Processor Systems (LSSR-2)

As in GSSR-2, we assume that the shared memory holds the control information. Figure 9 shows the LSSR-2 algorithm. Each processor ( $P_{id}$ ) invokes the LSSR-2 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 the function *signal( $P_k$ )* to wake processor  $P_k$ .

In the algorithm,  $RT_i^c$  is the ready time of task  $T_i$  during canonical execution at speed  $S_{jit}$  and all other variables are the same as before.  $RT_i^c (i = 1, \dots, n)$  are calculated as:  $RT_i^c = \max\{EET_k^c | T_k \rightarrow T_i \in E\}$ . Initially, all tasks are put in *Global-Q* in the canonical execution order.  $STNT_{id} (id = 1, 2)$  are set to 0 (not shown in the following figure).

```

1  if (Head(Global-Q) is ready)
2  {
3       $T_k = Dequeue (Global-Q)$ ;
4      if ( $STNT_{id} > STNT_{\bar{id}}$ )
           $STNT_{id} \leftrightarrow STNT_{\bar{id}}$ 
5       $EET_k = \max\{RT_k^c, STNT_{id}, t\} + c_k$ ;
6       $STNT_{id} = EET_k$ ;
7       $S_{id} = S_{jit} * c_k / (EET_k - t)$ ;
8      if ((Head(Global-Q) is ready) AND ( $P_{\bar{id}}$  is Idle))
9          Signal( $P_{\bar{id}}$ );
10     Execute  $T_k$  at speed  $S_{id}$ ;
11 } else wait();

```

Figure 9. The LSSR-2 Algorithm invoked by  $P_{id}$

If the algorithm is invoked by a signal from another processor, it will begin at the 'waiting for signal' point (line 11). If the algorithm is invoked at the beginning or when  $P_{id}$  finishes a task, it begins at line 1. If the head of *Global-Q* is ready,  $P_{id}$  picks a task  $T_k$  from the head of *Global-Q* (line 3). After selecting  $T_k$ ,  $P_{id}$  calculates the speed  $S_{id}$  to execute  $T_k$ . Then  $P_{id}$  signals  $P_{\bar{id}}$  if  $P_{\bar{id}}$  is sleeping and the head of *Global-Q* is ready (line 11). Finally,  $P_{id}$  runs  $T_k$  at the speed of  $S_{id}$ .

Based on this algorithm, we prove that at any time  $STNT_1$  and  $STNT_2$  of the processors are always equal to the largest two *EET*s of the tasks that have been started or finished, and one of them will be the task most recently started.

#### 4.2.2 Analysis of LSSR-2 Algorithm

Similar to GSSR-2, at any time (except when *Global-Q* is empty), the value of  $STNT_1$  and  $STNT_2$  of the processors are always equal to the biggest two *EET*s of the tasks running on the two processors. One of these two tasks is the most recently started task (from line 4, 5, 6). The task that starts next will follow the relatively smaller *STNT*.

**Lemma 2** For LSSR-2, at any time  $t$ , if  $T_k$  is the most recently started task, then:

$$EET_k \in \max_2\{EET_i | T_i \in H(t)\};$$

Moreover:

$$\{STNT_1, STNT_2\} = \max_2\{EET_i | T_i \in H(t)\}.$$

**Proof** The proof is by induction on  $T_i, i = 1, \dots, n$  and is very similar to the proof of Lemma 1.

Initially, after  $T_1$  and  $T_2$  start execution and before any of them finish, at any time  $t$ :

$$H(t) = \{T_1, T_2\};$$

$$EET_2 \in \max_2\{EET_i | T_i \in H(t)\}; \text{ and}$$

$$\{STNT_1, STNT_2\} = \max_2\{EET_i | T_i \in H(t)\}$$

Assume that before  $T_k$  started execution,  $T_{k-1}$  is the most recently started task. At any time  $t$ , we have:

$$H(t) = \{T_1, \dots, T_{k-1}\};$$

$$EET_{k-1} \in \max_2\{EET_i | T_i \in H(t)\}; \text{ and}$$

$$\{STNT_1, STNT_2\} = \max_2\{EET_i | T_i \in H(t)\};$$

And without loss of generality, assume  $STNT_1 \leq STNT_2$ .

After  $T_k$  starts and before any more tasks finish,  $T_k$  is the most recently started task, and at any time  $t$ :

$$H(t) = \{T_1, \dots, T_k\};$$

From line 4, 5 and 6 of the algorithm in Figure 9:

$$\begin{aligned} EET_k &= \max\{\min\{STNT_1, STNT_2\}, RT_k^c, t\} + c_k \\ &= \max\{STNT_1, RT_k^c, t\} + c_k; \end{aligned}$$

Then,

$$EET_k \in \max_2\{EET_i | T_i \in H(t)\};$$

The new values of  $STNT_1$  and  $STNT_2$  are thus given by:

$$\begin{aligned} \{STNT_1, STNT_2\} &= \{STNT_2, EET_k\} \\ &= \max_2\{EET_i | T_i \in H(t)\}; \end{aligned}$$

□

**Theorem 2** For list scheduling with shared slack reclamation in 2-processor systems (LSSR-2), if canonical execution completes at a time  $D$ , then any execution will complete by time  $D$ .

**Proof** We prove this theorem by showing that, for any execution of LSSR-2:  $EET_i = EET_i^c, i = 1, \dots, n$ . The proof is by induction on  $T_i, i = 1, \dots, n$ . Recall that tasks are numbered by the order in which they entered *Global-Q* during canonical execution and they are always executed in the canonical execution order.

Initially, LSSR-2 sets  $EET_1$  and  $EET_2$  at the beginning of execution without any consideration to the actual execution time of  $T_1$  and  $T_2$ . Hence,  $EET_i = EET_i^c$ , for  $i = 1, 2$ .

Assume that  $EET_i = EET_i^c, i = 1, \dots, k-1$ . Without loss of generality, at any time before  $T_k$  starts,  $T_{k-1}$  is the most recently started task and

$$\max_2\{EET_i | T_i \in H(t)\} = \{EET_{k-a}, EET_{k-1}\}, a > 1.$$

From Lemma 2:

$$\{STNT_1, STNT_2\} = \{EET_{k-a}, EET_{k-1}\}.$$

When  $T_k$  starts at time  $t$  (non-canonical execution) or  $t'$  (canonical execution):

$$\begin{aligned} EET_k &= \max\{\min\{STNT_1, STNT_2\}, RT_k^c, t\} + c_k \\ &= \max\{\min\{EET_{k-a}, EET_{k-1}\}, RT_k^c, t\} + c_k; \\ EET_i^c &= \max\{\min\{STNT_1, STNT_2\}, RT_k^c, t'\} + c_k \\ &= \max\{\min\{EET_{k-a}^c, EET_{k-1}^c\}, RT_k^c, t'\} + c_k; \end{aligned}$$

When  $T_k$  starts, either

$$t < RT_k^c \text{ and } t' < RT_k^c$$

or

$$t < \min\{STNT_1, STNT_2\} \text{ and}$$

$$t' < \min\{STNT_1, STNT_2\}$$

we will have:

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

Thus  $EET_k = EET_k^c$ .

So,  $EET_i = EET_i^c, i = 1, \dots, n$ .

□

### 4.2.3 LSSR for N (> 2) Processor Systems (LSSR-N)

The LSSR-2 algorithm may be extended to any number of processors by checking all processors and getting the minimal  $STNT$  as shown on line 4 in Figure 10. The minimal  $STNT$  is the time the next start task should follow. If in the canonical execution, LSSR-N finishes before  $D$ , LSSR-N will finish before  $D$  for any case. The proof is similar to the one above and is omitted for brevity.

## 5 Performance Comparison

In this section, we empirically demonstrate how shared slack reclamation reduces energy consumption. We compare the energy consumed when using the combination of static power management and dynamic supply voltage/speed adjustments by shared slack reclamation with that when using only static power management.

### 5.1 GSSR and Partition Scheduling with Greedy Slack Reclamation vs. SPM

First, we describe the simulation experiments. To get the actual execution time for each task, we define  $\alpha_i$  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  $\alpha_i * c_i$ . For independent task sets, we specify the lower ( $c_{min}$ ) and

```

1 if (Head(Global-Q) is ready)
2 {
3    $T_k = Dequeue (Global-Q)$ ;
4   Find  $P_r$  such that:
5      $STNT_r = \min\{STNT_1, \dots, STNT_n\}$ ;
6   if ( $STNT_{id} > STNT_r$ )
7      $STNT_{id} \leftrightarrow STNT_r$ 
8    $EET_k = \max\{RT_k^c, STNT_{id}, t\} + c_k$ ;
9    $STNT_{id} = EET_k$ ;
10   $S_{id} = S_{jit} * c_k / (EET_k - t)$ ;
11  if ((Head(Global-Q) is ready) AND ( $P_m$  is Idle))
12     $Signal(P_m)$ ;
13  Execute  $T_k$  at speed  $S_{id}$ ;
14 } else wait();

```

Figure 10. The LSSR-N Algorithm invoked by  $P_{id}$

upper ( $c_{max}$ ) bounds on the task's WCET and the average  $\alpha$  for the tasks in the set, which reflects the amount of dynamic slack in the system. The higher the value of  $\alpha$ , the less the dynamic slack. A task's WCET is generated randomly between ( $c_{min}, c_{max}$ ) and  $\alpha_i$  is generated as a uniform distribution around  $\alpha$ . For simplicity, energy consumption is assumed to be proportional to  $\int S^3(t) dt * C_{ef}$  and the idle speed is set to  $0.1 * S_{max}$  when the processor is idle. In the following experiments, energy is normalized to the energy consumed when using only SPM.

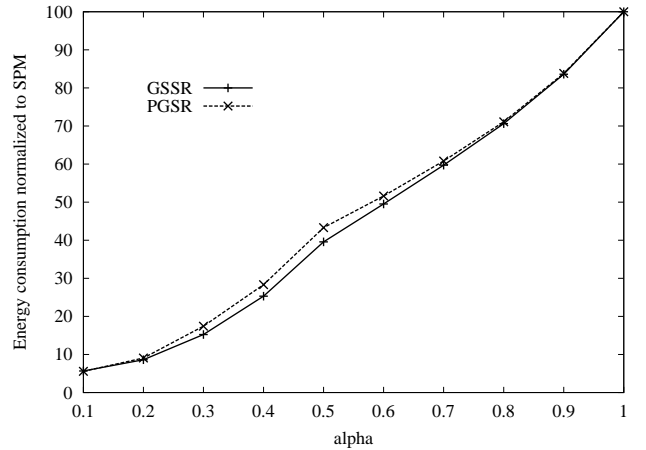


Figure 11. Energy Consumed by GSSR and PGSR vs. SPM

The results reported in this section are obtained by running the task set 100 times. There are 100 tasks in the task set. We set the value of WCET as  $c_{min} = 1$  and  $c_{max} = 50$ . In Figure 11, the number of processor is 2.  $\alpha$  is varied from 0.1 to 1.0. We compare the global scheduling with shared slack reclamation with partition scheduling with greedy slack reclamation (PGSR). For PGSR, we use longest task first partitioning to divide the tasks among processors, and then apply greedy slack reclamation scheme on each processor [6]. Shared slack

reclamation is one form of *greedy*. 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, with slack sharing longer tasks get more slack while short tasks get less. This balances the speed for each task and reduces energy consumption. When the average/worst case ratio( $\alpha$ ) is about 0.5 (that is, on the average we have 50% of time as dynamic slack), global scheduling with shared slack reclamation results in energy saving of more than 60% versus static power management. When  $\alpha$  increases, there is less dynamic slack and compared to SPM the energy saving of GSSR decreases.

To see the shared slack reclamation scheme's performance on systems with different number of processors, in Figure 12, we change the number of processors and set  $\alpha = 0.5$ . Compared to SPM, the energy saving of GSSR is almost the same when the processor number is less than or equal to 8. Because of lack of parallelism when the processor number is more than 8, the energy saving of GSSR decreases dramatically.

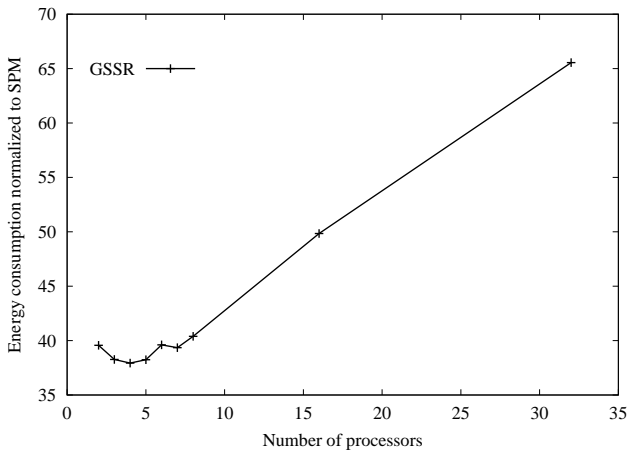


Figure 12. Energy Consumption of GSSR with Different Number of Processors

## 5.2 LSSR vs. SPM

For dependent tasks, we first consider an example with 20 tasks. The dependence graph for these tasks is generated randomly and shown in Figure 13 (a). The tasks' WCET are generated randomly from 1 to 50 and we assume a 2-processor system. In Figure 14, we vary  $\alpha$  from 0.1 to 1.0. The energy saving of fixed-order list scheduling with shared slack reclamation compared to that from static power management varies from 0% when  $\alpha$  is 1.0 to 72% when  $\alpha$  is 0.1. When  $\alpha$  increases, there is less dynamic slack and compared to SPM the energy saving of LSSR decreases. On average, when  $\alpha$  is 0.5, the energy saving is approximately 40%.

We next consider two matrix applications, matrix-multiplication and Gaussian-elimination, and measure the ef-

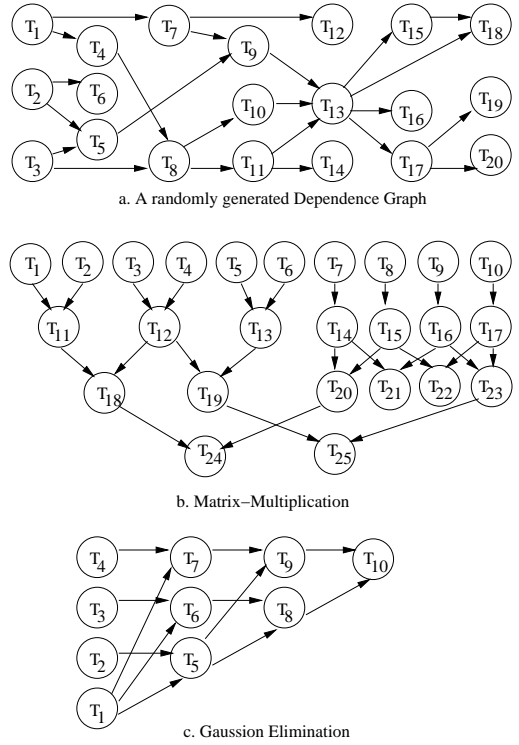


Figure 13. Dependent Graph

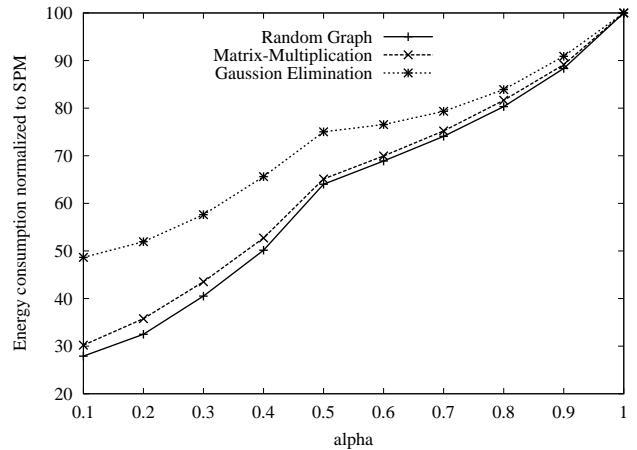
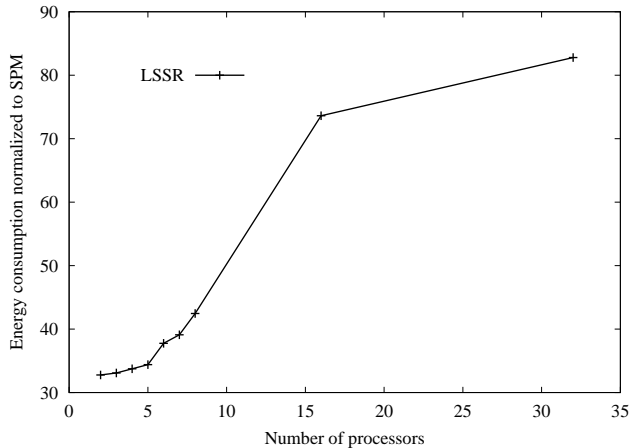


Figure 14. Energy consumed by LSSR vs. SPM

fectiveness of our techniques for these benchmarks. The dependence graph for matrix-multiplication is shown in Figure 13 (b) and Gaussian-elimination (assuming a  $5 \times 5$  matrix) is shown in Figure 13 (c) [16]. 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 also shown in Figures 14.

For Gaussian-elimination, we also considered a  $20 \times 20$



**Figure 15. Energy Consumption of LSSR with Different Number of Processors for Gaussian-elimination**

matrix to allow more parallelisms between tasks. With fixed  $\alpha = 0.5$ , we change the number of processors, and show the result in Figure 15. For this application, when the number of processor is larger than 8, most processors become idle (due to the lack of parallelism among tasks) and compared to SPM the energy consumption of LSSR increases dramatically.

## 6 Related Work

For uni-processor systems, Yao et al describe an optimal preemptive scheduling algorithm for independent tasks running with variable speed [4]. 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 [11]. By assigning lower voltages to the tasks with more switch activities and high 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 [12]. Their algorithms have an offline phase in which voltage is set for each task based on a task’s WCET and an on-line phase which adjusts the voltage on-the-fly to reclaim slack from prior tasks. Hsu et al describe a performance model to determine the efficient processor slow down factor under compier control [8]. Based on a superscalar target architecture and a machine with similar power dissipation to the Transmeta Crusoe TM5400, their simulation results show the potential of their proposed optimization technique. Mosse et al proposed and analyzed several techniques to dynamically adjust processor speed with slack reclamation [6]. The best scheme is an adaptive one that takes an aggressive approach while providing safeguards that avoid violating application deadline. For multi-processor systems, with fixed application sets and predictable execution time, static power management can be accomplished

by deciding beforehand the best supply voltage/speed for each processor. Flavius proposed two system-level designs for low-energy on architecture with variable voltage processors, and the simulation results show that both approach can save 50% of energy when deadline is relaxed by 50% [7].

Most of this previous work focused on uni-processor systems, and only a few focused on multi-processor systems. The work reported in this paper focuses on multi-processor systems with dynamic power management, which is different from static power management [7]. Our techniques are particularly beneficial for super-dense servers in which heat dissipation can adversely affect system cost and reliability.

## 7 Summary and Future Work

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 the canonical execution, where each task uses its worst case execution time. Our simulation results show that the scheduling algorithm with shared slack reclamation result in substantial energy saving compared to static power management.

Specifically, if canonical execution of a task set can finish before time  $D$ , then the two proposed algorithms, GSSR and LSSR, will finish the execution of the tasks before  $D$ . We show that, compared to static power management, GSSR and LSSR achieve considerable energy saving when the task’s execution time is smaller than their worst case execution time. With lower energy consumption, GSSR and LSSR potentially increase the reliability of the system.

Note that the energy results reported in this paper are based on the assumption that the processor supply voltage and speed can be scaled and changed *continuously*. In current processors, however, the processor voltage and speed can only be changed in incremental steps. The algorithms presented in this paper can be easily adapted to discrete voltage and speed levels. Specifically, after calculating a given CPU speed, setting the speed to the next higher discrete CPU speed will always guarantee that the deadline is met. The energy consumption, however, may be slightly higher than the one obtained from our algorithm with continuous voltage and speed. We are currently modifying our techniques and simulation infrastructure to account for discrete voltages and speeds.

The results reported do not account for the overhead of adjusting processor speed and supply voltage. However, in all of our schemes, speed adjustment is done only when the context switches between two tasks, and thus speed adjustment overheads can be added to the context switch overhead. With the knowledge that it takes only a few hundred cycles to adjust the processor speed and supply voltage [17], we do not expect GSSR and LSSR to increase the overhead substantially.

## References

- [1] T. D. Burd and R. W. Brodersen. Energy Efficient CMOS Micro-processor Design. *Proc. HICSS Conference*, pp. 288-297, Maui, Hawaii, January 1995
- [2] A.Chandrakasan, S.Sheng and R.Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-state circuit*, pp. 473-484, April 1992.
- [3] M.L.Dertouzos and A.K.Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, SE-15 (12): 1497-1505, 1989
- [4] F.Yao, A.Demers and S.Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science*. pages 374-382, Milwaukee, Wisconsin, October. 1995
- [5] T.D. Burd and R.W. Brodersen. Processor design for portable systems *Journal of VLSI Signal Processing*, 13(2/3):203-222, August 1996
- [6] D. Mossé, H. Aydin, B. Childers and R. Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications, *Workshop on Compiler and OS for Low Power*, Philadelphia , PA, October 2000
- [7] G. Flavius. System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors. *The Power-Aware Computing Systems 2000 Workshop at ASPLOS 2000*, Cambridge, MA, November 2000.
- [8] C.H.Hsu, U.Kremer and M.S. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. *Proceedings of Workshop on Power-Aware Computer Systems (PACS)*, Cambridge, MA, November 2000
- [9] R.Ernst and W.Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)97*. pp. 58-604. San Jose, CA, November 1997.
- [10] I.Hong, G.Qu, M.Potkonjak and M.Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Tasks on Variable Voltage Processors. In *Proceeding of 19th IEEE RTSS98*, Madrid, December 1998
- [11] T.Ishihara and H.Yauura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pp.197 – 202, Monterey, CA, August 1998.
- [12] C.M. Krishna and Y.H.Lee. Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-time Systems. In *Proceeding of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS00)*, Washington D.C., May 2000
- [13] K.D.Cooper, P.J.Schielke and D.Subramanian. An Experimental Evaluation of List Scheduling. *TR98-326*, Dept. of Computer Science, Rice University, September 30, 1998
- [14] <http://www.microprocessor.sccc.ru/>
- [15] <http://www.transmeta.com/crusoe/family.html>
- [16] M.Cosnard and D.Trystram. Parallel Algorithms and Architectures, *International Thomson Computer Press*, 1995.
- [17] W.Namgoong, M.Yu and T.Meng. A High-Efficiency variable-voltage CMOS dynamic DC-DC switching regulator. In *IEEE International Solid-State Circuits Conference* pp. 380-381, San Francisco, February 1997
- [18] D.Singh and V. Tiwari. Power Challenges in the Internet World. *Cool Chips Tutorial in conjunction with the 32<sup>nd</sup> Annual International Symposium on Micro-architecture*, Haifa, Israel, November 1999
- [19] F. Liberato, S. Lauzac, R. Melhem and D. Mosse. Fault Tolerant Real-Time Global Scheduling on Multiprocessors, *Euromicro Workshop in Real-Time Systems*, York, England June 1999

## Appendix

In this appendix we define the acronyms used in this paper.

**AET** : actual execution time of a task

**EET** : estimated end time of a task's execution

**GSR** : greedy slack reclamation

**GSSR** : global scheduling with shared slack reclamation

**LTF** : longest task first

**LSSR** : fixed-order list scheduling with shared slack reclamation

**PGSR** : partition scheduling with greedy slack reclamation

**RT** : ready time of a task

**SPM** : static power management

**SSR** : shared slack reclamation

**STNT** : start time of next task on a processor

**WCET** : worst case execution time of a task