Energy-Aware Primary/Backup Scheduling of Periodic Real-Time Tasks 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: dzhu@cs.utsa.edu

Abstract

Energy management and fault tolerance are often conflicting requirements, as the extra resources needed to tolerate faults significantly increase the energy consumption. Yet, for real-time embedded systems both objectives are crucial. In this paper, we consider energy-aware and fault-tolerant scheduling of preemptive fixed-priority periodic real-time tasks on heterogeneous multicore systems. To tolerate both transient and permanent faults, primary and backup copies of tasks are scheduled on different cores. Our framework consists of offline and online phases to manage energy and fault-tolerant scheduling of periodic tasks in tandem. We propose an array of techniques to minimize the energy consumption, including DVFS to scale the primary tasks, and mechanisms to maximize the opportunities to cancel the back-up tasks in fault-free execution scenarios. The latter objective is achieved through an explicit task priority assignment phase, coupled with a dual queue based back-up delaying algorithm. In particular, we propose a scheme called *Reverse Preference-Oriented Priority Assignment (RPPA)* which is experimentally shown to be very effective to reduce the energy consumption. RPPA, when coupled with the dual-queue based delaying mechanism, outperforms other schemes, and approaches the energy performance of a theoretical lower bound. All the proposed schemes satisfy the stringent timing and fault tolerance requirements of periodic real-time tasks while managing the energy consumption dynamically.

Keywords: Heterogeneous multicore systems, fault tolerance, energy management, periodic real-time scheduling.

1. Introduction

Recently, heterogeneous multi-core systems which combine cores of different types on the same chip have received increasing attention. On those systems, typically different types of cores have the same instruction-set-architecture (ISA); hence, the same binary can run on different cores. Those systems often include *out-of-order* cores that offer high performance (such as ARM Cortex A-15) as well as *in-order* cores with modest performance but very high energy-efficiency (such as ARM Cortex A-7). They are particularly attractive in settings that require dynamic adaption to the workload at hand: for instance, in energy-constrained settings, the system can rely mostly on low-power cores; but it can make maximum use of the high-power cores when superior performance is required. There is a growing body of research literature that investigates various aspects of heterogeneous multicore systems [1, 2, 3, 4, 5].

For many embedded systems, in particular those that are deployed in safety-critical applications,

real-time operation (in terms of meeting the timing constraints) and *fault tolerance* (in terms of meeting the reliability requirements) are important design and operation objectives. In particular it is imperative to detect and recover from run-time *faults* in a timely manner in those applications. Research studies indicate that most of those faults are *transient* in nature, meaning that they are short-lived [6]. Their sources can be traced back to electromagnetic interference and various cosmic rays. They cause erroneous computation in some task(s), and typically repeating the original computation or invoking an alternative recovery task gives the correct result [6]. Yet, another type of fault is the *permanent fault* which affects a specific system component (e.g., an individual core) and results in the unavailability of that component until it is replaced or repaired [6, 7]. Any realistic safety-critical system must provide mechanisms to tolerate both transient and permanent faults.

Fault tolerance mechanisms often rely on the *redundancy* principle. For instance, tolerating permanent faults requires the availability of an additional hardware component (e.g., a processing core) that can provide the functionality of a faulty component. Similarly, transient faults of individual tasks are often tolerated by deploying backup copies for each. As redundancy imposes additional resource requirements, fault tolerance and energy-awareness are frequently at odds with each other. In fact, a straightforward deployment of conventional fault tolerance techniques, such as tri-modular redundancy [7], would consume prohibitive amount of energy. As a result, numerous research studies explored various solutions to achieve fault tolerance objectives with minimum energy consumption [8, 9, 10, 11, 12, 13, 14, 15]. A more detailed discussion of Related Work can be found in Section 8.

This research effort investigates energy-efficient and fault-tolerant implementation of periodic real-time systems upon heterogeneous dual core systems. While energy-efficiency and fault tolerance on heterogeneous cores have been recently investigated [16, 17, 18, 19, 20, 21, 22], the focus was mostly on frame-based systems where all tasks share a common period and deadline. However, actual implementation of many real-time systems are based on general periodic tasks which are invoked at different rates [23]; so we believe this study fills an important gap.

In our framework each periodic real-time task (called the *primary*) is assigned to one of the cores: all the instances of that periodic task ("jobs") are released and executed at periodic intervals on that core. In addition, for every such primary task, a *backup* periodic task is assigned to the alternate core. Since every real-time job has a backup allocated to the other core, transient faults in all primary jobs can be tolerated. Similarly since there are two copies of each periodic task assigned to alternate cores, the system can tolerate one permanent fault of any core by switching to the functional core if necessary.

To manage the energy consumption, we use two mechanisms: i.) the primary tasks are executed at low voltage/frequency levels using *Dynamic Voltage and Frequency Scaling*, and, ii.) the backup copies are *delayed* to the extent it is possible to enable their cancellation in case the primary completes without a fault. In addition, we use the *Mixed Primary/Backup Scheduling* framework in which a given core may be assigned both primary and backup copies of (distinct) tasks [19]. This is in contrast to the so-called *Standby-Sparing* systems in which one core is exclusively dedicated to the primaries and the other one (spare) executes only the backups [11, 12, 24, 25, 26]. While the mixed primary-backup scheduling framework significantly improves the schedulability and energy saving potential, it also presents challenges in terms of task allocation and scheduling.

A significant challenge in these settings is to find an efficient way to delay the backup copies: because of periodic and preemptive execution settings, a backup copy can be preempted multiple times by other jobs; but it should still meet its deadline when needed. To tackle this, we leverage two mechanisms: one is *assigning proper priority levels* to periodic primary and backup tasks, and the other one is the actual delaying of the backups using the *dual-queue* mechanism [11, 27]. Specifically, the backup tasks are promoted and become eligible for execution after a pre-determined time interval after their release. These promotion times are shown to be safe, in that they are derived using the well-known critical instant analysis technique for fixed priority periodic real-time tasks [23]. In particular, we show that when combined with the dual queue based delaying mechanism, assigning tentatively high-priority levels to the backup tasks gives the maximum delaying and energy saving opportunities. To the best of our knowledge, this research effort is the first study that explores fault-tolerant and energy-aware mixed primary backup scheduling

of periodic real-time tasks on heterogeneous dual core systems.

The rest of the paper is organized as follows. In Section 2, we present our system model and assumptions, including the fault model. In Section 3, we present preliminary discussion about fixed priority scheduling. Section 4 describes the mixed primary/backup framework. Section 5 presents our proposed priority assignment scheme, and Section 6 presents the runtime algorithms of our framework. In Section 7, we present our experimental results. Section 8 has discussion about related works, and Section 9 concludes our paper.

2. System Model and Assumptions

2.1. 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 independent real-time tasks $\{\tau_1, ..., \tau_n\}$. We assume the *general periodic task* model in which each task τ_i generates a job instance periodically with the period P_i. Each instance of the periodic task τ_i must complete within the relative deadline D_i, which is equal to its period (implicit deadline). Each task is assigned a scheduling *priority* determined at design time. In this paper, we consider fixed-priority periodic real-time tasks.

Each of the two processing cores is equipped with the *Dynamic Voltage and Frequency Scaling (DVFS)* feature that allows changing the frequency (processing speed) at run-time to manage energy consumption, A task instance of τ_i that requires C_i number of cycles on a given core may take up to $W_i(f) = C_i/f$ units of execution time on that core, if executed at the frequency level f. The worst-case number of cycles required by the instances of the task τ_i is denoted by C_i. 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 $U_i = W_i^{HP}(f_{max}^{HP})/P_i = W_i^{HP}(1.0)/P_i = C_i^{HP}/P_i$.

2.2. Power Model

The power consumption characteristics of HP and LP cores differ by design. For any processing core, the dynamic power consumption of an executing instance of 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., $P_i^{HP}, \alpha_i^{HP}, a_i^{HP}$).

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 [28]. This threshold frequency, known as the *energy-efficient frequency (f_{ee})* can be derived through analytical techniques [28], and we never reduce the processing frequency below f_{ee} on a given core.

2.3. Fault Model

Our framework opts to provide high assurance to safety-critical real-time tasks in energy-aware manner. Hence, we aim to tolerate transient faults (that affect individual task instances) as well as permanent faults (that lead to the unavailability of a whole processing core). Specifically, in our framework, we tolerate the following type of faults:

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

Each (primary) task τ_i has an associated backup task B_i with exact same timing parameters. τ_i

and B_i are allocated to different processing cores. Whenever a task instance is released, its backup copy is also released and is allocated to the alternate core. In order to maintain energy efficiency, the backup copy instances are delayed as much as possible while respecting their deadlines. Should a permanent fault affect any of the processing cores, the alternative core can take over and finish the workload before deadline. When a primary copy

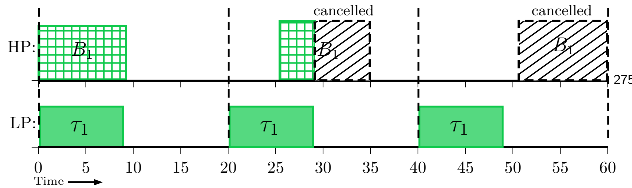


Figure 1: Primary/Backup Overlap

completes, the *acceptance* (or, *sanity*) tests [6] are performed to check the existence of errors induced by *transient* faults. If a fault is not detected, the corresponding backup copy (or, its remaining part) on the other core may be cancelled, as shown in Figure 1, to save energy. Otherwise, the backup copy runs to completion. If a permanent fault occurs on any of the cores, the other core can still execute one copy of each task's instances. However, note that, when a permanent fault occurs, the system loses the capability of tolerating any additional (transient or permanent) faults until the faulty core is repaired or replaced.

It should be noted that when a backup copy executes in the fault-free case, it is essentially a redundant execution which increases the energy consumption of the system significantly. Ideally, we would like to minimize redundant execution in order to conserve energy. As shown in Figure 1, the first instance of τ_1 and its backup B_1 execute in parallel, wasting a lot of energy. In the second instance of τ_1, we delayed B_1 to some extent and were able to cancel some parts of it. In the third instance, B_1 was delayed enough so that its execution could be entirely omitted.

Problem Statement. Given a set of real-time independent periodic tasks and a heterogeneous dual-core system, minimize the energy consumption by determining

1. The allocation of tasks to cores such that the primary and backup copy of each task are assigned to different cores, and,
2. The priority assignment, scheduling, and processing frequency assignment decisions for individual periodic task instances.

In the following sections, we first present preliminary (background) material and then we develop multiple components of our proposed framework.

3. Preliminaries

3.1. Work-Conserving Fixed-Priority Periodic Scheduling

Most of the traditional hard real-time scheduling theory is based on the **work-conserving** approach, in which the processor never idles as long as there are ready jobs to execute [23]. A well-known framework is **fixed-priority scheduling (FPS)** in which all the jobs generated by a given periodic task are assigned the same priority level during execution.

The real-time *feasibility analysis* is concerned with assessing if all the real-time jobs will meet their deadlines given the characteristics of the workload at hand [23]. In FPS, it is known that the worst-case response time of a periodic task occurs when it is released at the same time as all high-priority tasks. Specifically, the worst-case response time S_i of a task τ_i can be computed using the following iterative formula [29]:

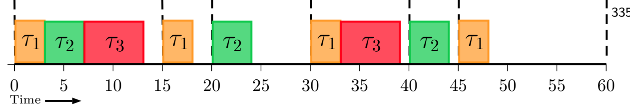
$$S_i^{(k+1)} = c_i + \Sigma_{\tau_j \in \ hp(\tau_i)} \lceil (S_i^{(k)}/P_j) \rceil \times c_j \quad (1)$$

Above c_i is the worst-case execution time of τ_i and $hp(\tau_i)$ denotes the set of tasks which are assigned a priority level higher than that of τ_i. In this iterative approach, initially $S_i^0 = c_i$ and the iterations continue until $S_i^{(k+1)} = S_i^{(k)}$. If at any point $S_i^{(k)}$ exceeds the period (relative deadline) P_i, then the task will not meet its deadline. Otherwise, the task will meet its deadline and the worst-case response time S_i is found as the last value obtained for $S_i^{(k)}$.

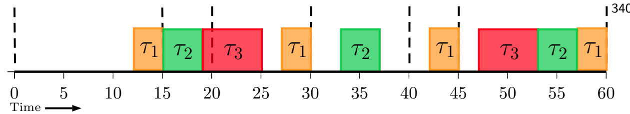
An important FPS policy is *Rate Monotonic Scheduling (RMS)* in which the priorities are inversely proportional to the periods. RMS is known to be optimal among all periodic fixed-priority assignments, in the sense that all task sets that can meet their deadline with any fixed-priority assignment can also do so using RMS [30]. This optimality makes RMS the most widely known and adopted fixed priority assignment policy for periodic real-time tasks [30].

Table 1: Example Task Set 1

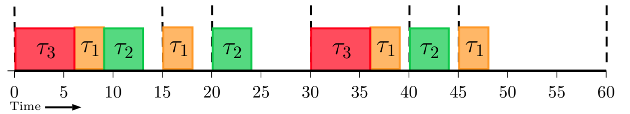
	Period	Execution Time
τ_1	15	3
τ_2	20	4
τ_3	30	6



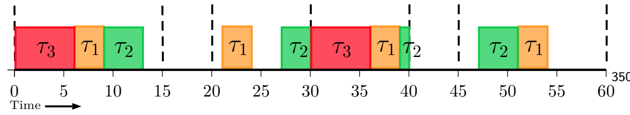
(a) Schedule obtained with RMS priorities



(b) Non-work-conserving schedule with dual queue based delaying (RMS priorities)



(c) Schedule obtained with Preference-Oriented Priority Assignment (PPA)



(d) Non-work-conserving schedule with dual queue based delaying (PPA priorities)

Figure 2: Work-conserving and non-work-conserving fixed-priority schedules

As an example consider the task set given in Table 1 with three periodic tasks, τ_1, τ_2 and τ_3. For illustration purposes, we assume all tasks execute at maximum frequency. The corresponding schedule obtained using RMS is shown in Figure 2a. The period boundaries are denoted by vertical dashed lines in the figure. As it can be observed, all periodic task instances meet their deadlines.

3.2. Non-Work-Conserving Fixed-Priority Periodic Scheduling

There are a number of scenarios where it is desirable to *delay* periodic tasks as long as they can still complete before their respective deadlines. For instance, when the workload includes non-real-time aperiodic jobs that arrive at unpredictable times, a common objective is to execute them as soon as possible to minimize their response time. In this case, periodic hard real-time tasks may be delayed maximally to enable early execution of the aperiodic jobs [23].

In these cases, work-conserving policies such as conventional RMS are no longer appropriate; instead non-work-conserving approaches are considered. For example the *dual-queue* based approach [11, 27] works as follows: The system is equipped with two (dual) queues, named *upper queue* and *lower queue*, respectively. Upon arrival, each periodic real-time job is first put to the lower queue, and remains there until a certain *promotion time* at which it is moved to the *upper queue*. Only jobs in the upper queue are eligible for execution; and they are dispatched according to the underlying fixed (e.g., RMS) priorities.

The crux of the scheme is to choose the promotion time safely and maximally, in order to delay the execution of the periodic jobs as much as possible. Specifically, the promotion time of a job of τ_i after its release time is computed as:

$$Y_i = P_i - S_i$$

where P_i is its period and relative deadline, and S_i is its worst-case response time computed through the iterative formula (1) based on the critical instant analysis. It is based on the observation that the job would still meet its deadline after being moved to the upper queue Y_i time units after its release time, even if it is subject to the maximum possible interference by higher-priority jobs [27]. Note that all task promotion times (Y_i values) may be computed offline (before execution) for all periodic tasks.

Returning to our example task set, we can compute the task promotion times as: $Y_i = P_i - S_i$. The S_i values are obtained by applying the formula (1) iteratively, and the promotion times are found as $Y_1 = 12, Y_2 = 13$ and $Y_3 = 17$, for τ_1, τ_2, and τ_3. The resulting non-work-conserving fixed-priority schedule (where tasks are first delayed in the lower queue and then eventually dispatched from the upper queue with RMS priorities) is shown in Figure 2b. It should be noted that many real-time jobs are significantly delayed, but they still meet their deadlines. This dual queue based approach will be instrumental in our mixed primary/backup energy-aware scheduling approach, as we elaborate in Section 4.

3.3. Preference-Oriented Priority Assignment (PPA)

A more recent study considers the *execution preferences* of real-time tasks explicitly in the scheduling phase [31]. Specifically, periodic real-time tasks are classified as ASAP or ALAP, depending on whether there is a preference to execute them *as soon as possible* or *as late as possible*, respectively, but still before all the hard deadlines.

In [31], the problem of finding a fixed priority assignment to satisfy the periodic real-time tasks' execution preferences while meeting the deadlines is considered. The solution is obtained through the Audsley's Optimal Priority Assignment Algorithm (AOPA) [32], which runs in time $O(n^2)$ for n periodic tasks. AOPA, which was originally proposed for tasks with potentially different release times [33], proceeds by first assigning a task to the lowest priority level, by making sure that that task would meet its deadline even in the worst-case activation pattern. Then it proceeds in iterative manner for priority levels $n-1$, ..., 1. The preference-oriented priority assignment (PPA) scheme, proposed in [31], proceeds in the same way, but it assigns low priority levels to the ALAP tasks and high priority levels to the ASAP tasks as much as possible, while still preserving the timing constraints.

For our example task set, now assume that τ_1 and τ_2 are ALAP tasks, while τ_3 is an ASAP task. PPA assigns the lowest priority to τ_2, medium priority to τ_1, and highest priority to τ_3. The resulting fixed-priority schedule where all the deadlines are met is presented in Figure 2c. It should be noted PPA is, just like RMS, an optimal fixed-priority assignment; but it incorporates the task execution preferences whenever possible in the priority assignment phase.

While PPA takes into account task's execution preferences, it is still by default a work-conserving approach. It is possible to combine PPA with the dual queue mechanism to further delay the ALAP tasks (thereby creating a non-work-conserving schedule). When applied to the schedule in Figure 2c, we obtain the solution in Figure 2d, where the ALAP tasks are delayed until their promotion times. This time promotion times for ALAP tasks τ_1 and τ_2 are computed as $Y_1 = 6$ and $Y_2 = 7$. It can be observed that using the dual queue mechanism helps to increase the delay in the execution of the ALAP tasks, and all the deadlines are still met.

4. Mixed Primary/Backup Scheduling of Periodic Tasks

Our dual objective in fault tolerance (Section 2.3), in terms of tolerating transient faults can be achieved by scheduling a separate backup copy of each periodic task instance. Moreover we require that the primary and backup copies of a given task are scheduled on different cores to provision for the permanent fault of any single core. Note that by scheduling a separate backup copy which is, if needed, executed at the maximum core speed, we also guarantee to fully mitigate the task-level reliability loss (with respect to the transient faults) induced by the application of DVFS [34].

Unlike the standby-sparing systems [11, 12, 16, 26] where one core is allocated only the primary copies of tasks and another one solely to the backups, in our work we adopt the *mixed primary/backup scheduling* approach: a given core can execute primary and backup copies of various tasks for maximum flexibility with respect to energy awareness and schedulability.

Table 2: Example Task Set 2

	P_i	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}	a_i^{LP}	α_i^{LP}
τ_1	15	1.8	3.8	1.98	1.50	0.36	0.036
τ_2	20	2.0	4.0	2.20	1.14	0.26	0.026
τ_3	30	3.5	7.9	3.85	3.30	0.38	0.038

For illustration purposes, we consider a running example throughout this section. Consider the task set with parameters given in Table 2. It has 3 tasks τ_1, τ_2 and τ_3, with respective backup copies B_1, B_2 and B_3 for fault tolerance, which are to be executed on the HP and LP cores. We chose $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$, and for each task, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. We assume that the primary copy of τ_2 and the backup copies B_1 and B_3 are allocated to the HP core, while the primary copies τ_1 and τ_3, along with the backup copy B_2 are allocated to the LP core. This makes sure that the two copies of the same task are always on different processing cores.

In Figure 3a we present the mixed primary/backup schedules that we obtain if we use RMS as the scheduling policy on each core. Observe that all primary and backup copies meet their deadlines, and the fault tolerance objectives are achieved. However, the total energy consumption (33.49 mJ) is significantly hampered by the duplicate execution of all backup tasks.

Figure 3: Mixed Primary/Backup Scheduling with RMS

As mentioned in Section 2.3, in case a fault is not detected at the end of execution of the primary, the remaining part of the corresponding backup copy can be cancelled. This gives a powerful mechanism to reduce the energy consumption. If we use this dynamic backup cancellation mechanism, we obtain the schedule in Figure 3b. It can be observed that some portions of the tasks (primary and backup) on the LP core are cancelled due to the completion of the counterpart task, and we obtain a reduced energy consumption of 29.17 mJ.

While it is important to guarantee the timely completion of backup tasks in case faults are detected, since faults are rare events, the average energy consumption in all execution scenarios will be dominated by fault-free execution scenarios. Consequently, in all subsequent examples we show only fault-free executions and assume that all backups (primaries) are cancelled when the corresponding primary (backup) completes successfully on the other core. For clarity, we do not show the cancelled parts of the tasks in the schedules.

While incorporating this fundamental dynamic backup cancellation mechanism, our framework consists of multiple solution layers aimed at reducing the energy consumption dynamically. In particular, we use DVFS to lower the execution speed of the primaries, and use appropriate mechanisms to delay the execution of the backups to maximize the opportunities for their cancellation.

Specifically, our solution consists of *offline* and *online* phases, as shown in Figure 4. In the offline phase, we use *task partitioning*, *priority assignment*,

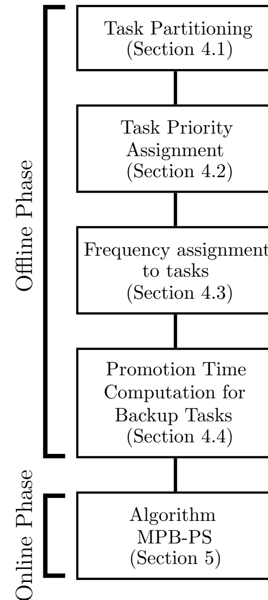


Figure 4: Mixed Primary/Backup Scheduling Components

frequency assignment and *backup promotion time computation* mechanisms. In the online phase, on each core the tasks (and backups, if needed) are executed at the pre-determined priority and frequency levels, and the backups are delayed until their pre-computed promotion times in order to enable their cancellations dynamically.

4.1. Task Partitioning

Our framework generates the task partitioning (allocation) decisions offline, based on the well-known *list scheduling* approach. Specifically we propose the following *List-Scheduling with Primary/Backup (LSPB)* variant. In this algorithm, we consider the primary copies of the tasks and employ list-scheduling algorithm to allocate them. First, the tasks are ordered according to their decreasing nominal utilizations. Then, each primary task is placed on a processing core on which it is *feasible* and which has the maximum *free capacity* after the placement. *Feasibility* is checked by using Time Demand Analysis and RMS priority assignment, which is known to be an optimal fixed priority assignment. *Free capacity* on a core is defined by $(1.0 - \sum_{\tau_i \epsilon \Gamma_p} \frac{C_i}{P_i} / f_{max})$, where Γ_p is the set of all primary tasks assigned to that core, augmented by the task under consideration. f_{max} and $\{C_i\}$ values are defined in the context of the core under consideration. After all the primary tasks are feasibly placed on the processing cores, their backup

copies are allocated on the respective alternative core. Finally, feasibility is checked again taking the backup copies into account. The partitioning shown in Figures 3a and 3b were in fact obtained using the LSPB technique. It can be observed that it generates a relatively balanced workload distribution which opens up opportunities to reduce energy consumption.

4.2. Priority Assignment

After determining the task partitioning and obtaining a task-set for each core, we turn our attention to the priority assignment to tasks. In fixed-priority scheduling, the execution order of task instances depends directly on their priorities (Section 3.1). The RMS priority scheme, in which tasks with smaller periods receive higher priorities, is a natural option on each core. However, in addition to RMS, there have been other fixed priority assignment algorithms proposed in the literature including the *Preference-Oriented Priority Assignment (PPA)* policy (Section 3.3), which considers execution preferences of different tasks [31]. In our setting, we can invoke the PPA scheme to assign priorities after designating the primary tasks and backup tasks as "as soon as possible (ASAP)" tasks and "as late as possible (ALAP)"' tasks, respectively. We show the execution schedule for PPA in Figure 5, using our example task set. Although we assign low priorities to the backup tasks, the schedule shows that it is not very effective in cancelling the backup copies– it incurs high energy consumption figure of 29.3 mJ. We can attribute this to the fact that the presented solution still generates work-conserving schedules for backup tasks.

In Section 5, we will introduce another priority assignment policy, which, when coupled with other components, gives much more improved energy saving opportunities.

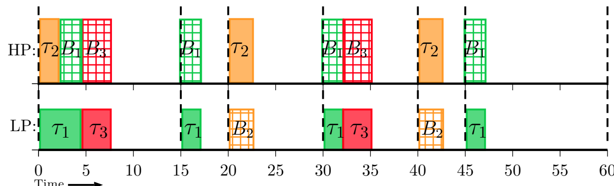
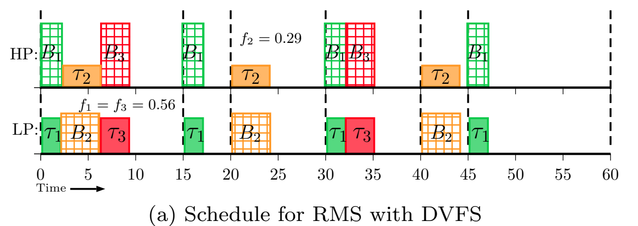


Figure 5: Schedule for PPA

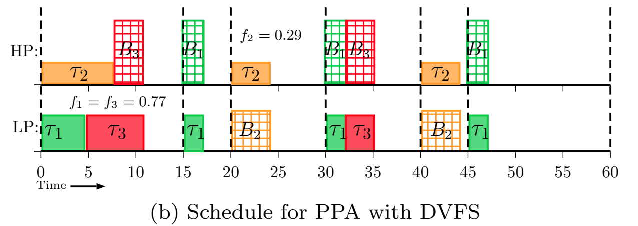
4.3. Frequency Assignment

After an allocation of tasks and their priority assignment is obtained on each core, we can use DVFS to slow down the primary copies and reduce energy consumption. We apply DVFS to primary copies only, and the backup copies are executed at the maximum speed. Not scaling the backups allows to delay their execution further, and also, it mitigates the task-level reliability-loss incurred due to DVFS [34]. For the primary tasks, we need to determine the speed (frequency) of the task-execution, such that the deadlines of all task instances (primary or backup) can be met.

We use a modified version of the well-known *Sys-Clock* algorithm proposed in [35], which assigns a common (minimum) execution-speed to all the tasks on a given core without violating any deadlines. *Sys-Clock* has very low computational overhead. In our modified version of *Sys-Clock*, only the primary tasks are scaled while the backup tasks are assigned the maximum speed of their respective cores. We modified the original algorithm by considering that backup tasks are always executed at maximum frequency levels, and only the primary tasks are scaled. For our example task set, Figures 6a and 6b show the execution schedules for RMS and PPA priorities, respectively when DVFS is applied. It shows that with DVFS, RMS consumes 22.86 mJ and PPA consumes 23.1 mJ, which is about 20% improvement in both cases compared to the cases without DVFS.



(a) Schedule for RMS with DVFS



(b) Schedule for PPA with DVFS

Figure 6: Schedules with DVFS

4.4. Promotion Time Computation for Backup Tasks

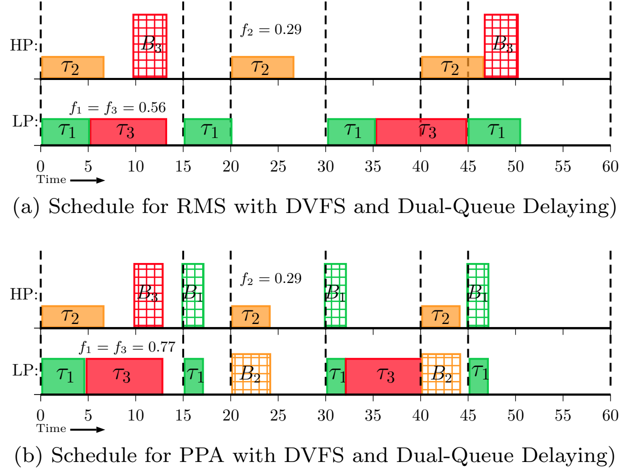
Once the task allocations, priorities, and frequencies are determined, we aim to delay the execution of the backup instances as much as possible, given that no instance should miss its deadline. A delayed backup copy has a greater chance of getting cancelled by its primary copy's completion, which helps to reduce their energy consumption in fault-free cases. In order to delay the backup copies, we used the dual queue based non-work-conserving scheduling algorithm discussed in Section 3.2, by adapting to heterogeneous processors and mixed primary/backup execution with DVFS.

In this delaying technique, when a backup task instance is released, it is first placed on a lower queue, and it can only be promoted to an upper queue at a precomputed promotion time. Promotion times are computed as the same way discussed in Section 3.2, by first computing the *worst-case response time*, using the iterative formula (1). Specifically, for a backup task B_i, its worst-case response time S_i can be computed as:

$$S_i^{(k+1)} = W_i + \Sigma_{\tau_j \in \, hp(B_i)} \lceil (S_i^{(k)}/P_j) \rceil \times (W_j(f_j)) \tag{2}$$

In this formula, W_i is the worst-case execution time of the backup task B_i on its assigned processing core under maximum speed. $hp(B_i)$ is the set of all higher priority (primary or backup) tasks on that specific core. For such a high priority primary task τ_j, we use its execution time, W_j, after scaling it with the *Sys-Clock* speed, f_j. If τ_j is a backup task, then we consider its execution time at the maximum speed of its processing core. Iterative computation continues until $S_i^{(k+1)} = S_i^{(k)}$, which gives the worst-case response time S_i. After that, the scheme computes the promotion time Y_i for each backup task by subtracting its worst case response time (S_i) from its relative deadline (period) P_i.

A backup task is dispatched on the processor only if the promotion time has elapsed and it is in the upper queue. Primary task instances, on the other hand, are always placed in the upper queue directly, and they are dispatched according to their assigned fixed priorities. Figure 7a and Figure 7b show the execution schedules with backup-delaying for RMS and PPA priorities respectively, when applied along with DVFS. It is evident that many of the backup task executions are cancelled in these schemes, which provide energy efficient performance (16.36 mJ for RMS, 19.33 mJ for PPA.) RMS with backup-delaying is almost 28% better than RMS without delaying (Figure 5).



(a) Schedule for RMS with DVFS and Dual-Queue Delaying)

(b) Schedule for PPA with DVFS and Dual-Queue Delaying)

Figure 7: Schedules with Backup Delaying

5. Reverse Preference-Oriented Priority Assignment (RPPA)

We observed that while PPA made an attempt to delay the backup copies by assigning lower priorities, in the motivational example, it did not translate directly to energy savings, even with dual queue based delaying, compared to RMS. In fact, our detailed experimental evaluation (Section 7) will confirm the generality of that observation.

We note that this is primarily due to the fact that during the computation of promotion times (delays) for low-priority backup tasks using Equation (2) the execution times of high-priority tasks act as a negative factors: the lower the scheduling priority of a backup, the higher will be the interference that need to be taken into account when computing the promotion times, and the smaller will be the promotion time (delay) we can afford for the backup.

This suggests an alternative but seemingly counter-intuitive solution: **assign high priorities to backup tasks before applying the dual queue based delaying technique**, relying on the fact that this will help to increase their promotion times, and the total amount we can delay in them in the lower queue. Even though they will indeed execute at high priority eventually, this will only happen when they are maximally delayed in the lower queue through the extended promotion times.

The proposed scheme, called the *Reverse Preference-Oriented Priority Assignment (RPPA)*, is very similar to PPA, but it assigns higher preference to backup tasks, and lower preference to the primary tasks to the extent it is possible. Like before, the backup tasks are delayed using the dual queue based delaying technique.

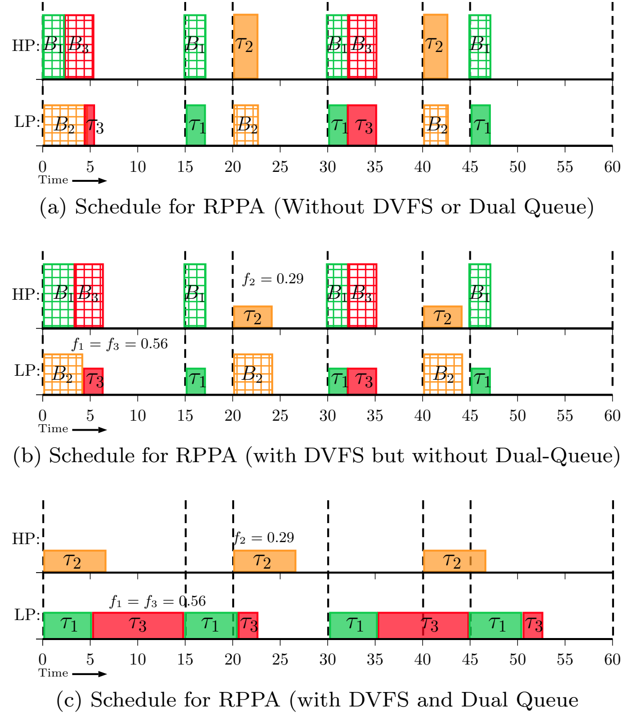
RPPA priority assignment is also optimal as PPA and RMS – it never results in a loss in schedulability as long as there exists a feasible solution. In the extreme case where a given task set cannot be scheduled by assigning low priority to (most of) backup tasks, it will generate another priority assignment which may resemble PPA or RMS, even though we observed that in practice it is able to generate a feasible priority assignment by assigning backup tasks high priorities in many cases.

For our example task set, the schedules with PPA and RPPA are shown in Figure 7b and 8c, respectively (with DVFS and backup-delaying enabled). It shows that for PPA, promotion times for B_1, B_2 and B_3 are 0.3, 0 and 10, respectively. This means B_2 gets promoted as soon as they arrive, and it could not be delayed at all. B_1 and B_3 are only marginally delayed. In contrast, for RPPA, promotion times for B_1, B_2 and B_3 are found as 13.2, 16 and 24.7, respectively. This big improvement in backup-delaying translates to more backup cancellation which reduces the overall energy consumption. With DVFS and backup-delaying enabled, RPPA consumes only 9.42 mJ of energy, which represents about 42% and 51% improvement compared to RMS and PPA, respectively.

We also observe that reversing the priorities in RPPA by itself (without dual-queue based delaying) does not help in consuming less energy, as it can be seen in Figure 8a and 8b, which consume 26.52 mJ and 22.23 mJ energy, respectively. However, when RPPA is combined with the non work-conserving dual priority algorithm, then its big potential for energy consumption becomes clear (in this case, giving more than 55% improvement.)

6. Algorithm MPB-PS

This section describes the algorithm executed in the *online* phase of our framework, called the *Mixed Primary/Backup Periodic Scheduling (MPB-PS)* algorithm. In this algorithm, the runtime events are processed on the HP and LP core separately. In the offline phase, the tasks are allocated to the HP and LP cores, priority assignment



(a) Schedule for RPPA (Without DVFS or Dual Queue)

(b) Schedule for RPPA (with DVFS but without Dual-Queue)

(c) Schedule for RPPA (with DVFS and Dual Queue

Figure 8: Schedules for RPPA

is made to tasks on each core, and the execution frequency and promotion times are computed.

At runtime, when a primary task is released, its backup copy is also released on the alternate core (with the same deadline). There are four important events that our runtime algorithm needs to consider: task release, completion, promotion and cancellation. The details of this algorithm are given in Algorithm 1.

On each processing core, we have two queues: the upper and the lower queue. Tasks are eligible for execution only if they are in the upper queue. Backup copies are initially put to the lower queue, and they get promoted to the upper queue at the precomputed promotion times. As shown in Algorithm 1, when a task is released, it is checked whether it is a primary or backup copy, and then it is added to the appropriate queue.

A timer for the "promotion event" is set in case of a backup task. After all events, the highest priority task in the upper queue (which may be primary or backup) on each core is dispatched, possibly preempting any running low-priority task. The algorithm sets the task's frequency to the precomputed frequency value if it is a primary task, otherwise it is executed at the maximum speed on the corre-

sponding core.

When a task completes, the corresponding actions are shown in Algorithm 1. An *acceptance test* is run to check whether there is an error in the task's output. If no error is detected, then its alternate copy is cancelled (on the alternate core.) If an error is detected, the algorithm does not take additional steps: it is expected that the alternate copy on the other core should produce correct results before the deadline. On the event when a backup task is promoted, it is moved to the upper queue.

Algorithm 1 MPB-PS

Event: A task τ_i (B_i) is released at time t
  if released task is *primary* **then**
    Add τ_i to the upper queue at the proper priority level
  else
    Add B_i to the lower queue
    Let Y_i be the promotion time computed at offline phase
    Set a timer for promotion event at $t + Y_i$
  end if
  Dispatch highest priority tasks in the upper queues of both cores at pre-computed frequencies
Event: A task τ_i (B_i) completes
  Run *acceptance test* for detection of transient fault in task τ_i (B_i)
  if no error is detected **then**
    Generate a "task cancelled" event for the alternate copy $B_i(\tau_i)$ on the alternate core
  end if
  Dispatch highest priority tasks in the upper queues of both cores at pre-computed frequencies
Event: Timer signals the promotion time of the backup task B_i
  Move B_i from the lower to the upper queue on its core at the proper priority level
  Dispatch B_i on its core if it has the highest priority in the upper queue at the maximum frequency
Event: A task τ_i (B_i) is cancelled
  if task τ_i (B_i) is currently executing **then**
    Dispatch the highest priority task in the upper queue of the core where τ_i (B_i) is cancelled
  end if

This runtime algorithm ensures that a primary copy of a task is executed at the scaled speed, while backup tasks are executed at maximum speed of its core. Whenever one of them completes, the other one gets cancelled to conserve energy. In case of a permanent fault, the remaining core can execute one copy of each task without missing any deadline. At each invocation, this algorithm runs in $O(n)$ time, where n is the number of tasks on each core.

7. Experimental Evaluation

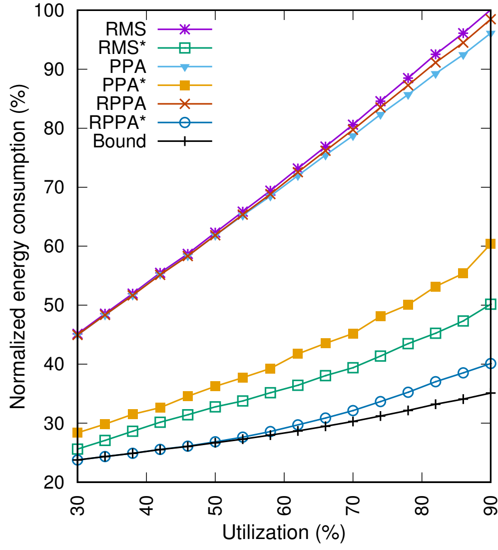
We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. The tasks are partitioned using the LSPB scheme (Section 4.1). The priority assignment schemes RMS, PPA and RPPA are evaluated, along with their variants which enable the dual queue based backup delaying technique (denoted as RMS*, PPA* and RPPA*, respectively). For all cases, we used DVFS to scale the frequency of the primary tasks and the *Sys-Clock* [35] algorithm was used for both cores.

We also implemented a scheme named *Bound*, where we remove all the backup tasks and allow the primary copies to scale their speed. This scheme does not offer any fault tolerance; but it is used as a theoretical lower bound on the energy consumption of all six schemes with fault tolerance features and backup task overheads.

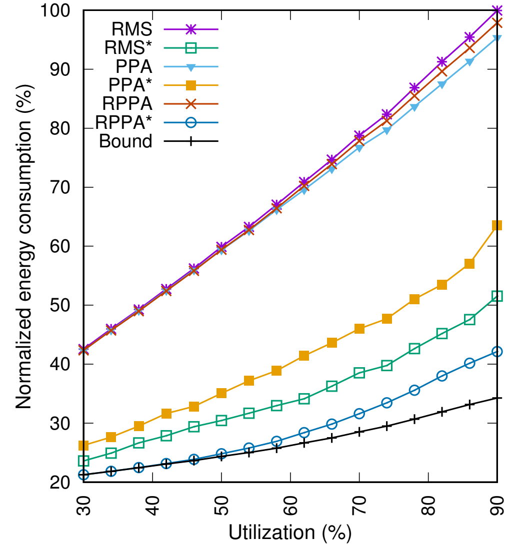
We simulated dual core systems with $f_{max}^{HP} = 1.0$ and f_{max}^{LP} varied from 0.6 to 1.0. Due to space limitations, we will show the results for $f_{max}^{LP} = 0.8$, and analyze the impact of varying f_{max}^{LP} in a separate plot.

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. Task periods are randomly chosen from a log uniform distribution ranging from 10 to 100. Hence, $U = (\sum \frac{C_i^{LP}}{P_i})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [36] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U.

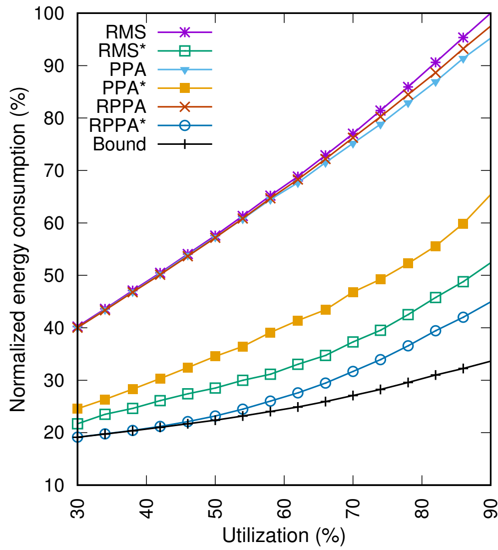
It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [37]. Therefore, as in [16, 37], we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which



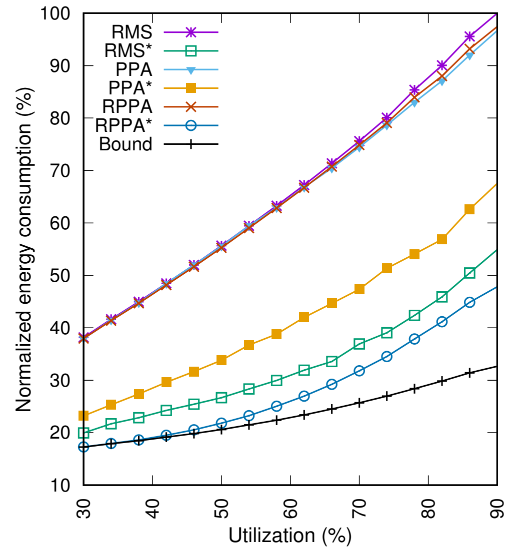
(a) Impact of Utilization ($f^{LP}_{max} = 0.6$)



(b) Impact of Utilization ($f^{LP}_{max} = 0.7$)



(c) Impact of Utilization ($f^{LP}_{max} = 0.8$)



(d) Impact of Utilization ($f^{LP}_{max} = 0.9$)

Figure 9: Impact of Utilization

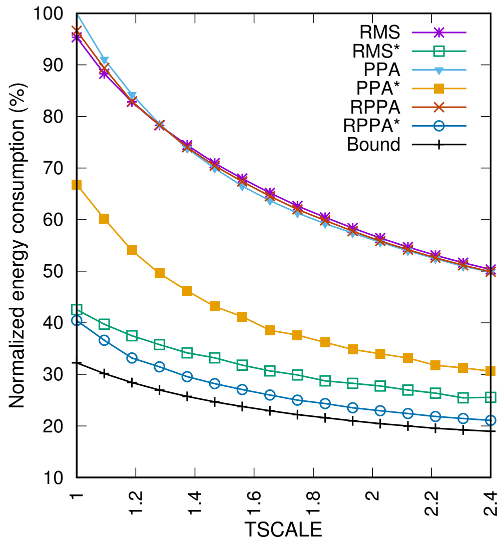
models how execution time changes on the LP core for a given task, τ_i. Typical values for $tscale_i$ are reported to be in the range [1.4, 2.3] [37]. Moreover, following [16], we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as $\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$. From experimental measurements, it has been found that $1.4 \le 1/(tscale_i * pscale_i) \le 2.1$ [37]. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within the ranges suggested in [37]. Specifically, $1.4 \le tscale_i \le 2.3$ and $1.4 \le 1/(tscale_i * pscale_i) \le 2.1$ 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. 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. The results in each plot are normalized with respect to the highest energy consumption of any scheme in that plot.

Impact of Utilization. Figures 9a, b, c, and d show the impact of utilization on normalized energy consumption for increasingly faster LP core (for f_{max}^{LP} set to 0.6, 0.7, 0.8 and 0.9, respectively). As expected, the normalized energy consumption of all schemes increase with the load. However, some schemes can save more energy than others. It also shows that, in all the cases the work-conserving RMS, PPA and RPPA schemes (without dual-queue based back-up delaying) perform worst and they perform very close to each other. This is because, in these schemes, the backup-delaying mechanism is not used, and therefore, backup copy executions overlapping with primaries are very frequent.

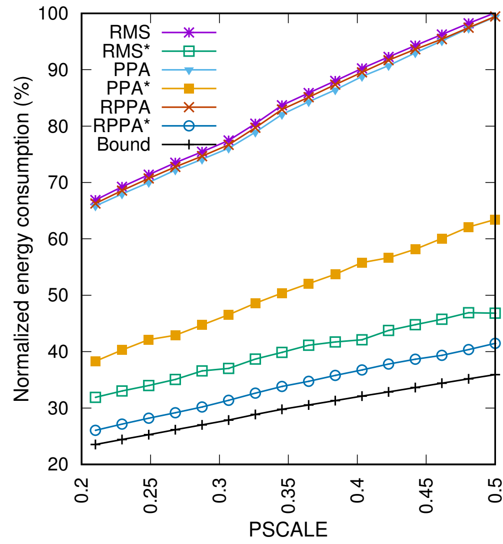
This problem is addressed by enabling the dual-queue based backup-delaying and non-work-conserving schedules in RMS*, PPA* and RPPA* schemes. As shown in Figure 9, PPA* can save more than 35% energy at average compared to the no-backup-delay schemes, throughout the entire range of system utilization values. RMS* performs even better than PPA* and it saves about 13% more energy on low-load and up to 20% for high-load task sets. Our proposed scheme, RPPA* performs the best and it saves 32% more energy than PPA*, and about 18% more energy than the RMS* scheme.

The performance level of PPA* warrants some elaboration. If backup copies are assigned lower priorities than primary copies, then due to the fact that primary copies are being slowed down through DVFS, we have only little room to delay the low-priority backups at run-time. This causes the backups to get activated early in the schedules, which results in increased energy consumption. On the other hand, the RPPA* scheme assigns higher priorities to the backup copies and the worst-case response time of backup tasks does not include the DVFS-enabled primary tasks, which allows the backups to remain in the lower queue for much longer time. This allows us to significantly delay, and in many cases, eventually cancel them. If a backup needs to be activated, it gets a higher priority on the processor, enabling it to finish still before deadline. This effect is reflected in the results and the RPPA* scheme performs much better throughout the entire spectrum. It performs very close to *Bound* for low utilization and it drifts away only moderately as the load increases. RMS* yields somewhat better results than PPA* (by virtue of the fact that some back-up tasks accidentally receive high priority based on their small periods), however, it is consistently worse than RPPA* in the entire spectrum. These observations hold true for all the utilization and the maximum speed configurations for the LP core.

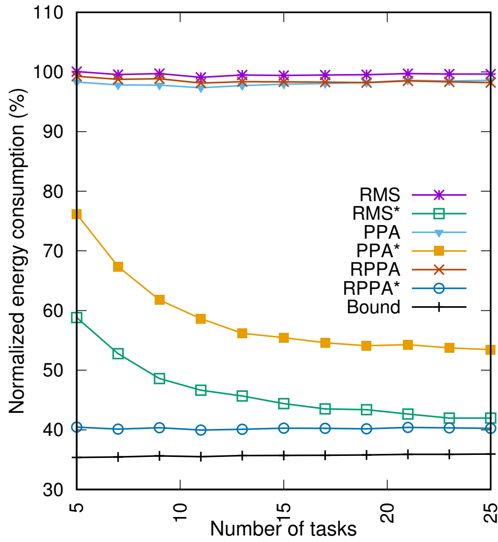
Impact of *tscale*. Figure 10a shows the impact of varying *tscale* for the tasks, while keeping the system utilization at 65%. A lower *tscale* means the increase in task cycle requirements is modest on the LP core, indicating higher energy efficiency. The figure shows that the overall energy consumption of all schemes decrease as *tscale* increases. This is because since the utilization is fixed, a higher *tscale* value represents a lower number of cycles for the HP core, and the overall energy consumption decreases. The results show that the no-backup-delaying schemes. RMS, PPA and RPPA are performing the worst throughout the entire region. PPA* scheme improves energy consumption by about 35% compared to PPA. RMS* outperforms PPA* by a moderate amount of 20%. The best performing scheme is RPPA*, which is about 15% better than RMS* and it performs very close to *Bound* in the entire *tscale* region. This is because the coupling of RPPA priority assignment and dual-queue based backup-delaying techniques was able to cancel many of the backup executions



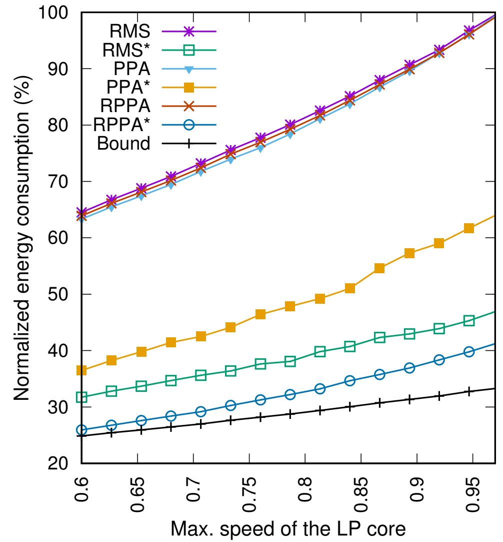
(a) Impact of tscale ($f_{max}^{LP} = 0.8$, Load = 65%)



(b) Impact of pscale ($f_{max}^{LP} = 0.8$, Load = 65%)



(c) Impact of Number of Tasks ($f_{max}^{LP} = 0.8$, Load = 65%)



(d) Impact of Max. speed of LP core (Load = 65%)

Figure 10: Impact of various system parameters

and reduce the overall energy consumption close to *Bound* (which does not consider the back-ups).

Impact of *pscale*. The impact of *pscale* on energy consumption is demonstrated in Figure 10b. As *pscale* grows, the LP core becomes less power efficient, and the effect is visible in the results. All the schemes show increased energy consumption with growing *pscale*, however, the RPPA* schemes can keep it very low. Throughout the entire spectrum of *pscale*, RPPA* schemes perform about 35% better than PPA* schemes, and about 20% better than RMS* schemes. Due to the effective cancellation of back-ups, RPPA* was able to conserve a lot of energy and it performs very close to *Bound* (within 10%.)

Impact of Number of Tasks. Figure 10c

shows the impact of number of tasks. It shows that RPPA* scheme performs the best throughout the entire spectrum and stays very close to *Bound*. The schemes with no backup-delaying shows very high energy consumption regardless of the number of tasks. The PPA* and RMS* schemes show higher energy consumption for small number of tasks, but as the number of tasks grow their energy consumption decreases. This is because, increasing the number of tasks while keeping the utilization same increases the granularity of the task set (giving smaller back-up tasks on the average), and the proposed algorithms are able to cancel back-ups more effectively. The observation that RPPA* outperforms PPA* and RMS* is still prevalent in these results.

Impact of the maximum speed of the LP core. Figure 10d shows the impact when we change the maximum speed of the LP core relative to the HP core, keeping the system utilization at 65%. As the LP core's maximum speed grows, its capacity and the system's actual workload also grows. That is reflected in the results by showing increased energy consumption for higher maximum speed of the LP core. The plot also shows that throughout our entire range of experiments, RPPA* schemes perform about 15% better than RMS* schemes, and about 35% better than PPA* schemes. *Bound* performs very close (within 2%) to RPPA* for lower values of f^{LP}_{max}, and it drifts away slowly as both cores' speeds become more similar.

8. Related Work

The research community has been recently exploring various aspects of heterogeneous multicore computing. In this section we review some of those recent works with particular emphasis on energy management, real-time operation, and/or fault tolerance.

Xu et al. [38] considered a framework which provides reliability on heterogeneous embedded systems and also minimizes energy consumption. They used a method to transform the application's reliability goal in to each task's reliability requirement, then proposed a method to minimize their energy consumption. They considered DAG-based embedded system applications; however, they did not consider any real-time constraints.

Devaraj [1] proposed a solution to the problem of scheduling real-time tasks on heterogenous systems which guarantees timeliness and feasibility. They used a linear programming algorithm to find feasible schedules for real-time task sets considering heterogeneous execution platforms. Li et al. [39] proposed a framework to execute real time applications on heterogeneous multicore processors, which minimizes temperature and energy consumption. They considered a graph based application model and scheduled the application tasks on to the processing cores in an energy/thermal-aware manner. Their results on real-world applications demonstrated the effectiveness of their approach, however, they did not consider task replication or fault-tolerance in their work.

Many real time systems are deployed in safety-critical environment which require very high reliability, therefore, fault-tolerant real time scheduling has been an active research area, with recent emphasis on heterogeneous systems. Zhou et al. [40] considered heterogeneous platforms for applications with deadline constraints and reliability. They developed a "earliest finish-time" based algorithm for heterogeneous MPSoCs, which executes graph based real-time tasks. Their solution strives to maximize reliability to transient and permanent faults, however, they did not take energy consumption in to account. Liu et al. [41] proposed an adaptive fault-tolerant scheduling mechanism for real time systems executing on heterogeneous multiprocessors. They used task-replication and computed the number of required replicas which guarantees reliability and deadline. However, their work also did not address the energy consumption dimension.

A number of recent research studies explored the joint management of timeliness, energy consumption, and reliability. For instance, Bansal et al. [42] proposed energy aware fixed priority scheduling for real-time tasks in which they considered execution-preferences for different tasks (primary or backup). They improved the energy consumption of the preference-oriented scheduling proposed in [31], by applying DVFS and DPM techniques. However, their work was based on a single processor system, and cannot be easily generalized to multicore platforms or tolerate permanent faults. Zhao et al. [8] proposed an energy-efficient standby-sparing technique, which executes a mix of high and low criticality tasks. They scaled the primary processor with DVFS, and also, they extended their algorithm for cluster/island systems, however, their work did not consider heterogeneous processors.

Safari et al. [43] also proposed a low-energy standby-sparing scheme for mixed criticality sys-

tems, in which, they considered graph based real time applications, and executed it on multicore system with fault tolerance. They used convex optimization to exploit DVFS along with DPM to save energy under timeliness and reliability constraints. That paper considered homogeneous multicore systems. Kumar et al. [44] addressed heterogeneous multicore systems and proposed a framework for energy-efficient scheduling of periodic real-time tasks with fault tolerance. They modeled the problem as a constraint optimization problem and also proposed several low-overhead heuristics. However, their setting does not allow preemption of real-time tasks, which may have a strong and negative impact on the schedulability of periodic real-time task sets for which preemptive scheduling is the norm [23].

Our research group also explored the subtle interaction of real-time, fault-tolerant, and energy-aware operation on heterogeneous multicore systems. For instance, the work in [16] addresses the standby sparing setting for fault tolerance on heterogeneous systems for real time tasks. We also studied mixed primary/backup scheduling on heterogeneous cores for both frame based [19] and DAG-based [21] real time applications. However, all those studies considered frame-based applications where tasks share a common deadline/period. The present work considers fixed-priority preemptive periodic real-time applications, and it has a wider applicability.

9. Conclusions

In this paper, we investigated energy-aware scheduling of preemptive fixed-priority tasks upon heterogeneous cores. The fault tolerance requirements dictate scheduling a separate backup copy of each primary real-time job on a separate core. In addition to scaling the primary jobs through DVFS to save energy, we developed a comprehensive framework to maximize the cancellation opportunities for back-ups through the use of priority assignment and dual queue based task delaying. We evaluated the performance of the proposed schemes under different workload conditions. Our proposed Reverse Preference-Oriented Priority Assignment (RPPA) scheme is shown to yield very high energy savings, by virtue of exploiting the dual-queue based delaying mechanism maximally.

References

[1] R. Devaraj, A solution to drawbacks in capturing execution requirements on heterogeneous platforms, The Journal of Supercomputing (2020) 1–16.

[2] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, V. Kumar, Hass: a scheduler for heterogeneous multicore systems, ACM SIGOPS Operating Systems Review 43 (2) (2009) 66–75.

[3] S. I. Kim, J.-K. Kim, A method to construct task scheduling algorithms for heterogeneous multi-core systems, IEEE Access 7 (2019) 142640–142651.

[4] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, S. Gobriel, Energy-efficient thread assignment optimization for heterogeneous multicore systems, ACM Transactions on Embedded Computing Systems (TECS) 14 (1) (2015) 1–26.

[5] A. Gamatié, G. Devic, G. Sassatelli, S. Bernabovi, P. Naudin, M. Chapman, Towards energy-efficient heterogeneous multicore architectures for edge computing, IEEE Access 7 (2019) 49474–49491.

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

[7] D. K. Pradhan, Fault-tolerant computer system design, Prentice-Hall, Inc., 1996.

[8] M. Zhao, D. Liu, X. Jiang, W. Liu, G. Xue, C. Xie, Y. Yang, Z. Guo, CASS: Criticality-aware standby-sparing for real-time systems, Journal of Systems Architecture 100 (2019) Article No. 101661.

[9] M. A. Haque, H. Aydin, D. Zhu, On reliability management of energy-aware real-time systems through task replication, IEEE Transactions on Parallel and Distributed Systems 28 (3) (2016) 813–825.

[10] S. Safari, M. Ansari, G. Ershadi, S. Hessabi, On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration, IEEE Transactions on Parallel and Distributed Systems 30 (10) (2019) 2338–2354.

[11] M. A. Haque, H. Aydin, D. Zhu, Energy-aware standby-sparing for fixed-priority real-time task sets, Journal of Sustainable Computing 6 (2015) 81–93.

[12] A. Ejlali, B. M. Al-Hashimi, P. Eles, Low-energy standby-sparing for hard real-time systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 31 (3) (2012) 329–342.

[13] Y. Guo, D. Zhu, H. Aydin, Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems, in: 2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE, 2013, pp. 62–71.

[14] K. Arora, S. Bansal, R. K. Bansal, Energy aware fault tolerant fixed priority task scheduling in multiprocessor system, in: 2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence), IEEE, 2018, pp. 658–663.

[15] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, H. Xiong, Energy-efficient real-time scheduling of DAG tasks, ACM Transactions on Embedded Computing Systems (TECS) 17 (5) (2018) 1–25.

[16] A. Roy, H. Aydin, D. Zhu, Energy-aware standby-sparing on heterogeneous multicore systems, in: 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), IEEE, 2017, pp. 1–6.

[17] P. P. Nair, R. Devaraj, A. Sarkar, Fest: Fault-tolerant energy-aware scheduling on two-core heterogeneous platform, in: 2018 8th International Symposium on Embedded Computing and System Design (ISED), IEEE, 2018, pp. 63–68.
[18] M. Ansari, M. Pasandideh, J. Saber-Latibari, A. Ejlali, Meeting thermal safe power in fault-tolerant heterogeneous embedded systems, IEEE Embedded Systems Letters (2019) 29–32.
[19] A. Roy, H. Aydin, D. Zhu, Energy-efficient primary/backup scheduling techniques for heterogeneous multicore systems, in: 2017 Eighth International Green and Sustainable Computing Conference (IGSC), IEEE, 2017, pp. 1–8.
[20] S. Moulik, R. Chaudhary, Z. Das, Hears: A heterogeneous energy-aware real-time scheduler, Microprocessors and Microsystems 72 (2020) Article No. 102939.
[21] A. Roy, H. Aydin, D. Zhu, Energy-efficient fault tolerance for real-time tasks with precedence constraints on heterogeneous multicore systems, in: 2019 Tenth International Green and Sustainable Computing Conference (IGSC), IEEE, 2019, pp. 1–8.
[22] M. Ansari, J. Saberlatibari, S. M. Pasandideh, A. Ejlali, Simultaneous management of peak-power and reliability in heterogeneous multicore embedded systems, IEEE Transactions on Parallel and Distributed Systems (2019) 623–633.
[23] J. W. S. Liu, Real-Time Systems, Pearson Education, 2000.
[24] M. A. Haque, H. Aydin, D. Zhu, Energy-aware standby-sparing technique for periodic real-time applications, in: 2011 IEEE 29th International Conference on Computer Design (ICCD), IEEE, 2011, pp. 190–197.
[25] A. Ejlali, B. M. Al-Hashimi, P. Eles, A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems, in: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, 2009, pp. 193–202.
[26] M. K. Tavana, M. Salehi, A. Ejlali, Feedback-based energy management in a standby-sparing scheme for hard real-time systems, in: 2011 IEEE 32nd Real-Time Systems Symposium, IEEE, 2011, pp. 349–356.
[27] R. Davis, A. Wellings, Dual priority scheduling, in: Proceedings 16th IEEE Real-Time Systems Symposium, IEEE, 1995, pp. 100–109.
[28] D. Zhu, R. Melhem, D. Mossé, The effects of energy management on reliability in real-time embedded systems, in: IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004., IEEE, 2004, pp. 35–40.
[29] M. Joseph, P. Pandya, Finding response times in a real-time system, The Computer Journal 29 (5) (1986) 390–395.
[30] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.
[31] R. Begam, Q. Xia, D. Zhu, H. Aydin, Preference-oriented fixed-priority scheduling for periodic real-time tasks, Journal of Systems Architecture 69 (2016) 1–14.
[32] N. C. Audsley, On priority assignment in fixed priority scheduling, Information Processing Letters 79 (1) (2001) 39–44.
[33] N. C. Audsley, Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, Citeseer, 1991.
[34] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Transactions on Computers 58 (10) (2009) 1382–1397.
[35] S. Saewong, R. Rajkumar, Practical voltage-scaling for fixed-priority rt-systems, in: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings., IEEE, 2003, pp. 106–114.
[36] P. Emberson, R. Stafford, 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.
[37] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, S. Vishin, Power-performance modeling on asymmetric multi-cores, in: 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), IEEE, 2013, pp. 1–10.
[38] H. Xu, R. Li, C. Pan, K. Li, Minimizing energy consumption with reliability goal on heterogeneous embedded systems, Journal of Parallel and Distributed Computing 127 (2019) 44–57.
[39] T. Li, T. Zhang, G. Yu, J. Song, J. Fan, Minimizing temperature and energy of real-time applications with precedence constraints on heterogeneous mpsoc systems, Journal of Systems Architecture 98 (2019) 79–91.
[40] J. Zhou, M. Zhang, J. Sun, T. Wang, X. Zhou, S. Hu, Drheft: Deadline-constrained reliability-aware heft algorithm for real-time heterogeneous mpsoc systems, IEEE Transactions on Reliability (2020) 1–12.
[41] Y. Liu, J. Liu, Z. Zhu, C. Deng, Z. Ren, X. Xu, Adaptive fault-tolerant scheduling in heterogeneous real-time systems, in: 2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA), IEEE, 2019, pp. 982–987.
[42] S. Bansal, R. K. Bansal, K. Arora, Energy-cognizant scheduling for preference-oriented fixed-priority real-time tasks, Journal of Systems Architecture 108 (2020) Article No. 101743.
[43] S. Safari, S. Hessabi, G. Ershadi, LESS-MICS: A low energy standby-sparing scheme for mixed-criticality systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020) 1–10.
[44] N. Kumar, J. Mayank, A. Mondal, Reliability aware energy optimized scheduling of non-preemptive periodic real-time tasks on heterogeneous multiprocessor system, IEEE Transactions on Parallel and Distributed Systems (2019) 871–885.