Dynamic Adaptation for Fault Tolerance and Power Management in Embedded Real-Time Systems

YING ZHANG and KRISHNENDU CHAKRABARTY
Duke University

Safety-critical embedded systems often operate in harsh environmental conditions that necessitate fault-tolerant computing techniques. In addition, many safety-critical systems execute real-time applications that require strict adherence to task deadlines. These embedded systems are also energy-constrained, since system lifetime is determined largely by the battery lifetime. In this paper, we investigate dynamic adaptation techniques based on checkpointing and dynamic voltage scaling (DVS) for fault tolerance and power management. We first present schedulability tests that provide the criteria under which checkpointing can provide fault tolerance and real-time guarantees. We then present an adaptive checkpointing scheme in which the checkpointing interval for a task is dynamically adjusted during execution, and checkpoints are inserted based not only on the available slack, but also on the occurrences of faults. Next, we combine adaptive checkpointing with DVS to achieve power reduction. Finally, we develop an adaptive checkpointing scheme for a set of multiple tasks in real-time systems. An offline preprocessing based on linear programming is used to determine the parameters that are provided as inputs to the online adaptive checkpointing procedure. Simulation results show that compared to previous methods, the proposed adaptive checkpointing approach increases the likelihood of timely task completion in the presence of faults. When combined with DVS, adaptive checkpointing also leads to considerable energy savings.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Fault Tolerance

General Terms: Algorithm, Performance

Additional Key Words and Phrases: Checkpointing, dynamic voltage scaling

1. INTRODUCTION

Embedded systems often operate in harsh environmental conditions that necessitate the use of fault-tolerant computing techniques to ensure dependability. These systems are also severely energy-constrained, since system lifetime is determined to a large extent by the battery lifetime. In addition, many embedded systems execute real-time applications that require strict adherence to task

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.
Authors' address: Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708; email: {yingzh,krish}@ee.duke.edu.

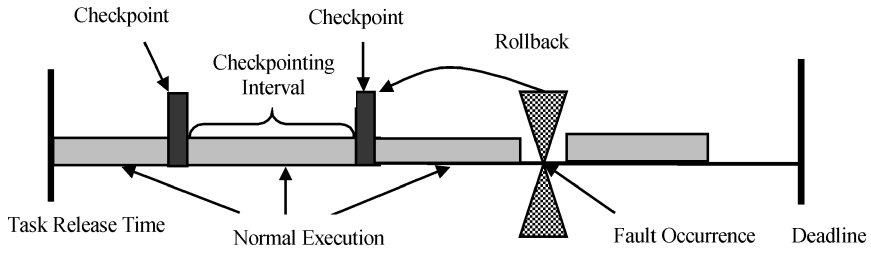

Fig. 1. Illustration of checkpointing and rollback recovery.

deadlines [Pop et al. 2000]. In this paper, we present an integrated approach that provides fault tolerance and dynamic power management for real-time tasks executing in an embedded system.

Dynamic voltage scaling (DVS) has emerged as a popular solution to the problem of reducing power consumption during system operation [Ishihara and Yasuura 1998; Shin et al. 2000; Quan and Hu 2001]. Many embedded processors are now equipped with the ability to dynamically scale the operating voltage. Examples of such embedded processors include the embedded SL enhanced Intel 486DX2 processor [Intel], which operates at 5.5 and 3.3 V, and the Motorola 6805 [Motorola], which operates at 5.5, 3.3, and 2.2 V. AMD's PowerNow! technology offers greater flexibility than many other commercially available processors in setting frequencies and core voltages. The AMD K-6 processor uses a small range of frequencies from 300 to 500 MHz, adjustable in 50 MHz increments [AMD]. Since a reduction in voltage results in a corresponding drop in the processor speed, a number of techniques have been proposed recently to balance real-time responsiveness with low-energy task execution [Bahbha et al. 2001; Luo and Jha 2000; Scmitz et al. 2002].

Fault tolerance is typically achieved in real-time systems through online fault detection [Shin and Lee 1984], checkpointing and rollback recovery [Chandy et al. 1975]. Figure 1 illustrates checkpointing and rollback recovery. 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.

Checkpointing increases task execution time and in the absence of faults, it might cause a missed deadline for a task that completes on time without checkpointing. In the presence of faults, however, checkpointing can increase the likelihood of a task completing on time with the correct result. Without checkpointing, a fault necessitates the restart of the task. Frequent checkpointing reduces recomputation time due to faults, but it increases task execution time. On the other hand, infrequent checkpointing has less impact on task execution in the absence of faults, but it increases the amount of rollback that must be performed after a fault is detected. Therefore, the checkpointing interval, that is, duration between two consecutive checkpoints, must be carefully chosen to balance checkpointing cost (the time needed to perform a single checkpoint) with the reexecution time.

Dynamic power management and fault tolerance for embedded real-time systems have been studied as separate problems in the literature. DVS techniques

for power management do not consider fault tolerance [Ishihara and Yasuura 1998; Shin et al. 2000; Quan and Hu 2001], and checkpoint placement strategies for fault tolerance do not address dynamic power management [Shin et al. 1987; Ziv and Bruck 1997; Kwak et al. 2001]. However, lower processor voltages and shrinking process technologies in the nanotechnology realm are likely to lead to lower noise margins and more transient faults, caused in part by single-event upsets [Dupont et al. 2002]. Hence a dynamic adaptation framework in which DVS techniques are tied to system-level fault tolerance, are of particular interest for embedded systems. We present here an integrated approach that facilitates fault tolerance through checkpointing and power management through DVS. To the best of our knowledge, this is the first approach that addresses these two issues in conjunction.

We assume throughout that faults are intermittent or transient in nature, and that permanent faults are handled through manufacturing testing or field-testing techniques [Bushnell and Agrawal 2000]. Typical examples of transient faults include errors caused by cosmic rays and high-energy particles in nanotechnology with shrinking processes [Dupont et al. 2002].

We address both hard and soft real-time systems in this paper. Systems in which a missed deadline results in disastrous consequences are termed hard real-time systems, while systems in which a missed deadline results in degraded performance, but with no extreme consequences, are termed soft real-time systems [Liu 2000]. Examples of power-constrained hard real-time systems include battery-driven autonomous airborne and seaborne systems. Stock price quotation systems, telephone switching systems, and multimedia applications are examples of soft real-time systems. Therefore, it is important to guarantee timeliness for hard real-time systems in worst-case scenarios, and provide a high likelihood of meeting task deadlines for soft real-time systems. In this work, the offline feasibility analysis is targeting at providing deterministic timeliness for hard real-time systems, and adaptive checkpointing is aiming at ensuring a high probability that task deadlines are met for soft real-time systems.

We first present feasibility tests for checkpointing schemes that use a fixed checkpointing interval for real-time tasks. These feasibility tests provide the criteria under which checkpointing can provide fault tolerance and real-time guarantees under two different transient fault arrival models. We also present two techniques to determine the fixed checkpointing interval in an offline manner. Following this, we present an adaptive checkpointing scheme for real-time systems in which a variable checkpointing interval is dynamically adjusted during task execution, and checkpoints are inserted based not only on the available slack, but also on the occurrences of faults during task execution. This approach is in contrast to "static" checkpointing schemes that fix the checkpointing interval *a priori* before task execution. Simulation results show that compared to previous methods, the proposed adaptive checkpointing approach increases the likelihood of timely task completion in the presence of faults. The proposed adaptive checkpointing is tailored to handle not only a random fault-arrival process, but it is also designed to be k-fault-tolerant—it attempts to tolerate up to k fault occurrences.

The proposed adaptive checkpointing is then extended in two ways. First, it is combined with DVS to achieve power reduction and fault tolerance simultaneously. The resulting energy-aware adaptive checkpointing scheme uses a dynamic speed-scaling criterion that is based not only on the slack in task execution but also on the occurrences of faults during task execution. The second extension applies adaptive checkpointing to a set of multiple real-time tasks. We develop a linear-programming model to determine, in an offline fashion, the parameters that are provided as inputs to the adaptive online checkpointing procedure.

The rest of the paper is organized as follows. Section 2 introduces some relevant background material on checkpointing. Section 3 provides offline feasibility analysis for checkpointing in real-time systems. Section 4 presents our adaptive checkpointing scheme for real-time systems. In Section 5, we describe two extensions of the adaptive checkpointing scheme: (i) incorporation of DVS; (ii) application to a set of multiple real-time tasks. Conclusions and directions for future work are presented in Section 6.

2. CHECKPOINTING IN REAL-TIME SYSTEMS

In this section, we present a classification of checkpointing schemes for real-time systems that have been presented in the literature.

2.1 Online Versus Offline Schemes

An offline checkpointing scheme determines the checkpointing interval for a task *a priori*, that is, before task execution. Most known checkpointing schemes for real-time systems belong to this category [Kwak et al. 2001; Duda 1983; Lee et al. 1999]. A drawback here is that the checkpointing interval cannot be adapted to the actual fault occurrence during task execution. An online scheme in which the checkpointing interval can be adapted to fault occurrences is therefore more desirable. However, current online checkpointing schemes [Ziv and Bruck 1997] provide only probabilistic guarantees on the timely completion of tasks, as described next.

2.2 Probabilistic Versus Deterministic Guarantees

Some checkpointing schemes [e.g., Kwak et al. 2001; Duda 1983] assume that faults occur as a Poisson process with arrival rate λ. These schemes use a checkpointing interval that maximizes the probability that a task completes on time for a given fault arrival rate λ. Hence the real-time guarantees in these schemes are probabilistic. Other checkpointing schemes [e.g., Lee et al. 1999; Bettati et al. 1992] offer deterministic real-time guarantees under different assumptions. For example, it is sometimes assumed that at most k faults occur during task execution. This assumption has been justified in the literature for safety-critical applications such as aerospace systems, where stringent dependability requirements mandate that a sufficient number of faults be tolerated with certainty [Sieworek and Swarz 1998]. Another common assumption is that two successive faults arrive with the minimum inter-arrival time T_F [Punnekkat et al. 2001]. This assumption, however, is not practical in many situations since

it is often hard to determine appropriate value for the parameter T_F. A drawback of most deterministic checkpointing schemes is that they cannot adapt to actual fault occurrences during task execution. In view of the above reasons, we focus here on the use of a stochastic fault arrival process and a probabilistic checkpointing scheme, while utilizing deterministic feasibility analysis as a guideline.

2.3 Equidistant Versus Variable Checkpointing Interval

Equidistant checkpointing, as the term implies, relies on the use of a constant checkpointing interval during task execution. This is typically used with offline checkpointing schemes. It has been shown in the literature that if the checkpointing cost is C and faults arrive as a Poisson process with rate λ, the mean execution time for the task is minimum if a constant checkpointing interval of $\sqrt{2C/\lambda}$ is used [Duda 1983]. We refer to this as the Poisson-arrival approach. However, the minimum execution time does not guarantee timely completion of a task under real-time deadlines. It has also been shown that if the fault-free execution time for a task is E, the worst-case execution time for up to k faults is minimum if the constant checkpointing interval is set to $\sqrt{EC/k}$ [Bettati et al. 1992]. We refer to this as the k-fault-tolerant approach. A drawback with these equidistant schemes is that they cannot adapt to actual fault arrivals. For example, due to the random nature of fault occurrences, the checkpointing interval can conceivably be increased halfway through task execution if only a few faults occur during the first half of task execution. Therefore, we consider online checkpointing with variable checkpointing intervals.

2.4 Constant Versus Variable Checkpointing Cost

Most prior work has been based on the assumption that all checkpoints take the same amount of time, that is, the checkpointing cost is constant. An alternative adaptive approach, taken in Ziv and Bruck [1997], but less well understood, is to assume that the checkpointing cost depends on the time at which it is taken. We use the constant checkpointing cost model in our work because of its inherent simplicity.

3. OFFLINE FEASIBILITY ANALYSIS

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.
- D_i is the deadline ($D_i \leq T_i$ for τ_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:

- The task set Γ is scheduled using fixed-priority methods such as the rate-monotonic scheme.

- The task set Γ is schedulable under fault-free conditions.
- The priority of tasks are in decreasing order of the index i, that is, task τ_i has higher priority than task τ_j if $i < j$.
- Each instance of the task is released at the beginning of the period.
- The checkpointing intervals are equal for the same task.
- The times for rollback and state restoration are zero.
- Faults are detected as soon as they occur.
- No faults occur during checkpointing and rollback recovery.

In Punnekkat et al. [2001], a feasibility analysis is provided under the assumption that two successive faults arrive with a minimum inter-arrival time T_F. This implies that the time between the occurrences of two faults is at least T_F. This assumption is not practical for realistic applications, where the fault occurrence can be bursty or memoryless. For example, it is difficult to obtain a minimum inter-arrival time for a typical Poisson-arrival process. 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; the other is to tolerate k faults within a hyperperiod, which is defined the least common multiple of all the task periods [Liu 2000]. In practical situations, the choice of an appropriate fault tolerance criterion can be made based on the needs of the applications.

We first consider the case of a single job. Suppose $m(m \geq 0)$ checkpoints are inserted equidistantly during the computation time to tolerate k faults in one job. The worst-case response time R for the jobis composed of three terms: the task execution time E, the checkpointing cost mC, and the recovery cost $kE/(m+1)$, that is, $R = E + mC + kE/(m+1)$.

To satisfy the deadline constraint, we must have $E + mC + kE/(m+1) \leq D$. 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 $m_0 = \lceil \max(\sqrt{kE/C} - 1, 0) \rceil$.

If $f(m_0) \leq 0$, 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.

Example 1. For a real-time job with parameters $C = 10, k = 1, E = 9{,}000$ and $D = 10{,}000$, we get $m_0 = 29$ and $f(m_0) = -410$. This implies that there exists an equidistant checkpointing scheme to tolerate a single fault for this job, and the response time is minimized when 29 checkpoints are inserted. Now we change k from 1 to 3, that is, the system is required to tolerate up to three faults. Then we get $m_0 = 51$ and $f(m_0) = 29$. Since $f(m_0) > 0$, no equidistant checkpointing scheme exists to tolerate up to three faults for this job.

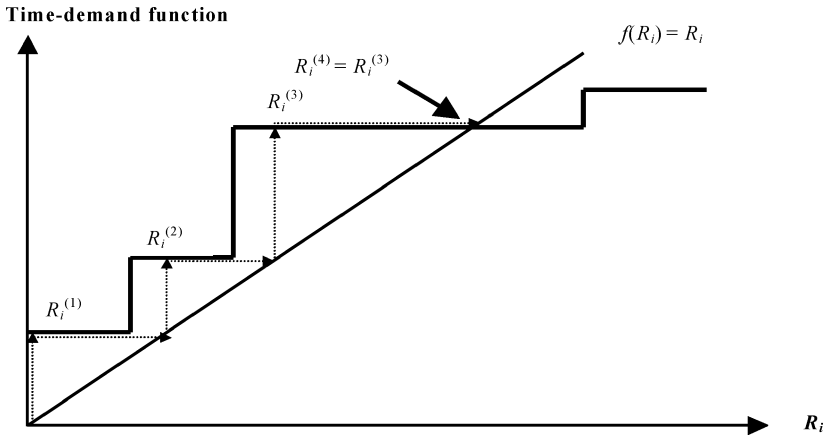

Fig. 2. Illustration of the iterative approach to determine the response time using time-demand analysis.

3.1 Tolerating *k* Faults in Each Job for Multiple Tasks

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. The feasibility analysis is based on the time-demand analysis for fixed-priority scheduling [Liu 2000]. The steps in the analysis are as following:

(1) Compute the response time R_i for τ_i according to the equation below (the right hand is defined as the time-demand function):

$$R_i = E_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil E_h.$$

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:

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

(2) The iteration is terminated either when $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \leq D_i$ for some i 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.

Figure 2 illustrates the iterative algorithm to determine the response time using time-demand analysis. The staircase function represents the magnitude of the time-demand function. The straight line simply represents R_i. Starting from the original, the iterative method finds the point of intersection of the staircase function and the straight line.

According to Liu [2000], the time complexity of the time-demand analysis for each task is $O(nW)$, where W is the ratio of the largest period to the smallest period.

Under faulty conditions, 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:

$$R_i = \left(E_i + m_i C + k\frac{E_i}{m_i+1}\right) + \sum_{h=1}^{i-1}\left\lceil \frac{R_i}{T_h}\right\rceil \left(E_h + m_h C + k\frac{E_h}{m_h+1}\right)$$

To minimize all response times, we must have: $m_i^* = \lceil \max(\sqrt{kE_i/C} - 1, 0)\rceil$ $(1 \le i \le n)$.

Then we can employ the recurrence equation as follows:

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

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.

Example 2. Consider a task set composed of two tasks: $\tau_1 = (60, 18, 7)$ and $\tau_2 = (80, 34, 8)$, and let $k = 3, C = 1$. Then $m_1^* = 4$ and $m_2^* = 4$. After applying the recurrence equation, we get the response times: $R_1 = 15.2 < 18$ and $R_2 = 33 < 34$. 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.

3.2 Tolerating k Faults in a Hyperperiod for Multiple Tasks

In Punnekkat et al. [2001], 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 [Punnekkat et al. 2001]:

$$R_i = (E_i + m_i C) + \sum_{h=1}^{i-1}\left\lceil \frac{R_i}{T_h}\right\rceil (E_h + m_h C) + \left\lceil \frac{R_i}{T_F}\right\rceil \max_{1\le j\le i}\{F_j\}, \quad \text{where } F_j = \frac{E_j}{m_j+1}.$$

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, that is, $\max_{1\le j\le i}\{F_j\}$. The details of the steps in Punnekkat et al. [2001] 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 need to be reduced further.

The only way to reduce R_i is to add more checkpoints to decrease the reexecution time caused by faults—F_j for $1 \leq j \leq i$. In fact, the parameter $\max_{1 \leq j \leq i}\{F_j\}$ is relevant here and should be reduced. Thus the task τ^* that contributes the most to the task reexecution time is found and one more checkpoint is added to τ^*. Then R_i is recalculated. This process is repeated until either $R_i \leq D_i$ or the deadline D_i is exceeded.

As discussed in Section 2.2, this algorithm is based on the restrictive assumption that two successive faults arrive with a minimum inter-arrival time T_F. In addition, while the schedulability test in Punnekkat et al. [2001] provides useful guidelines on task schedulability in the presence of faults, a drawback of this work is that two key issues that affect schedulability have not been 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 recalculate their response times and adjust their checkpoints. This issue has not been addressed in Punnekkat et al. [2001].
2. It is necessary to determine a bound on the number of checkpoints beyond which the addition of checkpoints does not improve schedulability. In another words, we need a criterion that can declare a task set to be not schedulable with a given number of checkpoints even though an arbitrary number of additional checkpoints can be added. In Punnekkat et al. [2001], 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 3. Consider two tasks $\tau_1 = (100, 18, 7.999)$ and $\tau_2 = (101, 21, 8)$, and let $T_F = 102, C = 0.1$. We follow the steps from Punnekkat et al. [2001] as 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 $\max_{1 \leq j \leq 2}\{F_j\} = 7.999$. We recalculate the response time $R_2 = 24.098 > 23.999$. According to Punnekkat et al. [2001], 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 $\max_{1 \leq j \leq 2}\{F_j\} = 4$. We recalculate the response time of τ_1 and τ_2 : $R_1 = 12.0985 < 18$ and $R_2 = 20.199 < 21$, which means 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 Punnekkat et al. [2001], we solve the two aforementioned problems as follows.

The response time R_i for τ_i is expressed as

$$R_i = (E_i + m_i C) + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil (E_h + m_h C) + k \max_{1 \le j \le i} \{F_j\}, \quad \text{where } F_j = \frac{E_j}{m_j + 1}.$$

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 reexamined. The second problem is more complicated since the response time R_i for task τ_i does not decrease monotonically when more checkpoints are added to higher-priority tasks. Suppose that in $\max_{1 \le h \le i}\{F_h\}$, we find task τ_{h_1} contributes the most to the response time R_i, and add one more checkpoint to τ_{h_1}. After recalculating R_i, we might find that R_i has increased. In this situation, we cannot simply claim the task is not schedulable, as has been shown in Example 3.

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 twofold. 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 reexecution 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 reexecution time under the k-fault-tolerance requirement is simply $kE/(m+1) - kE/(m+2) = \frac{kE}{(m+1)(m+2)}$. We combine the two impacts of checkpointing on the reexecution time to define the tradeoff function $tr(m)$ as follows: $tr(m) = C - kE/(m+1)(m+2)$.

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

An advanced checkpointing algorithm *ADV-CP* for offline feasibility analysis is described in Figure 3, which takes as an input parameter the real-time task set Γ. Line 1 initializes the parameters. The number of all checkpoints is set to 0. The bounds for all tasks are calculated. All tasks are set unschedulable.

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

Fig. 3. 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);}}}
```

Fig. 4. Recursive checkpointing procedure.

Line 2 calls the recursive checkpointing subroutine *CP* to add checkpoints from τ_1 to τ_n.

The recursive checkpointing procedure *CP(p, q)* is described in Figure 4, where p and q are the lowest and highest index for the task subset under consideration. Line 1 checks the deadline constraint to see if the current number of checkpoints can make the task subset schedulable. Line 2 checks to see if the bounds for the task subset are exceeded. If so, the whole task set is unschedulable and the recursive checkpointing should be exited. Line 3 further improves the feasibility of tasks from τ_p to τ_q. Line 3.1 calculates R_j. If the deadline cannot be met for τ_j using the current number of checkpoints, Line 3.2 adds more checkpoints to higher-priority tasks or to τ_j itself. Line 3.2.1 finds the task τ_h that contributes most to the task reexecution time. Line 3.2.2 adds one more checkpoint to τ_h, and Line 3.2.3 recalculates the reexecution time due to τ_h. Finally, Line 3.2.4 employs the procedure *CP* for tasks from τ_h to τ_j.

The time complexity for the feasibility test and the checkpointing procedure can be analyzed as follows. The computation of m_i^* for all the tasks takes $O(n^2W)$ in the worst case. Each time a checkpoint is added, the response time for lower-priority tasks need to be recalculated. Hence the recursive execution of $CP(p, q)$ takes $O(n^2W)\sum_{i=1}^{n} m_i^*$. Let $M^* = \sum_{i=1}^{n} m_i^*$. Adding all the cost together, the total complexity is $O(n^2WM^*)$, 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^*.

4. ADAPTIVE CHECKPOINTING

In the adaptive checkpointing scheme, the actual fault arrival process is modeled as a Poisson process with rate λ. At the same time, our goal is to tolerate up to k faults for each job during task execution. We first limit ourselves to a single job here, and then outline the extension to a set of multiple periodic tasks.

We next determine the maximum value of E for the Poisson-arrival and the k-fault-tolerant schemes beyond which these schemes will always miss the job deadline. Our proposed adaptive checkpointing scheme is more likely to meet the task deadline even when E exceeds these threshold values. If the Poisson-arrival scheme is used, the effective task execution time in the absence of faults must be less than the deadline D if the probability of timely completion of the task in the presence of faults is to be nonzero. This implies that $E + [E/(\sqrt{2C/\lambda}) - 1]C \leq D$, from which we get the threshold:

$$E_{\lambda th} = (D + C)/(1 + \sqrt{\lambda C/2}) \tag{2}$$

Here $[E/(\sqrt{2C/\lambda}) - 1]$ refers to the number of checkpoints. The reexecution time due to rollback is not included in the formula for $E_{\lambda th}$. If E exceeds $E_{\lambda th}$ for the Poisson-arrival approach, the probability of timely completion of the task is simply zero. Therefore, beyond this threshold, the checkpointing interval must be set by exploiting the slack time, instead of utilizing the optimum checkpointing interval for the Poisson-arrival approach. The checkpointing interval I_m that barely allows timely completion in the fault-free case is given by $E + (E/I_m - 1)C = D$ from which it follows that $I_m = EC/(D + C - E)$. To decrease the checkpointing cost, we set the checkpointing interval to $2I_m$ in our adaptive scheme.

A similar threshold on the execution time can easily be calculated for the k-fault-tolerant scheme. In order to satisfy the k-fault-tolerant requirement, the worst-case reexecution time is incorporated. The following inequality must hold:

$$E + [E/(\sqrt{EC/k}) - 1]C + k\sqrt{EC/k} \leq D.$$

This implies the following threshold on E:

$$E_{kth} = [(D + C) + 2kC] - 2\sqrt{kC(D + C) + (kC)^2} \tag{3}$$

If the execution time E exceeds E_{kth}, the k-fault-tolerant checkpointing scheme cannot provide a deterministic guarantee to tolerate up to k faults.

4.1 Checkpointing Algorithm for a Single Job

The adaptive checkpointing algorithm attempts to maximize the probability that the job completes before its deadline despite the arrival of faults as a Poisson process with rate λ. A secondary goal is to tolerate, as for a possible,

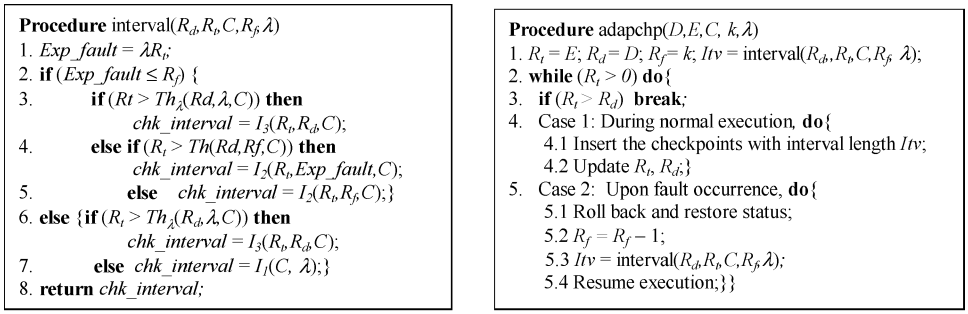

Fig. 5. (a) Procedure for calculating the checkpointing interval and (b) adaptive checkpointing procedure.

up to k faults. In this way, the algorithm accommodates a predefined fault-tolerance requirement (handle up to k faults) as well as dynamic fault arrivals modeled by the Poisson process. We list below some notation that we use in our description of the algorithm:

1. $I_1(C, \lambda) = \sqrt{2C/\lambda}$ denotes the checkpointing interval for the Poisson-arrival approach.
2. $I_2(E, k, C) = \sqrt{EC/k}$ denotes the checkpointing interval for the k-fault-tolerant approach.
3. $I_3(E, D, C) = 2EC/(D + C - E)$ denotes the checkpointing interval if the Poisson-arrival approach is not feasible for timely task completion.
4. R_t denotes the remaining execution time. It is obtained by subtracting from E the amount of time the job has executed (not including checkpointing and recovery). This parameter is updated during job execution.
5. R_d denotes the time left before the deadline. It is obtained by subtracting the current time from D. This parameter is repeatedly updated during job execution.
6. R_f denotes an upper bound on the remaining number of faults that must be tolerated. This parameter is also repeatedly updated during job execution.
7. The threshold $Th_\lambda(R_d, \lambda, C)$ is obtained by replacing D with R_d in (2). If the remaining time R_t is greater than this threshold, the task will miss its deadline even if no additional fault occurs in the system.
8. The threshold $Th(R_d, R_f, C)$ is obtained by replacing D with R_d and k with R_f in (3). If the remaining time R_t is greater than this threshold, the R_f-fault-tolerant requirement cannot be satisfied.

The procedure interval $(R_d, R_t, C, R_f, \lambda)$ for calculating the checkpointing interval is described in Figure 5(a), and the adaptive checkpointing scheme adapchp(D, E, C, k, λ) is described in Figure 5(b). The adaptive checkpointing procedure is event driven and the checkpointing interval is adjusted when a fault occurs and rollback recovery is performed.

In the checkpointing interval procedure, we first calculate the number of faults *Exp_fault* that are expected to occur in the remaining time R_t (Line 1). If *Exp_fault* is less than or equal to R_f, the k-fault-tolerant requirement is deemed to be more stringent than the Poisson-arrival criterion (Line 2). In Line 3, a check is performed to see if R_t exceeds the threshold $Th_\lambda(R_d, \lambda, C)$. If this condition is satisfied, the checkpointing interval is set to $I_3(R_t, R_d, C)$. In Line 4, a check is performed to see if R_t exceeds threshold $Th(R_d, R_f, C)$ but is below $Th_\lambda(R_d, \lambda, C)$. If this condition is satisfied, the checkpointing interval is set to $I_2(Rt, Exp\_fault, C)$. If the k-fault-tolerant threshold is met, the checkpointing interval is set to $I_2(R_t, R_f, C)$ in Line 5. Lines 6 and 7 handle the case when the k-fault-tolerant requirement is deemed to be less stringent than the Poisson-arrival criterion. In Line 6, if $Th_\lambda(R_d, \lambda, C)$ is exceeded, the checkpointing interval is set to $I_3(R_t, R_d, C)$. If this threshold is met, the checkpointing interval is set to $I_1(C, \lambda)$ in Line 7. Line 8 returns the interval value. In the adaptive checkpointing procedure, Line 1 initializes the parameters. In Line 2, a check is performed to see if the task has been completed. Line 3 checks for the deadline constraint. Line 4 handles the case for normal execution. It inserts checkpoints and updates R_d and R_t. Line 5 handles the case for fault occurrences. Line 5.1 rolls back to the nearest checkpoint and loads the stored status, line 5.2 updates the number of faults to be tolerated, line 5.3 recalculates the checkpointing interval, and line 5.4 resumes execution.

4.2 Simulation Results on Adaptive Checkpointing

We carried out a set of simulation experiments to evaluate the adaptive checkpointing scheme (referred to as ADT) and to compare it with the Poisson-arrival and the k-fault-tolerant checkpointing schemes. Faults are injected into the system using a Poisson process with various values for the arrival rate λ. The unit of time used here is milliseconds. Due to the stochastic nature of the fault arrival process, the experiment is repeated 10,000 times for the same task and the results are averaged over these runs. We are interested here in the probability P that the task completes on time—either on or before the stipulated deadline. As in Liu [2000], we use the term task utilization U to refer to the ratio E/D. We separately consider the following cases:

(1) $\lambda < 0.002$ and $U < 0.7$. This case corresponds to low fault arrival rate and low task utilization. For this case, the performances of the three schemes, measured by the probability of timely completion of the task, are comparable.

(2) $\lambda > 0.002$ and $U > 0.7$. This case corresponds to a relatively high fault arrival rate as well as high task utilization. The adaptive checkpointing scheme clearly outperforms the other two schemes in this case; the results are shown in Table I. The value of P is as much as 30% higher for the ADT scheme. Note that even though the results are for $D = 10{,}000$, $C = 10$, and $k = 10$, similar trends were observed for other values of D, C, and k.

(3) $\lambda < 0.002$ and $U \geq 0.92$ (low fault arrival rate and high task utilization). The ADT scheme outperforms the other two schemes in this case (see Table II).

To further illustrate the advantage of the ADT scheme, we note that if we set $U = 0.99$ and $k = 1$ (keeping the values of D and C), the value of P drops to zero

Table I. P Versus λ (Panel A) and P Versus U (Panel B) for $D = 10{,}000$, $C = 10$, and $k = 10$

$U = E/D$	Fault Arrival Rate λ ($\times 10^{-2}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel A) 0.80	0.22	0.658	0.554	0.703
	0.24	0.505	0.476	0.532
	0.28	0.229	0.243	0.273
	0.30	0.152	0.151	0.199
0.82	0.22	0.313	0.276	0.354
	0.24	0.204	0.168	0.235
	0.28	0.052	0.042	0.092
	0.30	0.027	0.035	0.039

λ ($\times 10^{-2}$)	$U = E/D$	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel B) 0.26	0.72	0.996	0.996	0.997
	0.76	0.887	0.888	0.909
	0.78	0.655	0.666	0.715
	0.80	0.357	0.369	0.394
0.30	0.72	0.995	0.996	0.998
	0.76	0.864	0.823	0.872
	0.78	0.589	0.597	0.626
	0.80	0.269	0.275	0.331

Table II. P Versus λ (Panel A) and P Versus U (Panel B) for $D = 10{,}000$, $C = 10$, and $k = 1$

$U = E/D$	Fault Arrival Rate λ ($\times 10^{-4}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel A) 0.92	1.0	0.902	0.945	0.947
	2.0	0.770	0.786	0.831
0.95	1.0	0.659	0.649	0.774
	2.0	0.372	0.387	0.513

λ ($\times 10^{-4}$)	$U = E/D$	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel B) 1.0	0.92	0.902	0.945	0.947
	0.94	0.747	0.818	0.852
	0.96	0.589	0.578	0.643
2.0	0.92	0.770	0.786	0.831
	0.94	0.573	0.558	0.643
	0.96	0.298	0.316	0.437

for both the Poisson-arrival and the k-fault-tolerant schemes if $\lambda > 3 \times 10^{-5}$. In contrast, the proposed ADT scheme continues to provide significant higher value of P as λ increases (Table III).

(4) $\lambda > 0.002$ and $U \geq 0.9$ (high fault arrival rate and high task utilization). Here again the ADT scheme outperforms the other two schemes (Table IV).

In many practical applications, the checkpointing cost C may be higher than used for the above examples, and the fault arrival rate λ may be lower. The fault arrival rate for systems operating in harsh environments has been shown to be in the range of 10^{-2} to 10^{2} per hour [Punnekkat et al. 1999]. We therefore choose value of λ in this range. To demonstrate the applicability of our scheme to such cases, we provide results for higher checkpointing cost and a lower fault

Table III. P Versus λ for $D = 10{,}000$, $C = 10$, and $k = 1$

$U = E/D$	Fault Arrival Rate λ ($\times 10^{-5}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
0.99	1.0	0.893	0.000	0.907
	3.0	0.000	0.000	0.732
	5.0	0.000	0.000	0.515
	7.0	0.000	0.000	0.224

Table IV. P Versus λ (Panel A) and P Versus U (Panel B) for $D = 10{,}000$, $C = 2$, and $k = 10$

$U = E/D$	Fault Arrival Rate λ ($\times 10^{-2}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel A) 0.90	0.22	0.818	0.808	0.825
	0.26	0.598	0.537	0.628
	0.30	0.348	0.285	0.394
0.92	0.22	0.091	0.104	0.131
	0.26	0.014	0.025	0.037
	0.28	0.006	0.015	0.016

λ ($\times 10^{-2}$)	$U = E/D$	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
(Panel B) 0.22	0.910	0.507	0.512	0.552
	0.915	0.326	0.334	0.367
	0.920	0.179	0.175	0.235
0.25	0.910	0.307	0.316	0.368
	0.915	0.145	0.148	0.211
	0.920	0.054	0.083	0.093

Table V. P Versus λ for $D = 10{,}000$, $C = 100$, and $k = 2$

$U = E/D$	Fault Arrival Rate λ ($\times 10^{-4}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT
0.80	0.5	0.900	0.963	0.988
	1.0	0.872	0.897	0.919
	1.5	0.801	0.778	0.834
	1.0	0.677	0.669	0.783
0.85	0.5	0.808	0.738	0.819
	1.0	0.663	0.516	0.729
	1.0	0.565	0.360	0.585
	2.0	0.389	0.257	0.518

arrival rate (see Table V) and (Table VI). The ADT scheme still outperforms the other two schemes.

In conclusion, we note that the ADT scheme is more likely to meet task deadlines when the task utilization is high and the fault arrival rate is not very low. In many cases, up to 50% increase is obtained in the probability of timely task completion.

5. EXTENSIONS OF ADAPTIVE CHECKPINTING TO DVS AND MULTIPLE TASKS

In this section, we present two extensions of the proposed adaptive checkpointing scheme. First, we combine adaptive checkpointing with DVS for achieving

Table VI. P Versus U for $D = 10{,}000$, $C = 500$, and $k = 1$

		Probability of Timely Completion of Tasks, P		
λ ($\times 10^{-5}$)	$U = E/D$	Poisson-Arrival	k-Fault-Tolerant	ADT
1.0	0.72	0.945	0.970	0.994
	0.74	0.940	0.956	0.986
	0.76	0.932	0.943	0.977
	0.78	0.925	0.935	0.970
	0.80	0.918	0.922	0.965
1.5	0.72	0.930	0.950	0.982
	0.74	0.928	0.949	0.974
	0.76	0.921	0.928	0.973
	0.78	0.905	0.924	0.968
	0.80	0.897	0.900	0.962

fault tolerance and energy savings in a unified manner. Second, we extend adaptive checkpointing to a set of multiple tasks.

5.1 ADT_DVS: Adaptive Checkpointing with DVS

We now show how adaptive checkpointing scheme can be combined with DVS to obtain fault tolerance and power savings in real-time systems. We consider adaptive intra-task voltage scaling, wherein the processor speed is scaled up in response to a need for increased slack for checkpointing, and scaled down to save power if the slack for a lower speed is adequate. We consider a two-speed processor here—the extension to more than two speeds appears to be straightforward and it is left for future work. For the sake of simplicity, we use the terms processor speed and processor frequency interchangeably.

We use the same notation as described in Section 4.1. In addition, we are given the following:

1. A single processor with two speeds f_1 and f_2. Without loss of generality, we assume that $f_2 = 2f_1$.
2. The processor can switch its speed in a negligible amount of time (relative to the task execution time).
3. The number of computation cycles N for the task in the fault-free condition.

The objective here consists of two priorities. The first priority is to maximize the probability that the task meets its deadline in the presence of faults. The second priority is to reduce energy consumption through DVS.

We note that if supply voltage V_{dd} is used for a task with N single-cycle instructions, the energy consumption is proportional to NV_{dd}^2 [Benini and Micheli 1998]. We also note that the clock period is proportional to $V_{dd}/(V_{dd} - V_t)^2$, where V_t is the transistor threshold voltage [Benini and Micheli 1998]. We assume here without loss of generality that $V_t = 0.8$ V, and the supply voltage V_{dd1} corresponding to speed f_1 is 2.0 V. Using the formula for the clock period, we find that the supply voltage V_{dd2} corresponding to speed f_2 is 2.8 V.

Let R_c be the number of instructions of the task that remain to be executed at the time of the voltage scaling decision. Let c be the number of clock cycles that a single checkpoint takes. We first determine if processor frequency f

```
Procedure adap_dvs(D,N, c,k,λ)
1. R_c = N; R_d = D; R_f = k;
2. if (t_est(R_c, f_1) ≤ R_d ) f = f_1; else f = f_2;
3. Itv = interval(R_d,R_c/f, c/f,R_f,λ);
4. while (R_f>0) do{
5.   if (R_t > R_d/f) break;
6.   Case 1: During normal execution, do{
       6.1 Insert checkpoints with interval length Itv;
       6.2 Update R_c, R_d according to speed f;}
7.   Case 2: Upon fault occurrence, do{
       7.1 Roll back and restore status;
       7.2 R_f = R_f − 1;
       7.3 if (t_est(R_c) ≤ R_d) f = f_1; else f = f_2;
       7.4 Itv = interval (R_d,R_c/f,c/f,R_f,λ);
       7.5 Resume execution; }}
```

Fig. 6. Energy-aware adaptive checkpointing procedure.

can be used to complete the task before the deadline. As before, let R_d be the amount of time left before the task deadline. The checkpointing cost C at frequency f is given by $C = c/f$. Let t_{est} be an estimate of the time that the task has to execute in the presence of faults and with checkpointing. The expected number of faults for the duration t_{est} is λt_{est}. We are assuming here that the checkpointing cost is negligible compared to the time for forward execution and rollback recovery; hence even though no faults occur during checkpointing, the expected number of faults is λt_{est}. To ensure λt_{est}-fault-tolerance during task execution, the checkpointing interval must be set to $\sqrt{t_{est}C/(\lambda t_{est})} = \sqrt{C/\lambda} = \sqrt{c/(\lambda f)}$. Now, the parameter t_{est} can be expressed as follows:

$$t_{est} = \frac{R_c}{f} + \lambda t_{est}\sqrt{\frac{c}{\lambda f}} + \frac{c}{f}\frac{R_c/f}{\sqrt{c/(\lambda f)}}. \tag{4}$$

The first term on the right-hand side of (4) denotes the time for forward execution, the second term denotes the recovery cost for λt_{est} faults, and the third term denotes the checkpointing cost. From (4), we get:

$$t_{est} = \frac{R_c(1+\sqrt{\lambda c/f})}{f(1-\sqrt{\lambda c/f})}.$$

We consider the voltage scaling (to frequency f) to be feasible if $t_{est} \le R_d$. This forms the basis of the energy-aware adaptive checkpointing procedure *adap_dvs* described in Figure 6. At every DVS decision point, an attempt is made to run the task at the lowest-possible speed. If we have multiple processor speeds, we can perform an exhaustive examination for all the speeds. The lowest speed that satisfies $t_{est} \le R_d$ will be selected. Here we can see the computational complexity is linear in the number of speeds. Furthermore, since the estimate of test requires only simple computation, incorporating multiple speeds will not affect the applicability of online adaptive checkpointing for real-time systems.

5.1.1 *Simulation Results on ADT_DVS.* We compare the adaptive DVS scheme, denoted by ADT_DVS, with the Poisson-arrival and k-fault-tolerant

Table VII. P Versus λ (Panel A) and P Versus U (Panel B) for $D = 10{,}000$, $c = 10$, and $k = 2$

U	Fault Arrival Rate λ ($\times 10^{-4}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT_DVS
(Panel A) 0.90	0.5	0.976	0.999	1.000
	1.0	0.963	0.991	1.000
	1.5	0.942	0.988	1.000
	2.0	0.919	0.972	1.000
0.95	0.5	0.790	0.704	1.000
	1.0	0.648	0.508	1.000
	1.5	0.501	0.367	1.000
	2.0	0.385	0.244	1.000

λ ($\times 10^{-4}$)	U	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT_DVS
(Panel B) 1.0	0.92	0.924	0.960	1.000
	0.94	0.750	0.735	1.000
	0.96	0.549	0.000	1.000
	0.98	0.000	0.000	1.000
	1.00	0.000	0.000	1.000
2.0	0.92	0.799	0.849	1.000
	0.94	0.530	0.439	1.000
	0.96	0.229	0.000	1.000
	0.98	0.000	0.000	1.000
	1.00	0.000	0.000	1.000

schemes in terms of the probability of timely completion and energy consumption. We use the same experimental set-up as in Section 4.2. In addition, we consider the normalized frequency values $f_1 = 1$ and $f_2 = 2$. First, we assume that both the Poisson-arrival and the k-fault-tolerant schemes, use the lower speed f_1. The task execution time at speed f_1 is chosen to be less than D—$N/f_1 < D$. The task utilization U in this case is simply $N/(f_1 D)$. Our experimental results are shown in Table VII. The ADT_DVS scheme always leads to timely completion of the task by appropriately choosing segments of time when the higher frequency f_2 is used. The other two schemes provide a rather low value for P, and for larger values of λ and U, P drops to zero. The energy consumption for the ADT_DVS scheme is slightly higher than that for the other two schemes; however, on average, the task runs at the lower speed f_1 for as much as 90% of the time. The combination of adaptive checkpointing and DVS utilizes the slack effectively and stretches the task completion time to as close to the deadline as possible.

Next we assume that both the Poisson-arrival and the k-fault-tolerant schemes use the higher speed f_2. The task execution time at speed f_2 is chosen to be less than D—$N/f_2 < D$, and the task utilization here is $N/(f_2 D)$. Table VIII shows that since even though ADT_DVS uses both f_1 and f_2, adaptive checkpointing allows it to provide a higher value for P than the other two methods that use only the higher speed f_2. The energy consumption for ADT_DVS is up to 50% less than for the other two methods for low to moderate values of λ and U (see Table IX). When either λ or U is high, the energy consumption of ADT_DVS is comparable to that of the other two schemes. (Energy is measured by summing the product of the square of the voltage and the

Table VIII. P Versus λ for $D = 10{,}000$, $c = 10$, and $k = 1$

U	Fault Arrival Rate λ ($\times 10^{-4}$)	Probability of Timely Completion of Tasks, P		
		Poisson-Arrival	k-Fault-Tolerant	ADT_DVS
0.95	0.8	0.898	0.939	0.965
	1.2	0.841	0.868	0.912
	1.6	0.754	0.785	0.871
	2.0	0.706	0.695	0.791

Table IX. Energy Consumption Versus λ (Panel A) and Energy Consumption Versus U (Panel B) for $D = 10{,}000$, $c = 10$, and $k = 10$

U	Fault Arrival Rate λ ($\times 10^{-4}$)	Energy Consumption		
		Poisson-Arrival	k-Fault-Tolerant	ADT_DVS
(Panel A) 0.60	2.0	25,067	26,327	21,568
	4.0	25,574	26,477	21,642
	6.0	25,915	26,635	21,714
	8.0	26,277	26,806	22,611

λ ($\times 10^{-4}$)	U	Energy Consumption		
		Poisson-Arrival	k-Fault-Tolerant	ADT_DVS
(Panel B) 5.0	0.10	4,295	4,909	2,508
	0.20	8,567	9,335	4,791
	0.30	12,862	13,862	7,026
	0.40	17,138	17,990	9,223
	0.50	21,474	22,300	15,333

number of computation cycles over all the segments of the task.) This is expected, since ADT_DVS attempts to meet the task deadline as the first priority and if either λ or U is high, ADT_DVS seldom scales down the processor speed.

5.2 Adaptive Checkpointing for Multiple Tasks

We next present the extended adaptive checkpointing procedure for a set of multiple tasks.

5.2.1 *System Model.* Our objective here is to extend the *adapchp*(D, E, C, k, λ) procedure to a set of multiple real-time tasks. We are given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$, T_i is the period of τ_i, D_i is the relative deadline ($D_i \leq T_i$), and E_i is the computation time under fault-free conditions. We are also aiming to tolerate up to k faults for each task instance.

The task set is first scheduled offline with a general scheduling algorithm scheme under fault-free conditions. Here we employ the earliest-deadline-first (EDF) algorithm [Liu 2000]. Alternative base scheduling techniques such as the rate-monotonic algorithm can also be used. We are using EDF here to simplify the presentation of the procedure. A sequence of m jobs $\Phi = \{\theta_1, \theta_2, \ldots, \theta_m\}$ is obtained for each hyperperiod. We further denote each job $\theta_i, 1 \leq i \leq m$, as a tuple $\theta_i = \langle a_i, b_i, c_i \rangle$, where a_i is the starting time, b_i is the execution time, and c_i is the deadline for θ_i. All these parameters are known *a priori*. Note that a_i is the absolute time, when θ_i starts execution instead of the relative release time, execution time is equal to that for the corresponding task, and

job deadline is equal to the task deadline plus the product of task period and corresponding number of periods. In addition, since we are employing EDF, we have $c_1 \leq c_2 \leq \cdots \leq c_m$.

Based on the job set Φ, we develop a checkpointing scheme that inserts checkpoints to each job by exploiting the slacks in a hyperperiod. The key issue here is to determine an appropriate value for the deadline that can be provided as an input parameter to the *adapchp*(D, E, C, k, λ) procedure. This deadline must be no greater than the exact job deadline such that the timing constraint can be met, and it should also be no less than the job execution time such that time redundancy can be exploited for fault-tolerance. To obtain this parameter, we incorporate a preprocessing step immediately after the offline EDF scheduling is carried out. The resulting values are then provided for subsequent online adaptive checkpointing procedure.

We denote the slack time for job θ_i, $1 \leq i \leq m$, as h_i. We also introduce the concept of checkpointing deadline v_i, which is the deadline parameter provided to the *adapchp*(D, E, C, k, λ) procedure. It is defined as the sum of job execution time b_i and slack time h_i. Furthermore, for the sake of convenience of problem formulation, we add a pseudojob θ_0, which has parameters $a_0 = b_0 = c_0 = h_0 = 0$. The benefit of introducing θ_0 will be demonstrated shortly.

5.2.2 *Linear-Programming Model.* Now we illustrate the preprocessing step needed to obtain the slack time h_i for each job θ_i, $1 \leq i \leq m$. According to the deadline constraints, we have

$$\max\{a_i + b_i + h_i, a_{i+1}\} + b_{i+1} + h_{i+1} \leq c_{i+1}, \quad \text{where } 0 \leq i \leq m-1.$$

On the other hand, to ensure each job has the minimum time-redundancy for fault-tolerance, we require the slack of each job to be greater than a constant threshold value Q, which is defined as a given number of checkpoints. Then we have

$$h_i \geq Q, \quad \text{where } 1 \leq i \leq m.$$

The problem now is that of allocating the slacks appropriately to the jobs subject to the above constraints. If we choose the total slack as the optimization function, then the problem is how to maximize the sum of all slacks $\sum_{i=1}^{m} h_i$. This is a linear-programming problem and h_i can be obtained by employing linear programming solver tools such as BPMPD [NEOS]. Since this processing step is done offline prior to the actual execution of the job set, the additional computation is acceptable. In our experiments, the CPU time is less than 1 s for moderate problem size (number of jobs < 20). For greater problem size, the CPU time is normally less than 1 min.

5.2.3 *Modification to the adapchp Procedure.* After obtaining the slacks for all jobs offline through the preprocessing step, we next incorporate them into the online adaptive checkpointing scheme. The *adapchp* procedure of Figure 5(b) needs to be modified due to the following reasons.

(1) In the *adapchp* procedure, a job is declared to be unschedulable if the deadline is missed. When this happens, the execution of the entire set of jobs is

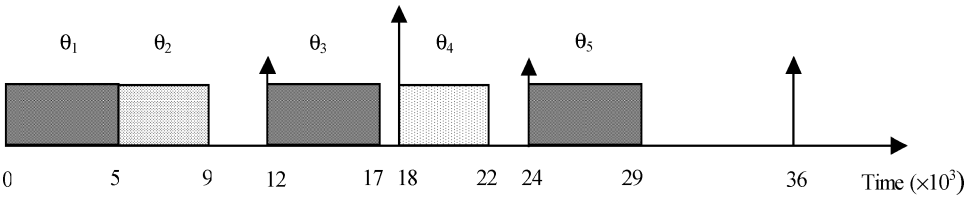

Fig. 7. Schedule of jobs with $\tau_1 = (5{,}000, 7{,}000, 12{,}000)$ and $\tau_2 = (4{,}000, 11{,}000, 18{,}000)$.

terminated. Here we are using checkpointing deadline as an input parameter for the adaptive checkpointing procedure. Sometimes, however, a job deadline is not really missed even if its checkpointing deadline is missed. Therefore it is not correct to always declare the job as unschedulable if its checkpointing deadline is missed. We overcome this problem by adopting the following solution: If the checkpointing deadline is missed but the actual job deadline is not missed, the job continues execution without inserting any more checkpoints.

(2) Since the actual execution time of a job is affected by fault arrival pattern, it is necessary to adjust the starting time and slack of the next job during execution. In our proposed solution, during actual execution, once the current job finishes its execution, the *adapchp* procedure returns its completion time. The next job starts its execution based on the previous job's completion time and its own precomputed starting time, which is obtained offline. Meanwhile, the precomputed slack of the next job is adjusted accordingly. We explain this formally below.

Let the actual starting time of θ_i be a'_i, and the actual execution time be b'_i. Then the actual starting time a'_{i+1} of the next job θ_{i+1} can be calculated as

$$a'_{i+1} = \max\{a_{i+1}, a'_i + b'_i\}.$$

The actual slack time h'_{i+1} of θ_{i+1} is adjusted as

$$h'_{i+1} = h_{i+1} - (a'_{i+1} - a_{i+1})$$

and the actual checkpointing deadline v'_{i+1} is adjusted as

$$v'_{i+1} = \begin{cases} b_{i+1} & \text{if} \quad h'_{i+1} < 0 \\ b_{i+1} + h'_{i+1} & \text{if} \quad h'_{i+1} \geq 0. \end{cases}$$

Then we can apply $adapchp(v'_{i+1}, b_i, C, k, \lambda)$ to job θ_{i+1}; the procedure returns b'_{i+1}, the value which is the actual execution time of θ_{i+1} including checkpointing and fault recovery cost.

5.2.4 *Experimental Results.* We consider two tasks $\tau_1 = (5{,}000, 7{,}000, 12{,}000)$ and $\tau_2 = (4{,}000, 11{,}000, 18{,}000)$.

After offline scheduling is carried out using EDF, we obtain the sequence of jobs shown in Figure 7. Note that θ_1, θ_3, and θ_5 are instances of τ_1, and θ_2 and θ_4 are instances of τ_2.

The parameters of the jobs are tabulated in Table X.

Here we assume that the single checkpointing cost is 10, and we require that at least 20 checkpoints are inserted for each slack. The slack values generated

Table X. Job Parameters for the Two-Task Example

Job	Starting Time (a_i)	Execution Time (b_i)	Deadline (c_i)
θ_1	0	5,000	7,000
θ_2	5,000	4,000	11.000
θ_3	12,000	5,000	19,000
θ_4	18,000	4,000	29,000
θ_5	24,000	5,000	31,000

Table XI. Results on Adaptive Checkpointing for Two Tasks and Five Jobs: (Panel A) P Versus λ for $c = 10$, and $k = 5$ and (Panel B) P Versus λ for $c = 20$, and $k = 5$

	Probability of Timely Completion of Tasks, P		
$\lambda(\times 10^{-4})$	ADT_MUL	Poisson-Arrival	k-Fault-Tolerant
Panel A			
6	1.000	0.996	1.000
8	0.998	0.992	1.000
10	0.987	0.967	0.983
12	0.976	0.941	0.970
14	0.914	0.870	0.899
16	0.823	0.765	0.762
18	0.696	0.647	0.610
20	0.547	0.484	0.457

	Probability of Timely Completion of Tasks, P		
$\lambda(\times 10^{-4})$	ADT_MUL	Poisson-Arrival	k-Fault-Tolerant
Panel B			
6	0.930	0.884	0.815
8	0.781	0.731	0.565
10	0.542	0.487	0.370
12	0.347	0.306	0.189
14	0.153	0.149	0.086
16	0.076	0.057	0.029
18	0.028	0.018	0.013
20	0.004	0.003	0.004

by the linear programming solver are $h_1 = 1{,}800$, $h_2 = 200$, $h_3 = 2{,}000$, $h_4 = 1{,}800$, and $h_5 = 200$.

We compare the multitask adaptive scheme, denoted by ADT_MUL, with the Poisson-arrival and k-fault-tolerant schemes in terms of the probability of timely completion. As in Section 4.2, faults are injected into the system using a Poisson process with various values for the arrival rate λ, the experiment is repeated 10,000 times for the same task, and the results are averaged over these runs. The experimental results are shown in Table XI. These results show that the adaptive scheme provides a higher probability of timely completion for multitask systems than the other two schemes.

6. CONCLUDING REMARKS

We have shown how dynamic adaptation can be used for fault tolerance and power management in embedded systems. Fault tolerance is achieved via checkpointing and power management is carried out using dynamic voltage

scheduling. We have presented feasibility-of-scheduling tests for checkpointing schemes that use a fixed checkpointing interval for real-time tasks. These tests provide the criteria under which checkpointing can provide fault tolerance and real-time guarantees. We have considered two different fault arrival models: up to k faults for a job, and up to k faults for a hyperperiod. We have also presented techniques to determine a fixed checkpointing interval in an offline manner. Following this, we have presented an adaptive checkpointing scheme for real-time systems in which a variable checkpointing interval can be determined dynamically. The checkpointing interval is dynamically adjusted during task execution, and checkpoints are inserted based not only on the slack in task execution but also on the occurrences of faults during task execution. We have presented simulation results to show that compared to previous methods, the proposed adaptive checkpointing approach increases the likelihood of timely task completion in the presence of faults. Together with the feasibility tests, the adaptive checkpointing method provides a useful framework for dependable computing in the presence of faults in real-time systems.

Next, we presented a unified approach for adaptive checkpointing and DVS for a real-time task executing in an embedded system. This approach provides fault tolerance and facilitates dynamic power management. We have presented simulation results to show that the proposed approach significantly reduces power consumption and increases the probability of tasks completing correctly on time despite the occurrences of faults. We have also extended the adaptive checkpointing scheme to a set of multiple tasks. A linear-programming model is employed in an offline manner to obtain the relevant parameters that are used by the adaptive checkpointing procedure. Simulation results show that the adaptive scheme is also capable of providing a high probability of timely completion in the presence of faults for a set of multiple tasks.

We are currently extending this work to unified checkpointing and DVS for a set of multiple tasks. We are also investigating checkpointing for distributed systems with multiple processing elements, where data dependencies and internode communication have a significant impact on the checkpointing strategy. We are examining ways to relax the restrictions of zero fault detection latency, state restoration costs, as well as the assumption of no fault occurrence during checkpointing and rollback recovery. It is quite straightforward to model nonzero fault detection and state restoration costs. Let the single checkpoint cost be C, single fault detection cost be B, and single rollback and state restoration cost be H, then we can incorporate the additional costs by replacing C with $C' = C + B + H$ in all the expressions. It appears though that it is more difficult to relax the assumption that no faults occur during checkpointing.

REFERENCES

AMD. AMD PowerNow! Technology. http://www.amd.com/epd/processors/6.32bitproc/8.amdk6fami/x24267/24267a.pdf.

BAHBHA, N., TELCH, J., AND ZITZLER, E. 2001. Hybrid global/local search strategies for DVS in embedded microprocessors. In *Proceedings of the International Symposium on Hardware/Software Codesign*, 243–248.

BENINI, L. AND MICHELI, G. D. 1998. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Norwell, MA.

BETTATI, R., BOWEN, N. S., AND CHUNG, J. Y. 1992. Checkpointing imprecise computation. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*. 45–49.

BUSHNELL, M. L. AND AGRAWAL, V. D. 2000. *Essentials of Electronic Testing*. Kluwer Academic Publishers, Norwell, MA.

CHANDY, K. M., BROWNE, J. C., DISSLY, C. W., AND UHRIG, W. R. 1975. Analytic models for rollback and recovery strategies in data base systems. *IEEE Trans. Software Eng*. 1 (March), 100–110.

DUDA, A. 1983. The effects of checkpointing on program execution time. *Inf. Process. Lett. 16* (June), 221–229.

DUPONT, E., NICOLAIDIS, M., AND ROHR, P. 2002. Embedded robustness IPs for transient-error-free Ics. *IEEE Des. Test Comput. 19*, 54–68.

INTEL. Intel embedded SL Enhanced 486DX2 processor. http://www.intel.com.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 197–202.

KWAK, S. W., CHOI, B. J., AND KIM, B. K. 2001. An optimal checkpointing-strategy for real-time control systems under transient faults. *IEEE Trans. Reliab.* 50 (Sep.), 293–301.

LEE, H., SHIN, H., AND MIN, S. 1999. Worst case timing requirement of real-time tasks with time redundancy. In *Proceedings of the Real-Time Computing Systems and Applications*. 410–414.

LIU, J. W. 2000. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ.

LUO, J. AND JHA, N. 2000. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*. 357–364.

MOTOROLA. Motorola 6805 processor. http://www.motorola.com.

NEOS SOLVERS. http://www-neos.mcs.anl.gov/neos/server-solver-types.html.

POP, P., ELES, P., AND PENG, Z. 2000. Schedulability analysis for systems with data and control dependencies. In *Proceedings of the Euromicro RTS*. 201–208.

PUNNEKKAT, S., BURNS, A., AND DAVIS, R. 1999. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the International Conference on Dependable Computing for Critical Applications*. 361–378.

PUNNEKKAT, S., BURNS, A., AND DAVIS, R. 2001. Analysis of checkpointing for real-time systems. *Real-Time Syst. J.* (Jan.), 83–102.

QUAN, G. AND HU, X. 2001. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the Design Automation Conference*. 828–833.

SCMITZ, M. T., AL-HASHIMI, B. M., AND ELES, P. 2002. Energy efficient mapping and scheduling for DVS enabled distributed embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference*. 514–521.

SHIN, K. G. AND LEE, Y.-H. 1984. Error detection process—Model, design and its impact on computer performance. *IEEE Trans. Comput.* 33 (June), 529–540.

SHIN, K. G., LIN, T., AND LEE, Y. 1987. Optimal checkpointing of real-time tasks. *IEEE Trans. Comput.* 36 (Nov.), 1328–1341.

SHIN, Y., CHOI, K., AND SAKURAI, T. 2000. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design*. 365–368.

SIEWIOREK, D. AND SWARZ, R. 1998. *Reliable Computer Systems: Design and Evaluation*. A. K. Peters Ltd, Natick, MA.

ZIV, A. AND BRUCK, J. 1997. An on-line algorithm for checkpoint placement. *IEEE Trans. Comput. 46* (Sep.), 976–985.

Received January 2003; revised July 2003; accepted August 2003