A Unified Approach for Fault Tolerance and Dynamic Power Management in Fixed-Priority Real-Time Embedded Systems

Ying Zhang and Krishnendu Chakrabarty, *Senior Member, IEEE*

***Abstract*—This paper investigates an integrated approach for achieving fault tolerance and energy savings in real-time embedded systems. Fault tolerance is achieved via checkpointing, and energy is saved using dynamic voltage scaling (DVS). The authors present a feasibility analysis for checkpointing schemes for a constant processor speed as well as for variable processor speeds. DVS is then carried out on the basis of the feasibility analysis. The authors incorporate important practical issues such as faults during checkpointing, rollback recovery time, memory access time, and energy needed for checkpointing, as well as DVS and context switching overhead. Numerical results based on real-life checkpointing data and processor data sheets show that compared to fault-oblivious methods, the proposed approach significantly reduces power consumption and guarantees timely task completion in the presence of faults.**

***Index Terms*—Checkpointing, dynamic voltage scaling (DVS), fault tolerance, real-time scheduling.**

I. Introduction

MANY embedded systems in use today rely on dynamic power management (DPM) techniques, where the operating system (OS) is responsible for managing system-level power consumption. There are two main types of DPM techniques. The first includes selective shut-off or slow down of system components that are idle or underutilized [1]. The second, termed as dynamic voltage scaling (DVS), refers to the dynamic control of the supply voltage level for the various components in a system [2]. DVS has emerged as a popular solution to the problem of reducing power consumption during system operation [3]–[5]. Many embedded processors such as the Intel XScale PXA260 [6], the Motorola 6805 [7], the Transmeta Crusoe [8], and the AMD K-6 [9] are now equipped with the ability to dynamically scale the processor frequency by adjusting the operating voltage.

A large number of embedded systems are also designed for real-time use [10], where a missed deadline can result in catastrophic consequences. When DVS is employed to achieve energy saving for real-time systems, a reduction in voltage results in a corresponding drop in the processor speed, hence the capability of the system to meet task deadlines might be undermined [11]. A number of techniques have been proposed recently to balance real-time responsiveness with low-energy task execution [12]–[15].

Embedded systems are often deployed in harsh operational environments. Many of these systems tend to be situated at remote and inaccessible locations; hence, repair and maintenance are often difficult and sometimes even impossible. This necessitates the use of fault-tolerant techniques.

Fault-tolerant computing refers to the correct execution of user programs and system software in the presence of faults [16]. It is typically achieved through task reexecution or component redundancy. In real-time embedded systems, it is necessary to ensure that task reexecution does not jeopardize the timely completion of tasks. Fault tolerance is typically achieved in real-time systems through online fault detection [17], checkpointing, and rollback recovery [18], [19]. Fig. 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 the 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 prevents the need for task restarts and increases the likelihood of a task completing on time with the correct result. Frequent checkpointing reduces reexecution time due to faults but increases task execution time. On the other hand, infrequent checkpointing has less impact on task execution in the absence of faults but increases the amount of reexecution that must be performed after a fault is detected. Therefore, the checkpointing interval, i.e., the 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.

There are three main reasons for combining DPM with fault tolerance in real-time embedded systems. Increased die temperatures due to higher processor speeds create thermal stresses on the die and result in more transient faults during system operation. In order to mitigate reliability problems caused by high die temperatures, we can either lower energy consumption through DPM techniques such as DVS or adopt fault tolerance

Manuscript received August 26, 2003; revised March 8, 2004. This work was supported by the Defense Advance Research Projects Agency (DARPA) and administered by the Army Research Office under Emergent Surveillance Plexus MURI Award DAAD19-01-1-0504. Parts of this paper were presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, 2003, and at the IEEE/ACM Design, Automation and Test in Europe (DATE) Conference, Paris, France, 2004. This paper was recommended by Associate Editor R. Gupta.

Y. Zhang is with Guidant Corporation, St. Paul, MN 55112 USA.

K. Chakrabarty is with Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: krish@ee.duke.edu).

Digital Object Identifier 10.1109/TCAD.2005.852657

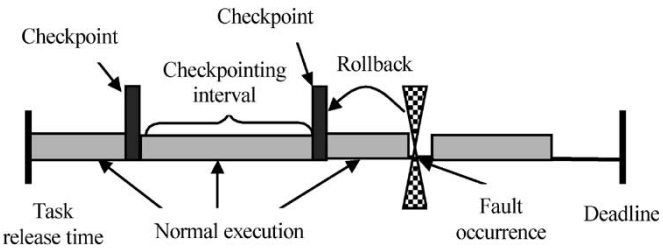

Fig. 1. Illustration of checkpointing and rollback recovery.

techniques such as checkpointing. Better still, a combination of DVS and checkpointing can be used to reduce energy consumption and improve the run time reliability of the system.

The second reason for combining DVS with adaptive checkpointing is motivated by the need to meet task deadlines in real-time systems. Checkpointing provides an effective method to reduce reexecution time in the presence of faults. DVS can also be used to enhance fault tolerance in a real-time system. If faults occur frequently, the processor speed can be scaled up dynamically (within limits imposed by higher die temperatures) and more slack can be provided to the task, which allows more time for rollback recovery. Hence, a combination of checkpointing and DVS can be used to increase the likelihood of timely task completion in the presence of faults and trade off energy with fault tolerance.

The third motivation arises from shrinking process technologies in the nanotechnology realm. Lower processor voltages are likely to lead to lower noise margins and more transient faults caused in part by single-event upsets [26]. Hence, a DPM framework, in which DVS techniques are tied to system-level fault tolerance, are of particular interest for embedded systems.

DPM and fault tolerance for embedded real-time systems have largely been studied as separate problems in the literature. DVS techniques for power management do not consider fault tolerance [3]–[5], [12]–[15], and checkpoint placement strategies for fault tolerance do not address DPM [20]–[22]. It is only recently that an attempt has been made to combine fault tolerance with DPM [23], [24].

In [23], adaptive checkpointing is combined with DVS for soft real-time systems. In [24], the analysis is based on the earliest-deadline-first (EDF) scheduling strategy; however, a number of simplifying assumptions are made, e.g., a task is subject to at most one fault occurrence before its deadline and always resumes execution after rollback with the maximum processor speed. In addition, it is assumed in [24] that the processor is capable of adjusting its speed continuously in the range $[S_{\min}, S_{\max}]$, the checkpointing cost is dependent on processor speed, and the state restoration cost is zero. Finally, faults during checkpointing and state restoration are not considered in [24], task priorities are assumed to be dynamic, and simulation results are presented only for simple task sets and hypothetical processor models.

In contrast to [24], this paper focuses on fixed-priority real-time systems and relaxes the restriction that no faults occur during checkpointing and state restoration. Although the EDF policy is used in some real-time systems, fixed-priority assignments are adopted in many real-time scheduling algorithms of practical interest due to their low overhead and predictability [25]. The number of fault occurrences in this paper is not limited to one, and voltage scaling and state restoration costs are incorporated in the analysis. Simulation results are presented for benchmark task sets and commercial embedded processors with a discrete set of voltage/speed settings. Typical parameters used in the simulation experiments are based on realistic scenarios.

The integrated approach presented here provides fault tolerance and DPM in hard real-time embedded systems. Feasibility tests for fixed-priority real-time systems with checkpointing under constant processor speed are presented first. We consider job-oriented feasibility tests, in which the goal is to tolerate k fault occurrences for each job, where an appropriate value of k is determined based on the task execution time and typical fault arrival rates. Hyperperiod-oriented feasibility tests are then presented, in which the goal is to tolerate up to k fault occurrences in a hyperperiod. The value of k in this case depends on the length of the hyperperiod and typical fault arrival rates. Following this, we extend these feasibility tests to variable-speed processors. The above feasibility analyses provide the criteria under which checkpointing can provide fault tolerance and real-time guarantees. Based on the feasibility analyses results, an on-line dynamic speed-scaling scheme is developed to reduce energy during task execution by exploiting the available slack. The proposed approach is compared with a recent fault-oblivious DVS scheme, referred to as voltage scaling for low power (VSLP) [5], in the presence of fault occurrences.

It was assumed throughout that faults are intermittent or transient in nature and that permanent faults are handled through manufacturing testing or field testing techniques [27]. Typical examples of transient faults include errors caused by cosmic rays and high-energy particles in nanotechnology with shrinking processes [26]. We assume that these faults affect the processor either during normal operation or during memory writes for checkpointing. In the latter case, the fault makes the checkpoint invalid.

The rest of the paper is organized as follows. Section II discusses practical issues related to checkpointing and DVS in real-time embedded systems. These include the size of checkpoints, the type of stable storage for the checkpoints, memory access time and power models, and the time and power needed for voltage scaling. Section III provides off-line feasibility analysis for checkpointing under constant processor speed in real-time systems. Section IV extends these results to a system with a variable-speed processor. Experimental results based on representative parameter values from Section II are presented in Section V, and Section VI presents conclusions.

II. Practical Issues in Checkpointing and DVS

This section reviews some practical issues for checkpointing and DVS in real-time embedded systems. The goal here is not to present a comprehensive survey of checkpointing and DVS, for which the reader is referred to [1], [11], and [28]. Rather, the objective here is to identify practical issues and parameter values that must be considered for analysis and for meaningful simulation experiments. While prior work on DVS has sometimes been based on realistic processor models, the cost of

voltage scaling (time and power) has largely been ignored. This paper addresses this issue for several real-time benchmark task sets and commercial embedded processors. These practical considerations provide a realistic basis for the work on system modeling, analysis, and experiments.

A. Stable Storage

Checkpoints need to be saved to stable storage. Stable storage must ensure that the recovery data persist through the tolerated faults [29]. Embedded systems have limited memory, and most of them do not contain a hard disk acting as a nonvolatile storage. In addition to static random access memory (SRAM) and dynamic RAM (DRAM), read-only memory (ROM) and flash memory are also used as a nonvolatile storage for embedded systems. Since an ROM is a read-only device, it cannot be used for saving checkpoints. Flash memory offers the highest capacity, followed by DRAM and SRAM. However, flash memory also suffers from the highest access time, followed by DRAM and SRAM [30].

SRAM in embedded systems is normally used for frequently accessed and time-critical storage such as caches and register files. Its typical capacity is in the order of kilobytes. In [31], it is shown that even a small game program such as Raptoids on a Palm handheld device can have a checkpoint size of 2.897 kB. Due to its limited capacity, SRAM is of limited use for checkpointing in embedded systems.

DRAM is used as a main memory in embedded systems while flash memory is used for storing boot images and other nonvolatile data, both with a capacity of tens of megabytes. Flash memory reads at almost DRAM speeds, but writes 10 to 100 times slower [30]. For example, reading a 64-kB block takes 4.3 ms while writing a 64-kB block takes 178.3 ms for a 2-MB flash memory. The large access time, especially for write operations, limits the use of flash memory as a stable storage for short-duration real-time tasks. In summary, DRAM is more appropriate for storing checkpoints in real-time embedded systems.

B. Saving of a Checkpoint

The checkpoint of a process corresponding to a task includes a copy of the process's state (including the program counter and stack pointers), operating system state (mainly the state of the open file table), and the data state (the process's stack and data segments) [32]. There are two types of checkpoints in embedded systems. Full checkpointing refers to the writing of the entire address space to a stable storage during each checkpoint. In contrast, incremental checkpointing reduces data volume by writing only the pages of the address space that have been modified since the previous checkpoint. This set of pages is determined using the dirty bit maintained by the memory management hardware in each page table entry [29].

In the case of full checkpointing, only the most recent checkpoint data need to be retained for recovery; older ones may be deleted. For incremental checkpointing, old checkpoint files cannot be automatically deleted because the program's data state is spread out over many checkpoint files [32]. In real-time embedded systems, hardware resources are scarce; it is therefore undesirable to introduce extra hardware overhead to maintain the page table necessary for incremental checkpointing. Hence, full checkpointing is more viable despite the drawback that relatively longer time is needed to read/write the data for a single checkpoint.

The size of a checkpoint depends strongly on the task set. For example, computation-intensive applications such as matrix operations produce large checkpoints with sizes in the order of megabytes [32]. Due to this reason, it is hard to characterize the checkpoint size with a single numerical value. Many embedded systems are targeted for real-time control in response to sensor inputs; hence, it is expected that the data volume for such applications is not too large. Furthermore, resource constraints in embedded systems limit the data volume. The only source available to in the literature that describes checkpoint size for embedded systems is [31]. It provides checkpointing data for games on Palm handheld devices. The checkpoint size ranges from 0.497 to 2.897 kB. In the absence of additional literature, we use [31] as a basis and assume that the checkpoint size is of the order of a few kilobytes.

C. Fault Arrival Rate

As indicated in [33], the typical fault arrival rate that must be tolerated in safety-critical real-time systems is in the range of 10^{-10} to 10^{-5} /h. However, for systems that operate in harsh environments, the fault arrival rate can be much higher, in the range of 10^{-2} to 10^{2} /h [34]. For example, in an orbiting satellite, the number of errors caused by protons and cosmic ray ions was measured to be as high as 35 in a 15-min interval, which amounts to 140/h [35]. Clearly, any embedded system cannot ensure system safety under such harsh environments without employing fault-tolerant techniques; the proposed technique is therefore targeted at such systems. Moreover, the fault arrival rates reported in [33] are for older process technologies. Higher transient fault arrival rates can be expected for newer technologies that operate at lower voltages and provide less noise margin.

It is important to know the typical fault arrival rate for a specific real-time task set since it provides the basis for the k-fault-tolerant requirement. Given a fault arrival rate λ and a task execution interval t, the mean number of faults that arrive during the interval is λt. Suppose the number of faults to be tolerated deterministically during this interval is k. If k is much smaller than λt, a sophisticated fault-tolerant scheme with its associated overhead is not appropriate. On the other hand, if k is much larger than λt, a fault-tolerant scheme that provides deterministic real-time guarantee may not exist. In order to target a system with reasonable real-time performance with fault tolerance, the value of k can be taken to be a small multiple of λt, e.g., $2\lambda t \leq k \leq 3\lambda t$.

Prior work on checkpointing is usually based on the assumptions that no faults occur during checkpointing and that the state restoration time is zero. These assumptions are unrealistic in practice, especially for high fault arrival rates and if the checkpointing time is not negligible due to a large checkpoint data size.

D. Cost of Voltage Scaling

The voltage scaling cost for DVS in real-time embedded systems has attracted relatively little attention in prior work on task scheduling and DPM. It is now recognized, however, that these costs cannot be ignored for real-time and power-constrained embedded systems [12].

Four typical real-time task sets are described in [12]: a generic avionics platform (GAP), an inertial navigation system (INS), a computerized numerical control (CNC), and a flight control system. The execution times for the tasks range from 35 to 720 μs for CNC, 1 to 9 ms for GAP, 1.18 to 100.28 ms for INS, and 10 to 60 ms for flight control. In [36], some task sets for real-time multimedia applications are presented. The typical execution times for those tasks range from 1.4 to 50.4 ms. We therefore see that the execution times of tasks in real-time applications range from microseconds to milliseconds. As discussed next, the execution times of these tasks must be viewed in the context of typical voltage scaling overheads.

The voltage scaling costs for embedded processors have been documented in a number of recent papers. The worst case voltage transition time for the ARM8 microprocessor core ranges from 10 [37] to 520 μs [38]. The time taken by a Power personal computer (PC)-embedded processor to switch between 0.9 and 1.95 V is 105 μs [39]. The StrongArm 1100 processor takes 250 μs to switch from a 1.5-V supply voltage to a 1.23-V supply voltage [40].

We note that the voltage scaling time is in the order of hundreds of microseconds for typical embedded processors. Hence, for short-duration real-time tasks such as CNC, where the execution time is also in the order of tens of microseconds, it might be counterproductive to employ DVS. The relative DVS time penalty is so high for CNC that the energy savings due to DVS are counterbalanced by the adverse impact on real-time responsiveness due to voltage scaling times. Most papers in the DVS literature that use CNC as a benchmark do not consider this issue. Even for longer-duration real-time tasks (with an execution time greater than 1 ms), the consideration of DVS overheads leads to more accurate conclusions. In this paper, we incorporate the times and energy overheads due to voltage scaling in the general theoretical framework. We also include these considerations in the simulations based on real-time benchmark task sets and commercial embedded processors.

III. Feasibility Analysis Under Constant Speed

We are given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$. The elements of the tuple are defined as follows: T_i is the period of τ_i and D_i is its deadline ($D_i \leq T_i$), and E_i is the execution time of τ_i under fault-free conditions. Let the time required to store a checkpoint be C_s and the time required to retrieve a checkpoint be C_r. We make the following assumptions related to task execution and fault arrivals: 1) the task set Γ is scheduled using fixed-priority methods such as the rate-monotonic scheme or the deadline-monotonic scheme [25]; 2) the task set Γ is schedulable under fault-free conditions; 3) the priority of tasks is in decreasing order of the index i, i.e., task τ_i has higher priority than task τ_j if $i < j$; 4) each instance of the task is released at the beginning of the period; 5) the checkpointing intervals for the same task are equal; and 6) faults are detected as soon as they occur.

In [41], a feasibility analysis is provided under the assumption that two successive faults arrive with a minimum interarrival 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 interarrival 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 is infinite if we consider an unlimited number of periods. We therefore need to identify an appropriate k-fault-tolerant condition for a shorter time duration. Here, we provide two solutions corresponding to two different fault-tolerance requirements. One is to tolerate k faults for each job, termed as job-oriented fault tolerance; the other is to tolerate k faults within a hyperperiod (defined as the least common multiple of all the task periods [25]), termed as hyperperiod-oriented fault tolerance. In practical situations, the choice of an appropriate fault tolerance criterion can be made based on the needs of the real-time application and a realistic fault arrival rate.

The rest of this section is organized as follows. Section III-A considers k fault tolerance for a single job. Section III-B reviews feasibility analysis based on time-demand analysis for a task set under fault-free conditions [25]. This analysis is extended in Section III-C to incorporate fault arrivals under the job-oriented fault model. Finally, Section III-D analyzes a hyperperiod-oriented fault tolerance for a task set.

A. Feasibility Analysis for a Single Job

We first consider the case of a single job. Suppose the checkpointing interval is $\Delta = E/(m+1)$, where m is the number of checkpoints inserted equidistantly during the computation time to tolerate k faults in one job. The objective here is to find the optimal checkpointing interval to minimize the worst case response time in case of faults.

The total execution time of the job can be divided into three categories: effective computation (the time when the job performs real computation), checkpoint saving, and checkpoint retrieval. Based on this classification, we can further divide the occurrences of the k faults during task execution. Suppose k_1 faults occur during checkpointing saving, k_2 faults occur during checkpoint retrieval, and k_3 faults occur during effective computation, where $k = k_1 + k_2 + k_3$; see Fig. 2. Whenever a fault occurs during job execution or checkpoint saving, the system rolls back to the most recent checkpoint and restores the system state. As a result, the maximum time penalty due to a fault during job execution is $\Delta + C_r$, as indicated in Fig. 2(c). Similarly, the maximum time penalty due to a fault during checkpoint saving is $\Delta + C_s + C_r$, as indicated in Fig. 2(a). If a fault occurs during state restoration, the system will roll

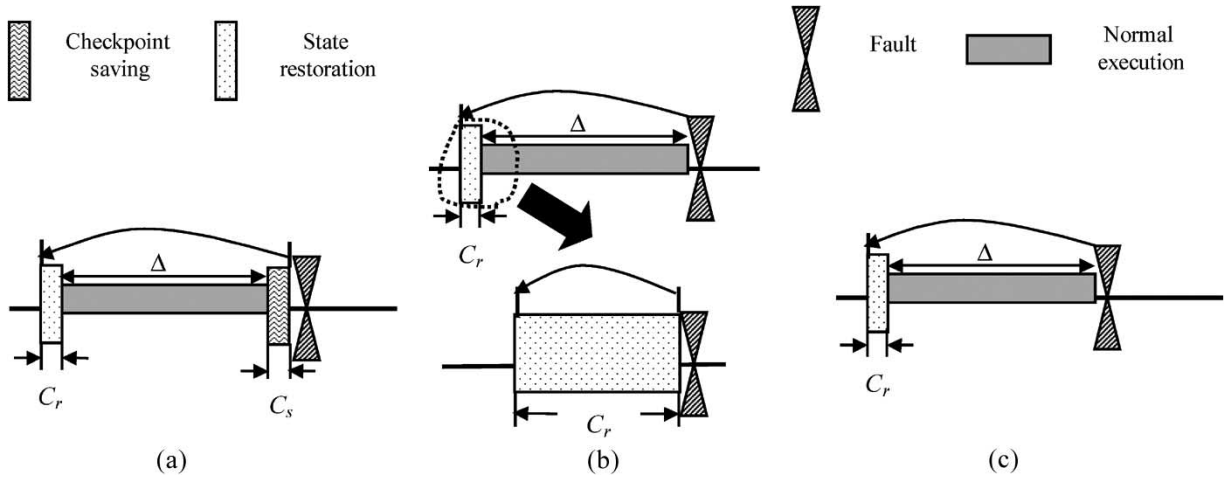

Fig. 2. Illustration of fault occurrence. (a) Fault during checkpoint saving. (b) Fault during state restoration. (c) Fault during job execution.

back to the checkpoint and attempt to restore state until it is successful, as demonstrated in Fig. 2(b) (the state restoration time is enlarged in order to show the effect of fault occurrence). Hence, the maximum time penalty due to a fault during checkpoint retrieval is C_r.

The response time R for the job is composed of five terms: 1) the fault-free job execution time E; 2) the total time for saving m checkpoints mC_s; 3) the additional penalty due to k_1 faults during checkpoint saving $k_1(\Delta + C_s + C_r)$; 4) the additional penalty due to k_2 faults during state restoration k_2C_r; and 5) the additional penalty due to k_3 faults during job execution $k_3(\Delta + C_r)$. Hence, the response time is expressed as $R = E + (m + k_1)C_s + (k_1 + k_3)\Delta + kC_r$. It can be seen that the worst case response time is obtained when $k_1 = k$ and $k_2 = k_3 = 0$. This means that all k faults occur at the end of checkpoint saving. Replacing Δ with $E/(m + 1)$, the worst case response time $R_{\text{worst case}}$ is further expressed as $R_{\text{worst case}}(m) = E + k(C_s + C_r) + mC_s + kE/(m + 1)$. In this expression, E, k, C_s, and C_r are constants, and m is a variable. We are aiming to find the optimal value of m such that $R_{\text{worst case}}$ is minimized. To satisfy the deadline constraint, they must have $R_{\text{worst case}}(m) \leq D$.

The minimum value of $R_{\text{worst case}}(m)$ is obtained for $m = \sqrt{kE/C_s} - 1$. Let $\|\sqrt{kE/C_s} - 1\|$ denote the value of m from the pair $\{\lceil\sqrt{kE/C_s} - 1\rceil, \lfloor\sqrt{kE/C_s} - 1\rfloor\}$ that minimizes $R_{\text{worst case}}$. Furthermore, since m is a nonnegative integer, we have $m^* = \max\{\|\sqrt{kE/C_s} - 1\|, 0\}$. Let $f(m^*) = R_{\text{worst case}}(m^*) - D$.

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

Example 1: For a hypothetical real-time job with parameters $C_s = 10$, $C_r = 10$, $k = 1$, $E = 9000$, and $D = 10\,000$, we get $m^* = 29$ and $f(m^*) = -390$. This implies that there exists an equidistant checkpointing scheme to tolerate a single fault for this job, and the worst case response time is minimized when 29 checkpoints are inserted. Now we change k from one to three, i.e., the system is required to tolerate up to three faults. Then, we get $m^* = 51$ and $f(m^*) = 89.23$. Since $f(m^*) > 0$, no equidistant checkpointing scheme exists to tolerate up to three faults for this job.

B. Feasibility Analysis for a Task Set Under Fault-Free Condition

Here, we examine the feasibility of a task set under fault-free conditions. The feasibility analysis is based on time-demand analysis for fixed-priority scheduling [25]. The steps in the analysis are as follows.

1) Compute the response time R_i for τ_i according to the equation: $R_i = E_i + \sum_{h=1}^{i-1}\lceil R_i/T_h\rceil E_h$. Here, T_h and E_h are the period and the execution time of a task τ_h with higher priority than τ_i. This equation can be solved by forming the following (delete) 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 j or when $R_i^{(j+1)} > D_i$, whichever occurs sooner. In the former case, τ_i is schedulable; in the later case, τ_i is not schedulable.

According to [25], the time complexity of the time-demand analysis for each task is $O(nR)$, where R is the ratio of the largest period to the smallest period.

C. Job-Oriented Fault Tolerance: Tolerating k Faults in Each Job

In order to tolerate k faults in each task instance (job), it is required that all tasks can meet their deadlines under the condition that at most k faults occur during the execution of each job. This case needs to be considered for task sets with longer duration tasks and for systems with higher fault arrival rates such that one or more faults can be expected during each job execution.

Under the worst case condition, the additional time due to checkpointing and recovery should be incorporated. When

there are m_j equidistant checkpoints for each instance of τ_j, we have

$$R_i = \left[E_i + k(C_s + C_r) + m_i C_s + \frac{kE_i}{m_i + 1}\right] + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \left[E_h + k(C_s + C_r) + m_h C_s + \frac{kE_h}{m_h + 1}\right].$$

Let $\psi_i(m_i) = E_i + k(C_s + C_r) + m_i C_s + kE_i/(m_i + 1)$. Then, $R_i = \psi_i(m_i) + \sum_{h=1}^{i-1} \lceil R_i/T_h \rceil \psi_h(m_h)$. To minimize all response times $R_i (1 \le i \le n)$, $\psi_i(m_i) (1 \le i \le n)$ must be minimized. We have $m_i^* = \max\{\|\sqrt{kE_i/C_s} - 1\|, 0\}$, $(1 \le i \le n)$. Then, the recurrence equation can be employed as

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

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

Example 2: Consider a hypothetical task set composed of two tasks, $\tau_1 = (60, 25, 7)$ and $\tau_2 = (80, 47, 8)$, and let $k = 3$, $C_s = C_r = 1$. Then, $m_1^* = 4$ and $m_2^* = 4$. After applying the recurrence equation, we get the response times $R_1 = 21.2 < 25$ and $R_2 = 44.0 < 47$. Thus, checkpointing is feasible for this task set if up to three faults occur during each job. Next, the case of $k = 4$ is examined. For this case, $m_1^* = 4$ and $m_2^* = 5$. The response times are $R_1 = 24.6 < 25$ and $R_2 = 50.9 > 47$. As a result, checkpointing is not feasible if up to four faults need to be tolerated for each job.

D. Hyperperiod-Oriented Fault Tolerance: Tolerating k Faults in a Hyperperiod

An algorithm is presented in [41] to determine the checkpointing interval under the following assumptions.

1) Two successive faults arrive with a minimum interarrival time T_F.
2) The time cost to retrieve a checkpoint is assumed to be zero, i.e., $C_r = 0$.
3) All checkpoints are assumed to be fault-free, i.e., no faults can occur during checkpoint saving and retrieval.

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 its execution. Here, F_j is also the checkpointing interval for τ_j. When there are m_j equidistant checkpoints for τ_j, the response time R_i for τ_i is expressed in [41] as

$$R_i = (E_i + m_i C_s) + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \times (E_h + m_h C_s) + \left\lceil \frac{R_i}{T_F} \right\rceil \max_{1 \le j \le i}\{F_j\}$$

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

The checkpoint is examined starting from high-priority tasks to low-priority tasks. For each task τ_j, the algorithm tries to reduce the response time by reducing the maximum additional computation time, i.e., $\max_{1 \le j \le i}\{F_j\}$. The details of the steps in [41] are as follows.

1) Initially, $m_i = 0$ for $1 \le i \le n$.
2) Starting from the highest priority task τ_1, calculate the minimum number of checkpoints m_1 required to make it schedulable.
3) In decreasing order of task priorities, calculate the response time R_i of task τ_i. If $R_i \le D_i$, move to the next task; otherwise, R_i needs to be reduced further. The only way to reduce R_i is to add more checkpoints to decrease the reexecution time caused by faults, i.e., F_j, for $1 \le j \le i$. In fact, the parameter $\max_{1 \le j \le i}\{F_j\}$ is relevant here and should be reduced. 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 $R_i \le D_i$ or the updated value of the response time is greater than the previous iteration, i.e., $R_i > R_{i-1}$.

This algorithm is based on the restrictive assumption that two successive faults arrive with a minimum interarrival time T_F. In addition, while the schedulability test in [41] provides useful guidelines on task schedulability in the presence of faults, a drawback of this work is that two key issues that affect schedulability are not addressed.

1) Checkpoints are added to 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 [41].
2) It is necessary to determine a bound on the number of checkpoints beyond which the addition of checkpoints does not improve schedulability. In other words, we need a criterion that can declare a task set infeasible with a given number of checkpoints even though an arbitrary number of additional checkpoints can be added. In [41], 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 hypothetical tasks $\tau_1 = (100, 18, 7.999)$ and $\tau_2 = (101, 21, 8)$, and let $T_F = 102$, $C_s = 0.1$, and $C_r = 0$. We follow the steps from [41] as shown below.

1) Initially, $m_1 = m_2 = 0$, and $F_1 = 7.999$, $F_2 = 8$;
2) Next, τ_1 is examined: $R_1 = 15.998 < 18$. No checkpoints are needed for τ_1. Thus, $m_1 = m_2 = 0$;
3) Next, τ_2 is examined: $R_2 = 23.999 > 21$. Since $F_2 > F_1$, one checkpoint is added to τ_2, thus $m_1 = 0$ and $m_2 = 1$. Then, $F_1 = 7.999$, $F_2 = 4$, and $\max_{1 \le j \le 2}\{F_j\} = 7.999$. We recalculate the response time $R_2 = 24.098 > 23.999$. According to [41], 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, $m_1 = 1$ and $m_2 = 1$. Then, $F_1 = 7.999/(1 + 1) = 3.9995$, $F_2 = 4$, and $\max_{1 \le j \le 2}\{F_j\} = 4$. We recalculate the response time of τ_1 and τ_2 : $R_1 = 12.0985 < 18$ and $R_2 = 20.199 < 21$, which implies that both tasks are schedulable.

It is required here that the tasks meet their deadlines under the condition that at most k faults occur during a hyperperiod. Furthermore, as indicated in Section II, in order to make the model more practical for real-time embedded systems, the two restrictive assumptions of zero state-restoration time and fault-free checkpointing are removed. Based on the schedulability test in [41], we incorporate the state-restoration time, take into account faults during checkpointing, and solve the two aforementioned problems as follows.

The response time R_i for τ_i is expressed as

$$R_i = (E_i + m_i C_s) + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil (E_h + m_h C_s) + k(C_s + C_r) + k \max_{1 \le j \le i}\{F_j\}$$

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

The problem of recalculating response times due to the addition of checkpoints to higher-priority tasks can be solved using a recursive method. Any time the number of checkpoints for a task is increased, 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 that task τ_{h1} contributes the most to the response time R_i and add one more checkpoint to τ_{h1}. After recalculating R_i, we might find that R_i has increased. In this situation, it cannot be simply claimed that 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) Bound Based on Checkpointing Tradeoffs: The effect of adding more checkpoints is twofold. First, it increases the execution time due to the checkpoint saving 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

$$\frac{kE}{(m+1)} - \frac{kE}{(m+2)} = \frac{kE}{[(m+1)(m+2)]}.$$

We combine the two impacts of checkpointing on the reexecution time to define the tradeoff function $\mathrm{tr}(m)$ as $\mathrm{tr}(m) = C_s - kE/[(m+1)(m+2)]$. If $\mathrm{tr}(m) < 0$, 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.

Procedure ADV-CP(Γ)
begin
1. **for** $i = 1$ to n **do**
   $m_i = 0$; compute m_i^*; $R_i = \infty$;
2. CP(1, n).
end

Fig. 3. Advanced checkpointing procedure.

Procedure CP (p, q)
1. **if** ($R_p \le D_p$ & $R_{p+1} \le D_{p+1}$ & ... & $R_q \le D_q$)
   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 $h \in [1, j]$ 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.

For each task τ_i with m_i checkpoints, the tradeoff function $\mathrm{tr}_i(m_i)$ is determined. Let m_i' be the number of checkpoints beyond which the addition of more checkpoints does not reduce the response time. To determine m_i', we need to solve the equation $\mathrm{tr}_i(m_i') = 0$. Solving this equation obtains $m_i' = (-3 + \sqrt{1 + 4kE_i/C_s})/2$ for $1 \le i \le n$. Since $m_i' \ge 0$, we further express it as $m_i' = \max\{\lfloor(-3 + \sqrt{1 + 4kE_i/C_s})/2\rfloor, 0\}$ for $1 \le i \le n$. This gives an upper bound on the number of checkpoints, which is based on the tradeoff function.

2) 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 checkpoint saving cost and timing constraints, we have $R_i^0 + m_i C_s \le D_i$, which implies that $m_i \le (D_i - R_i^0)/C_s$. Let $m_i^{\#} = \lfloor (D_i - R_i^0)/C_s \rfloor$.

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

A checkpointing algorithm ADV-CP for off-line feasibility analysis is described in Fig. 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 initially set unschedulable. Line 2 calls the recursive checkpointing subroutine CP to add checkpoints from τ_1 to τ_n.

The recursive checkpointing procedure $\mathrm{CP}(p, q)$ is described in Fig. 4, where p and q are the lowest and highest indexes 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 $O(n^2R)$ in the worst case. (Recall that R is the ratio of the largest period to the smallest period.) Each time a checkpoint is added, the response time for lower-priority tasks needs to be recalculated. Hence, the recursive execution of $\mathrm{CP}(p,q)$ takes $O(n^2R)\sum_{i=1}^{n} m_i^*$. Let $M^* = \sum_{i=1}^{n} m_i^*$. Here, M^* is a constant dependent on the timing parameters of the task set. Adding all the costs together, the overall complexity of $O(n^2RM^*)$ is obtained. Furthermore, we note that the complexity can be reduced if they can make M^* as small as possible. That is why they combine both the tradeoff function and the timing constraints to obtain a relatively tight bound for m_i^*.

IV. Feasibility Analysis With DVS

Here, 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 $i < j$. We are also given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$. The elements of the tuple are defined as follows: 1) T_i is the period of τ_i and D_i is its deadline ($D_i \le T_i$); 2) E_i is the number of computation cycles of τ_i under fault-free conditions.

We note here that although the processor is equipped with variable speeds, CPU scaling does not affect the cost of checkpoint saving and state restoration. The reason for this is that checkpointing is essentially a memory-access operation; checkpointing costs and state restoration costs are determined by the read and write costs, and these are independent of the processor speed. As in Section III, we use C_s and C_r to denote the time needed for checkpoint saving and data retrieval, respectively. Furthermore, we assume that the energy cost for saving one checkpoint is ξ_{cs} and the energy cost for retrieving one set of checkpoint data is ξ_{cr}. We incorporate the DVS penalty due to the time and energy needed for speed switch. We assume that a single speed switch incurs a time cost of t_{ss} and energy consumption of ξ_{ss}.

Given clock frequency f for a commercial embedded processor, its corresponding power consumption $P(f)$ can be found in the data sheets. For a task with N single-cycle instructions, the energy consumption can be expressed as $\mathcal{E}(N,f) = P(f)^*N/f$.

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

The primary objective here is to meet task deadlines deterministically, even though k faults occur, either during the execution of a job in the job-oriented model or during a hyperperiod in the hyperperiod-oriented model. A secondary goal is to minimize energy consumption. First, we need to identify the appropriate time duration to evaluate the energy consumption. They consider the hyperperiod as the time duration. Second, the criterion of minimizing energy consumption needs to be clarified. Based on the application requirement, we can choose either a best case or a worst case energy consumption value. By best case, we refer to the results obtained under the fault-free condition, while worst case refers to the results obtained when all k faults occur. This paper focuses on minimizing energy consumption under the worst case condition during a hyperperiod. Let the hyperperiod be denoted by Ht and the number of checkpoints for τ_i be denoted by m_i.

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

A. Job-Oriented Fault Tolerance With DVS

We first neglect the voltage switching cost in the analysis. The objective here is to highlight the impact of checkpointing on fault tolerance and of DVS on energy saving. Following this, we incorporate the switching cost and provide the formulation based on the most realistic scenario.

Without the voltage switching cost, the worst case response time for task τ_i can be expressed as

$$R_i = \left\lceil \frac{E_i + \frac{kE_i}{m_i+1}}{s(\tau_i)} + k(C_s + C_r) + m_i C_s \right\rceil + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \left\lceil \frac{E_h + \frac{kE_h}{m_h+1}}{s(\tau_h)} + k(C_s + C_r) + m_h C_s \right\rceil. \quad (2)$$

The total energy consumption Π during one hyperperiod can be expressed as

$$\Pi = \sum_{i=1}^{n} \frac{Ht}{T_i} \left[\mathcal{E}\left(E_i + \frac{kE_i}{m_i+1}, s(\tau_i) \right) + k(\xi_{\mathrm{cs}} + \xi_{\mathrm{cr}}) + m_i \xi_{\mathrm{cs}} \right]. \quad (3)$$

To minimize all response times, we must have $m_i^* = \max(\|\sqrt{kE_i/(s(\tau_i)C_s)} - 1\|, 0), 1 \le i \le n$. As a feasibility test, the recurrence equation can be employed as

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

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

Since the optimal number of checkpoints depends on the speed assignment, we first need to choose the appropriate processor speeds. After that, they can calculate the optimal number

of checkpoints, insert these values in (2), and carry out the feasibility test.

1) Application-Level Speed Scaling—All Tasks Have the Same Speed: Here, all tasks have the same speed f^* and $s(\tau_1) = s(\tau_2) = \cdots = s(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Then, (2) is simplified as

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

For each given speed f^*, in order to minimize all response times, we must have $m_i^* = \max(\lVert \sqrt{kE_i/(f^* C_s)} - 1 \rVert, 0), 1 \le i \le n$. The iterative method described in Section III-A can be used here. To examine the feasibility for each task, all possible speeds have to be examined. There are l possibilities in total. The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

2) Task-Level Speed Scaling—Different Tasks Can Have Different Speeds: To obtain an optimal solution, a straightforward solution is to use an exhaustive search method. Since each task can be run at l speeds, there are l^n possible speed combinations for n tasks. Given a speed assignment, in order to minimize all response times, we must have $m_i^* = \max(\lVert \sqrt{kE_i/(s(\tau_i) C_s)} - 1 \rVert, 0), 1 \le i \le n$. The feasibility test is performed according to (2). Meanwhile, the energy consumption is calculated from (3). The speed combination that satisfies the timing constraints with the minimum energy consumption is chosen as the optimal solution.

Finally, the voltage switching cost is incorporated. The difficulty in modeling this cost accurately is that exact switching events can only be known after the schedule is obtained; hence, it is not possible to characterize it during offline feasibility analysis. Therefore, we make two conservative assumptions. First, it was assumed that there is always a voltage switching between two consecutive jobs and that the time and energy costs for voltage switching are $t_{\rm ss}$ and $\xi_{\rm ss}$. Second, context switching due to preemption is taken into account. We assume that voltage switching occurs when a lower-priority task is preempted by a higher-priority task. As discussed in [25], each job preempts at most one job. Hence, the maximum number of preemptions for a job is bounded by the number of higher-priority jobs. Furthermore, since each preemption leads to two context switches, the voltage switching cost for one preemption should be multiplied by a factor of two, so the time and energy costs for voltage switching incurred by a single higher-priority job are $2t_{\rm ss}$ and $2\xi_{\rm ss}$. Adding the costs incurred by voltage switching between consecutive jobs ($t_{\rm ss}$ and $\xi_{\rm ss}$) and job preemption ($2t_{\rm ss}$ and $2\xi_{\rm ss}$), the maximum time and energy costs for voltage switching incurred by a single higher-priority job are $3t_{\rm ss}$ and $3\xi_{\rm ss}$, respectively. As a result, (2) and (3) are reexpressed as the equations shown at the bottom of the page.

The methods for application-level and task-level speed scaling can still be used here. If the task set can be scheduled under this conservative assumption, it is guaranteed that the task set can be scheduled under any voltage switching scenario.

B. Hyperperiod-Oriented Fault Tolerance With DVS

As in Section IV-A, initially we do not consider the voltage switching cost. Without the voltage switching cost, the worst case response time for task τ_i can be expressed as

$$R_i = \left\lceil \frac{E_i}{s(\tau_i)} + m_i C_s \right\rceil + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \left\lceil \frac{E_h}{s(\tau_h)} + m_h C_s \right\rceil + k(C_s + C_r) + k \max_{1 \le j \le i} \{F_j\} \quad (4)$$

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

The total energy consumption during one hyperperiod is expressed as

$$\Pi = \sum_{i=1}^{n} \frac{Ht}{T_i} \left[\mathcal{E}(E_i, s(\tau_i)) + m_i \xi_{\rm cs} \right] + k(\xi_{\rm cs} + \xi_{\rm cr}) + k\mathcal{E}(F^* s(\tau^*), s(\tau^*)) . \quad (5)$$

Here, τ^* is the task with the longest checkpointing interval, F^* represents its checkpointing interval, and $s(\tau^*)$ represents its corresponding speed assignment.

1) Application-Level Speed Scaling—All Tasks Have the Same Speed: Here, all tasks have the same speed f^* and $\mathrm{s}(\tau_1) = \mathrm{s}(\tau_2) = \cdots = \mathrm{s}(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Then, (4) is simplified to

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

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

For each given speed f^*, the feasibility of the task set is examined using the method in Section III-B. If it is schedulable, the corresponding number of checkpoints for each task can be obtained. The energy consumption is calculated from (5). The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

$$\begin{cases} R_i = \left\lceil \frac{E_i + \frac{kE_i}{m_i+1}}{s(\tau_i)} + k(C_s + C_r) + m_i C_s \right\rceil + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \left\lceil \frac{E_h + \frac{kE_h}{m_h+1}}{s(\tau_h)} + k(C_s + C_r) + m_h C_s + 3t_{\rm ss} \right\rceil \\ \Pi = \sum_{i=1}^{n} \frac{Ht}{T_i} \left[\mathcal{E}\left(E_i + \frac{kE_i}{m_i+1}, s(\tau_i) \right) + k(\xi_{\rm cs} + \xi_{\rm cr}) + m_i \xi_{\rm cs} + 3\xi_{\rm ss} \right] \end{cases}$$

2) Task-Level Speed Scaling—Different Tasks Can Have Different Speeds: To obtain an optimal solution, a straightforward solution is to use an exhaustive method. Since each task can be run at l speeds, there are l^n possible speed combinations for n tasks. For each speed combination, the feasibility test is performed according to (4). The method in Section III-B is employed and the corresponding number of checkpoints is obtained. Energy consumption is calculated from (5). The speed combination that satisfies the timing constraints with the minimum energy consumption is chosen as the optimal solution.

Next, the voltage switching cost is incorporated. As in the job-oriented case, the switching cost between two jobs and the switching cost due to preemption are incorporated. Based on this, (4) and (5) are reexpressed as

$$\begin{cases} R_i = \left\lceil \frac{E_i}{s(\tau_i)} + m_i C_s \right\rceil + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil \left\lceil \frac{E_h}{s(\tau_h)} + m_h C_s + 3t_{\rm ss} \right\rceil \\ \qquad + k(C_s + C_r) + k \max_{1 \le j \le i} \{F_j\} \\ \Pi = \sum_{i=1}^{n} \frac{Ht}{T_i} \left[\mathcal{E}(E_i, s(\tau_i)) + m_i \xi_{\rm cs} + 3\xi_{\rm ss} \right] + k(\xi_{\rm cs} + \xi_{\rm cr}) \\ \qquad + k\mathcal{E}(F^* s(\tau^*), s(\tau^*)) \end{cases}.$$

The methods for application-level and task-level speed scaling can again be used here. If the task set can be scheduled under this conservative assumption, it is guaranteed that the task set can be scheduled under any other voltage switching scenario.

C. Heuristic Method Based on a Genetic Algorithm (GA)

As expected, the exhaustive method for task-level speed scaling is very time consuming. For instance, we carried out simulation for a real-time application composed of 17 tasks with three variable speeds. The exhaustive search method examined all 3^{17} speed combinations and took 63 h of CPU time on a Pentium IV PC with a 1.4-GHz processor and 256-MB memory. It is therefore obvious that the exhaustive method can only be applied to moderate-sized problem instances. When the size of the task set or the number of processor speeds is large, a heuristic method needs to be employed to obtain acceptable performance with low computation cost. Heuristics based on GAs have been used to solve a number of combinatorial search problems. We use a GA-based heuristic here for task-level speed scaling. It is applicable for both job- and hyperperiod-oriented cases. The solution is approximate. Alternative heuristic approaches can also be developed to solve this problem. However, since the focus of this paper is not on the comparison between heuristic approaches, an investigation into other heuristic techniques is left for future work.

We choose a GA based on two reasons. First, they are targeting a multiobjective optimization problem for which researchers have often used GAs by formulating the problem in terms of two-priority optimizations. The primary objective here is to meet task deadlines deterministically, even though k faults occur, either during the execution of a job in the job-oriented model or during a hyperperiod in the hyperperiod-oriented model. A secondary goal is to minimize energy consumption.

Procedure Init(Γ)
Input: Task set Γ and processor speeds $f_1, f_2, \ldots, f_l$
Output: Initial chromosome population Ω with a size of P
(1) Find the lowest speed f^* which makes the task set schedulable;
(2) Generate one chromosome $\alpha_0 = (f^*, f^*, \ldots, f^*)$;
(2) Apply random mechanism to generate the other $(P-1)$ chromosomes.

Procedure GA(Ω)
Input: Initial chromosome population: $\Omega = \{\alpha_i | \alpha_i = (v_1, v_2, \ldots, v_n), 1 \le i \le P\}$
Output: chromosome $\alpha^* = (f_1^*, f_2^*, \ldots, f_n^*)$ which makes the task set schedulable and minimize energy consumption
(1) **while** number of generations not exhausted **do**
(2) **for** $j = 1$ **to** PopulationSize **do**
(3) Select two chromosomes with the highest fitness values, apply the crossover operator randomly, generate two children;
(4) Apply mutation to two children randomly, update their fitness values;
(5) **endfor**
(6) **endwhile**
(7) Report the best chromosome as the final solution.

Fig. 5. Heuristic search based on GA.

Among various randomized search algorithms, GAs and simulated annealing algorithms (SAs) have been deemed in the literature to be appropriate for multiobjective optimization [42]. Second, a simple implementation of SA deals with only one solution at a time, and no information from previous moves is used to guide the selection of new moves [43]. In contrast to SA, GA maintains a pool of solutions instead of a single solution and allows communication between solutions via crossover and mutation. In this way, GA is better equipped to escape the local minima and use information from previous moves.

The GA is divided into two stages: application-level population generation and task-level heuristic search. The procedure is described in Fig. 5. Each chromosome α_i is a n-dimensional vector $(v_{i1}, v_{i2}, \ldots, v_{in})$, where n is the number of tasks and v_{ij} is the corresponding speed for task τ_j. Furthermore, α_i is viable if the task set can be scheduled under the corresponding speed assignment, otherwise it is not viable. Procedure $\mathrm{Init}(\Gamma)$ initializes the search space (chromosome population). One chromosome is initially generated using the computationally feasible application-level speed scaling method. This is to ensure that the initial population always includes a schedulable solution if such a solution exists. The other chromosomes are generated randomly. The initial population Ω is composed of these chromosomes. Procedure $\mathrm{GA}(\Omega)$ applies crossover and mutation operators to Ω based on the fitness values. The operations are repeated for a predefined number of generations Q. The fitness value $\mathrm{fit}(\alpha_i)$ is defined as follows.

1) If α_i is not viable: $\mathrm{fit}(\alpha_i) = \mathrm{rand}()$, where rand() is a uniform random function that returns a value between 0 and 1.
2) If α_i is viable, we need to make sure that it has a higher probability to be chosen compared to the case when it is not viable. In the scheme, for the sake of simplicity, we design a function with two terms for this case: $\mathrm{fit}(\alpha_i) = A + C \times B/\mathrm{Energy}(\alpha_i)$, in which A, B, and C are constants, and $\mathrm{Energy}(\alpha_i)$ is the energy consumption for the task set under chromosome α_i. The constants are determined as follows.
   a) Constant A needs to be greater than 0.5 to ensure a greater probability than the uniform distribution.
   b) Constant C is set to $(1 - A)$ to make sure that the fitness value ranges from A to 1.

c) Constant B is set to the value of the energy consumption under fault-free conditions using some well-known schemes such as VSLP.
d) A chromosome α_i with low-energy consumption $\mathrm{Energy}(\alpha_i)$ has a high fitness value, which makes it more likely to be selected.

The choice of the value for A is based on preprocessing using a large number of experiments. It has to tradeoff between task feasibility and energy consumption. If A is too small (i.e., slightly greater than 0.50), then energy consumption plays a more important role and the difference between a viable chromosome and that which is not viable is not significant; if A is too large, then the effect of energy consumption becomes small in the selection procedure. Based on experiments with random task sets, we choose $A = 0.6$ and $C = 0.4$ for the GA-based algorithm.

The mutation and crossover operators used in the procedure are defined as follows.

1) Crossover: find an index randomly; then one child keeps the information of its parent to the left of the index and fills the right with the other parent chromosome, and the other child keeps the information of its parent to the right of the index and fills the left with the other parent chromosome.
2) Mutation: choose a certain number of bits from two children randomly and replace them with a different information.

The choices for the number of generations Q and the population size P are also based on experiments. For the benchmarks in this paper, $Q = 1000$ yields good results. The value of P depends on the size of the task set and on the number of speeds.

The complexity of this heuristic method is linear with the number of generations Q and the population size P. In the 17-task, three-speed example for which the exhaustive method took 63 h, the heuristic method takes only 3 min. While the proposed GA-based technique is found to yield good results in Section V, it is particularly difficult to establish a theoretical basis that explains its effectiveness for this problem. By their very nature, it is very hard to theoretically justify the suitability of GAs for a given optimization problem, or explain analytically why they work well. As a result, it is typical in the area of evolutionary algorithms to demonstrate the suitability of a GA-based approach for a problem experimentally.

D. Job-Level Online Speed Scaling

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

The on-line speed scaling procedure, done at the job level, is adaptive with respect to fault occurrence. It makes use of a simple run-time adaptation mechanism. The key features of this procedure are: 1) once a job completes, the release time of the next job is adjusted dynamically during run time and 2) the processor is run at an appropriate speed such that the current job completes either before its deadline or before the static release time of the next job, whichever is sooner.

V. Numerical Results

This section compares the performance of the energy-aware fault-tolerance scheme with the DVS technique proposed in [5], referred to as VSLP. In the absence of any published experimental results on energy-aware fault tolerance, we are only able to compare the approach with DVS schemes that do not consider fault occurrences. On the other hand, current fault-tolerant schemes do not incorporate DVS, which makes them energy inefficient. We therefore compare the approach with fault-tolerant schemes that do not consider energy. The goal here is to highlight the impact of fault occurrences on a fault-oblivious DVS scheme and quantify the energy saving of the scheme over a DVS-oblivious checkpointing scheme.

We use the following notation to refer to the various types of fault tolerance and energy-aware fault tolerance schemes: 1) JFTC: job-oriented fault tolerance under constant speed; 2) JFTA: job-oriented fault tolerance with application-level speed scaling; 3) JFTT: job-oriented fault tolerance with task-level speed scaling; 4) HFTC: hyperperiod-oriented fault tolerance under constant speed; 5) HFTA: hyperperiod-oriented fault tolerance with application-level speed scaling; and 6) HFTT: hyperperiod-oriented fault tolerance with task-level speed scaling.

Since the VSLP scheme presented in [5] does not provide fault tolerance, we assume that it simply reexecutes a job when a fault occurs. As for the DVS-oblivious constant-speed schemes (JFTC and HFTC), it was assumed that the tasks are executed under the highest processor speed. Furthermore, since JFTA is a special case of JFTT and HFTA is a special case of HFTT, the other schemes were compared with the JFTT and HFTT schemes. For both cases, we first show that JFTT and HFTT can schedule task sets even when the VSLP cannot do so, that these schemes can save more energy via checkpointing in the presence of faults, and finally that these schemes can also save energy via DVS compared to the DVS-oblivious schemes.

Two low-power embedded processors were chosen for the experiments: the Intel XScale PXA260 [6] and the Transmeta Crusoe [8]. The parameter values that are listed in data sheets were used. The frequencies, voltages, and corresponding power consumptions for these processors are listed in Table I.

We evaluate the schemes on three real-life task sets. These task sets include a CNC task set [44], an inertial navigation system (INS) task set [45], and a generic aviation platform (GAP) task set [46], respectively. The characteristics of the three task sets are listed in Table II. The task execution times for these task sets are assumed for a nominal CPU frequency of 200 MHz.

The choices of k are based on the characteristics of the task sets and typical fault arrival rates. For the job-oriented case, the

TABLE I
PROCESSOR FREQUENCIES, VOLTAGES, AND POWER [6], [8]

Intel XScale PXA260			Transmeta Crusoe		
CPU Frequency (MHz)	Voltage (V)	CPU Power (mW)	CPU Frequency (MHz)	Voltage (V)	CPU Power (W)
200	1.0	178	300	1.2	1.3
300	1.1	283	400	1.225	1.9
400	1.3	411	533	1.35	3.0
			600	1.5	4.2
			667	1.6	5.3

TABLE II
CHARACTERISTICS OF THE BENCHMARKS

Task Set	Number of Tasks	Execution Times of the Tasks (ms)	Period of the Tasks (ms)	Hyperperiod (s)
CNC	8	0.035 – 0.72	2.4 – 9.6	0.1248
INS	6	1.18 – 25	2.5 – 1250	5
GAP	17	1 – 9	5 – 1000	118

TABLE III
JFTT PERFORMANCE COMPARISON USING INTEL XScale

Task Set	k	Zero DVS Cost					Nonzero DVS Cost				
		VSLP: E_1 (mJ)	JFTC: E_2 (mJ)	JFTT: E_3 (mJ)	E_{13}	E_{23}	VSLP: E_1 (mJ)	JFTC: E_2 (mJ)	JFTT: E_3 (mJ)	E_{13}	E_{23}
INS	1	1764.2	1879.3	1573.8	10.8	5.1	1963.9	1879.3	1837.6	6.4	2.2
GAP	1	40 131.2	36 724.6	33 228.6	17.2	4.4	41 394.8	36 724.6	35 418.1	14.4	3.6

typical value of k should be relatively small since each task instance is short. It is therefore enough to require that each task instance tolerate one fault ($k = 1$). Therefore, we choose $k = 1$ for INS and GAP. In addition, since the execution times and periods for CNC are extremely short, it is impractical to incorporate job-oriented fault tolerance for this task set. For the hyperperiod-oriented case, since a hyperperiod can be as long as hundreds of seconds, it is reasonable to choose a larger value of k. For example, the hyperperiod for GAP is 118 s. If we assume that the fault arrival rate is as high as 140/h, as in [34], then the number of faults during this hyperperiod can be as high as 5. In the experiments for the hyperperiod-oriented case, we therefore set the values of k to 1, 2, and 4 for CNC, INS, and GAP, respectively.

Based on the discussion in Section II, it was assumed that the checkpoint size is 5 kB and that checkpoint data are saved in DRAM. Based on the typical access speeds of DRAM described in Section II, the time to read or write a checkpoint of size 5 kB is assumed to be 0.4 ms. We choose a power consumption value of 400 mW for the DRAM [47]. Hence, the energy consumption for saving or retrieving a checkpoint is 160 μJ. In addition, based on data provided in the literature in [37]–[40], it was assumed that a single DVS transition takes 100 μs and consumes 30 μJ. We further classify the simulation into two categories: zero DVS cost and nonzero DVS cost. This classification is done in order to highlight the effect of DVS cost on the task set feasibility and the energy consumption. For some processors, it has been reported that task execution can proceed concurrently with voltage scaling [48]; hence, we also consider the case of zero voltage switching time.

Since the number of tasks for CNC and INS is relatively small, the simulation results for CNC and INS are obtained using the exhaustive search method. The simulation results for GAP are obtained using the heuristic method. Typical DVS costs due to context switching have also been considered in these experiments.

A. JFTT Results Based on Data Sheets of the Intel XScale Processor

The simulation results for zero voltage-scaling cost are shown in Table III. The last two columns of the table show the energy saving of JFTT compared to VSLP and JFTC, respectively. In the table, $E_{13} = (E_1 - E_3)/E_1 \times 100\%$ and $E_{23} = (E_2 - E_3)/E_2 \times 100\%$. It can be seen that JFTT saves as much as 17% more energy compared to VSLP. The performances of JFTC and JFTT are comparable. This is because JFTT has to often run at the highest processor speed to ensure timely task completion. As a result, the energy saving is not so significant.

Next, we assume nonzero DVS cost. The performance comparisons are also shown in Table III. From Table III, it can be seen that JFTT saves approximately 15% more energy compared to VSLP. The performances of JFTT and JFTC are once again comparable.

B. JFTT Results Based on Data Sheets of the Transmeta Crusoe Processor

Since the Transmeta Crusoe processor consumes more power compared to the Intel XScale processor, the checkpointing energy is relatively small in this case compared to the task execution energy. The performance gain of JFTT over VSLP is therefore more significant in this case. Furthermore, since the power consumption varies more for different processor speeds, it is expected that JFTT can achieve more energy saving than the DVS-oblivious JFTC scheme.

The simulation results for zero DVS cost are shown in Table IV. It can be seen that JFTT performs significantly better

TABLE IV
JFTT PERFORMANCE COMPARISON USING TRANSMETA CRUSOE

Task Set	k	Zero DVS Cost					Nonzero DVS Cost				
		VSLP: E_1 (mJ)	JFTC: E_2 (mJ)	JFTT: E_3 (mJ)	E_{13}	E_{23}	VSLP: E_1 (mJ)	JFTC: E_2 (mJ)	JFTT: E_3 (mJ)	E_{13}	E_{23}
INS	1	8138.6	11 967.9	5698.1	30.0	51.9	8417.3	11 967.9	5793.1	31.2	51.6
GAP	1	207 474.0	240 372.5	138 563.0	33.2	42.1	210 456.3	240 372.5	139 684.2	33.6	41.9

TABLE V
HFTT PERFORMANCE COMPARISON USING INTEL XScale

Task Set	k	Zero DVS Cost					Nonzero DVS Cost				
		VSLP: E_1 (mJ)	HFTC: E_2 (mJ)	HFTT: E_3 (mJ)	E_{13}	E_{23}	VSLP: E_1 (mJ)	HFTC: E_2 (mJ)	HFTT: E_3 (mJ)	E_{13}	E_{23}
CNC	1	22.9	25.6	21.6	5.7	15.6	NF	25.6	25.6	–	0
INS	1	1764.2	929.1	812.9	53.9	12.5	1895.7	929.1	910.2	49.9	2.0
	2	NF	949.9	845.2	–	11.0	NF	949.9	935.1	–	1.6
GAP	1	40 131.2	20 615.9	20 200.9	49.7	2.0	40 131.2	20 615.9	20 583.2	48.7	0.2
	2	NF	22 729.5	22 431.4	–	1.3	NF	22 729.5	22 729.5	–	0
	3	NF	25 307.4	25 002.3	–	1.2	NF	25 307.4	25 307.4	–	0

TABLE VI
HFTT PERFORMANCE COMPARISON USING TRANSMETA CRUSOE

Task Set	k	Zero DVS Cost					Nonzero DVS Cost				
		VSLP: E_1 (mJ)	HFTC: E_2 (mJ)	HFTT: E_3 (mJ)	E_{13}	E_{23}	VSLP: E_1 (mJ)	HFTC: E_2 (mJ)	HFTT: E_3 (mJ)	E_{13}	E_{23}
CNC	1	105.7	98.2	60.5	42.8	38.4	NF	98.2	98.2	–	0
INS	1	8138.6	7184.1	3917.9	51.9	45.5	8233.2	7184.1	4238.6	48.5	41.0
	2	15 009.8	7243.7	3931.7	73.8	45.7	15 147.6	7243.7	4241.2	72.0	41.4
GAP	1	207 474.0	159 429.2	89 057.0	57.1	44.1	208 314.0	159 429.2	92 343.6	55.7	42.1
	2	462 897.6	159 443.7	97 277.4	79.0	39.0	463 715.6	159 443.7	105 271.7	77.3	34.0
	3	NF	159 995.1	100 002.4	–	37.5	NF	159 995.1	118 118.4	–	26.2
	4	NF	160 003.2	111 595.8	–	30.3	NF	160 003.2	131 591.5	–	17.8

than VSLP. For example, the energy saving for GAP is as high as 33.2%. Next, we compare JFTT and JFTC. As expected, JFTT saves much more energy. For example, JFTT saves 51.9% energy over JFTC for INS. This is because JFTT can scale down the processor speed more often when faults occur less frequently.

Finally, we examine the case of nonzero DVS cost. The performance comparisons are also shown in Table IV. JFTT saves up to 33.6% energy compared to VSLP and up to 51.6% energy compared to JFTC.

C. HFTT Results Based on Data Sheets of the Intel XScale Processor

The performance comparison for zero DVS cost is shown in Table V. In the table, "NF" denotes that the task set cannot be feasibly scheduled. HFTT performs better than VSLP and HFTC in all cases. First, HFTT saves more energy when all schemes are feasible. The energy saving for HFTT over VSLP is as high as 53.9%, and it is as high as 15.6% over HFTC. Second, HFTT and HFTC tolerate more faults compared to VSLP. For example, when k is greater than one for INS and GAP, VSLP is not feasible while both HFTT and HFTC still guarantee the feasibility.

The performance comparison for nonzero DVS cost is also shown in Table V. HFTT saves as much as 49.9% in energy compared to VSLP. The performances of HFTT and HFTC are comparable.

D. HFTT Results Based on Data Sheets of the Transmeta Crusoe Processor

The performance comparison for zero DVS cost is shown in Table VI. HFTT outperforms VSLP and HFTC in all cases. The energy saving for HFTT over VSLP is as high as 79.0%, and it is as high as 45.7% over HFTC.

The performance comparison for nonzero DVS cost is also shown in Table VI. HFTT saves as much as 77.3% in energy over VSLP, and as much as 42.1% in energy over HFTC.

To further demonstrate the performance of the three schemes, we show the performance comparison for nonzero DVS cost for the GAP benchmark in Fig. 6. The energy consumption is normalized according to the value of the energy consumption for HFTT when $k = 1$. It is noticed that VSLP can only tolerate up to two faults, beyond that it is not feasible. In addition, it consumes much more energy than HFTC and HFTT. Between HFTC and HFTT, HFTT always achieves more energy consumption.

The performance of the GA-based heuristic is compared to simple alternatives. First, we use the constant slowdown scheme as a baseline and compare it to the GA-based heuristic. Constant slowdown corresponds to the application-level

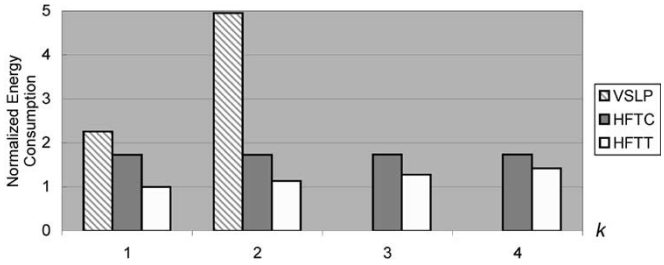

Fig. 6. Performance comparison for GAP using Transmeta Crusoe (nonzero DVS cost).

TABLE VII
COMPARISON OF THE GA-BASED HEURISTIC WITH CONSTANT SLOWDOWN SCHEMES (TRANSMETA CRUSOE PROCESSOR AND NONZERO DVS COST)

Task Set	Job-Oriented Schemes				Hyperperiod-Oriented Schemes			
	k	JFTA: E_0 (mJ)	JFTT: E_3 (mJ)	E_{03} (percent)	k	HFTA: E_0 (mJ)	HFTT: E_3 (mJ)	E_{03} (percent)
GAP	1	164 336.5	139 684.2	15.0	1	121 591.2	89 057.0	26.8
					2	122 685.0	97 277.4	20.7
					3	123 772.1	100 002.4	19.2
					4	126 910.7	111 595.8	12.1

TABLE VIII
COMPARISON OF THE GA-BASED HEURISTIC WITH RANDOM SELECTION (HFTT, TRANSMETA CRUSOE PROCESSOR, AND NONZERO DVS COST)

Task Set	k	Random Selection E_4 (mJ)	GA-based Heuristic E_3 (mJ)	$E_{43} = (E_4 - E_3)/E_4 \times 100\%$
GAP	1	95 132.5	92 343.6	2.9
	2	111 287.2	105 271.7	5.4
	3	123 492.6	118 118.4	4.4
	4	140 678.1	131 591.5	6.5

speed-scaling scheme (JFTA for the job-oriented scheme and HFTA for the hyperperiod-oriented scheme). For such a scheme, the lowest speed that satisfies the timing constraints is selected for all tasks. The comparison between the constant slowdown scheme and the GA-based heuristic is shown in Table VII. Let $E_{03} = (E_0 - E_3)/E_0 \times 100\%$. We find that for the GAP benchmark, GA-based JFTT can achieve 15.0% energy savings over the constant slowdown scheme (JFTA) and GA-based HFTT can achieve as much as 26.8% energy savings over the constant slowdown scheme (HFTA). The CPU time for the GA-based heuristic is in the order of minutes, while the CPU time for the constant slowdown scheme is in the order of seconds. Compared to constant slowdown, the CPU time for the GA-based heuristic is slightly higher but the energy saving is significant.

Next, we have compared the GA-based heuristic to a random selection method. In the latter case, we use the same initial set of chromosomes used in the GA-based heuristic and randomly select one of these chromosomes. Thus, they do not apply the mutation and the crossover operations. The performance comparison between the GA-based heuristic and the random selection method for HFTT is shown in Table VIII. The results show that the mutation and crossover operations can lead to measurable energy savings in a single hyperperiod over the random selection method. Over several hyperperiods, the energy savings can be significant. The CPU times for both the GA-based heuristic and the random selection scheme are in the order of minutes.

VI. CONCLUSION

A unified approach for achieving fault tolerance and energy savings in embedded systems has been presented. Fault tolerance is achieved via checkpointing and energy is saved using dynamic voltage scaling (DVS). We have presented feasibility-of-scheduling tests for checkpointing schemes under both constant processor speed and variable processor speed. Two feasibility tests have been developed for application-level and task-level speed scaling, respectively. A heuristic method based on a GA has been proposed to reduce computational complexity. We have presented numerical results for two commercial embedded processors—the Intel XScale PXA260 and the Transmeta Crusoe—using real-time benchmark task sets and realistic values of parameters such as checkpointing cost, memory access time and power consumption, and voltage scaling cost. The values of these parameters have been derived from processor data sheets. As part of a future work, We are developing alternative heuristic methods to improve the GA-based heuristic approach.

ACKNOWLEDGMENT

The authors thank the reviewers for their valuable suggestions that have improved the content and presentation of the paper.

REFERENCES

[1] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 3, pp. 299–316, Jun. 2000.
[2] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable-voltage core-based systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 12, pp. 1702–1714, Dec. 1999.
[3] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low Power Electronics and Design*, Monterey, CA, 1998, pp. 197–202.
[4] Y. Shin, K. Choi, and T. Sakurai, "Power optimization of real-time embedded systems on variable speed processors," in *Proc. Int. Conf. Computer-Aided Design*, San Jose, CA, 2000, pp. 365–368.
[5] G. Quan and X. Hu, "Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors," in *Proc. Design Automation Conf.*, Las Vegas, NV, 2001, pp. 828–833.
[6] Intel PXA26x Processor Family Electrical, Mechanical, and Thermal Specification Datasheet. [Online]. Available: http://developer.intel.com/design/pca/applicationsprocessors/datashts/27864002.pdf
[7] Motorola 6805 Processor. [Online]. Available: http://www.motorola.com
[8] Transmeta LongRun Power Management—Dynamic Power Management for Crusoe Processors. [Online]. Available: http://www.transmeta.com/pdf/white_papers/paper_mfleischmann_17jan01.pdf
[9] AMD PowerNow! Technology. [Online]. Available: http://www.amd.com/epd/processors/6.32bitproc/8.amdk6fami/x24404/24404a.pdf
[10] J. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, Oct. 1988.
[11] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. Int. Conf. Computer-Aided Design*, San Jose, CA, 2001, pp. 259–263.
[12] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Design Automation Conf.*, New Orleans, LA, 1999, pp. 134–139.
[13] G. Qu "What is the limit of energy saving by dynamic voltage scaling?" in *Proc. Int. Conf. Computer-Aided Design*, San Jose, CA, 2001, pp. 560–563.
[14] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles, "Energy efficient mapping and scheduling for DVS enabled distributed embedded systems," in *Proc. Design, Automation and Test Europe Conf.*, Paris, France, 2002, pp. 514–521.
[15] R. Jejurikar and R. Gupta, "Energy aware task scheduling with task synchronization for embedded real time systems," in *Proc. Int. Conf.*

Compilers, Architecture and Synthesis Embedded Systems, Grenoble, France, 2002, pp. 164–169.
[16] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*. Natick, MA: A. K. Peters, Ltd., 1998.
[17] K. G. Shin and Y.-H. Lee, "Error detection process—Model, design and its impact on computer performance," *IEEE Trans. Comput.*, vol. C-33, no. 6, pp. 529–540, Jun. 1984.
[18] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic models for rollback and recovery strategies in data base systems," *IEEE Trans. Softw. Eng.*, vol. 1, no. 1, pp. 100–110, Mar. 1975.
[19] D. K. Pradhan and N. H. Vaidya, "Roll-forward and rollback recovery: Performance reliability trade-off," *IEEE Trans. Comput.*, vol. 46, no. 3, pp. 372–378, Mar. 1997.
[20] K. Shin, T. Lin, and Y. Lee, "Optimal checkpointing of real-time tasks," *IEEE Trans. Comput.*, vol. 36, no. 11, pp. 1328–1341, Nov. 1987.
[21] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Trans. Comput.*, vol. 46, no. 9, pp. 976–985, Sep. 1997.
[22] S. W. Kwak, B. J. Choi, and B. K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *IEEE Trans. Reliab.*, vol. 50, no. 3, pp. 293–301, Sep. 2001.
[23] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proc. Design, Automation and Test Europe Conf.*, Munich, Germany, 2003, pp. 918–923.
[24] R. Melhem, D. Mosse, and E. N. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Trans. Comput.*, vol. 53, no. 2, pp. 217–231, Feb. 2004.
[25] J. W. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice-Hall, 2000.
[26] E. Dupont, M. Nicolaidis, and P. Rohr, "Embedded robustness IPs for transient-error-free ICs," *IEEE Des. Test. Comput.*, vol. 19, no. 3, pp. 54–68, May–Jun. 2002.
[27] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing*. Norwell, MA: Kluwer, 2000.
[28] E. N. Elnozahy, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.
[29] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proc. Symp. Reliable Distributed Systems*, Houston, TX, 1992, pp. 39–47.
[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.
[31] C.-Y. Lin, S.-Y. Kuo, and Y. Huang, "A checkpointing tool for Palm operating system," in *Proc. Dependable Systems and Networks*, Göteborg, Sweden, 2001, pp. 71–76.
[32] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Proc. Usenix Technical Conf.*, New Orleans, LA, 1995, pp. 213–223.
[33] C. M. Krishna and A. D. Singh, "Reliability of checkpointed real-time systems using time redundancy," *IEEE Trans. Reliab.*, vol. 42, no. 3, pp. 427–435, Sep. 1993.
[34] S. Punnekkat, A. Burns, and R. Davis, "Probabilistic scheduling guarantees for fault-tolerant real-time systems," in *Proc. Int. Conf. Dependable Computing Critical Applications*, San Jose, CA, 1999, pp. 361–378.
[35] A. Campbell, P. McDonald, and K. Ray, "Single event upset rates in space," *IEEE Trans. Nucl. Sci.*, vol. 39, no. 6, pp. 1828–1835, Dec. 1992.
[36] D. Shin, S. Lee, and J. Kim, "Intra-task voltage scheduling for low-energy hard real-time applications," *IEEE Des. Test. Comput.*, vol. 18, no. 2, pp. 20–30, Mar.–Apr. 2001.
[37] T. Pering, T. Burd, and R. Broderson, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics and Design*, Monterey, CA, 1998, pp. 76–81.
[38] T. Burd and R. Broderson, "Design issues for dynamic voltage scaling," in *Proc. Int. Symp. Low Power Electronics and Design*, Rapallo, Italy, 2000, pp. 9–14.
[39] K. Nowka, G. Carpenter, E. M. Donald, H. Ngo, B. Brock, K. Ishii, K. Nguyen, and J. Burns, "A 0.9 V to 1.95 V dynamic voltage-scalable and frequency-scalable 32b PowerPC processor," in *Proc. IEEE Int. Solid-State Circuits Conf.*, San Francisco, CA, 2002, pp. 340–341.
[40] D. Grunwald, P. Levis, C. Morrey, III, M. Neufeld, and K. Farkas, "Policies for dynamic clock scheduling," in *Proc. Symp. Operating Systems Design and Implementation*, San Diego, CA, 2000, pp. 73–86.
[41] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Syst. J.*, vol. 20, no. 1, pp. 83–102, Jan. 2001.
[42] R. P. Dick, "Multiobjective synthesis of low-power real-time distributed embedded systems," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, Nov. 2002.
[43] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines*. Chichester, U.K.: Wiley, 1989.
[44] N. Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin, "Visual assessment of a real-time system design: Case study on a CNC controller," in *Proc. Real-Time Systems Symp.*, Washington, DC, 1996, pp. 300–310.
[45] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. Softw. Eng.*, vol. 19, no. 9, pp. 920–934, Sep. 1993.
[46] D. C. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics platform in Ada: A case study," in *Proc. Real-Time Systems Symp.*, San Antonio, TX, 1991, pp. 181–189.
[47] T. Simunic, L. Benini, P. Glynn, and G. D. Micheli, "Event-driven power management," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 7, pp. 840–857, Jul. 2001.
[48] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE J. Solid-State Circuits*, vol. 35, no. 11, pp. 1571–1580, Nov. 2000.

Ying Zhang received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 1997 and 1999, respectively, and the Ph.D. degree in electrical and computer engineering from Duke University, Durham, NC, in 2004.

He is currently a Senior Software engineer with the Research and Development Department, Guidant Corporation, St. Paul, MN. His research interests include real-time scheduling, low-power design, and fault tolerance.

Krishnendu Chakrabarty (S'92–M'96–SM'00) received the B.Tech. degree from the Indian Institute of Technology, Kharagpur, India, in 1990, and the M.S.E. and Ph.D. degrees, all from the University of Michigan, Ann Arbor, in 1992 and 1995, respectively.

He is currently an Associate Professor of Electrical and Computer Engineering at Duke University, Durham, NC. From 2000 to 2002, he was also a Mercator Visiting Professor at the University of Potsdam, Germany. He is a coauthor of two books, *Microelectrofluidic Systems: Modeling and Simulation* (Boca Raton, FL: CRC Press, 2002) and *Test Resource Partitioning for System-on-a-Chip* (Norwell, MA: Kluwer, 2002), and the Editor of *SOC (System-on-a-Chip) Testing for Plug and Play Test Automation* (Norwell, MA: Kluwer, 2002). He is also a coauthor of the forthcoming book *Scalable Infrastructure for Distributed Sensor Networks* (London, U.K.: Springer). He has published over 190 papers in journals and refereed conference proceedings, and he holds a U.S. patent in integrated circuit testing. His current research projects include design and testing of system-on-chip integrated circuits, embedded real-time systems, distributed sensor networks, design automation of microfluidics-based biochips, and microfluidics-based chip cooling.

Dr. Chakrabarty is the recipient of the National Science Foundation Early Faculty (CAREER) Award, the Office of Naval Research Young Investigator Award, the Best Paper Award at the 2001 Design, Automation and Test in Europe (DATE) Conference, and the Humboldt Research Fellowship, awarded by the Alexander von Humboldt Foundation, Germany. He is a Distinguished Visitor of the IEEE Computer Society from 2005 to 2007. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, an Associate Editor of IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, an Editor of the *Journal of Electronic Testing: Theory and Applications (JETTA),* and an Associate Editor for the *ACM Journal on Emerging Technologies in Computing Systems*. He serves on the Editorial Board of *Sensor Letters* and the *Journal of Embedded Computing*. He serves as a Subject Area Editor for the *International Journal of Distributed Sensor Networks*. He has also served as an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: ANALOG AND DIGITAL SIGNAL PROCESSING. He is a Member of ACM and ACM SIGDA, and a Member of Sigma Xi. He serves as Vice Chair of Technical Activities of the IEEE Test Technology Technical Council and is a Member of the program committees of several IEEE/ACM conferences and workshops. He served as the Tutorial Cochair for the 2005 IEEE International Conference on VLSI Design and is a designated Program Cochair for the 2005 IEEE Asian Test Symposium.