Energy-Efficient Fault Tolerance for Real-Time Tasks with Precedence Constraints on Heterogeneous Multicore Systems

Abhishek Roy, Hakan Aydin
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
Email: {aroy6, aydin}@gmu.edu

Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
Email: Dakai.Zhu@utsa.edu

***Abstract*—Heterogeneous multicore systems have been recently received much attention due to their power efficiency and ability to handle different workloads. In this paper, we consider real-time tasks with precedence constraints and fault tolerance requirements, and investigate how they can be implemented on heterogeneous dual-core systems in energy-aware fashion. Our framework is able to tolerate one transient fault per task, and one permanent processing core fault simultaneously. We develop a number of task partitioning, ordering, and frequency assignment techniques for energy efficiency. Our experimental results indicate that the proposed techniques significantly reduce energy consumption while satisfying the fault tolerance requirements.**

I. Introduction

Real-time embedded systems are widely used in applications requiring predictability in terms of timing performance such as those in autonomous vehicles, industrial control, medical support systems, and avionics. Energy management on such systems is an important objective that has been widely explored. For instance, with *Dynamic Voltage Scaling (DVFS)* technique, the tradeoffs between processor speed and power dissipation are exploited. Similarly, *Dynamic Power Management (DPM)* enables idle system components to transition to low-power sleep states to save power. As a recent development, *heterogeneous (asymmetric)* multicore systems have been increasingly investigated and deployed in settings where power and performance are equally important. In those systems, processing units with different power/performance reside on the same chip. This enables the system to activate the combination of cores that are most suitable for the current workload. As a well-known example, ARM's big.LITTLE systems combine "big" out-of-order and high-performance cores with "LITTLE" in-order and power-efficient cores. In most cases, heterogeneous cores have the same instruction-set-architecture (ISA), thus the same binary executable can run on any core but with different execution time/power dissipation characteristics. Numerous recent studies investigated power management issues on heterogeneous multicore systems [1]–[5].

Safety-critical real-time systems are designed to be fault-tolerant. This is generally accomplished by incorporating extra (redundant) hardware or software components [6], [7]. The fault detection and recovery techniques are well-studied [8]. Those techniques typically depend on the type of the fault affecting the computer system. Most of the run-time faults are instantaneous and they result from temporary environmental factors such as electromagnetic interference and cosmic rays. These are *transient* faults – they cause an error in the output of a single task. Typically re-execution of the task (or a recovery task) gives the correct result [8], [9]. Moreover, state-of-the-art power management techniques (e.g., near-threshold voltage operation) tend to increase the susceptibility of CMOS circuits to transient faults [10]. Another important fault category is *permanent* faults – this is when a processor ceases to function, more commonly because of manufacturing defects or wear-out effects. When a permanent fault occurs, the system can continue execution only with additional units (e.g., spare cores) on which the application can resume execution [8].

The joint problem of minimizing energy consumption while providing fault tolerance guarantees has recently received attention [11]–[14]. However these studies consider either uniprocessor systems or homogeneous multiprocessors. Two recent studies considered heterogeneous dual-core systems and proposed energy-aware fault tolerance solutions using standby sparing [15] and primary/backup techniques [16]. However those works consider independent real-time tasks and hence they are not applicable in cases when tasks are dependent.

In this paper, we propose a fault-tolerant and energy-efficient framework for periodic real-time tasks executing on heterogeneous dual-core systems. In practice, there are often dependency relationships among real-time tasks [17]–[19]; so we model the dependencies using precedence constraints on directed acyclic graphs (DAGs) [20], [21]. Our solution has two components: 1.) A main schedule where tasks are executed at low processing frequency (speed) levels using DVFS to save energy as long as faults are not encountered, and, 2.) The *contingency schedule* according to which recovery tasks are

978-1-7281-5416-9/19 $31.00

executed upon the detection of transient and/or permanent faults. Task partitioning and speed assignment algorithms are designed by respecting the precedence constraints and allowing the necessary recovery times in the contingency schedule, while minimizing energy consumption in most common (fault-free) execution scenarios. Our framework has the distinct feature of *tolerating a separate transient fault for each real-time task, as well as the permanent fault of any single core* – in fact, the system can recover from the permanent fault of a processing core, even after multiple tasks have incurred transient faults and have been re-executed thanks to the hardware and time redundancy offered by the contingency schedule. All the components of the framework guarantee the precedence and timing constraints. The experimental evaluation suggests that our proposed schemes can offer non-trivial energy gains over a broad parameter spectrum. To the best of our knowledge, this is the first study on the energy-aware fault-tolerant operation of real-time tasks with precedence constraints, executing on heterogeneous multicore systems.

The rest of the paper is organized as follows: Section II presents our system model and assumptions. Section III describes our proposed framework, including fault recovery mode, contingency schedule, task partitioning and speed assignmen solutions. Section IV presents our experimental evaluation and Section V concludes the paper.

II. System Model and Assumptions

A. Platform and Application model

We consider a heterogenous dual-core system with a high-performance (big) core and a low-power (little) core. Throughout the paper, we denote the high-performance and low-power cores by HP and LP, respectively. The target application consists of n dependent real-time tasks $\{\tau_1, ..., \tau_n\}$. We assume the *frame-based execution* model [22], [23] in which all tasks will be executed in a *frame*, which is invoked periodically with a deadline, D. D is also the period of the frame and the common deadline for all the tasks in the frame. The precedence constraints among tasks are represented by a directed acyclic graph (DAG), where an edge $\tau_j \rightarrow \tau_k$ indicates that τ_k can only start to execute when τ_j completes successfully, usually due to the input-output dependencies. Since our platform is a shared memory multicore system, communication time to transfer data between tasks is considered negligible.

Each of the processing core is equipped with the *Dynamic Voltage and Frequency Scaling (DVFS)* feature that allows changing the frequency (processing speed) at run-time. A task τ_i that requires C_i number of cycles on a given core may take up to $W_i = C_i/f$ units of execution time on that core, if executed at the frequency level f. Due to the architectural differences, a task's required number of cycles, and hence execution time, can be different on the HP and LP cores. Therefore, we use superscripts HP and LP to denote the variables on the HP or the LP core (e.g., C_i^{LP}, W_i^{LP}, C_i^{HP}, W_i^{HP}). The maximum frequency levels supported by the HP and LP cores are denoted by f_{max}^{HP} and f_{max}^{LP}, respectively. We assume $f_{max}^{HP} = 1.0$, and normalize all other frequency values with respect to that value. We define the *nominal utilization* of a task τ_i as (C_i^{HP}/D).

Our framework, as discussed in Section III-A, includes copies of each task τ_i to be executed in a potential recovery mode, upon the detection of run-time faults. Thus, to distinguish each task τ_i from its recovery mode copies, we use the term *primary task* throughout the paper.

B. Power Model

The power consumption characteristics of the HP and LP cores differ by design. For any processing core, the dynamic power consumption of an executing task τ_i is modeled as, $P_i(f) = a_i f^3 + \alpha_i$, where a_i denotes the switching capacitance, α_i denotes the frequency-independent power consumption, and f is the processing frequency of the task adjustable through the DVFS feature. Due to the asymmetry of the cores, these parameters are different for each core and again we use superscripts HP and LP to denote the core-specific power parameters (e.g., P_i^{HP}, α_i^{HP}).

Each core executes tasks in the *active* state, dissipating power as determined by the characteristics of the current task and processing frequency. The *Dynamic Power Management (DPM)* feature allows a given core to switch to a *low-power (idle)* mode when it is not actively executing tasks. The low-power (idle) power consumption of the high-performance and low-power cores are denoted by P_{idle}^{HP} and P_{idle}^{LP}, respectively. We assume those figures include the static power consumption of the corresponding core as well. The energy consumption during a time interval is given by the aggregate power consumption during the same interval.

Existing research indicates that scaling down the frequency below a certain threshold is no longer effective for saving energy, due to the impact of the frequency-independent power component [23]. This threshold frequency, known as the *energy-efficient frequency (f_{ee})* can be derived through analytical techniques [23], and we never reduce the processing frequency below f_{ee} on a given core.

C. Fault Model

Our framework targets providing high assurance to safety-critical real-time tasks in energy-aware manner. Hence, we aim to tolerate transient faults (that affect individual tasks) as well as permanent faults (that lead to the unavailability of a whole processing core). Specifically, in our framework, we tolerate within each execution frame:

- A transient fault per each (primary) task, and,
- A permanent fault of any of the processing cores

When a primary task τ_i completes, the *acceptance* (or, *sanity*) tests [8] are performed to check the existence of errors induced by *transient* faults which may have affected τ_i. If no fault is detected, the result of the task is committed to, and the system continues with the execution of subsequent tasks. Otherwise, upon the detection of a transient or permanent fault, the system switches to the *recovery mode* and executes all the incomplete tasks at the maximum speed according to a *contingency schedule*, whose details are provided in Section

III-A. In essence, the contingency schedule allows re-executing faulty tasks and also tolerating additional transient faults that may affect other tasks. Moreover, it offers the capability to recover from a permanent processing core fault that may occur after any number of transient faults. Note that, should a permanent fault occur, the system loses the capability of tolerating any more (transient or permanent) faults until the faulty core is repaired or replaced.

III. Proposed Framework

Our proposed framework has two complementary phases for task mapping and scheduling. First, since the faults are rare events, there is a need to determine the default schedule according to which tasks are executed in each frame as long as faults are not enocuntered. We call this default schedule the *main schedule*. An important objective for the main schedule is to minimize energy consumption in most common (i.e., fault-free) frame execution scenarios. Consequently, computing the main schedule involves:

- Allocating the primary tasks on the HP and LP cores and determining their execution order (*task partitioning and ordering*), and,
- Determining the voltage/frequency levels for individual primary tasks (*speed assignment*),

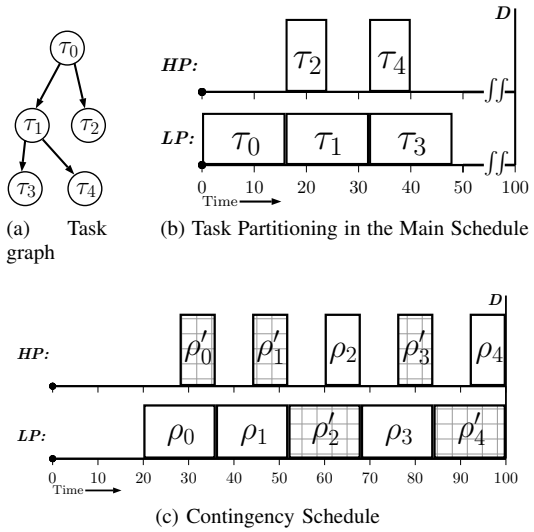
to minimize energy consumption while meeting the precedence and timing constraints. However, while generating the main schedule, it is necessary to take provisions to tolerate transient and permanent faults in a timely manner. In what follows, we first discuss the *recovery mode* of execution and the derivation of the *contingency schedule* according to which the tasks are (re-)executed in a frame upon the detection of a fault. Then we elaborate on the two components of the main schedule computation (task partitioning and speed assignment).

A. Recovery Mode and Contingency Schedule

When a transient fault is detected at the end of task τ_i, the task must be re-executed, in addition to other tasks that are yet to be executed before the end of the frame (deadline), while satisfying the precedence constraints. In our framework, upon the detection of a fault, the system switches to the *recovery mode* and executes the (incomplete) tasks according to a pre-computed *contingency schedule*.

As long as both cores are functional, the system must preserve its capability to recover from the permanent fault of any of the cores. Since this invariant must hold even during the recovery mode which may have been triggered by a transient fault, in the contingency schedule *it is necessary to schedule two distinct copies of tasks allocated on two different cores* – with only one copy of a task τ_j allocated on a specific core, it would not be possible to re-execute τ_j, should that core experience a permanent fault.

Consequently, in the contingency schedule, associated with each task τ_i there are two contingency tasks ρ_i and ρ'_i with the exact same timing parameters as those of τ_i. We make sure that ρ_i and ρ'_i are allocated to different cores, as a precaution against a permanent fault. If the primary task τ_i completes



(a) Task graph

(b) Task Partitioning in the Main Schedule

(c) Contingency Schedule

Fig. 1: An Example Contingency Schedule

successfully, both contingency tasks ρ_i and ρ'_i are cancelled; hence they do not incur any time or energy overhead in most common fault-free execution scenarios. Thus, the contingency schedule consists of a sequence of paired contingency tasks executing at the maximum speed of their respective cores, if the system enters the recovery mode for that frame. Moreover, their executions are delayed as much as possible to minimize overlaps with the tasks in the main schedule (see Fig. 1c as an example).

Specifically, once the primary tasks are mapped to the HP and LP cores, the contingency schedule is determined according to the following rules:

1. A topological order of tasks satisfying the precedence constraints implied by DAG is obtained. In addition, this task sequence complies with the execution order of tasks observed on each core in the main schedule.
2. Two contingency copies of each task (ρ_i and ρ'_i) are placed in parallel on the HP and LP cores, according to the order derived in Step 1. ρ_i is placed on the same core as its primary copy in the main schedule, whereas ρ'_i is placed on the alternative core.

The contingency tasks are shifted towards the deadline as much as possible such that each primary copy can get a larger execution-window and run at a slower speed by applying DVFS in order to save energy (Fig. 1c). The activation times of the contingency tasks are computed such that both copies complete at the same time with their respective worst-case execution time on the HP and LP cores. These activation times represent the latest start time of a contingency copy such that any faulty task and all of its subsequent tasks can be executed before the frame deadline if needed.

As an example, consider a set of tasks given by the DAG shown in Fig. 1a and deadline $D = 100ms$. For each task τ_i, we have $C_i^{HP} = 8$ and $C_i^{LP} = 12$, expressed in millions of cycles. Assuming $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.75$ GHz, therefore, each task takes 16 ms and 8 ms when executed

on the LP and HP cores at maximum speed, respectively. Considering the allocation of the primary tasks shown in Fig. 1b, we show an example contingency schedule in Fig. 1c. First the topological order τ_0, τ_1, τ_2, τ_3, τ_4 is obtained and the corresponding contingency tasks are maximally pushed towards the deadline.

In summary, the system operates according to the following rules at run-time:

R1. As long as there are no faults, the system continues with the main schedule. When a primary task τ_i completes successfully, the contingency schedule is updated to reflect the cancellation of the contingency tasks ρ_i and ρ_i'.

R2. If a fault is detected at the end of τ_i, the system immediately transitions to the recovery mode, and it starts executing the contingency copies of all incomplete tasks at the maximum speed of both cores according to the contingency schedule, including ρ_i and ρ_i'. Once the end of the frame is reached, the system resumes the execution according to the main schedule in the new frame, at low processing speeds to save energy.

R3. If a permanent fault is detected on HP or LP, the system cancels the execution of main schedule on the remaining operational core, and immediately starts executing the incomplete tasks in the contingency schedule at the maximum speed. The system operates on a single core executing the contingency schedule until the faulty core is replaced or repaired.

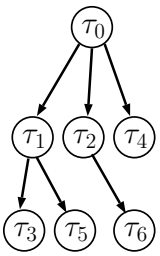
It should be noted that even though the task start times are computed according to the *as late as possible* principle in the contingency schedule, when the system transitions to the recovery mode upon the detection of a fault, the required contingency tasks are dispatched *immediately* in the specified order, without waiting until the latest possible start times indicated in the contingency schedule. We conclude this section by the following remarks that justify fault tolerance capability of the proposed framework:

FT1. As long as both cores are functional, the system is able to recover from transient faults affecting any number of tasks, even when some of these faults may occur in the recovery mode, affecting a single copy of each pair of the contingency tasks ρ_i and ρ_i'.

FT2. The system can tolerate one permanent fault of any of the cores, even when the fault may occur during the execution of the contingency schedule thanks to the paired arrangement of the contingency tasks.

B. Task partitioning and ordering

Now we turn our attention to the problem of allocating the primary tasks on two cores and ordering them to satisfy the precedence constraints. In general, partitioning a set of real-time tasks on a multiprocessor system is a well-known NP-Hard problem. For this reason, our framework generates the task partitions decisions offline, based on the *list scheduling* approach which is widely used to schedule tasks with precedence constraints [21], [24]. In list scheduling, tasks

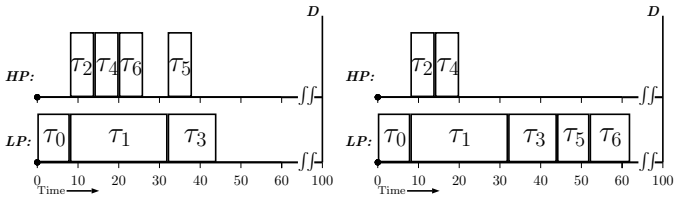


(a) Task graph

	W_i^{HP}	W_i^{LP}	a_i^{LP}	α_i^{LP}
τ_0	3.0	7.5	0.28	0.028
τ_1	10.0	23.1	0.45	0.045
τ_2	6.0	13.5	0.30	0.03
τ_3	6.5	14.6	0.40	0.04
τ_4	5.0	9.4	0.25	0.025
τ_5	4.5	7.9	0.20	0.02
τ_6	4.0	7.0	0.20	0.02

(b) Task parameters

Fig. 2: Task Set for the Running Example



(a) LTF Partition (b) TBLS Partition

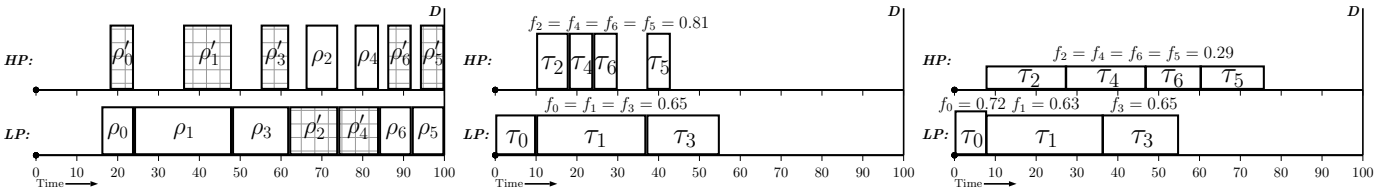
Fig. 3: Task partitioning algorithms

are allocated to the available cores one at a time, starting with tasks with no predecessors (root tasks). A task whose all predecessors have been already allocated becomes also eligible for allocation. The algorithm keeps track of the tasks allocated to individual cores, as well as the current length of the schedule (*makespan*) on every core. If multiple tasks are eligible for allocation, ties may be broken using various parameters, such as execution time or power consumption of the tasks. Below we describe two heuristics based on this list scheduling technique. It should be noted that once the task partitioning is determined through our heuristics, it is not changed at run-time; i.e., migration of the tasks is not allowed.

We use a running example to demonstrate the operation of our heuristics. As shown in Fig. 2, we have a task set with 7 tasks along with their dependencies indicated by the task graph. For each task, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. The other parameters are shown in Table 1b, where W_i values are computed assuming maximum speed on the respective core. For the HP and LP cores, we assume $f_{max}^{HP} = 1.0$, $f_{max}^{LP} = 0.8$, $P_{idle}^{HP} = 0.05$, and, $P_{idle}^{LP} = 0.02$.

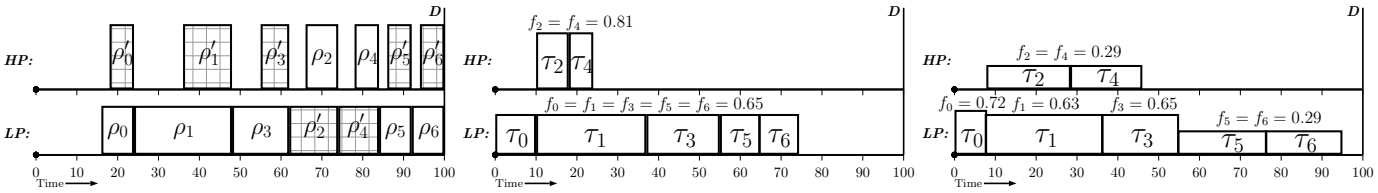
Largest Task First (LTF). In this variant of list scheduling, tasks again are allocated one by one, and in each iteration, the next highest-priority eligible task is allocated to the earliest available processor. The priority of a task is determined by its size (the C_i^{HP} value). In case both of LP and HP cores are available at a given iteration, then we choose the LP core (assuming the deadline is still met), to save energy.

This method produces a good mix of tasks on the HP and LP core and generally produces a schedule with a short makespan. The overall complexity of the algorithm is $O(n^2+nE)$, where n is the number of tasks and E is the number of dependencies in the task graph. The operation of LTF for our running example task is shown in Fig. 3a. All tasks can finish execution



(a) LTF Contingency Schedule (b) LTF-US Schedule (c) LTF-CPSS Schedule

Fig. 4: Contingency Schedules and Speed Assignments under LTF scheme



(a) TBLS Contingency Schedule (b) TBLS-US Schedule (c) TBLS-CPSS Schedule

Fig. 5: Contingency Schedules and Speed Assignments under TBLS scheme

by $t = 45$ (when executed at the maximum speed of their respective cores.) The corresponding contingency schedule for this task-allocation is shown in Fig. 4a.

Threshold-based List Scheduling (TBLS). This method attempts to exploit the low-power feature of the LP core. Specifically, we define a *threshold* value for the utilization on the LP core. If the total task utilization (on the LP core) is less than this threshold value, then all tasks are placed on LP. Otherwise, the algorithm allocates some tasks on the HP core in order to keep the LP core's total utilization under the predefined *threshold* value.

Specifically, the tasks are again ordered according to their C_i^{HP} values and we use the list scheduling technique to allocate eligible tasks (by considering the precedence constraints) to the LP core as long as its utilization does not exceed *threshold*. When this threshold is about to be exceeded on LP considering the size of the next task, the algorithm attempts to put it on HP core. This method prefers LP core to allocate most of the tasks, but can take advantage of the HP core when the task set's utilization is high. The runtime complexity of the algorithms is the same as LTF: $O(n^2 + nE)$. The operation of this algorithm on our example task set with a *threshold* value of 65% is shown in Fig. 3b. The corresponding contingency schedule is shown in Fig. 5a.

C. Speed assignment

After an allocation of tasks and their execution order is obtained on each core, we need to determine the execution speed (frequency) of the tasks, such that the frame deadline can be met by also considering the slots reserved for the contingency schedule. Assuming maximum frequency on each core, we first compute the activation and completion time of each task with worst case number of cycles. We call this the *canonical schedule*. When assigning speeds to tasks, we make sure that each primary task can complete its execution before the activation time of its contingency copy on the alternate core. By doing so, in the fault-free case, contingency copies are never activated, since we cancel the reservation in the contingency schedule as soon as the primary copy completes, thereby saving energy. For each primary task, its contingency copy activation time can be seen as its *pseudo-deadline* which is earlier than the frame deadline D. Using this method, we developed the following two techniques.

Uniform Scaling (US). In this method, we start with the canonical schedule and slow down all the tasks by a uniform scaling factor $S \leq 1.0$, ensuring that all tasks complete by their contingency activation times. The scaling factor, multiplied by the maximum speed of the respective core, gives the execution frequency (speed) of that core. This method determines the task with the *tightest* timing constraint (the task with the minimum [*contingency activation time - primary completion time*] difference), and scales the frequencies accordingly. The algorithm (with $O(n)$ complexity) ensures that no tasks would execute beyond its contingency activation time. Therefore, scaling all tasks by the same (minimum) amount cannot possibly result in a deadline miss.

The schedules for our running example under LTF and TBLS and Uniform Scaling, named as LTF-US and TBLS-US, are shown in Fig. 4b and 5b, respectively. The uniform scaling is determined by the pseudo-deadline of τ_1 (which is the tightest) and gives a speed of 0.81 on HP and 0.65 on LP, for both LTF and TBLS schemes. The energy consumption of LTF-US and TBLS-US schemes are 28.49 mJ and 23.34 mJ, respectively. We can see that by limiting the use of HP core, TBLS consumes 18% less energy than LTF under US.

Critical Path based Static Speed (CPSS). This method is developed as an extension to the critical-path based DVFS algorithm originally proposed in [25]. As opposed to imposing the same scaling factor to all the tasks, this method computes

different scaling factors on the basis of each execution path, starting from the most critical one in terms of timing constraints. However, an assumption of the technique is that all processors are homogeneous and hence have the same power dissipation characteristics.

In our adaptation of the algorithm from [25], we differ in the following way: i) we consider all possible paths from any source to any sink to be in our set of critical paths, and ii) when scaling a path, we use the system-level DVFS algorithm [26] to exploit the heterogeneity of the cores to minimize energy consumption, as opposed to using a common scaling factor for all tasks on a given path as done in [25].

The schedules produced by CPSS are shown in Fig. 4c and 5c, for LTF and TBLS partitioning and our running example, respectively. They show that CPSS assigns relatively low execution-speed for tasks on the HP core, and also for τ_5 and τ_6 on the LP core. The energy consumption for LTF-CPSS is 19.30 mJ, which is 32% less than the uniform scaling in LTF-US scheme. By limiting the use of HP core, TBLS-CPSS consumes even less energy, 17.33 mJ, which is 25% and 10% better than TBLS-US and LTF-CPSS, respectively.

The time complexity of this algorithm depends on the number of all distinct paths from all source to all sink nodes– we denote it by k. The system-level DVFS technique takes $O(n^2 \log n)$ time for a path with n tasks. In each iteration of our algorithm, finding the *most critical path* would take $O(kn^2 \log n)$ time, and then applying ENERGY-LU for a final set of frequencies would take another $O(n^2 \log n)$. The iterations would run for at most n times, therefore, the algorithm's overall complexity is $O(kn^3 \log n)$.

D. Dynamic Reclamation

Real-time systems must be designed to deal with the worst-case workload scenarios. However, real-time tasks often finish earlier than their worst-case estimates. Thus, to exploit the early completions and save more energy at run-time by dynamic slow-down, we developed a dynamic slack reclamation algorithm. In this algorithm first we compute offline speeds for each task based on any of our speed assignment algorithms. Then, using these speeds and the worst case number of cycles for each task, we compute a *reference activation time*, which corresponds to the time point when a task should start its execution when all tasks run at their assigned speeds and present their worst-case workload.

At runtime, when a task is about to be dispatched, we check the difference between its reference activation time and the current time. This difference is denoted as its *slack*. We recompute the assigned speed of the ready task by giving all the slack time to it, i.e., slow it down further such that it completes at its original (offline) reference completion time. Let f_i be the offline assigned speed for τ_i with C_i worst-case number of cycles, and s_i be the dynamically generated slack available at its dispatch time. Then, its dynamically adjusted speed, f_i^* is computed as $f_i^* = \frac{C_i f_i}{C_i + s_i f_i}$. The algorithm is invoked at dispatch time for every task and it has constant time complexity ($O(1)$).

IV. Experimental Evaluation

We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. In our simulator, we implemented the task partitioning schemes LTF and TBLS, as well as the speed assignment schemes US and CPSS, giving four combinations named as LTF-US, LTF-CPSS, TBLS-US, and TBLS-CPSS.

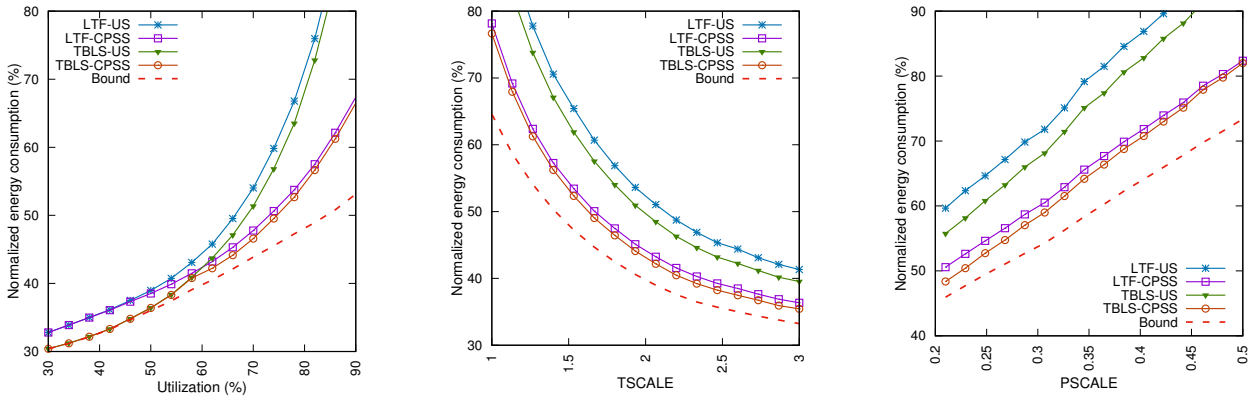
We also implemented a scheme named *Bound*: This scheme is based on brute-force search for all possible task partitioning and choosing the one with lowest energy consumption. *Bound* does not implement any fault tolerance, and removes all contingency tasks, setting the deadline of all tasks to D. The tasks are allocated on two cores using the topological order and the CPSS technique is used for speed assignment. We use this scheme as a lower bound for energy consumption and compare our proposed schemes that offer fault tolerance. The obtained energy consumption numbers are normalized with respect to the maximum energy consumption (observed in the considered parameter spectrum) of LTF-US scheme.

For each experiment, the simulator generates a task set containing n tasks, and a given total utilization, U. The utilization is calculated with respect to the LP core (which is more constrained in terms of performance) and normalized considering its maximum speed. Hence, $U = (\sum \frac{C_i^{LP}}{D})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [27] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U. We set the frame deadline $D = 100ms$. In order to experiment with arbitrary task-graphs, we use TGFF tool [28] to randomly generate a DAG with n nodes.

It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [29]. Therefore, as in [15], we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which models how execution time changes on the LP core for a given task, τ_i. Moreover, following [15], we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as $\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within ranges suggested in [29]. Specifically, $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ hold. We assume for all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. In addition, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$ for all experiments.

We use task sets with $n = 10$ tasks, $f_{max}^{LP} = 0.8$ and $f_{max}^{HP} = 1.0$, unless otherwise stated. The value of *threshold* is set to 0.6 for TBLS. Every reported data point is the average of 1000 runs. We report the average energy consumption in fault-free executions, since faults are very rare events.

Impact of Utilization. Figure 6a shows the impact of utilization on normalized energy consumption. When the utilization is low, the energy consumption is largely dependent on the partitioning method, and not very much on the speed assignment schemes. This is because at low load, most tasks would typically be able to run at their energy-efficient



(a) Impact of Utilization (b) Impact of tscale at load = 75% (c) Impact of pscale at load = 75%

Fig. 6: Impacts of Utilization, tscale, and pscale.

frequency f_{ee} in both speed assignment techniques. TBLS puts all the tasks in to the LP core resulting in a better performance, whereas LTF utilizes the HP core for some tasks and spends somewhat more energy. As the load increases, all the schemes show an increase in the energy consumption, however, the CPSS schemes can keep the energy consumption low compared to the US schemes. This is because CPSS can set a suitable speed to each of the task, whereas US has to commit to a common speed for all tasks. With increase in the load, the advantage of CPSS becomes very significant (up to 35%). For a given speed assignment technique, TBLS partitioning performs slightly better than the LTF technique. Among all schemes, TBLS-CPSS performs the best and stays within 10% of Bound for up to 75% of system load.

Impact of tscale. Figure 6b shows the impact of varying *tscale* value for all tasks when the system load is fixed at 75%. As *tscale* increases, the normalized energy consumption of the system decreases. This is because a low value for *tscale* indicates a very efficient LP core. The plot shows that for the entire range of *tscale* values, TBLS-CPSS is performing the best, very closely followed by the LTF-CPSS scheme. Compared to Bound, both of the CPSS schemes perform very close (around 6-12%) for the entire spectrum.

Impact of pscale. Varying *pscale* also has similar effect as shown in Fig. 6c. As *pscale* increases, the overall energy consumption of the system also increases. Increasing the value of *pscale* implies making the LP core more power-hungry, resulting in higher overall energy consumption. We can see that TBLS-CPSS performs best throughout the entire *pscale* spectrum, closely followed by LTF-CPSS.

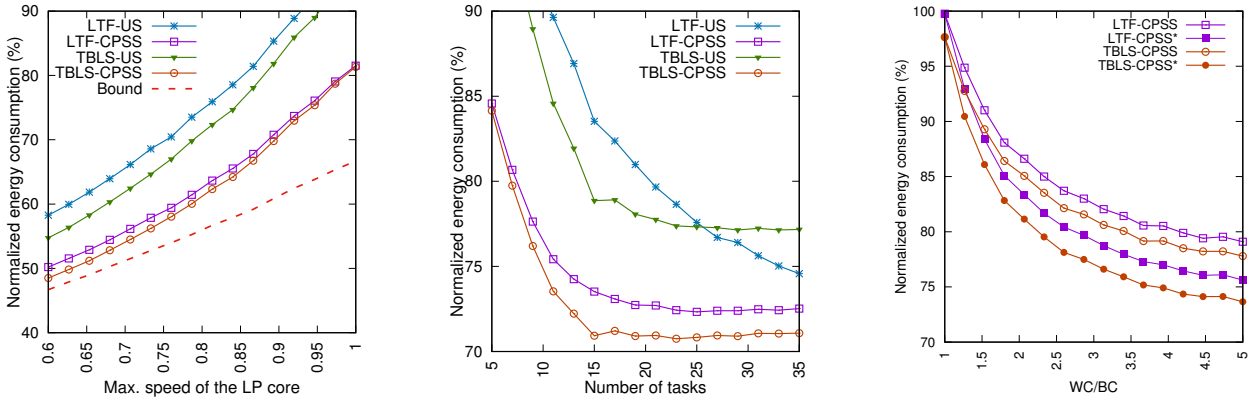
Impact of maximum speed of LP core. In this set of experiments, we varied the maximum speed of the LP core while fixing the load at 75%, as shown in Fig. 7a. We see that the energy consumption of all schemes increase with increasing f_{LP}^{max}. This is because, when the utilization is kept fixed at 75% (which is computed relative to f_{LP}^{max}) the effective amount of workload on the system increases with increasing LP core speed, which is reflected in the results. We observe again that TBLS-CPSS performs the best, closely followed by LTF-CPSS. In fact, Bound scheme performs only 5% better than TBLS-CPSS when LP core's maximum speed is low. However, their difference increases with the LP core's maximum speed.

Impact of number of tasks. Fig 7b shows the impact of number of tasks for a system with 75% load. We see that for small number of tasks, the performance of all the schemes is affected. As the number of tasks grows, the average task size decreases and the performances of various schemes stabilize. For TBLS partitioning, the advantage of using CPSS scheme over US can be up to 18% when number of tasks is low, but it stabilizes at 10% with the increase in number of tasks. The plot also shows that LTF-US starts to outperform TBLS-US as soon as number of tasks exceeds 25. This is because when the average task size decreases, this favors the LTF scheme greatly due to the reduced cost of using the HP core with US. *Bound* is not shown in these experiments due to its prohibitive running time with increased number of tasks.

Impact of Workload Variability. To evaluate the gains due to our dynamic reclamation scheme in the presence of the workload variability, we first define the ratio WC/BC as the ratio of the worst-case execution time to the best-case execution time. During the experiments, the actual execution time of every task is randomly generated between its worst-case and best-case execution time, using a uniform probability distribution. A higher value of WC/BC indicates larger amount of runtime slack being generated, providing opportunities for further energy savings. In these experiments, we evaluated 1000 execution frames for each task set. Figure 7c shows that our techniques with dynamic reclamation enabled (indicated by '*') are able to save additional energy at runtime. The dynamic schemes are about 6-8% more efficient than their static counterparts.

V. Conclusions

In this paper, we proposed a fault-tolerant framework for dependent real time tasks executing on a heterogeneous dual



(a) Impact of maximum speed of LP core at load = 75%

(b) Impact of number of tasks at load = 75%

(c) Impact of WC/BC at load = 75%

Fig. 7: Impact of the LP core's maximum speed, number of tasks, and workload variability

core system. We presented task partitioning heuristics for allocation and ordering of tasks to the available processing cores, and developed speed assignment techniques which can ensure low-energy consumption while providing reliability against transient and permanent faults. Simulation results demonstrate that our proposed techniques are capable of energy efficient operation and perform close to a theoretical lower bound.

REFERENCES

[1] A. Colin, A. Kandhalu, and R. R. Rajkumar, "Energy-efficient allocation of real-time applications onto single-isa heterogeneous multi-core processors," *Journal of Signal Processing Systems, 84(1)*, 2016.

[2] Y. Qin, G. Zeng, R. Kurachi, Y. Matsubara, and H. Takada, "Execution-variance-aware task allocation for energy minimization on the big. little architecture," *Sustainable Computing: Informatics and Systems*, 2018.

[3] P. P. Nair, R. Devaraj, and A. Sarkar, "Fest: Fault-tolerant energy-aware scheduling on two-core heterogeneous platform," in *Proc. IEEE ISED*, 2018.

[4] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. of ACM/IEEE DAC*, 2013.

[5] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *Proc. of IEEE DATE*, 2015.

[6] T. Wei, P. Mishra, K. Wu, and H. Liang, "Fixed-priority allocation and scheduling for energy-efficient fault tolerance in hard real-time multiprocessor systems," *IEEE Trans. on Parallel and Distributed Systems, 19(11)*, 2008.

[7] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich, *et al.*, "Design and architectures for dependable embedded systems," in *Proc of IEEE/ACM/IFIP CODES+ISSS*, 2011.

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

[9] H. Aydin, R. Melhem, and D. Mossé, "Tolerating faults while maximizing reward," in *Proc. of IEEE ECRTS*, 2000.

[10] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro, 24(6)*, 2004.

[11] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE Trans. on Parallel and Distributed Systems, 28(3)*, 2017.

[12] M. Fan, Q. Han, and X. Yang, "Energy minimization for on-line real-time scheduling with reliability awareness," *Journal of Systems and Software, 127*, 2017.

[13] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 31(3)*, 2012.

[14] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *IEEE RTAS*, 2012.

[15] A. Roy, H. Aydin, and D. Zhu, "Energy-aware standby-sparing on heterogeneous multicore systems," in *Proc. of IEEE/ACM DAC*, 2017.

[16] A. Roy, H. Aydin, and D. Zhu, "Energy-efficient primary/backup scheduling techniques for heterogeneous multicore systems," in *Proc. of IEEE IGSC*, 2017.

[17] S. K. Baruah, "A general model for recurring real-time tasks," in *Proc. IEEE RTSS*, 1998.

[18] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *IEEE Trans. on Industrial Informatics, 7(2)*, 2011.

[19] Y. C. Lee and A. Y. Zomaya, "Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling," in *Proc. IEEE/ACM CCGrid*, 2009.

[20] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, and H. Xiong, "Energy-efficient real-time scheduling of dag tasks," *ACM Trans. on Embedded Computing Systems, 17(5)*, 2018.

[21] K. D. Cooper, P. J. Schielke, and D. Subramanian, "An experimental evaluation of list scheduling," *TR98, 326*, 1998.

[22] B. Zhao, H. Aydin, and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints," *ACM Trans. on Design Automation of Electronic Systems, 18(2)*, 2013.

[23] D. Zhu, R. Melhem, and D. Mossé, "The effects of energy management on reliability in real-time embedded systems," in *Proc. of IEEE ICCAD*, 2004.

[24] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Trans. on software engineering, 15(12)*, 1989.

[25] Y. Liu, B. Veeravalli, and S. Viswanathan, "Novel critical-path based low-energy scheduling algorithms for heterogeneous multiprocessor real-time embedded systems," in *Proc. IEEE ICPADS*, 2007.

[26] H. Aydin, V. Devadas, and D. Zhu, "System-level energy management for periodic real-time tasks," in *Proc. IEEE RTSS*, 2006.

[27] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. of the Int. WS on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.

[28] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proc. IEEE CODES/CASHE*, 1998.

[29] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. of IEEE CASES*, 2013.