Energy-Aware Deterministic Fault Tolerance in Distributed Real-Time Embedded Systems*

Ying Zhang
Electrical and Computer Engineering
Duke University
Durham, NC 27708, USA
yingzh@ee.duke.edu

Robert Dick
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
dickrp@ece.northwestern.edu

Krishnendu Chakrabarty
Electrical and Computer Engineering
Duke University
Durham, NC 27708, USA
krish@ee.duke.edu

ABSTRACT

We investigate a unified approach for fault tolerance and dynamic power management in distributed real-time embedded systems. Coordinated checkpointing is used to achieve fault tolerance, and power management is carried out using dynamic voltage scaling. We present feasibility-of-scheduling tests for coordinated checkpointing schemes for a constant processor speed as well as for DVS-enabled processors that can operate at variable speeds. Simulation results based on the CORDS hardware/software co-synthesis system show that, compared to fault-oblivious methods, the proposed approach significantly reduces power consumption while guaranteeing timely task completion in the presence of faults.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Algorithms, Performance, Design, Reliability

Keywords

Real-time systems, fault tolerance, checkpointing, voltage scaling.

1. INTRODUCTION

Fault tolerance techniques are needed to ensure the dependability of embedded systems that operate in harsh environmental conditions. Tolerance to transient faults is especially important due to reduced noise margins caused by lower supply voltages. In addition, embedded systems are often energy-constrained and they are frequently used for hard real-time applications, which require strict adherence to task deadlines.

Many real-time systems consist of a distributed system of embedded processors. In this paper, we investigate a unified approach that provides fault tolerance and dynamic power management (DPM) in distributed real-time embedded systems. The proposed approach provides deterministic guarantees on the timely completion of tasks despite the occurrences of transient faults.

Dynamic voltage scaling (DVS) is a popular technique for reducing power consumption [1]. Fault tolerance can be achieved in embedded systems through checkpointing [2]. 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.

A number of techniques have been presented in the literature on scheduling real-time tasks for DVS-enabled uniprocessor systems under fault-free conditions [1]. The problem of fault tolerance in distributed systems has also received attention [3, 4]. Dynamic power management in distributed embedded systems has recently emerged as an active research area. DVS has been applied to distributed real-time systems in which the inter-job communication cost is assumed to be zero. Some DVS techniques also consider communication costs [5]. However, none of the above papers address fault tolerance and power management in conjunction for distributed real-time embedded systems. While the combination of checkpointing and DVS has recently been proposed for energy-aware fault tolerance in real-time embedded systems [2], these techniques are restricted to uniprocessor systems.

We first present feasibility-of-scheduling tests for distributed real-time task sets when checkpointing is carried out for constant processor speed. The real-time application is modeled using a directed acyclic graph (DAG). Following this, we extend these feasibility tests to DVS-enabled variable-speed processors.

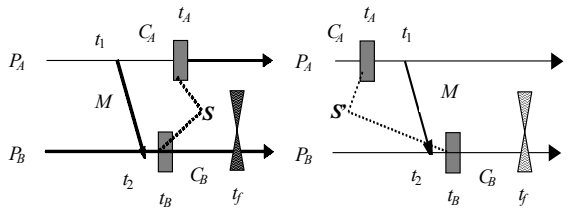
The contributions of this paper are as follows. First, it introduces the concept of *k*-fault-tolerance in distributed real-time systems and presents a deterministic fault-tolerant scheme. Second, it achieves global system consistency and saves fault-recovery time through coordinated checkpointing in distributed real-time systems. Finally, it investigates the relationships between fault tolerance and dynamic power management.

2. CHECKPOINTING IN DISTRIBUTED SYSTEMS

In this section, we describe previously proposed checkpointing methods, and choose an appropriate checkpointing scheme for distributed real-time systems.

* The work of Y. Zhang and K. Chakrabarty was sponsored in part by DARPA, and administered by the Army Research Office under Emergent Surveillance Plexus MURI Award No. DAAD19-01-1-0504.

DAC 2004, June 7-11, 2004, San Diego, California, USA



(a) Consistent state: $t_1 < t_A < t_f$. (b) Inconsistent state: $t_A < t_1 < t_f$.

Figure 1: Consistent and inconsistent states.

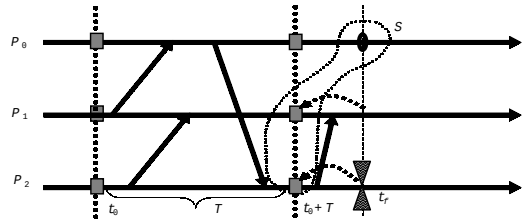


Figure 2: Synchronized checkpointing.

2.1 Consistent System States

In a message-passing system, it is critical to maintain the state consistency of the system. The global state of a message-passing system is a collection of the individual states of all processors and communication channels [6]. A consistent system is one in which, if the checkpointing state of a processor reflects a message receipt, then the checkpointing state of the corresponding sender processor indicates that the message has been sent out [7].

Consider a system of two processors A and B, and suppose both processors perform local checkpointing to store individual states. In the event of a fault, the system uses checkpoints from both processors to determine a global state. Suppose processor A sends a message M to processor B at time t_1 and B receives it at time t_2 ($t_1 < t_2$). Meanwhile, processor A takes a checkpoint C_A at t_A, and processor B takes a checkpoint C_B at t_B, after t_2. Assume a fault occurs at t_f after t_B.

In Figure 1(a), $t_1 < t_A < t_f$, and the global state is S. The content of C_A indicates that message M has been sent to processor B, which is consistent with the actual communication precedence. Hence, S is a consistent state. In Figure 1(b), $t_A < t_1 < t_f$, and the global state is S'. The content of C_A indicates that message M has not yet been sent to processor B. However, as mentioned earlier, the state of C_B reflects the receipt of the message M from A. According to the global state S' composed of C_A and C_B, an apparent discrepancy emerges: message M is received by processor B without being sent by processor A. Thus S' is an inconsistent state.

Inconsistent states result in wasted checkpoint cost and they incur unnecessary recovery time. The worst possible scenario is that all processors have to roll back to the beginning of the program and restart execution. This phenomenon is called the Domino Effect [6]. Restoring a consistent global state when faults cause inconsistencies is a critical problem in the design of reliable distributed systems.

2.2 Uncoordinated vs. Coordinated Checkpointing for Ensuring Consistency

In uncoordinated checkpointing [6], a processor decides when to make a checkpoint without dependence on the other processors. Processors record dependencies among the checkpoints during the fault-free execution. Once a fault occurs in a processor, this processor broadcasts a dependency-request message. Upon receipt of the dependency request, each processor stops execution and replies with the local dependency information. The initiator then calculates the most-recent consistent global checkpoint (defined as the recovery line) based on the received data and broadcasts a rollback request message containing the recovery line. A processor whose current state belongs to the recovery line resumes execution; otherwise, it rolls back to the checkpoint indicated by the recovery line. Uncoordinated checkpointing is susceptible to the Domino Effect; moreover, checkpoints that will never be part of a global consistent state can be taken, and each processor needs to maintain multiple checkpoints.

In coordinated checkpointing [6], processors cooperate to form a consistent global state. Synchronized coordination, which has relatively small protocol overhead, uses a global synchronization signal, i.e., a coarse-grained or fine-grained clock, to trigger the local checkpointing actions at the same time instead of waiting for a request from an initiator; all processors take checkpoints at predefined times according to the global clock. Note that this synchronization signal may (and should) have a frequency dramatically lower than the local clock frequencies of the individual processors, minimizing its power consumption and preserving the assumption of continuous local time, i.e., there are many local clock ticks between subsequent checkpoints. Figure 2 shows an example of synchronized checkpointing. Starting from time t_0, all processors take checkpoints simultaneously and periodically based on a global clock with a checkpointing period of T. Once a fault occurs at time t_f in processor P_2, the consistent state S is formed as follows: P_1 and P_2 roll back to their most recent checkpoints because a message was sent from P_2 to P_1 after the last checkpointing operation, while P_0 remains at its current state since no dependency exists between P_0 and the other two processors. Synchronized checkpointing is easy to implement and cost-effective. Hence, we use this technique for fault tolerance in distributed real-time systems.

3. FEASIBILITY ANALYSIS UNDER CONSTANT SPEED

In this section, we present feasibility analysis based on synchronized checkpointing under constant processor speed.

Given an embedded system specification, a hardware-software co-synthesis system [8] solves the following three problems under fault-free conditions: (1) allocation of the resources composed of processing elements (PEs) and communication links; (2) assignment of jobs or communications on different PEs/links; and (3) task scheduling. Both the scheduling problem and the allocation/assignment problem are NP-complete for distributed systems [9]. To handle problem complexity, we start with a valid resource allocation, task assignment, and schedule produced under the assumption fault-free conditions.

We are given a program modeled by a DAG: $G = (V, E)$. The system is composed of PEs and communication links. All timing parameters for the DAG are based on a global common clock. However, the operational frequency of each PE need not necessarily be the same as the global clock, i.e., the global clock can be used for global synchronization without forcing all PEs to

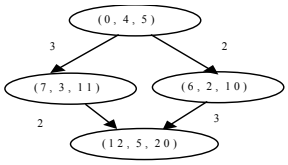


Figure 3: Program modeled by a DAG.

operate at its frequency. Information about distributed system topology, allocation of PEs and communication links, and task assignment is obtained through a system-level synthesis tool such as CORDS [8]. After mapping the DAG to the underlying system and scheduling the jobs (performed by the synthesis tool), each job v_i is modeled by a three-tuple (a_i, t_i, d_i), where a_i is the arrival time, t_i is the fault-free computation time, and d_i is the deadline. Edge e_{ij} represents a message sent from v_i to v_j with communication cost c_{ij}. Figure 3 shows a program modeled by a DAG.

We assume that at most k faults can occur in the system before the maximum deadline over all tasks. The time to store (retrieve) a checkpoint is c_w (c_r). A fault occurring at one PE does not affect the other PEs. The PEs in the system employ synchronized checkpointing to achieve global consistent state, i.e., all PEs take checkpoints simultaneously under a global clock. The checkpointing interval is Δ. Finally, we assume that the protocol overhead is negligible compared to the typical values of execution time of real-time jobs. This assumption is reasonable because only extremely short protocol-specific messages need to be transmitted.

Based on the properties of synchronized checkpointing, we make the following key observations:

(1) For a uniprocessor system, the maximum time penalty for one fault is $\sigma = \Delta + c_w + c_r$.

(2) For a message-passing system that employs a global clock to save checkpoints every Δ units of time, each fault incurs the maximum time penalty σ; i.e., any job whose execution suffers one fault will be delayed by at most σ.

Our goal here is to determine whether the program can be completed using the checkpointing interval Δ without violating its timing constraints, even if k faults occur during program execution.

As an example, consider a real-time program composed of three jobs v_1, v_2, and v_3. These jobs are assigned to three PEs, as illustrated in Figure 4. Each job v_i is modeled by the tuple (a_i, t_i, d_i).

We now examine all the jobs and see if they can meet the timing constraints in the presence of k faults during the execution of each job. These k faults can occur at any time either before or during the execution of v_i.

1) For job v_1: the worst-case finish time in the presence of k faults can be expressed as:

$$wcft_1(k) = a_1 + t_1 + k\sigma + t_1 c_w / \Delta. \quad (1)$$

In this expression, (a_1+t_1) represents the finish time under fault-free conditions, $k\sigma$ represents the time penalty due to rollback recovery in the presence of k faults, and $t_1 c_w/\Delta$ represents the checkpointing cost.

If $wcft_1(k) \le d_1$, v_1 can be completed before its deadline in the presence of k faults. It is straightforward to see that $wcft_1(k)$ is a monotonically increasing function of k, a property that we exploit in subsequent analysis.

2) For job v_2: the worst-case finish time in the presence of k faults can be expressed as:

$$wcft_2(k) = \max\{wcft_1(n_1) + c_{12}, a_2\} + t_2 + (k - n_1)\sigma + t_2 c_w / \Delta.$$

Let $g_2(n_1) = t_2 + (k - n_1)\sigma + t_2 c_w / \Delta$

and $f_1(n_1) = wcft_1(n_1) + c_{12}$. This gives us:

$$wcft_2(k) = \max(f_1(n_1), a_2) + g_2(n_1). \quad (2)$$

Next we divide the k possible faults into two parts: n_1 ($n_1 \le k$) faults that occur before the execution of v_2, and ($k - n_1$) faults that occur during the execution of v_2. For the first group of n_1 faults, it is easy to verify that no matter how these n_1 faults are distributed between execution and communication, the worst-case time for message c_{12} to arrive at PE_2 is $wcft_1(n_1) + c_{12}$. Therefore, we need not distinguish between faults during execution and faults during communication in the remaining analysis.

It still remains to distribute the k fault occurrences such that the worst-case scenario for v_2 is obtained in order to ensure that the deadline will be met in this case. Based on Equation (2) we have three mutually-exclusive cases.

Case 1: $wcft_1(k) + c_{12} \le a_2$. This is illustrated in Figure 5.

Since $wcft_1(k)$ is a monotonically-increasing function of k, we have: $f_1(n_1) = wcft_1(n_1) + c_{12} \le wcft_1(k) + c_{12} \le a_2$. Hence $\max\{f_1(n_1), a_2\} = a_2$, and $wcft_2(k) = a_2 + g_2(n_1)$.

It is shown in Figure 5 that the line segment $f_1(n_1) = wcft_1(n_1) + c_{12}$ ($0 \le n_1 \le k$) always lies below the line corresponding to $h(n_1) = a_1$, while the position of $g_2(n_1)$ relative to $h(n_1)$ is arbitrary. Obviously, the maximum value of $wcft_2(k)$ is obtained when $n_1 = 0$. This means that, in the worst case, all k faults occur during the execution of v_2. It can, therefore, be concluded that $wcft_2(k) = a_2 + g_2(0)$.

In this case, $wcft_2(k)$ is independent of $wcft_1(n_1)$. This follows from the observation that, as long as $n_1 \le k$, the execution of v_1 does not affect the actual starting time of v_2 (see Figure 6).

Case 2: $wcft_1(0) + c_{12} \ge a_2$. We now have:

$$f_1(n_1) = wcft_1(n_1) + c_{12} \ge wcft_1(0) + c_{12} \ge a_2.$$

Hence $\max(f_1(n_1), a_2) = f_1(n_1)$.

It is shown in Figure 7 that the line segment $f_1(n_1) = wcft_1(n_1) + c_{12}$ ($0 \le n_1 \le k$) lies above the line $h(n_1) = a_1$, while the position of $g_2(n_1)$ relative to $h(n_1)$ is arbitrary. Now we have:

$$\begin{aligned} wcft_2(k) &= f_1(n_1) + g_2(n_1) \\ &= wcft_1(n_1) + c_{12} + t_2 + (k - n_1)\sigma + t_2 c_w / \Delta. \end{aligned}$$

According to Equation (1), we have:

$$\begin{aligned} wcft_2(k) &= (a_1 + t_1 + n_1\sigma + t_1 c_w / \Delta) + c_{12} \\ &\quad + t_2 + (k - n_1)\sigma + t_2 c_w / \Delta \\ &= a_1 + t_1 + c_{12} + t_2 + k\sigma + (t_1 + t_2) c_w / \Delta. \end{aligned}$$

We make the interesting observation here that the value of $wcft_2(k)$ is independent of n_1. From a mathematical perspective,

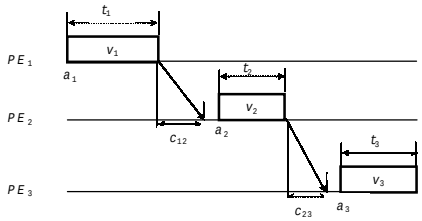


Figure 4: A program composed of three jobs.

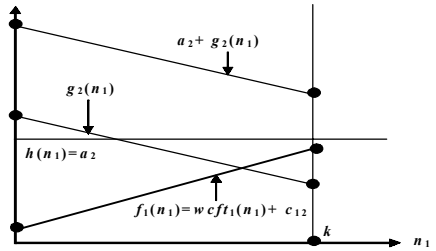


Figure 5: Case 1 corresponding to $wcft_1(k) + c_{12} \le a_2$.

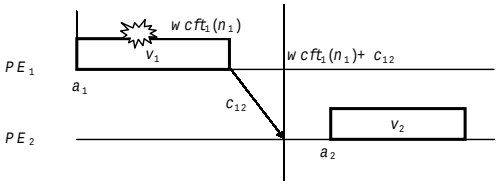


Figure 6: Explanation for Case 1.

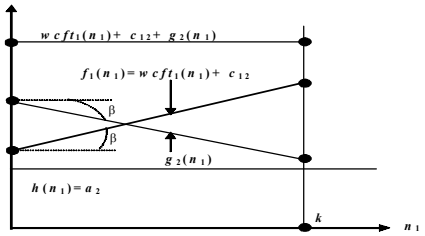


Figure 7: Case 2 corresponding to $wcft_1(0) + c_{12} \ge a_2$.

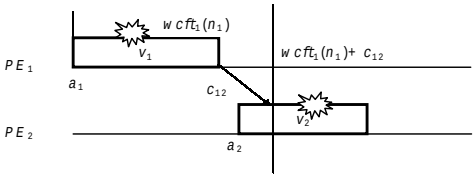


Figure 8: Explanation for Case 2.

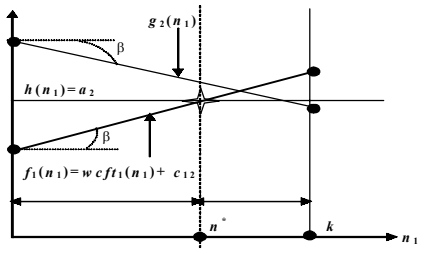


Figure 9: Case 3 corresponding to $wcft_1(0)+c_{12} < a_2 < wcft_1(k)+c_{12}$.

this is because the slopes of $g_2(n_1)$ and $f_1(n_1)$ are equal in magnitude but opposite in sign (as indicated in Figure 7:$\beta = tg^{-1}\sigma$). Therefore, their sum is independent of n_1. This can also be explained as follows. When $f_1(n_1) \ge a_2$, the time when the message from job v_1 is received by job v_2 is always later than the arrival time of v_2. Since v_1's execution is overlapped with v_2's arrival time, we cannot tell whether the delay for v_2 is caused by fault occurrences during v_1 or during v_2. As a result, $wcft_2(k)$ is independent of n_1. This is shown in Figure 8, where the two faults have the same effect on the delay in the execution of v_2.

Based on this property, we can further express $wcft_2(k)$ as: $wcft_2(k) = f_1(0) + g_2(0)$.

Case 3: $wcft_1(0) + c_{12} < a_2 < wcft_1(k) + c_{12}$; see Figure 9.

In this case, depending on the value of n_1, $f_1(n_1)$ can either be greater or less than a_2. It is shown in Figure 9 that the line segment $f_1(n_1) = wcft_1(n_1)+c_{12}\ (0 \le n_1 \le k)$ has an intersection point with the line $h(n_1) = a_1$, while the position of $g_2(n_1)$ relative to $h(n_1)$ is arbitrary. Now we determine the value of n_1 to maximize $wcft_2(k)$.

First, we solve the equation $f_1(n^*) = a_2$, and get the value of n^*, as indicated in Figure 9. Next, we divide n_1 into two intervals based on n^*, obtain two local maxima separately, and choose the greater one as the global maximum. Consider the two cases below.

i) $n_1 \in [0, n^*)$.

We have $f_1(n_1) = wcft_1(n_1) + c_{12} < wcft_1(n^*) + c_{12} = a_2$. Hence $\max\{f_1(n_1), a_2\} = a_2$, and $wcft_2(k) = a_2 + g_2(n_1)$.

Similar to the case in Figure 5, the maximum value of $wcft_2(k)$ is obtained when $n_1 = 0$. In addition, $n_1 = 0 \le n^*$, and this also satisfies the initial condition of $n_1 \in [0, n^*)$. This means that all k faults occur during v_2. Consequently, we have $wcft_2(k) = a_2 + g_2(0)$ and we denote it by $wcft_2^1(k)$.

ii) $n_1 \in [n^*, k]$

We have $f_1(n_1) = wcft_1(n_1) + c_{12} \ge wcft_1(n^*) + c_{12} = a_2$. Hence, $\max\{f_1(n_1), a_2\} = f_1(n_1)$, and $wcft_2(k) = f_1(n_1) + g_2(n_1)$. Similar to the case in Figure 7, the value of $wcft_2(k)$ is independent of n_1. Consequently, $wcft_2(k) = f_1(0) + g_2(0)$, and denote it by $wcft_2^2(k)$. Now that we have two local maxima, the greater one is chosen as the global maximum:

$$wcft_2(k) = \max\{wcft_2^1(k), wcft_2^2(k)\}$$
$$= \max\{a_2 + g_2(0), f_1(0) + g_2(0)\}$$

According to the pre-specified condition for Case 3, $wcft_1(0) + c_{12} < a_2 < wcft_1(k) + c_{12}$, which is equivalent to $f_1(0) < a_2 < f_1(k)$. We further simplify the above expression as:

$$wcft_2(k) = \max\{a_2 + g_2(0), f_1(0) + g_2(0)\} = a_2 + g_2(0).$$

This expression is the same as the one in Case 1, hence we can merge Case 1 with Case 3 using a single expression.

Procedure *Chkp* (G, Δ, k)
1. Perform topological-sort until G is traversed {
2. for each $v_j \in pred(v_i)$, **Do** {
3. calculate $wcft_i(k, v_j)$;
4. **if** ($wcft_i(k, v_j) > d_i$)
5. **exit**("Cannot tolerate k faults");
6. }
7. }
8. **return**("Feasible under k faults")

Figure 10: Procedure for feasibility analysis of a DAG.

Procedure *Eng_Chkp*
1. Calculate fault-free timing parameters (a_i, t_i) under f_l for each $v_j \in V$;
2. Find the appropriate checkpointing interval Δ using binary search;
3. Perform voltage scaling according to the criterion C.

Figure 11: Procedure for energy-aware checkpointing.

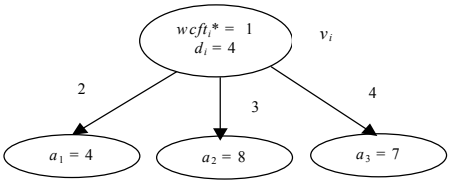


Figure 12: Illustrative example for voltage scaling.

To summarize, the worst-case scenario for v_2 depends on the relationship between v_1's completion time and v_2's arrival time. We combine all these cases as follows:

$$wcft_2(k) = \begin{cases} f_1(0) + g_2(0), & \text{if } wcft_1(0) + c_{12} \geq a_2 \\ a_2 + g_2(0), & \text{otherwise.} \end{cases}$$

3) For job v_3: the worst-case finish time in the presence of k faults can be expressed as

$$wcft_3(k) = \max\{wcft_2(n_2) + c_{23}, a_3\} + t_3 + (k - n_2)\sigma + t_3 c_w / \Delta.$$

Let $g_3(n_2) = t_3 + (k - n_2)\sigma + t_3 c_w / \Delta$ and $f_2(n_2) = wcft_2(n_2) + c_{23}$. Then $wcft_3(k) = \max\{f_2(n_2), a_3\} + g_3(n_2)$. We again divide the k faults into two parts: n_2 ($n_2 \leq k$) faults that occur before the execution of v_3 and ($k - n_2$) faults that occur during the execution of v_3. By employing the same method used for job v_2, we obtain:

$$wcft_3(k) = \begin{cases} f_2(0) + g_3(0), & \text{if } wcft_2(0) + c_{23} \geq a_3 \\ a_3 + g_3(0), & \text{otherwise.} \end{cases}$$

4. FEASIBILITY TEST

In this section, we present an algorithm to analyze the feasibility of a real-time program running on a message-passing distributed system. Our goal is to determine whether the program can be completed with a checkpointing interval of Δ without violating its timing constraints in the presence of up to k faults during execution.

We first state results that follow directly from the analysis in Section 3.2. The proofs are omitted due to lack of space.

Lemma 1: For the source job (a job without any predecessors) denoted by v_1, the corresponding worst-case finish time in the presence of k faults can be expressed as

$$wcft_1(k) = a_1 + t_1 + k\sigma + t_1 c_w / \Delta.$$

Lemma 2: For a job v_i, let the set of its predecessor jobs be $pred(v_i)$. Let $g_i(n_j) = t_i + (k - n_j)\sigma + t_i c_w / \Delta$ and $f_j(n_j) = wcft_j(n_j) + c_{ji}$. For each $v_j \in pred(v_i)$, there is a corresponding $wcft_{ij}(k, v_j)$ for v_i, which denotes the worst-case finish time determined by v_j in the presence of k faults. The parameter $wcft_{ij}(k, v_j)$ can be expressed as

$$wcft_{ij}(k, v_j) = \begin{cases} f_j(0) + g_i(0), & \text{if } wcft_j(0) + c_{ji} \geq a_i \\ a_i + g_i(0), & \text{otherwise.} \end{cases}$$

Theorem 1: Job v_i can be completed in the presence of k faults if and only if $\max_j \{wcft_{ij}(k, v_j) \mid v_j \in pred(v_i)\} \leq d_i$.

We next define the worst-case finish time implied by the worst-case finish times of all predecessors:

$$wcft_i^* = \max_j \{wcft_{ij}(k, v_j) \mid v_j \in pred(v_i)\}. \quad (3)$$

Theorem 2: Let $G = (V, E)$ be the DAG corresponding to a real-time program. This program is feasible in the presence of k faults if and only if $\forall v_i \in V$, $wcft_i^* \leq d_i$.

Based on the above theorems, we now describe the algorithm for analyzing the feasibility of a DAG in the presence of k faults under synchronized checkpointing. The pseudocode for the procedure is described in Figure 10. The complexity for our algorithm is O(|V|+|E|).

5. INCORPORATING DVS

In this section, we extend the results of Section 3 by considering DVS-capable processors. 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 $i < j$. Our goal is to find an appropriate checkpointing interval Δ and appropriate speed assignment for each job to save energy. To simplify the problem, we assume that the checkpointing interval Δ can be chosen from [Itv_{min}, Itv_{max}]. Here Itv_{min} is constrained by the minimum clock period, and Itv_{max} is constrained by the limits imposed by the program deadline.

The procedure *Eng_chkp* for energy-aware fault tolerance is summarized in Figure 11. First, we assign the maximum speed f_l to all processors and calculate the fault-free timing parameters, including arrival time and execution time. Next, we employ a binary-search based technique to choose the appropriate checkpointing interval Δ for the system under the highest processor speed f_l. The successors of each job $v_i \in V$ are denoted by $succ(v_i)$. Based on the results obtained under the highest processor speed f_l, we compare each job's worst-case finish time (denoted by $wcft_i^{**}$) with its successor's arrival time, and perform voltage scaling according to criterion C as defined below:

Criterion C:

(1) If $wcft_i^{**} \geq \min_j \{a_j - c_{ij} \mid v_j \in succ(v_i)\}$, do not scale down the speed, i.e., processor clock frequency, and voltage of v_i;

(2) If $wcft_i^{**} < \min_j \{a_j - c_{ij} \mid v_j \in succ(v_i)\}$, scale down the speed and voltage of v_i to the lowest speed $s(i) \in \{f_1, f_2, \ldots, f_l\}$ such that v_i completes before its deadline d_i, or before the time $\min_j \{a_j - c_{ij} \mid v_j \in succ(v_i)\}$ whichever is sooner.

Figure 12 shows an illustrative example.

Table 1: Feasibility and checkpointing intervals.

Benchmark	No. of faults (k)	Fault-oblivious method: feasible?	Proposed method	
			Feasible?	Δ (ms)
Automotive/ industrial ($\mathcal{B}_1$)	1	No	Yes	903.5
	2	No	Yes	363.5
	3	No	Yes	273.5
	4	No	Yes	183.5
	5	No	Yes	93.5
Consumer ($\mathcal{B}_2$)	1	No	Yes	36.1
	2	No	Yes	17.6
Network ($\mathcal{B}_3$)	1	No	Yes	82.4
	2	No	Yes	39.3
Office automation ($\mathcal{B}_4$)	1	Yes	Yes	23.2
	2	No	Yes	11.2
	3	No	Yes	7.2
Telecom ($\mathcal{B}_5$)	1	No	Yes	13.0
	2	No	Yes	4.0

Table 2: Degree of fault tolerance and energy consumption for $\mathcal{B}_4$.

Scheme	Checkpointing?	DVS?	No. of faults tolerated	Energy (mJ)
$\mathcal{S}_1$	No	No	1	150.2
$\mathcal{S}_2$	Yes	No	3	152.3
$\mathcal{S}_3$	No	Yes	1	130.2
$\mathcal{S}_4$	Yes	Yes	3	132.4

Since $wcft_i^{**} = 1 < \min_j \{a_j - c_{ij} \mid v_j \in succ(v_i)\} = 2 < d_i$, we scale down the speed of v_i by choosing the lowest possible speed that makes v_i complete before time $t = 2$.

6. SIMULATION RESULTS

In this section, we first demonstrate how the proposed checkpointing scheme can provide fault tolerance in distributed real-time systems. Following this, we show how energy saving can be achieved by employing DVS in combination with checkpointing.

We use the E3S benchmark set [10] for our experiments. The benchmarks are based on the Embedded Microprocessor Benchmark Consortium (EEMBC) and include tasks in the application domains of automotive systems, telecommunications, and consumer electronics [11].

Based on previous work [12], we assume that the time to read or write a checkpoint of size 5 KB is 0.4 ms. The results for the E3S-based benchmarks under constant processor speed are shown in Table 1. Procedure *Chkp* is able to find an appropriate value of Δ in each case. When Δ is set as shown in the table, the proposed scheme with synchronized checkpointing guarantees that all hard deadlines will be met in the presence of up to k faults. We have also compared our method with the fault-oblivious method without checkpointing. Simulations show that our proposed method outperforms the fault-oblivious method. For instance, our method can guarantee the timely completion of the automotive/industrial benchmark ($\mathcal{B}_1$) when up to 5 faults occur while the fault-oblivious method cannot complete on time when any faults occur during execution.

We next show how the checkpointing scheme combined with DVS can achieve energy saving while guaranteeing real-time responsiveness in the presence of faults. We consider 4 schemes in our simulation: (1) without checkpointing and DVS ($\mathcal{S}_1$), (2) with checkpointing but without DVS ($\mathcal{S}_2$), (3) without checkpointing and with DVS ($\mathcal{S}_3$), and (4) with checkpointing and DVS ($\mathcal{S}_4$). The degree of fault tolerance and energy consumption for the office-automation benchmark ($\mathcal{B}_4$) is shown in Table 2. We consider the AMD K6 processor in our simulation. First, we note that the proposed checkpointing schemes improve the degree of fault tolerance. As seen from Table 2, $\mathcal{S}_2$ and $\mathcal{S}_4$ can tolerate more faults than $\mathcal{S}_1$ and $\mathcal{S}_3$. Second, as expected, DVS saves energy in a distributed real-time embedded system. $\mathcal{S}_3$ ($\mathcal{S}_4$) achieves a 13.3% reduction in energy consumption compared to $\mathcal{S}_1$ ($\mathcal{S}_2$). Finally, the energy cost of incorporating checkpointing is negligible compared to the increase in fault tolerance, as seen from the comparison between $\mathcal{S}_1$ and $\mathcal{S}_2$, as well as $\mathcal{S}_3$ and $\mathcal{S}_4$.

7. CONCLUSIONS

We have shown how deterministic fault tolerance can be achieved in conjunction with dynamic power management in distributed real-time embedded systems. Deterministic fault tolerance is achieved via synchronized checkpointing. Power management is carried out using DVS. We have presented feasibility analysis for checkpointing schemes under constant processor speed. The proposed checkpointing scheme has then been combined with DVS to reduce energy consumption without violating deadline constraints in the presence of transient faults.

8. REFERENCES

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

[2] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems", *Proc. DATE*, pp. 918-923, 2003.

[3] L. Li et al, "Adaptive error protection for energy efficiency", *Proc. ICCAD*, pp. 2-7, 2003.

[4] D. Marculescu et al., "Fault-tolerant techniques for ambient intelligent distributed systems", *Proc. ICCAD*, pp. 348-355, 2003.

[5] J. Liu et al., "Communication speed selection for embedded systems with networked voltage-scalable processors", *Proc. CODES*, pp. 169-174, 2002.

[6] E. N. Elnozahy et al., "A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys,* vol. 34, pp. 375-408, September 2002.

[7] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Computer Systems*, pp. 63-75, February 1985.

[8] R. P. Dick and N. K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Proc. ICCAD*, pp. 62-68, 1998.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, NY, 1979.

[10] Embedded System Synthesis Benchmarks Suite (E3S): http://www.ece.northwestern.edu/~dickrp/e3s/.

[11] Embedded Microprocessor Benchmark Consortium (EEMBC): http://www.eembc.org.

[12] C.-Y. Lin et al., "A checkpointing tool for Palm operating system", *Proc. DSN*, pp. 71-76, 2001.