Energy-Aware Fault Tolerance in Fixed-Priority Real-Time Embedded Systems*

Ying Zhang, Krishnendu Chakrabarty and Vishnu Swaminathan
Department of Electrical & Computer Engineering
Duke University, Durham, NC 27708, USA

Abstract

We investigate an integrated approach to fault tolerance and dynamic power management in real-time embedded systems. Fault tolerance is achieved via checkpointing and power management is carried out using dynamic voltage scaling (DVS). We present feasibility-of-scheduling tests for checkpointing schemes for a constant processor speed as well as for variable processor speeds. DVS is then carried out on the basis of these feasibility analyses. Experimental results show that compared to fault-oblivious methods, the proposed approach significantly reduces power consumption and guarantees timely task completion in the presence of faults.

1. Introduction

Fault tolerance techniques are needed to ensure the dependability of embedded systems that operate in harsh environmental conditions. These embedded systems also operate under severe energy limitations. In addition, many embedded systems execute real-time applications that require strict adherence to task deadlines. In this paper, we investigate an integrated approach that provides fault tolerance and dynamic power management (DPM) in hard real-time embedded systems. We extend a recent energy-aware adaptive checkpointing scheme that considers a single task in a soft real-time system [1].

Dynamic voltage scaling (DVS) is a popular technique for reducing power consumption during system operation [2, 3]. Fault tolerance is typically achieved in real-time systems through checkpointing [4]. At each checkpoint, the system saves its state in a secure device. When a fault is detected, the system rolls back to the most recent checkpoint and resumes normal execution. The checkpointing interval, i.e., duration between two consecutive checkpoints, must be carefully chosen to balance checkpointing cost with the re-execution time.

DPM and fault tolerance for embedded real-time systems have largely been studied as separate problems in the literature. DVS techniques for power management do not consider fault tolerance [2, 3], and checkpoint placement strategies for fault tolerance do not address DPM [5, 6]. It is only recently that an attempt has been made to combine fault tolerance with DPM [1].

There are three main reasons for combining DPM with fault tolerance in real-time embedded systems. Increased die temperatures due to higher processor speeds create thermal stresses on the die and undermine system reliability. In order to mitigate reliability problems caused by high die temperatures, we can either lower energy consumption through DPM techniques such as DVS, or we can adopt fault tolerance techniques such as checkpointing. Better still, a combination of DVS and checkpointing can be used.

The second reason is motivated by the need to meet the task deadlines in real-time systems. If faults occur frequently, the processor speed can be scaled up dynamically (within limits imposed by higher die temperatures) and more slack can be provided to the task, which allows more time for rollback recovery.

The third motivation arises from shrinking process technologies in the nanotechnology realm. Lower processor voltages are likely to lead to lower noise margins and more transient faults, caused in part by single-event upsets.

We first present feasibility tests for fixed-priority real-time systems with checkpointing under constant processor speed. Following this, we extend these feasibility tests to variable-speed processors. Based on the results of the feasibility analyses, an on-line dynamic speed-scaling scheme is further developed to reduce energy during task execution. The proposed approach is compared with a fault-oblivious DVS scheme in the presence of faults.

2. Feasibility Analysis Under Constant Speed

We are given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$. The elements of the tuple are defined as follows: T_i is the period of τ_i, and D_i is its deadline ($D_i \le T_i$); E_i is the execution time of τ_i under fault-free conditions. Let the checkpointing cost be C. We make the following assumptions related to task execution and fault arrivals: (i) The task set Γ is scheduled using fixed-priority methods such as the rate-monotonic scheme [7]; (ii) the task set Γ is schedulable under fault free conditions; (iii) the priority of tasks are in decreasing order of the index i, i.e., task τ_i has higher priority than task τ_j if $i < j$; (iv) each instance of the task is released at the beginning of the period; (v) the checkpointing intervals for a task are equal; (vi) the times for rollback and state restoration are zero; (vii) faults are detected as soon as they occur, and (viii) no faults occur during checkpointing and rollback recovery.

In [8], a feasibility analysis is provided under the assumption that two successive faults arrive with a minimum inter-arrival time T_F. This is not practical for realistic applications, where the fault occurrence can be bursty or memoryless. Therefore, we focus here on tolerating up to a given number of faults during task execution. No additional assumption is made regarding fault arrivals.

Since the task set is periodic, the total execution time can be very high if we consider a large number of periods. We therefore need to identify an appropriate k-fault-tolerant condition for shorter time duration. Here we provide two solutions corresponding to two different fault-tolerance requirements. One is to tolerate k faults for each job, termed as job-oriented fault-tolerance; the other is to tolerate k faults within a hyperperiod (defined as the least common multiple of all the task periods [7]), termed as hyperperiod-oriented fault-tolerance.

We first consider the case of a single job. Suppose m checkpoints are inserted equidistantly to tolerate k faults in one job. The worst-case response time R for the job is composed of three terms: the task execution time E, the checkpointing cost mC, and the recovery cost $kE/(m+1)$, i.e. $R = E + mC + kE/(m+1)$. To

*This research was sponsored in part by DARPA, and administered by the Army Research Office under Emergent Surveillance Plexus MURI Award No. DAAD19-01-1-0504. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

satisfy the deadline constraint, we must have $E + mC + kE/(m+1) \le D$.

Let $f(m) = E + mC + kE/(m+1) - D$. The minimum value of $f(m)$ is obtained for $m = m_0 = \sqrt{kE/C} - 1$. Since m is a non-negative integer, we have $m_0 = \left\lceil \max(\sqrt{kE/C} - 1, 0) \right\rceil$.

If $f(m_0) \le 0$, there exists equidistant checkpointing schemes for k-fault-tolerance, and the response time is minimum when m_0 checkpoints are inserted. If $f(m_0) > 0$, then no equidistant checkpointing schemes exists for tolerating up to k faults.

The feasibility analysis for more than one job is based on the time-demand analysis for fixed-priority scheduling [7]. The steps in the analysis are as following:

(1) Compute the response time R_i for τ_i according to the equation: $R_i = E_i + \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil E_h$. Here T_h and E_h are the period and the execution time of a task τ_h with higher priority than τ_i. This equation can be solved by forming a recurrence relation:

$$R_i^{(j+1)} = E_i + \sum_{h=1}^{i-1} \left\lceil R_i^{(j)} / T_h \right\rceil E_h . \quad (1)$$

(2) The iteration is terminated either when $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j or when $R_i^{(j+1)} > D_i$, whichever occurs sooner. In the former case, τ_i is schedulable; in the later case, τ_i is not schedulable.

According to [7], the time complexity of the time-demand analysis for each task is $O(nR)$, where R is the ratio of the largest period to the smallest period.

2.1 Job-oriented fault-tolerance: tolerating *k* faults in each job

In this case, we require that the task set can meet the deadline requirement under the condition that at most k faults occur during the execution of each job.

Under the worst-case condition, the additional time due to checkpointing and recovery should be incorporated. When there are m_j equidistant checkpoints for each instance of τ_j, we have:

$$R_i = (E_i + m_i C + kE_i/(m_i+1)) + \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil (E_h + m_h C + kE_h/(m_h+1))$$

To minimize all response times, we must have: $m_i^* = \left\lceil \max(\sqrt{kEi/C} - 1, 0) \right\rceil$ $(1 \le i \le n)$. Then we can employ the recurrence equation as follows:

$$R_i^{(j+1)} = (E_i + m_i^* C + kE_i/(m_i^*+1)) + \sum_{h=1}^{i-1} \left\lceil R_i^{(j)} / T_h \right\rceil (E_h + m_h^* C + kE_h/(m_h^*+1)).$$

When $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j, τ_i is schedulable; when $R_i^{(j+1)} > D_i$, τ_i is not schedulable. The total time complexity here is $O(n^2 R)$, where R is the ratio of the largest period to the smallest period.

Example 1: Consider a task set composed of two tasks: τ_1 = (60, 18, 7) and τ_2 = (80, 34, 8), and let k = 3, C = 1. Then m_1^* = 4 and m_2^* = 4. After applying the recurrence equation, we get the response times: R_1 = 15.2 < 18; R_2 = 33 < 34. Thus checkpointing is feasible for this task set if up to three faults occur during each job. Next we examine the case of k = 4. For this case m_1^* = 5 and m_2^* = 5. The response times are: R_1 = 16.7 < 18 and R_2 = 35 > 34. As a result, checkpointing is not feasible if up to four faults need to be tolerated for each job.

2.2 Hyperperiod-oriented fault-tolerance: tolerating *k* faults in a hyperperiod

In [8], an algorithm is presented to determine the checkpointing interval under the assumption that two successive faults arrive with a minimum inter-arrival time T_F. Let $F_j, 1 \le j \le i$, be the extra computation time needed by $\tau_j, 1 \le j \le i$, if one fault occurs during the execution. When there are m_j equidistant checkpoints for τ_j, the response time R_i for τ_i is expressed as follows in [8]:

$$R_i = (E_i + m_i C) + \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil \ (E_h + m_h C) + \lceil R_i / T_F \rceil \max_{1 \le j \le i} \{F_j\},$$

where $F_j = E_j/(m_j+1)$.

The checkpoint is examined starting from high-priority tasks to low-priority tasks. For each task τ_j, the algorithm tries to reduce the response time by reducing the maximum additional computation time, i.e., $\max_{1 \le j \le i} \{F_j\}$. The details in [8] are as follows:

(1) Initially $m_i = 0$ for $1 \le i \le n$.

(2) Starting from the highest-priority task τ_1, calculate the minimum number of checkpoints m_1 required to make it schedulable.

(3) In decreasing order of task priorities, calculate the response time R_i of task τ_i. If $R_i \le D_i$, move to the next task; otherwise R_i needs to be reduced further. The only way to reduce R_i is to add more checkpoints to decrease the re-execution time caused by faults, i.e., F_j, for $1 \le j \le i$. In fact, the parameter $\max_{1 \le j \le i} \{F_j\}$ is relevant here and should be reduced. The task τ^* that contributes the most to the task re-execution time is found and one more checkpoint is added to τ^*. Then R_i is recalculated. This process is repeated until either $R_i \le D_i$ or the deadline D_i is exceeded.

While the schedulability test in [8] provides useful guidelines on task schedulability in the presence of faults, its drawback is that two key issues that affect schedulability are not addressed.

1. Checkpoints are added to the higher-priority tasks in certain iterations in order to satisfy deadline constraints for all the tasks. These higher-priority tasks, however, have met their deadline in earlier iterations. The addition of more checkpoints to them inevitably changes their response times. As a result, it is necessary to trace back to re-calculate their response times and adjust their checkpoints. This issue has not been addressed in [8].

2. It is necessary to determine a bound on the number of checkpoints beyond which the addition of checkpoints does not improve schedulability. In [8], the schedulability test concludes that τ_i is not schedulable once R_i increases during the addition of checkpoints. However, this does not always hold. We present a counterexample below.

Example 2: Consider two tasks τ_1 = (100, 18, 7.999) and τ_2 = (101, 21, 8), and let T_F = 102, C = 0.1. We follow the steps from [8] as shown below:

(1) Initially $m_1 = m_2 = 0$, and F_1 = 7.999, F_2 = 8;

(2) Next τ_1 is examined: R_1 =15.998 < 18. No checkpoints are needed for τ_1. Thus $m_1 = m_2 = 0$.

(3) Next τ_2 is examined: R_2 = 23.999 > 21. Since $F_2 > F_1$, one checkpoint is added to τ_2, thus m_1 = 0 and m_2 =1. Then F_1 = 7.999, F_2 = 4 and $\max_{1 \le j \le 2} \{F_j\} = 7.999$. We recalculate the response time R_2 = 24.098 > 23.999. According to [8], τ_2 is not schedulable. However, this is not correct. We continue the above step and find $F_1 > F_2$, then one more checkpoint is added to τ_1; as a result m_1 = 1, m_2 = 1. Then $F_1 = 7.999/(1+1) = 3.9995$, F_2 = 4, and

$\max_{1\le j\le 2}\{F_j\} = 4$. We recalculate the response time of τ_1 and τ_2: $R_1 = 12.0985 < 18$ and $R_2 = 20.199 < 21$, which implies that both tasks are schedulable.

We require here that the tasks meet their deadlines under the condition that at most *k* faults occur during a hyperperiod. Based on the schedulability test in [8], we solve the two aforementioned problems as follows.

The response time R_i for τ_i is expressed as:

$$R_i = (E_i + m_i C) + \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil \ (E_h + m_h C) + k \max_{1\le j\le i}\{F_j\},$$

where $F_j = E_j/(m_j + 1)$.

The first problem can be solved using a recursive method. Any time we increase the number of checkpoints for a task, all the lower-priority tasks need to be re-examined. We solve the second problem by determining a bound on the number of checkpoints such that if the task set cannot be made schedulable using this number of checkpoints, it cannot be scheduled by adding more checkpoints. Both the checkpointing cost and the timing constraints must be taken into account.

(1) Analysis of a bound based on checkpointing tradeoffs

The effect of adding more checkpoints is two-fold. First it increases the execution time due to the checkpoint cost, which runs contrary to the goal of reducing the response time. On the other hand, it decreases re-execution due to a fault, which helps in reducing the response time. Suppose the task execution time is *E* and *m* checkpoints have already been added. If another checkpoint is now added, the reduction of re-execution time under the *k*-fault-tolerance requirement is simply:

$kE/(m+1) - kE/(m+2) = kE/[(m+1)(m+2)]$.

We combine the two impacts of checkpointing on the re-execution time to define the tradeoff function $tr(m)$ as: $tr(m) = C - kE/[(m+1)(m+2)]$.

If $tr(m) < 0$, then adding one more checkpoint can potentially reduce the response time; otherwise, it is not helpful since it increases the task re-execution time due to the *k* faults.

For each task τ_i with m_i checkpoints, we can calculate the tradeoff function $tr_i(m_i)$. Solving for $tr_i(m_i') = 0$, we get: $m_i' = (-3 + \sqrt{1 + 4kE_i/C})/2$ for $1 \le i \le n$. Since $m_i' \ge 0$, we further express it as: $m_i' = \max(\lfloor(-3 + \sqrt{1 + 4kE_i/C})/2\rfloor, 0)$ for $1 \le i \le n$. This gives an upper bound on the number of checkpoints, which is based on the tradeoff function.

(2) Analysis of a bound based on timing constraints

Under fault-free conditions, the response time R_i^0 for task τ_i can be easily obtained. After incorporating the checkpointing cost and timing constraints, we have: $R_i^0 + m_i C \le D_i$, which implies that $m_i \le (D_i - R_i^0)/C$. Let $m_i^{\#} = \lfloor (D_i - R_i^0)/C \rfloor$.

Combining the two bounds, we define $m_i^* = \min(m_i', m_i^{\#})$ $(1 \le i \le n)$. Then m_i^* is a tighter upper bound on the number of checkpoints required to make τ_i schedulable.

A checkpointing algorithm *ADV-CP* for off-line feasibility analysis is described in Figure 1, which takes as an input parameter the real-time task set Γ. All tasks are initially set unschedulable. The recursive checkpointing procedure *CP*(*p*,*q*) is described in Figure 2, where *p* and *q* are the lowest and highest index for the task subset under consideration.

The recursive execution of $CP(p,q)$ takes $O(n^2R)\sum_{i=1}^{n} m_i^*$ time. Let $M^* = \sum_{i=1}^{n} m_i^*$. Adding all the cost together, the total complexity for the feasibility test & checkpointing procedure is $O(n^2RM^*)$, which is only quadratic in the number of tasks *n*. Furthermore, we note that the complexity can be reduced if we can make M^* as small as possible. That is why we combine both the tradeoff function and timing constraints to obtain a relatively tight bound for m_i^*.

3. Feasibility Analysis with DVS

We are given a variable-speed processor, which is equipped with *l* speeds $f_1, f_2, \ldots, f_l$. In addition, $f_i < f_j$ if $i < j$. Let *c* be the number of clock cycles that a single checkpoint takes. We are also given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of *n* periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$. The elements of the tuple are defined as follows: T_i is the period of τ_i and D_i is its deadline ($D_i \le T_i$); E_i is the number of computation cycles of τ_i under fault-free conditions.

In addition to the assumptions in Section 2, we assume the task set Γ is schedulable under fault free conditions at the lowest speed. For the sake of simplicity of presentation, we also assume without loss of generality that speed switching does not incur extra cost in terms of time and energy.

We note that if supply voltage V_{dd} is used for a task with *N* single-cycle instructions, the energy consumption can be expressed as (α is a constant): $Eng(N) = \alpha N V_{dd}^2$ (2)

We also note that the processor clock frequency *f* can be expressed in terms of the supply voltage V_{dd} and threshold voltage V_t as $f = \beta(V_{dd} - V_t)^2/V_{dd}$, where β is a constant.

From above, we obtain V_{dd} as a function of *f*:

$$V_{dd}(f) = (V_t + f/(2\beta)) + \sqrt{(V_t + f/(2\beta))^2 - V_t^2} \qquad (3)$$

According to Equation (2), energy consumption is a function of *N* and *f*: $Eng(N, f) = \alpha N V_{dd}^2(f)$, where $V_{dd}(f)$ is expressed in Equation (3). Here we assume $V_t = 0$ without loss of generality.

In our proposed scheme, speed scaling can be done for a particular application, i.e., all tasks for the application are assigned the same speed, or at the task level, i.e., different tasks can be assigned different speed. Speed scaling can also be carried out at the job level, i.e., different jobs for a task can have different speeds. Let $s(\tau_i) : \tau_i \to f_j$ $(1 \le i \le n,\ 1 \le j \le l)$ denote the speed scaling function, which maps a task τ_i to speed f_j.

Our aim is to meet task deadlines deterministically, even though *k* faults occur, while minimizing energy consumption. First, we need to identify appropriate time duration to evaluate the energy consumption. We consider the hyperperiod as the time duration. Second, the criterion of minimizing energy consumption needs to be clarified. Based on the application requirement, we can choose either a best-case or a worst-case energy consumption value. By best-case, we refer to the results obtained under the fault-free condition, while worst-case refers to the results obtained when all *k* faults occur. In our work, we focus on minimizing energy consumption under the worst-case condition during a hyperperiod. Let the hyperperiod denoted by *Ht* and the number of checkpoints for τ_i denoted by m_i; the total energy consumption during one hyperperiod is expressed as:

$$Total\_eng = \sum_{i=1}^{n} (Ht/T_i) Eng(E_i + m_i C + kE_i/(m_i + 1), s(\tau_i)) \qquad (4)$$

```
Procedure ADV-CP (I)
begin
1. for i = 1 to n do
     m_i = 0; compute m_i^*; R_j = ∞;
2. CP(1, n).
end
```

Figure 1: Advanced checkpointing procedure.

```
Procedure CP (p, q)
1. if (R_p ≤ D_p & R_{p+1} ≤ D_{p+1} & …& R_q ≤ D_q)
     return("task subset schedulable");
2. elseif (m_1 > m_1^* & m_2 > m_2^* & …& m_q > m_q^*)
     exit("task set unschedulable");
3. else{for j = p to q do{
   3.1 compute R_j;
   3.2 while (R_j > D_j) do{
     3.2.1 find h∈[1, j] such that F_h = max(F_1, F_2, …, F_j);
     3.2.2 m_h = m_h + 1;
     3.2.3 F_h = E_h/(m_h + 1);
     3.2.4 CP (h, j);}}}
```

Figure 2: Recursive checkpointing procedure.

The off-line feasibility analysis with DVS provides two important pieces of information: first, it provides the feasibility analysis under the worst-case scenario; second, it provides static results such as speed assignment and checkpoint interval, which can be further used for on-line adjustment during task execution.

3.1 Job-oriented fault-tolerance with DVS

The worst-case response time for task τ_i can be expressed as:

$$R_i = \frac{E_i + m_i c + kE_i/(m_i+1)}{s(\tau_i)} + \sum_{h=1}^{i-1}\left\lceil \frac{R_i}{T_h} \right\rceil \frac{E_h + m_h c + kE_h/(m_h+1)}{s(\tau_h)} \quad (5)$$

To minimize all response times, we must have: $m_i^* = \left\lceil \max(\sqrt{kE_i/C} - 1, 0) \right\rceil$ $(1 \le i \le n)$. Then we can employ the recurrence equation as follows:

$$R_i^{(j+1)} = \frac{E_i + m_i^* c + kE_i/(m_i^*+1)}{s(\tau_i)} + \sum_{h=1}^{i-1}\left\lceil \frac{R_i^{(j)}}{T_h} \right\rceil \frac{E_h + m_h^* c + kE_h/(m_h^*+1)}{s(\tau_h)}$$

If $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j, τ_i is schedulable; if $R_i^{(j+1)} > D_i$, τ_i is not schedulable.

Since the optimal number of checkpoints is fixed a *priori* for each task, we need to choose appropriate processor speeds to satisfy the deadline constraint for each task.

(1) Application-level speed scaling: all tasks have the same speed. Here all tasks have the same speed f^* and $s(\tau_1) = s(\tau_2) = \ldots = s(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Equation (5) is simplified as:

$$R_i = \frac{E_i + m_i^* c + kE_i/(m_i^*+1)}{f^*} + \sum_{h=1}^{i-1}\left\lceil \frac{R_i}{T_h} \right\rceil \frac{E_h + m_h^* c + kE_h/(m_h^*+1)}{f^*}$$

The iterative method described in Section 2.1 can be used here to determine f^*. To examine the feasibility for each task, all possible speeds have to be tested. There are l possibilities in total. The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

(2) Task-level speed scaling: different tasks can have different speeds. To obtain an optimal solution, we use an exhaustive method. Since each task can be run at l speeds, there are l^n possible speed combinations for n tasks. For each speed combination, the feasibility test is performed according to Equation (5). Meanwhile, the energy consumption is calculated from Equation (4). The speed combination that satisfies the timing constraints with the minimum energy consumption is chosen as the optimal solution.

3.2 Hyperperiod-oriented fault-tolerance with DVS

The worst-case response time for task τ_i can be expressed as:

$$R_i = (E_i + m_i c)/s(\tau_i) + \sum_{h=1}^{i-1}\left\lceil R_i/T_h \right\rceil (E_h + m_h c)/s(\tau_h) + k \max_{1\le j\le i}\{F_j\}$$

where $F_j = E_j/[s(\tau_j)(m_j+1)]$. (6)

(1) Application-level speed scaling: all tasks have the same speed. Here all tasks have the same speed f^* and $s(\tau_1) = s(\tau_2) = \ldots = s(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Equation (6) is simplified as:

$$R_i = (E_i + m_i c)/f^* + \sum_{h=1}^{i-1}\left\lceil R_i/T_h \right\rceil (E_h + m_h c)/f^* + k \max_{1\le j\le i}\{F_j\},$$

where $F_j = E_j/[f^*(m_j+1)]$.

In contrast to (1) in Section 3.1, we first fix the speed instead of the number of checkpoints. For each given speed f^*, we examine the feasibility of the task set using the method in Section 2.2. If it is schedulable, the corresponding number of checkpoints for each task can be obtained. The energy consumption is calculated from Equation (4). The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

(2) Task-level speed scaling: different tasks can have different speeds. To obtain an optimal solution, we use an exhaustive method. Since each task can be run at l speeds, there are l^n possible speed combinations for n tasks. For each speed combination, the feasibility test is performed according to Equation (6). The method in Section 2.2 is employed and the corresponding number of checkpoints is obtained. Meanwhile, the energy consumption is calculated from Equation (4). The speed combination that satisfies the timing constraints with the minimum energy consumption is chosen as the optimal solution.

3.3 Job-level on-line speed scaling

As discussed in Sections 3.1 and 3.2, the speed assignment and the checkpointing interval are determined by the off-line feasibility analysis. A static sequence of jobs is obtained and their timing parameters such as release times and execution times are known *a priori* under the worst case. However, if only such static measures are used during run-time, it will not be possible to make use of idle intervals. Clearly, further energy saving is possible through additional on-line speed scaling.

The on-line speed scaling procedure, done at the job-level, is adaptive with respect to fault occurrence. It makes use of a simple run-time adaptation mechanism. The key features are:

- Once a job completes, the release time of the next job is adjusted dynamically during run-time.
- The processor is run at an appropriate speed such that either the current job completes before its deadline, or before the static release time of the next job, whichever is sooner.

4. Experimental Results

In this section, we compare the performance of our energy-aware fault-tolerance scheme with the DVS technique proposed in [3], referred to as VSLP. Our goal here is to highlight the impact of fault occurrences on a fault-oblivious DVS scheme.

We use the following notation to refer to the various types of schemes: (1) JFTA: job-oriented fault tolerance with application-level speed scaling; (2) JFTT: job-oriented fault tolerance with task-level speed scaling; (3) HFTA: hyperperiod-oriented fault tolerance with application-level speed scaling; (4) HFTT: hyperperiod-oriented fault-tolerance with task-level speed scaling.

Since the VSLP scheme as presented in [3] does not provide fault tolerance, we assume that it simply re-executes a job when a fault occurs. Furthermore, since JFTA is a special case of JFTT and HFTA is a special case of HFTT, we compare VSLP with the JFTT and HFTT schemes. For both cases, we first show that JFTT and HFTT can schedule task sets even when VSLP cannot do so; we then show that these schemes can save more energy via checkpointing in the presence of faults.

4.1 JFTT vs. VSLP

As pointed out in [8], a system of periodic preemptable tasks, each of whose relative deadline D is equal to its period T, is schedulable on one processor according to the rate monotonic algorithm if and only if its total task utilization is equal to or less than 1. In the presence of faults, since the re-execution takes extra time and the total task utilization will be increased accordingly, a task set that is schedulable under fault-free conditions may no longer be schedulable. Here we construct a task set whose total utilization is greater than 1 for VSLP under faulty conditions even though the entire task set is executed with the highest speed, and show that this task set is schedulable using JFTT.

Suppose we are given three tasks τ_1 = (12000, 12000, 2200), τ_2 = (18000, 18000, 3000) and τ_3 = (24000, 24000, 4000), and three normalized processor speeds 1.0, 0.8 and 0.6. Let a single checkpoint take c = 50 cycles. If only k = 1 fault occurs during each job, the total utilization for VSLP under the highest speed is found to be 1.033. This implies that VSLP cannot schedule the task set when one or more faults occur during each job. However, the experiments show that JFTT can tolerate up to 6 faults during each job. The speed assignment (s_1, s_2, s_3) and number of checkpoints (m_1, m_2, m_3) for τ_1, τ_2 and τ_3, and total energy consumption are shown in Table 1.

Next we show that JFTT saves more energy than VSLP in the presence of faults when both schemes are feasible. Consider three tasks τ_1 = (12000, 10000, 500), τ_2 = (18000, 16000, 1000) and τ_3 = (24000, 22000, 2000), and three normalized processor speeds 1.0, 0.8 and 0.6. Let a single checkpoint take c = 50 cycles. The energy saving for JFTT over VSLP is shown in Table 2. Compared to VSLP, JFTT can save up to 75% energy in the presence of faults.

To demonstrate the effect of checkpointing cost, we fix the value of k and change the value of c for the same task set used in Table 2. The results are shown in Table 3.

4.2 HFTT vs. VSLP

We now show that HFTT can schedule task sets in the presence of faults, even when VSLP fails to do so. Suppose we are given three tasks τ_1 = (12000, 12000, 2200), τ_2 = (18000, 18000, 3000) and τ_3 = (24000, 24000, 4000), and three normalized processor speeds 1.0, 0.8 and 0.6. Let a single checkpoint take c = 50 cycles. As indicated in Section 4.1, VSLP cannot schedule this task set when one or more faults occur during each job. Here although we examine the fault occurrence in one hyperperiod, the *WCET* value of each task for VSLP remains the same as that in Section 4.1. As a result, VSLP still cannot schedule this task set if there are any fault occurrences. On the other hand, HFTT can tolerate more than 10 faults during a hyperperiod. The speed assignment (s_1, s_2, s_3) and number of checkpoints (m_1, m_2, m_3) for τ_1, τ_2 and τ_3, and total energy consumption are shown in Table 4.

Next we show that HFTT saves more energy than VSLP in the presence of faults when both schemes are feasible. Consider three tasks τ_1 = (12000, 10000, 500), τ_2 = (18000, 16000, 1000) and τ_3 = (24000, 22000, 2000), and three normalized processor speeds 1.0, 0.8 and 0.6. Let a single checkpoint take c = 100 cycles. The energy saving for HFTT over VSLP is demonstrated in Table 5.

k	JFTT			VSLP
	(s_1, s_2, s_3)	(m_1, m_2, m_3)	Energy	
1	(0.8, 0.8, 0.8)	(7, 8, 9)	29717	Infeasible
3	(0.8, 1.0, 0.8)	(12, 14, 16)	40473	
6	(1.0, 1.0, 1.0)	(17, 19, 22)	60530	

Table 1: JFTT vs. VSLP (Part 1).

k	Engy_JFTT	Engy_VSLP	Engy_JFTT/ Engy_VSLP
1	6522	9360	0.70
2	7362	14040	0.52
3	7981	33280	0.24

Table 2: JFTT vs. VSLP (Part 2).

c	Engy_JFTT	Engy_VSLP	Engy_JFTT/ Engy_VSLP
50	7981	33280	0.24
150	10206	33280	0.31
250	11844	33280	0.36

Table 3: JFTT vs. VSLP (Part 3).

k	HFTT			VSLP
	(s_1, s_2, s_3)	(m_1, m_2, m_3)	Energy	
1	(0.6, 0.8, 0.8)	(3, 3, 4)	21716	Infeasible
4	(0.8, 1.0, 0.8)	(5, 6, 10)	32187	
10	(1.0, 1.0, 1.0)	(10, 14, 19)	47850	

Table 4: HFTT vs. VSLP (Part 1).

k	Engy_HFTT	Engy_VSLP	Engy_HFTT/ Engy_VSLP
1	5400	9360	0.58
2	5508	14040	0.39
3	5760	33280	0.17

Table 5: HFTT vs. VSLP (Part 2).

5. Conclusions

We have shown how dynamic adaptation for fault tolerance and power management can be carried out in embedded systems. Fault tolerance is achieved via checkpointing and power management is carried out using DVS. We have presented feasibility-of-scheduling tests for checkpointing schemes under both constant processor speed and variable processor speed. Two feasibility tests have been developed for application-level and task-level speed scaling, respectively. Based on the results of the feasibility analyses, on-line dynamic speed scaling can be employed to further reduce energy.

References

[1] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems", *Proc. Design, Automation and Test in Europe Conference*, pp. 918-923, 2003.

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

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

[4] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Data Base Systems", *IEEE Trans. Software Eng*., vol. 1, pp. 100-110, March 1975.

[5] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement", *IEEE Trans. Computer*s, vol. 46, pp. 976-985, September 1997.

[6] S. W. Kwak, B. J. Choi and B. K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults", *IEEE Trans. Reliability*, vol. 50, pp. 293-301, September 2001.

[7] J. W. Liu, *Real-Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[8] S. Punnekkat, A. Burns and R. Davis, "Analysis of checkpointing for real-time systems", *Real-Time Systems Journal*, vol. 20, pp. 83-102, January 2001.