On Reliability Management of Energy-Aware Real-Time Systems through Task Replication

Mohammad A. Haque, Hakan Aydin, and Dakai Zhu,

Abstract—On emerging multicore systems, task replication is a powerful way to achieve high reliability targets. In this paper, we consider the problem of achieving a given reliability target for a set of periodic real-time tasks running on a multicore system with minimum energy consumption. Our framework explicitly takes into account the coverage factor of the fault detection techniques and the negative impact of Dynamic Voltage Scaling (DVS) on the rate of transient faults leading to soft errors. We characterize the subtle interplay between the processing frequency, replication level, reliability, fault coverage, and energy consumption on DVS-enabled multicore systems. We first develop static solutions and then propose dynamic adaptation schemes in order to reduce the concurrent execution of the replicas of a given task and to take advantage of early completions. Our simulation results indicate that through our algorithms, a very broad spectrum of reliability targets can be achieved with minimum energy consumption thanks to the judicious task replication and frequency assignment.

Index Terms—Energy-aware Systems, Real-time and Embedded Systems, Reliability, Multicore Systems, Scheduling

1 Introduction

In real-time systems, the timeliness of the output is as important as its logical correctness; in other words, timing constraints, often expressed in the form of *deadlines*, must be satisfied. A wide range of applications require real-time computing, including those in industrial automation, embedded control, avionics, and environmental acquisition/monitoring domains. As real-time embedded systems often control safety-critical applications tolerating faults and achieving high reliability levels is of utmost importance: faults must be detected, and appropriate recovery tasks must be successfully completed before the deadlines. Another very desirable feature of such systems is energy awareness. However, energy management and fault tolerance, are in general, conflicting system objectives as fault tolerance typically requires *redundancy* of system resources, such as extra CPU time and/or spare processing units [1], [2] [3], [4], [5].

Computer systems are susceptible to *faults*, leading to various *run-time errors*. Faults can be broadly categorized into two types: *transient* and *permanent* faults [6], [7]. Transient faults are known to occur much more frequently in practice, [8], [9]. Since transient faults are non-persistent, re-execution of the affected task or invocation of an alternate task are commonly used recovery techniques [6], [7].

Transient faults may manifest in the form of *single event upsets* or *soft errors* with incorrect computation results. These errors are commonly caused by alpha particles and cosmic rays (e.g., neutrons) that hit silicon chips in unpredictable way, creating large number of electron-hole pairs due to ionization mechanism and flipping the output of a logic gate [10]. These errors are called *soft* because they do not lead to a permanent failure on the affected hardware component [11]. Moreover, with increasing scaling levels in CMOS technologies, the susceptibility of computer systems to transient faults is known to increase, as more and more transistors are deployed per unit area [12], [13]. This is also considered as one of the great challenges for the *Near-threshold-voltage (NTV)* designs that are considered key for the next generation energy-efficient systems [14].

Dynamic Voltage Scaling (DVS), which is based on adjusting the CPU voltage and frequency, is a well-known energy management technique that trades off the processing speed with energy savings [15], [16]. The consideration of reliability for systems employing DVS for energy is equally important, as the current research suggests the negative impact of DVS on the transient fault rate [17], [18]: as we decrease the supply voltage and frequency to save power, the transient fault rate (and the corresponding soft errors) significantly increase. Therefore, energy management techniques must take into account the reliability degradation and make provisions accordingly, in particular for real-time applications. A set of techniques, called the *Reliability-Aware Power Management (RAPM)* [19], [20], exploit *time redundancy* available in the system for both energy and reliability management. These works consider the problem of preserving the system's *original reliability*, which is the reliability of the system executing tasks without any slowdown. The scheme in [20] resorts to *backward recovery* approach

• *Mohammad A Haque and Hakan Aydin are with the Department of Computer Science, George Mason University, Fairfax, VA, 22030. E-mail: aydin@cs.gmu.edu*
• *Dakai Zhu is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX, 78249.*

and assigns a recovery task for every task that has been slowed down, while the technique in [19] uses recovery blocks shared by all the scaled tasks.

More recently, the *reliability-oriented energy management* framework has been proposed for periodic real-time tasks running on a single-core system [21]. The main objective is to achieve *arbitrary reliability levels*, in terms of tolerance to transient faults, with minimum energy consumption. Unlike RAPM studies, the target reliability level can be *lower* or *higher* than the original reliability. This flexibility is important, as some high-criticality tasks may require very high-levels of reliability, while for some other tasks a modest reliability degradation may be acceptable to save energy. The solution still focuses on single-core systems and hence resorts to a limited number of shared recovery jobs that are invoked upon the detection of soft errors.

The main proposal of this paper is a *reliability-oriented energy management* framework for *multicore systems*. In the last decade, due to the advances in the CMOS technology, we witnessed the proliferation of systems with multiple processing cores. Systems with 2-4 cores are commonplace, and systems with more processing cores are also available. We observe that on these systems with multiple processing cores, *task replication* is likely to become a quite viable option for reliability management. By scheduling multiple copies of the same task on multiple cores, the likelihood of completing at least one of them successfully (i.e., without encountering transient faults) increases significantly. Replication has several advantages as an effective reliability management tool. Firstly, very high reliability targets can only be achieved through task replication. Secondly, replication has the potential of tolerating permanent faults in addition to improving reliability in terms of tolerance to transient faults. Thirdly, and most importantly, it creates an additional and powerful dimension to reduce the energy consumption by executing multiple copies at *lower* frequencies, while achieving the same reliability figures.

In our framework, we consider the energy-efficient replication problem for preemptive, periodic applications running on a multicore system. Our setting is *reliability-oriented* in the sense that we consider minimizing energy to meet arbitrary task-level or system-level reliability targets. Specifically, for a given reliability target, our goal is to find the *degree of replication* (number of copies) and the *frequency assignment* for all tasks, such that the overall energy consumption is minimized, while ensuring that all timing constraints will be met. Our reliability formulation explicitly considers the probability that the fault detection technique will work successfully during execution. This paper is an extension of the work presented in [22]. The main contributions of this paper can be summarized as follows:

- We present an extensive analysis to show the viability of replication as a tool for joint management of reliability and energy, while also taking into account the imperfect nature of the fault detection techniques and the impact of DVS on transient fault rates.
- We formulate the *Generalized Energy-Efficient Replication Problem (GEERP)* and show its intractability,
- We propose an efficient and approximate solution to GEERP,
- We develop a dynamic adaptation scheme based on *adaptive delaying*, that enables partial or complete cancellation of the additional task replicas to save further energy, and,
- We evaluate the performance of our framework under a wide range of system parameters.

2 System Model

2.1 Workload and Processor Model

We consider a set of N periodic real-time tasks $\Psi = \{\tau_1, ..., \tau_N\}$. Each task τ_i has worst-case execution time c_i under the maximum available CPU frequency f_{max}. τ_i generates a sequence of *task instances* (or, *jobs*) with the period of P_i. The relative deadline of each of these jobs is equal to the period value P_i, in other words, each job must complete by the arrival of the next job of the same task. The *utilization* of task τ_i, u_i, is defined as $\frac{c_i}{P_i}$. The *total utilization* U_{tot} is the sum of all the individual task utilizations.

The workload executes on a set of M identical cores. Each core can operate at one of the K different frequency settings ranging from a minimum frequency f_{min} to a maximum frequency f_{max}. We denote by F the set of available frequency settings. Without loss of generality, we normalize the frequency levels with respect to f_{max} (i.e., $f_{max} = 1.0$). At frequency f, a core may require up to $\frac{c_i}{f}$ time units to complete a job of task τ_i.

Our framework assigns k_i **replicas (copies)** of task τ_i to $k_i \leq M$ distinct processing cores, to achieve reliability targets. The entire set of $\sum k_i$ task copies are partitioned upon the multicore system. Each replica task, being a periodic task itself, generates a sequence of jobs on the core it is assigned to. The k^{th} replica of the j^{th} instance/job of task τ_i is denoted by $T^k_{i,j}$.

The preemptive Earliest-Deadline-First policy, which is known to be optimal on a single processing unit [23], is adopted to execute tasks on each core.

2.2 Power Model

We consider a power model adopted in previous reliability-aware power management research [24], [21], [20]. The power consumption of each core consists of static and dynamic power components. The static power P_s is determined mainly by the leakage current of the system. The dynamic power

P_d includes a frequency-dependent power component (determined by voltage and frequency levels), and a frequency-independent power component P_{ind}, driven by the modules such as memory and I/O subsystem in the active state. In DVS technique, the supply voltage is scaled in almost linear fashion with the processing frequency. Consequently, the power consumption of a core can be approximated by:

$$P_{core} = P_s + P_d = P_s + P_{ind} + C_e f^3 \tag{1}$$

Above, C_e is a system-dependent constant, reflecting the effective switching capacitance. When a core is not executing any task (idle state), its power consumption is primarily determined by the static power. We assume that the static power consumption can only be eliminated by the complete power-down of the core.

Existing research indicates that arbitrarily slowing down a task is not always energy-efficient [25], [26], [27], due to the frequency-independent power components. In other words, there is a processing frequency below which the total energy consumption increases. This frequency is called the *energy-efficient frequency* and denoted by f_{ee}. f_{ee} can be computed analytically through the well-known techniques [26], [18].

While the power model we adopt is commonly used in reliability-aware power management research, we note that our replication-based reliability management techniques would remain valid as long as P_{core} is a non-linear and convex function of the processing frequency f (and not necessarily a cubic function of f); however, numerical examples would need to be re-visited to obtain updated energy values.

2.3 Fault Model

Transient faults are typically modelled using an exponential distribution with an average arrival rate λ [28]. The fault rate λ increases significantly as frequency is scaled down when using DVS [17], [18]. The average fault rate at the maximum frequency is denoted by λ_0. The fault rate at frequency f can be expressed as [1], [18], [24], [29]:

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \tag{2}$$

Above, the exponent d, called the *sensitivity factor* in the paper, is a measure of how quickly the transient fault rate increases when the system supply voltage and frequency are scaled. Typical values range from 2 to 6 [18], [20], [24].

The *reliability* of a single task instance is defined as the probability of executing the task successfully, in the absence of transient faults [20]. Using the exponential distribution assumption, then the reliability of a single instance of task τ_i running at frequency f_i can be expressed as $e^{-\lambda(f_i)\frac{c_i}{f_i}}$ as in [1], [18], [24], [29].

At the end of execution of each task copy (replica), an *acceptance test* (or, *sanity check*) [6], [7] is conducted to check the occurrence of soft errors induced by the transient faults. If the test indicates no error, then the output of the task copy is committed to; otherwise, it is discarded. Therefore, in replicated execution settings of a given task, it is sufficient to have at least one task copy execution that passes the acceptance test.

Acceptance tests are not 100% accurate. Sometimes a fault may remain undetected or the acceptance test may declare a correct outcome faulty [6]. Consequently the reliability of a job in the presence of an imperfect acceptance test must be derived by factoring in the probability that the acceptance test will perform correctly. The latter probability is called *coverage factor* of the acceptance test in [6]. In our paper, we denote it by $(1-\alpha)$, where α is the probability of making an incorrect decision during the the acceptance test. Thus, the reliability of a job can be expressed as:

$$R_i(f_i) = (1-\alpha) \times e^{-\lambda(f_i)\frac{c_i}{f_i}} \tag{3}$$

Conversely, the *probability of failure (PoF)* of a task instance of τ_i is given by:

$$\phi_i(f_i) = 1 - R_i(f_i) \tag{4}$$

Throughout the paper, we consider reliability in terms of tolerance to transient faults only. We also focus on transient faults that affect the core logic; we assume that the memory subsystem is protected against transient faults through the use of error-correcting codes (ECCs) and memory interleaving which are known to provide strong protection against such faults [12], [30]. As our focus is the transient faults that affect the core hardware logic, failures that are due to software faults (bugs) are not considered.

3 Interplay of Energy, Reliability, Frequency, and Replication

Before presenting our detailed analysis, we start by illustrating how the level of replication and processing frequency jointly determine the reliability and energy savings. Consider a single instance of task τ_i running at frequency f_i. Its probability of failure is given by Equation (3) and is a function of the processing frequency f, λ_0, coverage factor and the sensitivity factor d. Now consider two replicas of τ_i running at frequency f. The execution with replication will be unsuccessful only if both replicas encounter transient faults during their respective executions. Hence, the new reliability with two replicas is found as:

$$R_i' = 1 - (1 - R_i(f))^2$$

Its new probability of failure is given by:

$$\phi_i^{(2)} = (\phi_i(f))^2$$

In general, with k replicas, the probability of failure decreases exponentially with k:

$$\phi_i^{(k)} = (\phi_i(f))^k \tag{5}$$

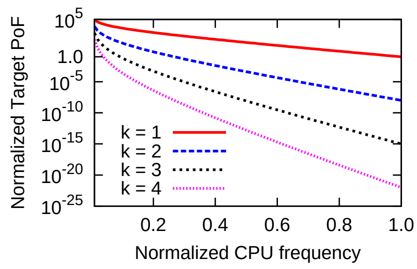


Fig. 1: Impact of the frequency on reliability

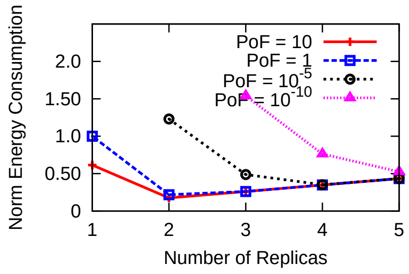


Fig. 2: Impact of replication level on energy

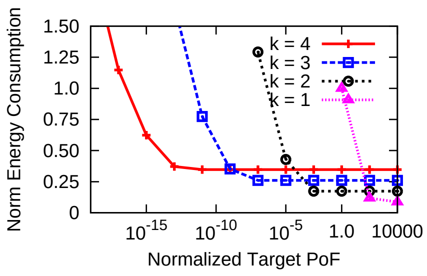


Fig. 3: Impact of target *PoF* on energy

We observe that the use of replication may be a powerful tool to mitigate the negative impact of the voltage/frequency scaling on the probability of failure, and improve energy savings through parallel execution. However, there are several non-trivial design dimensions that must be considered, including the energy cost of additional replica execution(s). As a concrete example, consider a single task with worst-case execution time $c_i = 100$ ms. Following [21], in this and subsequent examples used in the paper, we assume that transient fault arrival rate at the maximum frequency is $\lambda_0 = 10^{-6}$, and the system sensitivity factor d is 4. For illustration purposes, we assume a perfect acceptance test with coverage factor 1. We normalize the target *PoF* values with respect to a single copy running at the maximum frequency.

Figure 1 shows how the execution frequency determines the achievable *PoF*, for different number (k) of replicas. The *PoF* values are given in normalized form, with respect to the probability of failure of a single copy running at maximum frequency. Note that the $y-$axis in the plot is in logarithmic scale. Hence, for a given number of replicas, the *PoF* increases (the reliability decreases) rapidly with decreasing frequency. However, for a given frequency, there is also an exponential improvement on reliability (decrease in *PoF*), with increasing number of replicas. This simple fact points to an interesting design spectrum: the same target *PoF* value can be achieved at different frequency and replication levels.

We note that there is a lower bound on the number of replicas required to achieve a certain *PoF* target, ϕ_{target}. In particular, high reliability levels necessitate the use of multiple replicas. Using Equation (5), we can easily determine the minimum number of replicas (k) needed to achieve the target *PoF* at a given frequency level f:

$$\phi_{target} \geq (\phi_i(f))^k$$

$$k \geq \left\lceil \frac{\log(\phi_{target})}{\log(\phi_i(f))} \right\rceil \qquad (6)$$

On the other hand, the energy consumption of different replication/frequency levels yielding the same reliability level may vary significantly. Figure 2 shows the impact of replication on energy consumption for our example task, under various target (normalized) *PoF* values. In these experiments, for a given target *PoF* value ϕ_{target} and for each replication level (k), we compute the minimum energy consumption when we use the best common frequency f for all k replicas to achieve ϕ_{target}. The energy consumption is normalized with respect to a single replica running at the maximum frequency[1].

A few key observations are in order. First, some very high reliability (very low *PoF*) targets can only be achieved with large number of replicas. Second, for a fixed reliability target, using a minimum number of replicas generally consumes high energy. This is due to the fact that the minimum number of replicas are typically executed at high frequency levels to meet the reliability target ϕ_{target}, giving high energy figures. On the other hand, as we start increasing the number of replicas, we can afford reducing the frequency of individual replicas and thus the total energy consumption starts to decrease. But as we deploy further replicas, after a certain point, the energy consumption due to additional replicas offsets the energy savings due to execution at low frequencies. As a result, the energy consumption starts increasing. This is also coupled by the fact that reducing the frequency below the energy-efficient frequency f_{ee} is not helpful, even if we can use additional replicas. Consequently, the energy consumption figures continue to increase beyond a certain threshold point. Finally, it is clear that the optimal number of replicas to minimize energy depends highly on the target reliability.

Figure 3 depicts the interplay of target probability of failure and energy consumption. Notice that, as we increase the target *PoF*, the total energy consumption decreases for a fixed number of replicas, as we are more tolerant of reliability degradation and we can afford running the replicas at lower frequencies. Once the frequency required to achieve the reliability target reaches the energy-efficient frequency, the energy curve flattens out. At that point, increasing the target probability of failure does not yield any energy savings. Moreover, in our example, ϕ_{target} values smaller than 10^{-7} were not achievable by deploying < 3 replicas.

1. We assume $P_{ind} = 0.1$ in the examples.

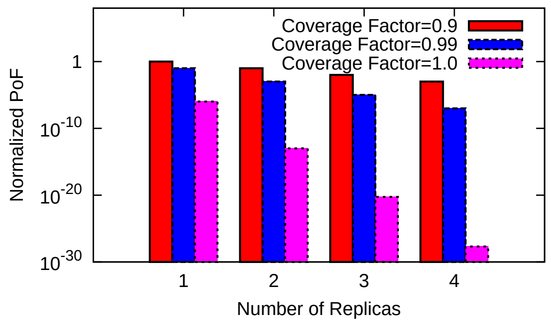


Fig. 4: Impact of the coverage factor

The coverage factor has also a significant impact in the results. In Fig. 4, we present the *PoF* as a function of the degree of replication and coverage factor. The CPU frequency is fixed at f_{max}. The *PoF* values are normalized with respect to a system with single replica running at f_{max} and coverage factor 0.9. As expected, the *PoF* decreases quickly with better coverage factor. We observe that we can achieve the same reliability target despite a lower coverage factor by deploying additional replicas. Consequently, replication can be used as a tool to compensate for an acceptance test with lower coverage factor. Finally, some reliability targets can only be achieved in the presence of an acceptance test with higher coverage factor and higher number of replicas.

The key observations in this section are as follows:

- In addition to the processing frequency, the replication can be used to manage reliability on DVS-enabled multicore systems, giving a broad design spectrum involving multiple dimensions, and in particular, *energy consumption*.
- The same target reliability can be typically achieved through multiple ways: using small number of replicas running at high frequencies or large number of replicas running at low frequencies. While the use of replication allows the system to use lower frequencies to mitigate the reliability loss, this may have a negative impact on the energy consumption due to the cost of executing additional replicas. The configuration that minimizes energy consumption varies depending on the system parameters and target reliability.
- Coverage factor provides another dimension of constraint and optimization opportunity. Replication can be used to compensate the impact of an acceptance test with low coverage factor.

4 Task-Level Analysis

We first address our problem of finding the energy-optimal configuration (i.e., the number and frequency assignment of replicas) in the context of a single task to achieve a target reliability level. This single-task setting enables us to illustrate some additional non-trivial aspects of the problem as well as to present the terminology and definitions that will be instrumental for the eventual system-level analysis that will be carried out in Section 5.

Ideally the system can choose different speed assignment for different replicas of the same job. Even though theoretically it should minimize the energy consumption, obtaining the speed assignment for all the replicas of all the tasks will be computationally prohibitive as the number of such possible combinations will increase exponentially with the number of replicas. Another alternative is to assign the same speed for all the replicas of a job. This choice reduces the computational overhead of determining the speeds and as observed in [22], the energy consumption increases only marginally compared to optimal but computationally prohibitive speed assignment method. Consequently, we will assume uniform speed for all replicas of a given task in the rest of the paper.

We start by observing that, for a given execution frequency, the minimum number of replicas needed to obtain a target reliability level is provided by Equation (6). Moreover, the number of available frequency levels is very limited in existing processors; for example, Intel Pentium M processor with 1.6 GHz maximum frequency supports only six frequency steps in the active state [31]. Consequently, for a given reliability target, one can easily compute the number of replicas needed, as well as the corresponding overall energy consumption for every available frequency setting in time $O(K)$. An interesting question is whether the energy numbers obtained as a function of decreasing replica frequency exhibit a certain (e.g., convex or concave) pattern. Unfortunately, as [22] illustrates, the answer is negative, and hence standard nonlinear optimization techniques do not apply.

In general, we can construct for every task τ_j a table with K rows on a system with K frequency levels. For each frequency setting, we compute the minimum number of replicas (through Equation (6)) to achieve its target reliability $\phi_{j,target}$ and the corresponding overall energy consumption. In preparation for our system-level analysis, we also include a column that indicates the total CPU time needed by all the replicas.

The entries given in each row corresponds to a separate *configuration* of the system. We denote the i^{th} configuration for a task by *RfConfig(i)*, which contains information about the frequency of each replica, number of replicas, total energy consumption, and total CPU time needed by all replicas. This *Energy-Frequency-Reliability (EFR)* table can be clearly constructed in time $O(K)$ for a given task.

Table 1 shows an example EFR table for a task whose execution time is $c = 100$ ms. The normalized *Pof* target is 10^{-6}. For simplicity, assume a perfect acceptance test and P_s and P_{ind} are negligible. Assuming the processor has 10 different frequency steps, we have 10 different configurations (*RfConfig(i)* $i = 1, \ldots, 10$).

TABLE 1: An Example EFR Table

Frequency, f	Replica # r	Energy	CPU time
1	2	0.2	0.2
0.9	2	0.162	0.222222
0.8	**3**	**0.192**	**0.375**
0.7	3	0.147	0.428571
0.6	3	0.108	0.5
0.5	3	0.075	0.6
0.4	4	0.064	1
0.3	4	0.036	1.33333
0.2	5	0.02	2.5
0.1	6	0.006	6

Note that, in general, the number of *valid* frequency levels may be smaller than the number of available frequency levels. This is because frequencies below the energy-efficient frequency f_{ee} should not be considered. In addition, since the task τ_j would miss its deadline at the frequency levels below its utilization value of $u_j = \frac{c_j}{P_j}$, we do not need to consider frequencies $< max\{f_{ee}, u_j\}$.

As we decrease the CPU frequency, the time required for executing each replica increases. The number of required replicas may remain the same or increase. As a result, the total CPU time consumption keeps increasing. However, in terms of energy consumption patterns, the trends are not always obvious. For example, looking at Table 1 and comparing the entries for $f = 0.9$ and $f = 0.8$, we observe an interesting phenomenon. Specifically, the energy consumption at the lower frequency configuration $f = 0.8$ is *higher* than that of the higher frequency configuration $f = 0.9$. Obviously, there is no benefit in using the frequency $f = 0.8$ as it consumes more energy while also using more CPU time (potentially affecting the feasibility of other tasks that may exist in the system), compared to the adjacent higher frequency level $f = 0.9$.

Clearly such *inefficient* frequency levels can be also removed from the table in a linear pass. Considering that the frequency levels below $max\{f_{ee}, u_j\}$ are not valid either, the trimming of the entire table can be achieved $O(K)$ time. After such a trimming, the energy consumption values in the table will be in decreasing order and the minimum energy configuration for the task will be at the last row of the table. While choosing this minimum energy configuration is ideal for the task, the feasibility and reliability requirements of other tasks may not allow the use of this frequency. This issue will be further analyzed in Section 5.

5 System-level Analysis

In this section, we address the system-level problem that involves the consideration of all the tasks in the system, each with potentially different reliability targets. We first give additional details necessary to formulate and manage reliability of periodic tasks each with multiple task instances (jobs).

5.1 Reliability Formulation for Periodic Tasks

We consider a reliability formulation similar to the one used in [21]. The reliability of a periodic task is defined as the probability of successfully executing all instances of that task during the hyperperiod, which is defined as the least common multiple of all the periods. Specifically, if the task τ_i has h_i instances in the hyperperiod, the *PoF* of the task is given by:

$$\phi_i = 1 - \Pi_{j=1}^{h_i}(1 - \phi_{i,j}) \tag{7}$$

where $\phi_{i,j}$ denotes the *PoF* of the job $T_{i,j}$.

The *system reliability* is the probability of executing all instances of all the tasks successfully. Therefore, it can be easily computed as the product of individual task reliabilities. The system *PoF* is then given by:

$$\phi_{syst} = 1 - \Pi_{i=1}^{N}(1 - \phi_i)$$

In reliability-oriented energy management problem, the task-level reliability targets may be given as part of the problem input. However, if only the system-level target reliability is given, we first need to compute the task level reliability target from the given system level reliability target ϕ'_{syst}. In this case, we can use the technique called the *Uniform Reliability Scaling* in [21]. This technique scales up or down all original task reliabilities by the same factor to achieve the new system-level target reliability. Specifically, assume that when all instances of a periodic task during the hyperperiod are executed at f_{max} and there are no additional replicas, the task level *Pof* is $\hat{\phi_i}$. Then, the task-level target $\phi_{i,target}$ values are determined such that $\phi'_{syst} = 1 - \Pi_{i=1}^{N}(1 - \phi_{i,target})$, and

$$\forall_i \frac{\phi_{i,target}}{\hat{\phi_i}} = \omega \tag{8}$$

Above, ω is called the *(uniform) PoF scaling factor*. Clearly, small (large) ω values correspond to higher (lower) reliability objectives.

5.2 Problem Definition

We now address the problem for a generalized setting, where there are multiple periodic tasks in the system. With multiple tasks, feasibility (deadline guarantees) becomes a major concern. Therefore, many tasks cannot be scheduled according to the most energy-efficient configuration from the EFR table since the total CPU time of all the replicas may exceed the time available on existing number of cores. Consequently, some tasks may have to be executed in a different 'configuration'. Given the EFR tables, determining the configuration (i.e., the degree of replication and frequency assignment) for each task such that the overall energy consumption is minimized while meeting the reliability target is a non-trivial problem.

Generalized Energy-Efficient Replication Problem (GEERP): Given a set of periodic tasks and task-level reliability targets, determine the number of replicas to execute and the frequency assignment for each replica such that the energy consumption is minimized, while ensuring a feasible partitioning such that the deadline constraints are met and no two replicas of the same task are assigned to the same core.

To present the general optimization problem formulation, we first introduce some additional notation. Let k_i be the number of replicas assigned to task τ_i and $E_i(f_i)$ be the energy consumption for each replica of τ_i running at frequency f_i during the hyperperiod. Γ_m denotes the set of all tasks for which a replica is assigned to core m. $\rho(i,j)$ represents the core where the j^{th} replica of τ_i is assigned. Then our problem is to find k_i and f_i values along with the replica-to-core allocation (partitioning) decisions $\{\rho(i,k_i)\}$, $i = 1,\ldots,N$, so as to:

$$\text{minimize} \quad \Sigma_{i=1}^{N} k_i \times E_i(f_i) \tag{9}$$

$$\text{subject to} \quad \forall_i \; f_i \; \in F \tag{10}$$

$$\forall_i \; k_i \; \leq M \tag{11}$$

$$\forall_m \; \Sigma_{\tau_i \in \Gamma_m} \frac{c_i}{P_i \times f_i} \; \leq \; 1 \tag{12}$$

$$\forall_i \; (\phi_i(f_i))^{k_i} \; \leq \; \phi_{i,target} \tag{13}$$

$$\forall_i \forall_{j \neq k} \; \rho(i,j) \; \neq \; \rho(i,k) \tag{14}$$

Above, the constraint (10) ensures a legitimate frequency assignment for every task and the constraint (11) enforces that the number of replicas does not exceed the available number of cores. The constraint (12) ensures that a feasible partitioning is obtained for all the cores, using the well-known schedulability condition with preemptive EDF [23]. The constraint set (13) represents the task level reliability targets. Finally, the constraint (14) ensures that no two replicas of a task are assigned to the same core.

The problem can be easily shown to be NP-hard in the strong sense: If we consider the special case of tasks with identical periods (deadlines), target reliabilities equal to the original reliability levels (requiring only one copy of each task), and a system without any DVS capability (where all tasks are executed at constant speed), GEERP reduces to the problem of packing variable-size items on M bins – this is the classical bin-packing problem, which is known to be NP-hard in the strong sense [32].

We consider a two-step solution for GEERP. In the first step, we construct the EFR tables for all tasks, separately. In the second step, using the tables, we search for a solution configuration that can be feasibly partitioned, while obtaining as much energy savings as possible. Due to the intractability of the problem, we resort to an efficient heuristic-based solution that still satisfies all the constraints of the problem.

5.3 **Algorithm** Energy-Efficient Replication (EER)

In this section, we present our solution. First, using Equation (7), the algorithm determines the job level reliability target for each periodic task, given the task-level reliability targets. Then as described in Section 4 the EFR tables are constructed. Assume that, the j^{th} configuration of τ_i is denoted by *RfConfig(i,j)*. In the rest of the paper, *f(i,j)*, *r(i,j)*, *E(i,j)* and *S(i,j)* denote respectively the frequency, the number, total energy consumption, and total CPU time of all replicas in *RfConfig(i,j)*. The specific quantities for *RfConfig(i,j)* can be obtained from the j^{th} row of the corresponding EFR table. From the tables, the algorithm first determines the minimum energy configuration for a given task.

The algorithm then tries to partition the workload for various configurations on M cores. We use the well-known *First-Fit-Decreasing (FFD)* heuristic [33] to allocate the replicas to the cores. However, we modified the FFD heuristic such that a different core is chosen for each replica of a task.

As the first attempt, the algorithm checks if it is possible to obtain a feasible partitioning where every task has its preferred (i.e., minimum-energy) replica-frequency configuration. If so, this is clearly the optimal solution for the entire problem. Otherwise, we check the other extreme, where every replica is forced to run at f_{max} to minimize the number and total CPU time of all the replicas. If there is no feasible solution for this case, the algorithm exits with an error report. Otherwise, the algorithm moves on to the next phase, which we call the *relaxation phase*.

In the relaxation phase, the algorithm starts with a feasible configuration where every replica runs at f_{max} and all tasks are marked as *eligible* for relaxation. Then in each step, based on the specific task selection heuristic (that will be discussed shortly), one eligible task is chosen and its frequency is reduced by one level according to its EFR table. If the resulting configuration is also feasible, the new configuration is committed to and the algorithm proceeds to the next step. Otherwise, the algorithm backtracks to the previous configuration and the chosen task is marked as *ineligible* for future relaxations. If a task reaches its minimum energy configuration level in the EFR table, it is also marked as ineligible for additional slowdown. The algorithm stops when there is no more eligible task for relaxation. Algorithm 1 shows the pseudo-code of the algorithm.

Several heuristics can be applied to choose the task for relaxation in every iteration. For this, we considered the following heuristics.

Largest-Energy-First (LEF): In this heuristic we choose the task that will provide the largest energy savings when relaxed to the next level in the EFR table. For task τ_i, let the current configuration be the row j in the EFR table. Then, the task with the largest

Algorithm 1 Algorithm Energy-Efficient Replication

```
Construct the EFR tables for all tasks
for i = 1 to N do
/* assume $j_i$ is the most energy-efficient frequency-level for $\tau_i$ in the EFR table */
    CurConfig[i] ← $j_i$;
end for
Partition the workload in CurConfig with modified FFD
if ( feasible(CurConfig) ) then
    return CurConfig and the partitioning $\rho$
    exit
end if
for i = 1 to N do
    CurConfig[i] ← 1;
    eligible[i] ← true;
end for
Partition the workload in CurConfig with modified FFD
if ( !feasible(CurConfig) ) then
    return error; /* No feasible solution exists */
    exit
end if
while ($\exists j$ eligible[j] ) do /* relaxation phase */
    Choose eligible task according to LEF, LPF or LUF
    /* $\tau_i$ is chosen for relaxation */
    CurConfig[i] ++;
    Partition the workload in CurConfig with modified FFD
    if( !feasible(CurConfig) ) then
        CurConfig[i] −−;
        eligible[i] ← false;
    end if
end while
return CurConfig and the partitioning $\rho$;
```

$$\Delta E \;=\; E(i,j) - E(i,j+1)$$

value is selected according to this heuristic.

Largest-Power-First (LPF): We choose the task that provides the largest energy savings per unit time for the additional CPU time required for the next level in the corresponding EFR table. Therefore, task τ_i is selected to maximize:

$$\frac{\Delta E}{\Delta S} \;=\; \frac{E(i,j) - E(i,j+1)}{S(i,j+1) - S(i,j)}$$

Largest Utilization First (LUF): This is a simple heuristic where the task with largest utilization value is chosen first.

We now analyse the complexity of the proposed solution. In the first phase, we construct the EFR tables for each task. As discussed in Section 4, each table can be constructed in $O(K)$ time. Therefore, the running time of phase 1 is $O(NK)$ in the worst case. In the relaxation phase, we can have at most NK relaxation steps and for obtaining the partitioning the cost is $O(NM)$ in the worst case. Therefore, the overall running time of the algorithm is $O(N^2MK)$. Note that, for most practical systems K and and M are small constants. In addition, the algorithm is executed only once as a pre-processing phase.

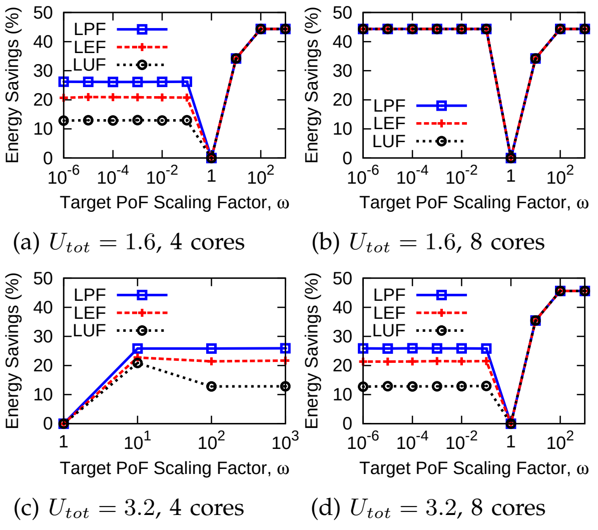


Fig. 5: Impact of target *Pof*

6 Performance Evaluation

In this section, we present our simulation results to evaluate the performance of our proposed scheme. We considered three different heuristics - LEF, LPF and LUF - for choosing the tasks for relaxation. As a *baseline* scheme, we considered the case where all replicas run at f_{max} and with the minimum number of replicas required to achieve the target reliability. We report the energy savings of our proposed scheme compared to the baseline scheme.

We constructed a discrete event simulator to evaluate the performance of our schemes. For each data point, we considered 1000 data sets with 20 tasks. The task utilizations are generated randomly using the UUnifast scheme [34]. Task periods are generated between 10 ms and 100 ms. The default number of speed steps in the system is 10 starting from 0.1 to 1.0 with steps of 0.1. The static power and the frequency-dependent power are set to 5% and 15% respectively of the maximum dynamic power consumption. We assumed an imperfect acceptance test with coverage factor 0.9 unless otherwise stated.

Impact of Target Reliability. We first consider the impact of target uniform *Pof* scaling factor ω on the system energy consumption. Figure 5 shows the energy savings for task sets with total utilization 1.6 and 3.2 running on 4- and 8-core systems. There is almost no energy gain for target $\omega = 1$. The reason is, when the

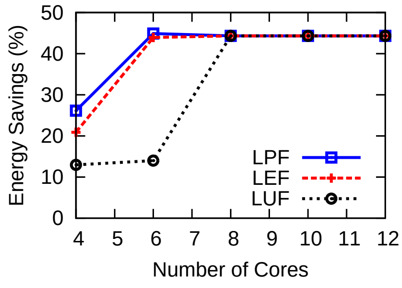


Fig. 6: Impact of number of cores

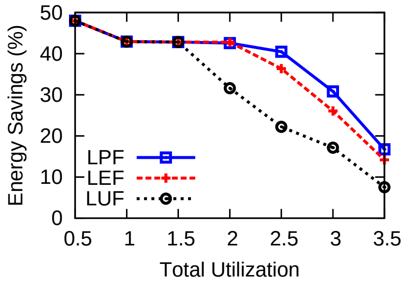


Fig. 7: Impact of system load

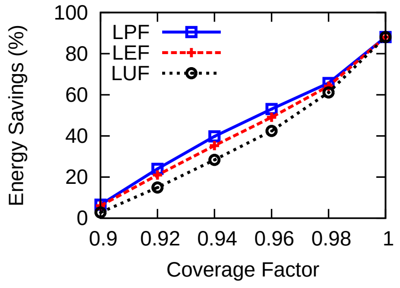


Fig. 8: Impact of coverage factor

target $\omega = 1$, the target reliability becomes equal to the reliability obtained trivially by the baseline scheme with exactly 1 replica for each task. We observe that for all settings, as we increase ω beyond 1, the energy savings increase as we can afford running the replicas at lower frequencies. When the frequency reaches the energy-efficient frequency, the savings reach a stable level. On the other hand, when $\omega < 1$, the baseline scheme uses more than one replicas running at f_{max} and LEF, LPF and LUF can execute replicas at lower frequencies. Even though LEF, LPF and LUF may use more replicas compared to the baseline scheme, they still save energy thanks to execution at lower frequencies.

In Figure 5b, the system utilization is very low compared to the available cores. As a result, the minimum energy configurations are feasible. Therefore, the energy savings for all schemes converge. In Figure 5c, on the other hand, the system utilization is very high. Therefore, it is not possible to find a feasible solution for target $\omega < 1$. For Figure 5a and 5d, the utilization is moderate considering the number of cores. When $\omega > 1$, all schemes were able to achieve the minimum energy. But as ω becomes smaller than one, there is no feasible solution for the most energy-efficient settings. However, some tasks can be feasibly relaxed to run at a lower frequency. Notice that, typically LPF performs slightly better than LEF. Further, both LEF and LUF outperform LUF. While LPF chooses the task that provides the highest energy savings per unit time, LEF chooses tasks based on energy savings only. Some tasks may provide higher energy savings at the cost of large increase of CPU time. As a result, LEF performs slightly worse than LPF.

Impact of the number of cores. Next, we evaluate the impact of the number of cores on the system performance. Figure 6 shows the energy consumption for a task set with total utilization 1.75 and target *Pof* scaling factor set to $\omega = 10^{-3}$. Observe that increasing the number of cores allows greater slowdown of replicas and hence typically provides greater energy savings. Initially LPF and LEF were able to improve energy savings significantly by making limited number of high energy-saving relaxation. LUF requires some additional cores before it can perform enough relaxation steps to achieve significant energy savings. Then with even additional cores, the energy savings for all schemes converge at a stable point, when choosing the most energy-efficient configurations become feasible for all tasks.

Impact of the System Load. Figure 7 represents the impact of total utilization U_{tot} on the energy savings. For this experiment, we set the number of cores in the system to 8 and the target ω is set to 10^{-3}. We vary the utilization from 0.5 to 3.5 and note the energy savings. We observe that, at very low utilization the energy savings is the highest, as we can find feasible partitioning for the minimum energy configurations. When the minimum energy configurations become infeasible ($U \geq 1.5$), the schemes start to differ. Then as we increase the utilization the energy savings keep decreasing. The energy savings for LPF and LEF drops at a slower rate as the relaxation steps provide higher energy savings. LUF has a sharper drop in energy savings as the relaxation steps it chooses does not always provide high energy savings. At very high utilization the energy savings approach to convergence as relaxation becomes difficult for all schemes.

Impact of the Coverage Factor. Figure 8 shows the impact of coverage factor on the energy consumption. For this experiment we vary the coverage factor from 0.9 to 1.0 while keeping the utilization and number of cores fixed at 1.25 and 16 respectively. The target PoF is set to $10^{-3} \times$ the PoF of a system with perfect acceptance test, executing single replica for all tasks at f_{max}. We record the energy savings compared to the baseline scheme with coverage factor 0.9.

We observe that the energy savings increases from 3% to 85% as the coverage factor increases. This is because the system can achieve the reliability target with less replicas or same number of replicas executing at a lower frequency with better coverage factors. At very low coverage factor, all schemes perform similarly as the number of replicas required for all schemes are high and there is very little slack for relaxation. As we increase the coverage factor the number of replica required decreases and schemes have some slack for relaxation which leads to different energy savings for different schemes. At coverage factor 1, all schemes require the lowest number of replicas and can achieve the optimum energy setting.

7 Dynamic Adaptations for Enhanced Energy Savings

Our proposed solution in Sections 4 and 5 was based on static analysis of workload characteristics and reliability requirement. In this section, we present dynamic adaptation techniques to further enhance energy savings.

First, we note that it is sufficient to complete only one replica successfully. So, if one replica completes and no fault is detected, we can cancel other replicas of that task immediately to avoid the energy consumption of the replicas. Consequently, it is beneficial to delay all the replicas except for the one that completes first, as this increases the chance of cancelation for the additional replicas. In other words, by minimizing the overlapped execution among the replicas of the same task, entire or partial execution of the additional replicas can be canceled in many scenarios. Second, such replica delaying strategy becomes even more attractive as real-time tasks often complete earlier than the estimated worst-case execution times: the dynamic slack generated by the early completions can be utilized for reducing the execution overlaps of replicas of other tasks.

In order to implement this replica delaying approach, our strategy will consist in executing one replica as usual and delay the remaining replicas without compromising the deadline constraints. To this aim, we introduce the concept of *primary* and *secondary* replicas. The primary replica is allowed to execute as soon as it becomes eligible for execution. The other (secondary) replicas are delayed whenever possible. Note that, the selection of the primary replica is done at run time; in fact, on a specific processor there can be a mix of primary and secondary replicas from different tasks. Specifically, when a replica of a job T_i^j is about to be dispatched, the scheduler marks it as primary if it is the first replica of T_i^j to be dispatched on any processor. The primary replica is then dispatched immediately at the speed determined according to the scheme presented in Section 5. Other replicas of the job T_i^j become secondary and are delayed according to the strategy explained below.

The algorithm introduces an idle period before dispatching a secondary replica. The length of the idle period is determined dynamically. A simple approach for delaying is to execute the secondary replicas at the maximum frequency by allocating only the minimum execution time needed for their completion. Then the difference between the execution times at the scaled and maximum frequencies can be used as the *delay duration*. Even tough the secondary execution at the maximum frequency involves an energy cost, recall that in many scenarios the secondary's execution will be partially or entirely canceled thanks to the delaying strategy.

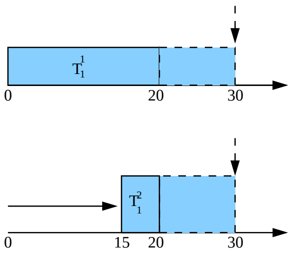


Fig. 9: Slow down on CPU1 (top) and delayed execution on CPU2 (bottom) for a single task

***Example** 1:* Let us consider a single task with period 30 and worst case execution time 15 under f_{max}. The task has two replicas. At the top of Figure 9 we first show the slowed down execution scenario (at $f = 0.5$) on the first CPU. On CPU 2, we can delay the execution of the secondary replica until $t = 15$ and then execute it at f_{max} as shown at the bottom of Figure 9. Assume the actual execution time of this job is 2/3 of the worst-case. Then T_1^1 will complete at $t = 20$. At that time we can cancel the remaining portion of the secondary copy.

While simple, we call this strategy *naive delaying*, because i.) for preemptive periodic tasks, determining the exact future dispatch times of replicas becomes a nontrivial problem, and ii.) it is possible to improve the delay durations efficiently by using additional data structures. To achieve this goal, we first introduce the concept of *canonical schedule* (CS). A CS for a task set is the expected schedule when all jobs present their worst case workloads under the scaled speeds given by the static solution. The main idea is to determine, at the dispatch time t of a secondary replica R, the total available CPU time for R such that it will complete no later than its completion time in the canonical schedule. This total available time is denoted by γ. Then if the remaining execution time for the secondary replica is c under f_{max}, we can delay the secondary at time t by $\gamma - c$ time units without missing the deadline. We call this technique *adaptive delaying* strategy. We present the following example to illustrate the concept.

***Example** 2:* Let us consider a task set with two periodic tasks with the parameters $c_1 = 7.5$, $P_1 = 25$, $c_2 = 5$, $P_2 = 25$. Assume both tasks can be slowed down to speed 0.5 without compromising the reliability and deadline requirements. Figure 10 shows the canonical schedule for the task set on CPU 1. Suppose the actual execution times at speed 0.5 are 10 and 7.5 respectively on CPU 1 as shown in Figure 11. In Figure 12, we first show the application of the naive delaying approach on CPU 2. T_1^2 is delayed for $7.5(= 15 - 7.5)$ units of time. As T_1^1 completes at time 10, T_1^2 can be cancelled. The primary T_2^1 is then dispatched immediately. However, the secondary T_2^2

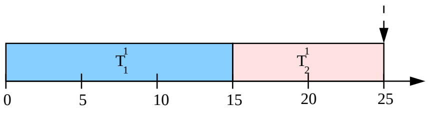


Fig. 10: Canonical Schedule on CPU 1

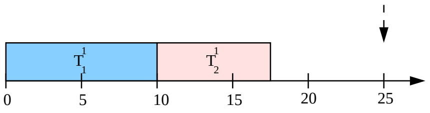


Fig. 11: Actual Execution Scenario on CPU 1

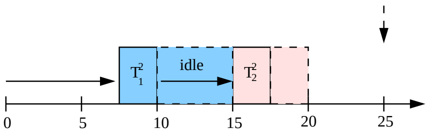


Fig. 12: Naive Delaying Strategy on CPU 2

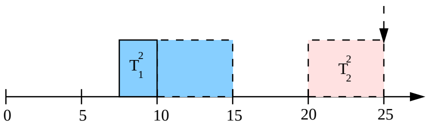


Fig. 13: Adaptive Delaying on CPU 2

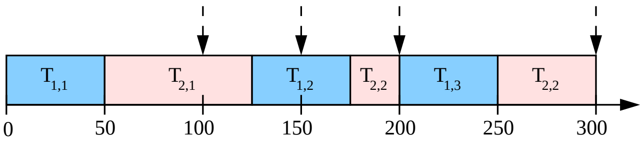


Fig. 14: Canonical Schedule

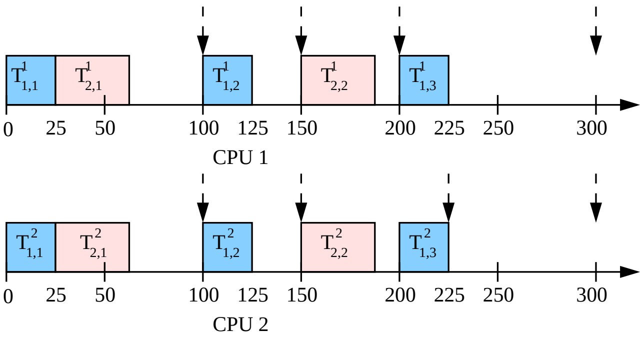


Fig. 15: Parallel Execution based on Static Schedule

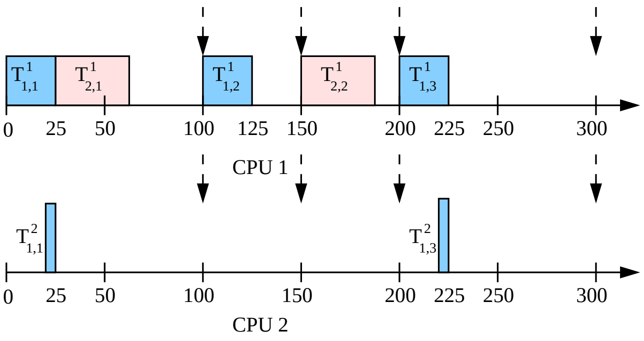


Fig. 16: Adaptive Delaying Scheme

is not dispatched immediately and using the naive delaying strategy it is delayed by 5 units of time and dispatched at time 15. In that scenario the execution of T_2^1 and T_2^2 will overlap between time 15 and 17.5. However, with our adaptive scheduling policy (Fig. 13), consulting the canonical schedule, we can take advantage of the early cancellation of T_1^2 and delay T_2^2 until $t = 20$ which is the difference between the completion time in the canonical schedule and its remaining workload ($25 - 5$). Thus we can avoid the execution of T_2^2 entirely.

While the idea of using the canonical schedule to compute the safe delaying durations is attractive, generating the canonical schedule in advance for preemptive periodic tasks with arbitrarily large hyperperiod is not a computationally acceptable solution. Noting that what we really need is the *difference* between the completion times in the worst-case canonical schedule and the remaining execution time in the current schedule, we can use a simple data structure originally proposed in [16] to implement dynamic reclaiming algorithms. This data structure that we call *Canonical Execution Queue* (CEQ) is a priority queue, where jobs are sorted in ascending order of their absolute deadline. Every processor has its own CEQ. The CEQ follow two rules.

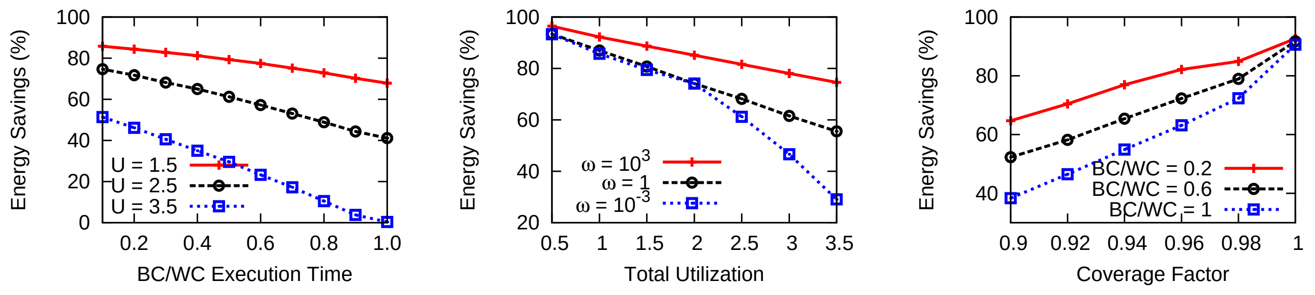
1) At the arrival of a replica $T_{i,j}^k$ in the processor, add a corresponding job to the CEQ with execution time c_i/f_i, where f_i is the speed assigned to τ_i according to the static scheme.
2) At every time unit, the execution time of the job at the head of the CEQ is decreased by one and when it reaches zero, the job is removed from the queue.

During execution, CEQ effectively gives a *snapshot* of the ready queue of the corresponding processor in the canonical schedule, with remaining workload of every ready job at that time. As shown in [16], it is sufficient to update CEQ only at task arrival and completion times and the complexity of update is linear in the number of tasks.

At dispatch time of a secondary replica job, the amount of safe delay may be computed by inspecting CEQ. Specifically, all jobs in the CEQ that are ahead of the current job have higher priority and hence they must have already been completed. So the sum of execution time for all those jobs and the current job according in CEQ can entirely be used by the current job without missing any deadlines [16]. This total available CPU time is denoted by γ. Specifically, for a secondary replica $T_{i,j}^k$, we can compute γ as,

$$\gamma = \Sigma_{\{T_{x,y}^z \in EQ \ \& \ d_x < d_i\}} rem(T_{x,y}^k) + rem(T_{i,j}^k) \quad (15)$$

where $rem(T_{i,j}^k)$ is the remaining execution of $T_{i,j}^k$ in the CEQ. Then we can set the safe delay duration for $T_{i,j}^k$ to $(\gamma - cr_i)$, where cr_i is the remaining worst-case execution time of $T_{i,j}^k$ under f_{max} to ensure feasibility.



(a) Impact of Workload Variability (b) Impact of Utilization (c) Impact of Coverage Factor

Fig. 17: Impact of Dynamic Adaptations

Example 3: Consider two tasks with parameters $p_1 = 100, p_2 = 150, c_1 = 30, c_2 = 45$. Assume it is sufficient to have two replicas for each task running at frequency 0.6 to achieve the reliability target. Figure 14 shows the canonical schedule for the task set. If the actual execution time of each job is 50% of the worst case estimate, we obtain the schedule generated by the static solution (Figure 15). Notice that the replicas are executed in parallel, each CPU is busy for 60% of time, the energy consumption is significant due to overlapped execution of the replicas. Figure 16 shows the execution with our proposed adaptive delaying technique. At time 0, $T^1_{1,1}$ in CPU 1 is marked as primary and dispatched immediately. $T^2_{1,1}$ on CPU 2 therefore becomes secondary. According to CEQ γ = 50 and we can delay it until 20 (= 50 - 30), when it starts running at f_{max}. At time 25, $T^1_{1,1}$ completes in CPU 1 and the corresponding secondary in CPU 2 is canceled. At time t = 25, CPU 1 dispatches $T^1_{2,1}$ as primary. From the CEQ, we get $\gamma = 100$, while the worst-case execution time is 45. So the secondary can be delayed for (100 - 45) = 55 units of time i.e. till 80. As $T^1_{2,1}$ completes at time 62.5 at CPU 1, the secondary in CPU 2 can be canceled entirely. In this way the secondaries $T^2_{1,2}$, $T^2_{2,2}$ are canceled entirely, while the secondary $T^2_{1,3}$ executes only briefly. Using this strategy CPU 2 is used for only 10 time units. Using the power parameters from Section 6, the dynamic energy consumption can be found to decrease by 35% compared to the static solution.

8 Evaluation of Dynamic Adaptation Techniques

In this section we present our experimental results concerning the performance of the dynamic adaptation techniques. The main simulation settings are essentially the same as those in Section 6. In addition, to model the workload variability, we determined the actual execution time of each job according to a probability distribution. The actual execution time is constrained between a best-case (BC) and a worst-case (WC) execution time and is determined according to the normal distribution. The mean and the standard deviation of the normal distribution are set to $(BC + WC)/2$ and $(WC - BC)/6$, respectively, as in [16]; this guarantees that the actual execution time falls in the [BC, WC] range with the 99.7% probability. Moreover, we varied the BC/WC ratio to model different workload variability settings.

First we study the impact of the workload variability in Figure 17a. We change the BC/WC ratio from 0.1 to 1 while keeping the number of cores and ω fixed at 8 and 10^{-3} respectively. We report the normalized energy savings compared to the static scheme with utilization is 3.5 and BC/WC = 1. The higher BC/WC ratio, the higher workload variability, and the lower energy savings compared to the static scheme, for a fixed system load (utilization). This is because as BC/WC increases, jobs have larger execution time and there is a higher chance of overlap between the primary and the secondary replicas. For utilization = 1.5, at very low BC/WC ratio, dynamic adaptation techniques provide over 82% energy savings. At BC/WC = 1, it provides up to 75% energy savings. Note that even when BC/WC = 1, the dynamic delaying technique is able to postpone the execution of the secondaries and save energy, by exploiting the available static slack in the system. With increasing utilization, however, static slack disappears, and the energy savings solely depend on the BC/WC ratio.

Next we study the impact of system load in Figure 17b. The settings for this experiment are similar to those in Figure 7. For the BC/WC ratio = 0.5, the energy savings are reported with respect to the same static scheme in Figure 17a. We report the energy savings for three different ω values. As a general trend the energy savings decrease as we increase the utilization. The energy savings are typically higher for lower ω values as we can afford lower reliability. Notice that at lower utilization ($U < 2$), the energy savings are similar for both $\omega = 1$ and $= 10^{-3}$. This is because, even though the number of replicas are higher for $\omega = 10^{-3}$, larger slack allows the secondary to be delayed significantly and thus avoid execution. But as the utilization increases, we can no longer avoid executing the secondary and the energy savings drop at a higher rate for $\omega = 10^{-3}$.

Finally, we consider the impact of coverage factor in Figure 17c. For three different BC/WC values, we increase the coverage factor from 0.9 to 1.0. The rest of the settings are identical to those in Figure 8. We report the energy savings compared to the static scheme with BC/WC = 1 and coverage factor 0.9. As a general trend, as we increase the coverage factor the energy savings increases; this is because at higher coverage values it is sufficient to deploy less replica. For a fixed coverage factor value, the energy savings increase as we decrease BC/WC value. For BC/WC = 1, the energy savings increase from 38% to 90% as we increase the coverage factor from 0.9 to 1.

9 Related Work

Most fault tolerance techniques on multiprocessor/multicore real-time systems rely on scheduling multiple versions of a task, called *replicas*, on different processors. The *overloading* technique has been proposed to improve the schedulability of the system. The idea is to schedule multiple tasks/versions on overlapping time intervals on the same processor. The key is to ensure that at most one of them will need to be executed at run-time and at least one version of every task will complete before deadline even in the presence of faults. One such approach is called the *primary-backup* (PB) model, where each task has a primary and backup copy. [35], [36] considered a simple PB model, where a backup can be executed at the same time with any other backup on any of the processors except the one where the corresponding primary task is running. This provides greater flexibility, but it can tolerate only one fault at a time. Manimaran et al. statically grouped processors, allowing backups to overlap only when the corresponding primary tasks run on the same group [37]. This allows the system to tolerate one fault per group. [38], [39] used dynamic grouping to avoid the limitation of static grouping. Al-Omari et al. proposed a primary-backup overloading technique, where primary tasks can overlap with backups of other tasks [40].

With the advance of low-power computing area, researchers started to consider reliability / fault tolerance issues in the context of power awareness. A set of techniques, called the *Reliability-Aware Power Management (RAPM)* [19], [20], exploit *time redundancy* available in the system for both energy and reliability management. These works consider the problem of preserving the system's *original reliability*, which is the reliability of the system executing tasks without any slowdown. The scheme in [20] resorts to *backward recovery* approach and assigns a recovery task for every task that has been slowed down, while the technique in [19] uses recovery blocks shared by all the scaled tasks. The work in [41] exploits the probability distribution of the task execution times to improve the energy savings in the RAPM framework. The *reliability-oriented energy management* framework has been proposed for periodic real-time tasks running on a single-core system [21]. The main objective is to achieve *arbitrary reliability levels*, in terms of tolerance to transient faults, with minimum energy consumption. The solution still focuses on single-core systems and hence resorts to a limited number of shared recovery jobs.

10 Conclusions

In this paper, we considered the reliability-oriented energy-management problem for preemptive periodic real-time applications running on a multi-core system. We showed how replication can be used to achieve the given task-level reliability targets that are expressed in terms of tolerance to transient faults. We presented techniques to determine the degree of replication and the frequency assignment for each task while minimizing overall energy. Although the problem is intractable in the general case our efficient heuristics are shown to satisfy the given reliability targets with considerable energy savings through simulations. We then proposed dynamic replica delaying techniques to further reduce energy consumption.

As more and more real-time embedded applications many of which exhibiting reliability requirements of varying degrees are implemented upon multi-core platforms, tackling both the energy-efficiency and reliability challenges simultaneously becomes critical. Our work illustrates the potential of the joint use of replication and DVS on such platforms while guaranteeing the timing constraints. We believe that as more cores becomes available on such systems, replication will be an even more attractive option; however, to optimize energy consumption the number of replicas and frequency levels should be determined carefully. A very interesting future research direction is the re-consideration of the same problem on emerging asymmetric multicore systems.

References

[1] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 329–342, 2012.

[2] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *Proceedings of the IEEE International Conference onComputer Design (ICCD)*, 2011.

[3] R. Melhem, D. Mossé, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, vol. 53, pp. 217–231, 2004.

[4] O. S. Unsal, I. Koren, and C. M. Krishna, "Towards energy-aware software-based fault tolerance in real-time systems," in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design*, 2002.

[5] Y. Zhang and K. Chakrabarty, "Dynamic adaptation for fault tolerance and power management in embedded real-time systems," *ACM Transactions on Embedded Computer Systems*, vol. 3, pp. 336–360, May 2004.

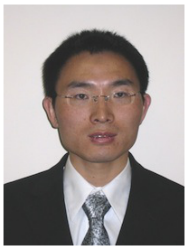
[6] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[7] D. Pradhan, *Fault Tolerant Computer System Design*. Prentice Hall, 1996.

[8] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Transactions on Computers*, vol. 31, pp. 658–671, 1982.
[9] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems*, vol. 4, pp. 214–237, 1986.
[10] V. Ferlet-Cavrois, L. W. Massengill, and P. Gouker, "Single event transients in digital cmos: a review," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1767–1790, 2013.
[11] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 243–247.
[12] M. Ebrahimi *et al.*, "Comprehensive analysis of alpha and neutron particle-induced soft errors in an embedded processor at nanoscales," in *Proc. of the IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.
[13] N. Seifert *et al.*, "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2666–2673, 2012.
[14] S. Borkar, "Extreme energy efficiency by near threshold voltage operation," in *Near Threshold Computing*. Springer, 2016.
[15] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Mobile Computing*, 1996.
[16] H. Aydin and R. Melhem, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584 – 600, 2004.
[17] D. Ernst *et al.*, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 6, pp. 10–20, 2004.
[18] D. Zhu, R. Melhem, and Mossé, "The effects of energy management on reliability in real-time embedded systems," in *Proc. of the IEEE Conf. on Computer Aided Design*, 2004.
[19] B. Zhao, H. Aydin, and D. Zhu, "Enhanced reliability-aware power management through shared recovery technique," in *Proc. of the IEEE Conf. on Computer Aided Design*, 2009.
[20] D. Zhu and H. Aydin, "Reliability-aware energy management for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1382 – 1397, 2009.
[21] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012.
[22] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms," in *Proceedings of the IEEE International Green Computing Conference (IGCC)*, 2013.
[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[24] R. Sridharan and R. Mahapatra, "Reliability aware power management for dual-processor real-time embedded systems," in *Proc. of the IEEE Design Automation Conference*, 2010.
[25] X. Fan, C. Ellis, and A. Lebeck, "The synergy between power-aware memory systems and processor voltage scaling," in *Power - Aware Computer Systems*. Springer, 2005, pp. 151–166.
[26] R. Jejurikar and R. Gupta, "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," in *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, 2004.
[27] J. Zhuo and C. Chakrabarti, "System-level energy-efficient dynamic task scheduling," in *Proceedings of the IEEE annual Design Automation Conference*, 2005.
[28] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proc. of the IEEE Conf. on Design, Automation and Test in Europe*, 2003.
[29] Q. Han, L. Niu, G. Quan, S. Ren, and S. Ren, "Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems," *Journal of Real-Time Systems*, vol. 50, no. 5-6, pp. 592–619, 2014.
[30] N. Dutt *et al.*, "Multi-layer memory resiliency," in *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2014.
[31] Intel, *Intel Pentium M Processor Datasheet*, 2004. [Online]. Available: http://download.intel.com/support/processors/mobile/pm/sb/25261203.pdf
[32] M. R. Garey and D. S. Johnson, *Computers and Intractability*. Freeman, 1979.
[33] D. S. Johnson *et al.*, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.
[34] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Journal of Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.
[35] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerant scheduling on a hard real-time multiprocessor system," in *Proceedings of the IEEE International Parallel Processing Symposium*, 1994.
[36] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272–284, 1997.
[37] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1137–1152, 1998.
[38] R. Al-Omari, G. Manimaran, and A. K. Somani, "An efficient backup-overloading for fault-tolerant scheduling of real-time tasks," in *Proceedings of the IEEE Workshop on Parallel and Distributed Processing*, 2000.
[39] K. Yu and I. Koren, "Reliability enhancement of real-time multiprocessor systems through dynamic reconfiguration," in *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994.
[40] R. Al-Omari, A. K. Somani, and G. Manimaran, "Efficient overloading techniques for primary-backup scheduling in real-time systems," *Journal of Parallel and Distributed Computing*, vol. 64, no. 5, pp. 629–648, 2004.
[41] D. Zhu, H. Aydin, and J.-J. Chen, "Optimistic reliability aware energy management for real-time tasks with probabilistic execution times," in *Proceedings of the IEEE Real-Time Systems Symposium*, 2008.

Mohammad A. Haque Mohammad A. Haque received the M.Sc. degree in Computer Science from George Mason University, Fairfax, Virginia, USA, in 2010. He is currently a PhD candidate in the same department. His area of research includes low-power computing, real-time systems, and operating systems.

Hakan Aydin Hakan Aydin received the Ph.D. degree in computer science from the University of Pittsburgh. He is currently an Associate Professor in the Computer Science Department at George Mason University. He has served on the program committees of several conferences and workshops. He served as the Technical Program Committee Chair of IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'11). He was the General Chair of IEEE RTAS in 2012. He received NSF CAREER Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance.



Dakai Zhu Dakai Zhu received the PhD degree in Computer Science from University of Pittsburgh in 2004. He is currently an Associate Professor in the Department of Computer Science at the University of Texas at San Antonio. His research interests include real-time systems, power-aware computing and fault-tolerant systems. He has served on program committees (PCs) for several major conferences (e.g., RTSS and RTAS). He received NSF CAREER Award in 2010.